What this book will do for you...

- Implement PostgreSQL features for reliability and performance
- Harness the power of the latest PostgreSQL 9.4 features
- Manage PostgreSQL versions 9.0, 9.1, 9.2, 9.3, and 9.4
- Advanced technical tips for experienced users
- Explore best practices for planning and designing live databases
- Select and implement robust backup and recovery techniques
- Concise and clear guidance on replication and high availability
- Latest details on Logical Replication and Bi-Directional Replication

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

PostgreSQL 9 Administration Cookbook
Second Edition

Simon Riggs            Gianni Ciolli
Hannu Krosing        Gabriele Bartolini

Over 150 recipes to help you run an efficient PostgreSQL database in the cloud

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'First Steps'
- A synopsis of the book’s content
- More information on *PostgreSQL 9 Administration Cookbook Second Edition*

About the Authors

**Simon Riggs** is the CTO of 2ndQuadrant and an active PostgreSQL committer. He has contributed to PostgreSQL as a major developer for more than 10 years, having written and designed many new features in every release over that period. His feature credits include replication, performance, business intelligence, management, and security. Under his guidance, 2ndQuadrant is now a leading developer of open source PostgreSQL and a platinum sponsor of the PostgreSQL Project, serving hundreds of clients in USA, Europe, Asia-Pacific, the Middle East, and Africa.

Simon is a frequent speaker at many conferences and is well known for his speeches on PostgreSQL Futures and different aspects of replication. He has worked with many different databases as a developer, architect, data analyst, and designer with companies across USA and Europe for nearly 30 years.

**Gianni Ciolli** is a principal consultant at 2ndQuadrant Italia, where he has been working since 2008 as a developer, consultant, and trainer. He has spoken at PostgreSQL conferences in Europe and abroad, and his other IT skills include functional languages and symbolic computing.

Gianni has a PhD in mathematics, and is the author of published research on Algebraic Geometry, Theoretical Physics, and Formal Proof Theory. He previously worked at the University of Florence as a researcher and teacher.
Gianni has been working on free and open source software for almost 20 years. From 2001 to 2004, he was a cofounder and the president of PLUG, short for Prato Linux User Group. He organized many sessions of the Italian PostgreSQL conference, and in 2013, he was elected to the board of ITPUG, the Italian PostgreSQL Users Group. He currently lives in London with his son. His other interests include music, drama, poetry, and sport—athletics in particular, where he competes in combined events.

Hannu Krosing is a principal consultant at 2ndQuadrant and a technical advisor at Ambient Sound Investments. As the original database architect at Skype Technologies, he was responsible for designing the SkyTools suite of replication and scalability technology. He has worked with and contributed to the PostgreSQL project for more than 12 years.

Gabriele Bartolini is a long-time open source programmer, a principal consultant with 2ndQuadrant, and an active member of the international PostgreSQL community. Gabriele has a degree in statistics from the University of Florence. His areas of expertise are data mining and data warehousing, and he has worked on web traffic analysis in Australia and Italy.

He currently lives in Prato, a small but vibrant city located in the northern part of Tuscany, Italy. His second home is Melbourne, Australia, where he studied at Monash University and worked in the ICT sector. Gabriele's hobbies include playing his Fender Stratocaster electric guitar and "calcio" (football or soccer, depending on which part of the world you come from).
PostgreSQL 9 Administration Cookbook  
Second Edition

PostgreSQL is an advanced SQL database server available on a wide range of platforms, and is fast becoming one of the world's most popular server databases, with an enviable reputation for performance, stability, and an enormous range of advanced features. PostgreSQL is one of the oldest open source projects, completely free to use, and developed by a very diverse worldwide community. Most of all, it just works!

One of the clearest benefits of PostgreSQL is that it is open source, meaning that you have a very permissive license to install, use, and distribute PostgreSQL without paying anyone any fees or royalties. On top of that, PostgreSQL is well-known as a database that stays up for long periods, and requires little or no maintenance in many cases. Overall, PostgreSQL provides a very low total cost of ownership.

*PostgreSQL 9 Administration Cookbook Second Edition* offers the information you need to manage your live production databases on PostgreSQL. The book contains insights straight from the main author of the PostgreSQL replication and recovery features, and the database architect of the most successful start-up that uses PostgreSQL: Skype. This hands-on guide will assist developers working on live databases, supporting web or enterprise software applications using Java, Python, Ruby, and .NET from any development framework. It's easy to manage your database when you've got *PostgreSQL 9 Administration Cookbook Second Edition* at hand.

This practical guide gives you quick answers to common questions and problems, building on the authors' experience as trainers, users, and core developers of the PostgreSQL database server.

Each technical aspect is broken down into short recipes that demonstrate solutions with working code, and then explain why and how they work. This book is intended to be a desk reference for both new users and technical experts.

The book covers all the latest features available in PostgreSQL 9. Soon, you will be running a smooth database with ease.
What This Book Covers

Chapter 1, First Steps, covers topics such as introduction to PostgreSQL 9, downloading and installing PostgreSQL 9, connecting to a PostgreSQL server, enabling server access to network/remote users, using graphical administration tools, using the psql query and scripting tools, changing your password securely, avoiding hardcoding your password, using a connection service file, and troubleshooting a failed connection.

Chapter 2, Exploring the Database, helps you identify the version of the database server you are using and also the server uptime. This chapter helps you locate the database server files, database server message log, and database's system identifier. It shows you how to list a database on the database server and contains recipes that let you know the number of tables in your database, how much disk space is used by the database and tables, which are the biggest tables, how many rows a table has, how to estimate rows in a table, and how to understand object dependencies.

Chapter 3, Configuration, covers topics such as reading the fine manual (RTFM), planning a new database, changing parameters in your programs, the current configuration settings, parameters that are at non-default settings, updating the parameter file, setting parameters for particular groups of users, the basic server configuration checklist, and adding an external module to the PostgreSQL server.

Chapter 4, Server Control, provides information about starting the database server manually, stopping the server quickly and safely, stopping the server in an emergency, reloading the server configuration files, restarting the server quickly, preventing new connections, restricting users to one session each, and pushing users off the system. It contains recipes that help you decide on a design for multitenancy. You can learn how to use multiple schemas, give users their own private database, run multiple database servers on one system, and set up a connection pool.

Chapter 5, Tables and Data, guides you through the process of choosing good names for database objects, handling objects with quoted names, enforcing the same name and the same definition for columns, identifying and removing duplicate rows, preventing duplicate rows, finding a unique key for a set of data, generating test data, randomly sampling data, loading data from a spreadsheet, and loading data from flat files.

Chapter 6, Security, provides recipes on revoking user access to a table, granting user access to a table, creating a new user, temporarily preventing a user from connecting, removing a user without dropping their data, checking whether all users have a secure password, giving limited superuser powers to specific users, auditing DDL changes, auditing data changes, integrating with LDAP, connecting using SSL, and encrypting sensitive data.
Chapter 7, Database Administration, covers useful topics such as writing a script wherein either all succeed or all fail, writing a psql script that exits immediately after the first error, performing actions on many tables, adding or removing columns from tables, changing the data type of a column, adding or removing schemas, moving objects between schemas, adding or removing tablespaces, moving objects between tablespaces, accessing objects in other PostgreSQL databases, and making views updatable.

Chapter 8, Monitoring and Diagnosis, provides recipes that answer questions such as, "Is the user connected? What are they running? Are they active or blocked? Who is blocking them? Is anybody using a specific table? When did anybody last use it? How much disk space is used by temporary data? And why are my queries slowing down?" It also helps you with investigating and reporting a bug, producing a daily summary report of log file errors, killing a specific session, and resolving an in-doubt prepared transaction.

Chapter 9, Regular Maintenance, includes useful recipes on controlling automatic database maintenance, avoiding auto-freezing and page corruptions, avoiding transaction wraparound, removing old prepared transactions, actions for heavy users of temporary tables, identifying and fixing bloated tables and indexes, maintaining indexes, finding unused indexes, carefully removing unwanted indexes, and planning maintenance.

Chapter 10, Performance and Concurrency, covers topics such as finding slow SQL statements, collecting regular statistics from pg_stat* views, finding out what makes SQL slow, reducing the number of rows returned, simplifying complex SQL code, speeding up queries without rewriting them, finding out why a query is not using an index, forcing a query to use an index, using optimistic locking, and reporting performance problems.

Chapter 11, Backup and Recovery, provides useful information about backup and recovery of your PostgreSQL database through recipes on understanding and controlling crash recovery, planning backups, hot logical backup of one database, hot logical backup of all databases, hot logical backup of all tables in a tablespace, backup of database object definitions, standalone hot physical database backup, and hot physical backup and continuous archiving. It also includes topics such as recovery of all databases; recovery to a point in time; recovery of a dropped or damaged table, database, or tablespace; improving performance of backup and recovery; and incremental/differential backup and restore.

Chapter 12, Replication and Upgrades, covers replication best practices; setting up file-based or streaming replication; setting up streaming replication security; Hot Standby and read scalability; managing Streaming Replication; using repmgr; using replication slots; monitoring replication; performance and synchronous replication; delaying, pausing, and synchronizing replication; Logical Replication; Bi-Directional Replication; archiving transaction log data, upgrading minor release upgrades, and major release upgrades, both in-place and online.
In this chapter, we will cover the following recipes:

- Getting PostgreSQL
- Connecting to the PostgreSQL server
- Enabling access for network/remote users
- Using graphical administration tools
- Using the psql query and scripting tool
- Changing your password securely
- Avoiding hardcoding your password
- Using a connection service file
- Troubleshooting a failed connection

**Introduction**

PostgreSQL is a feature-rich, general-purpose database management system. It's a complex piece of software, but every journey begins with the first step.

We'll start with your first connection. Many people fall at the first hurdle, so we'll try not to skip that too swiftly. We'll quickly move on to enabling remote users, and from there, we will move to access through GUI administration tools.

We will also introduce the `psql` query tool, which is the tool used to load our sample database, as well as many other examples in the book.

For additional help, we've included a few useful recipes that you may need for reference.
Introducing PostgreSQL 9

PostgreSQL is an advanced SQL database server, available on a wide range of platforms. One of the clearest benefits of PostgreSQL is that it is open source, meaning that you have a very permissive license to install, use, and distribute PostgreSQL without paying anyone fees or royalties. On top of that, PostgreSQL is well-known as a database that stays up for long periods and requires little or no maintenance in most cases. Overall, PostgreSQL provides a very low total cost of ownership.

PostgreSQL is also noted for its huge range of advanced features, developed over the course of more than 20 years of continuous development and enhancement. Originally developed by the Database Research Group at the University of California, Berkeley, PostgreSQL is now developed and maintained by a huge army of developers and contributors. Many of those contributors have full-time jobs related to PostgreSQL, working as designers, developers, database administrators, and trainers. Some, but not many, of those contributors work for companies that specialize in support for PostgreSQL, like we (the authors) do. No single company owns PostgreSQL, nor are you required (or even encouraged) to register your usage.

PostgreSQL has the following main features:

- Excellent SQL standards compliance up to SQL:2011
- Client-server architecture
- Highly concurrent design where readers and writers don't block each other
- Highly configurable and extensible for many types of applications
- Excellent scalability and performance with extensive tuning features
- Support for many kinds of data models: relational, document (JSON and XML), and key/value

What makes PostgreSQL different?

The PostgreSQL project focuses on the following objectives:

- Robust, high-quality software with maintainable, well-commented code
- Low maintenance administration for both embedded and enterprise use
- Standards-compliant SQL, interoperability, and compatibility
- Performance, security, and high availability

What surprises many people is that PostgreSQL's feature set is more comparable with Oracle or SQL Server than it is with MySQL. The only connection between MySQL and PostgreSQL is that these two projects are open source; apart from that, the features and philosophies are almost totally different.
One of the key features of Oracle, since Oracle 7, has been snapshot isolation, where readers don't block writers and writers don't block readers. You may be surprised to learn that PostgreSQL was the first database to be designed with this feature, and it offers a complete implementation. In PostgreSQL, this feature is called Multiversion Concurrency Control (MVCC), and we will discuss this in more detail later in this book.

PostgreSQL is a general-purpose database management system. You define the database that you would like to manage with it. PostgreSQL offers you many ways to work. You can use a normalized database model, augmented with features such as arrays and record subtypes, or use a fully dynamic schema with the help of JSONB and an extension named hstore. PostgreSQL also allows you to create your own server-side functions in any of a dozen different languages.

PostgreSQL is highly extensible, so you can add your own data types, operators, index types, and functional languages. You can even override different parts of the system using plugins to alter the execution of commands or add a new optimizer.

All of these features offer a huge range of implementation options to software architects. There are many ways out of trouble when building applications and maintaining them over long periods of time. Regrettably, we simply don't have space in this book for all the cool features for developers; this book is about administration, maintenance, and backup.

In the early days, when PostgreSQL was still a research database, the focus was solely on the cool new features. Over the last 15 years, enormous amounts of code have been rewritten and improved, giving us one of the most stable and largest software servers available for operational use.

You may have read that PostgreSQL was, or is, slower than My Favorite DBMS, whichever that is. It's been a personal mission of mine over the last ten years to improve server performance, and the team has been successful in making the server highly performant and very scalable. That gives PostgreSQL enormous headroom for growth.

Who is using PostgreSQL? Prominent users include Apple, BASF, Genentech, Heroku, IMDB.com, Skype, McAfee, NTT, The UK Met Office, and The U. S. National Weather Service. 5 years ago, PostgreSQL received well in excess of 1 million downloads per year, according to data submitted to the European Commission, which concluded, "PostgreSQL is considered by many database users to be a credible alternative."

We need to mention one last thing. When PostgreSQL was first developed, it was named Postgres, and therefore many aspects of the project still refer to the word "postgres"; for example, the default database is named postgres, and the software is frequently installed using the postgres user ID. As a result, people shorten the name PostgreSQL to simply Postgres, and in many cases use the two names interchangeably.

PostgreSQL is pronounced as "post-grez-q-l". Postgres is pronounced as "post-grez."

Some people get confused, and refer to "Postgre", which is hard to say, and likely to confuse people. Two names are enough, so please don't use a third name!

The following sections explain the key areas in more detail.
Robustness
PostgreSQL is robust, high-quality software, supported by automated testing for both features and concurrency. By default, the database provides strong disk-write guarantees, and the developers take the risk of data loss very seriously in everything they do. Options to trade robustness for performance exist, though they are not enabled by default.

All actions on the database are performed within transactions, protected by a transaction log that will perform automatic crash recovery in case of software failure.

Databases may be optionally created with data block checksums to help diagnose hardware faults. Multiple backup mechanisms exist, with full and detailed Point-In-Time Recovery, in case of the need for detailed recovery. A variety of diagnostic tools are available.

Database replication is supported natively. Synchronous Replication can provide greater than "5 Nines" (99.999 percent) availability and data protection, if properly configured and managed.

Security
Access to PostgreSQL is controllable via host-based access rules. Authentication is flexible and pluggable, allowing easy integration with any external security architecture.

Full SSL-encrypted access is supported natively. A full-featured cryptographic function library is available for database users.

PostgreSQL provides role-based access privileges to access data, by command type.

Functions may execute with the permissions of the definer, while views may be defined with security barriers to ensure that security is enforced ahead of other processing.

All aspects of PostgreSQL are assessed by an active security team, while known exploits are categorized and reported at http://www.postgresql.org/support/security/.

Ease of use
Clear, full, and accurate documentation exists as a result of a development process where doc changes are required. Hundreds of small changes occur with each release that smooth out any rough edges of usage, supplied directly by knowledgeable users.

PostgreSQL works in the same way on small or large systems and across operating systems.

Client access and drivers exist for every language and environment, so there is no restriction on what type of development environment is chosen now, or in the future.

SQL Standard is followed very closely; there is no weird behavior, such as silent truncation of data.
Text data is supported via a single data type that allows storage of anything from 1 byte to 1 gigabyte. This storage is optimized in multiple ways, so 1 byte is stored efficiently, and much larger values are automatically managed and compressed.

PostgreSQL has a clear policy to minimize the number of configuration parameters, and with each release, we work out ways to auto-tune settings.

**Extensibility**

PostgreSQL is designed to be highly extensible. Database extensions can be loaded simply and easily using `CREATE EXTENSION`, which automates version checks, dependencies, and other aspects of configuration.

PostgreSQL supports user-defined data types, operators, indexes, functions and languages.

Many extensions are available for PostgreSQL, including the PostGIS extension that provides world-class Geographical Information System (GIS) features.

**Performance and concurrency**

PostgreSQL 9.4 can achieve more than 300,000 reads per second on a 32-CPU server, and it benchmarks at more than 20,000 write transactions per second with full durability.

PostgreSQL has an advanced optimizer that considers a variety of join types, utilizing user data statistics to guide its choices.

PostgreSQL provides MVCC, which enables readers and writers to avoid blocking each other.

Taken together, the performance features of PostgreSQL allow a mixed workload of transactional systems and complex search and analytical tasks. This is important because it means we don't always need to unload our data from production systems and reload them into analytical data stores just to execute a few ad hoc queries. PostgreSQL's capabilities make it the database of choice for new systems, as well as the right long-term choice in almost every case.

**Scalability**

PostgreSQL 9.4 scales well on a single node up to 32 CPUs. PostgreSQL scales well up to hundreds of active sessions, and up to thousands of connected sessions when using a session pool. Further scalability is achieved in each annual release.

PostgreSQL provides multinode read scalability using the Hot Standby feature. Multinode write scalability is under active development. The starting point for this is Bi-Directional Replication (discussed in Chapter 12, Replication and Upgrades).
SQL and NoSQL

PostgreSQL follows SQL Standard very closely. SQL itself does not force any particular type of model to be used, so PostgreSQL can easily be used for many types of models at the same time, in the same database. PostgreSQL supports the more normal SQL language statement.

With PostgreSQL acting as a relational database, we can utilize any level of denormalization, from the full Third Normal Form, to the more normalized Star Schema models. PostgreSQL extends the relational model to provide arrays, row types, and range types.

A document-centric database is also possible using PostgreSQL's text, XML, and binary JSON (JSONB) data types, supported by indexes optimized for documents and by full text search capabilities.

Key/value stores are supported using the hstore extension.

Popularity

When MySQL was taken over some years back, it was agreed in the EU monopoly investigation that followed that PostgreSQL was a viable competitor. That's been certainly true, with the PostgreSQL user base expanding consistently for more than a decade.

Various polls have indicated that PostgreSQL is the favorite database for building new, enterprise-class applications. The PostgreSQL feature set attracts serious users who have serious applications. Financial services companies may be PostgreSQL’s largest user group, though governments, telecommunication companies, and many other segments are strong users as well. This popularity extends across the world; Japan, Ecuador, Argentina, and Russia have very large user groups, and so do USA, Europe, and Australasia.

Amazon Web Services' chief technology officer Dr. Werner Vogels described PostgreSQL as "an amazing database", going on to say that "PostgreSQL has become the preferred open source relational database for many enterprise developers and start-ups, powering leading geospatial and mobile applications".

Commercial support

Many people have commented that strong commercial support is what enterprises need before they can invest in open source technology. Strong support is available worldwide from a number of companies.

2ndQuadrant provides commercial support for open source PostgreSQL, offering 24 x 7 support in English and Spanish with bug-fix resolution times.

EnterpriseDB provides commercial support for PostgreSQL as well as their main product, which is a variant of Postgres that offers enhanced Oracle compatibility.
Many other companies provide strong and knowledgeable support to specific geographic regions, vertical markets, and specialized technology stacks.

PostgreSQL is also available as hosted or cloud solutions from a variety of companies, since it runs very well in cloud environments.

A full list of companies is kept up to date at http://www.postgresql.org/support/professional_support/.

**Research and development funding**

PostgreSQL was originally developed as a research project at the University of California, Berkeley in the late 1980s and early 1990s. Further work was carried out by volunteers until the late 1990s. Then, the first professional developer became involved. Over time, more and more companies and research groups became involved, supporting many professional contributors. Further funding for research and development was provided by the NSF. The project also received funding from the EU FP7 Programme in the form of the 4CaaST project for cloud computing and the AXLE project for scalable data analytics. AXLE deserves a special mention because it is a 3-year project aimed at enhancing PostgreSQL's business intelligence capabilities, specifically for very large databases. The project covers security, privacy, integration with data mining, and visualization tools and interfaces for new hardware. Further details of it are available at http://www.axleproject.eu.

Other funding for PostgreSQL development comes from users who directly sponsor features and companies selling products and services based around PostgreSQL.

**Getting PostgreSQL**

PostgreSQL is 100 percent open source software.

PostgreSQL is freely available to use, alter, or redistribute in any way you choose. Its license is an approved open source license, very similar to the Berkeley Distribution Software (BSD) license, though only just different enough that it is now known as The PostgreSQL License (TPL).

**How to do it...**

PostgreSQL is already being used by many different application packages, so you may find it already installed on your servers. Many Linux distributions include PostgreSQL as part of the basic installation, or include it with the installation disk.

One thing to be wary of, is that the version of PostgreSQL included may not be the latest release. It will typically be the latest major release that was available when that operating system release was published. There is usually no good reason to stick to that level—there is no increased stability implied there—and later production versions are just as well supported by the various Linux distributions as the earlier versions.
If you don't have a copy yet, or you don't have the latest version, you can download the source code or binary packages for a wide variety of operating systems from http://www.postgresql.org/download/.

Installation details vary significantly from platform to platform, and there aren't any special tricks or recipes to mention. Just follow the installation guide, and away you go! We've consciously avoided describing the installation processes here to make sure we don't garble or override the information published to assist you.

If you would like to receive e-mail updates of the latest news, then you can subscribe to the PostgreSQL announce mailing list, which contains updates from all the vendors that support PostgreSQL. You'll get a few e-mails each month about new releases of core PostgreSQL, related software, conferences, and user group information. It's worth keeping in touch with these developments.

For more information about the PostgreSQL announce mailing list, visit http://archives.postgresql.org/pgsql-announce/.

How it works...

Many people ask questions such as "how can this be free?", "are you sure I don't have to pay someone?", or "who gives this stuff away for nothing?"

Open source applications such as PostgreSQL work on a community basis, where many contributors perform tasks that make the whole process work. For many of those people, their involvement is professional, rather a hobby, and they can do this because there is generally a great value for both contributors and their employers alike.

You might not believe it. You don't have to, because It Just Works!

There's more...

Remember that PostgreSQL is more than just the core software. There is a huge range of websites offering add-ons, extensions, and tools for PostgreSQL. You'll also find an army of bloggers describing useful tricks and discoveries that will help you in your work.

Besides, there is a range of professional companies that are able to offer you help when you need it.
Connecting to the PostgreSQL server

How do we access PostgreSQL?

Connecting to the database is most people's first experience of PostgreSQL, so we want to make it a good one. Let's do it now and fix any problems we have along the way. Remember that a connection needs to be made securely, so there may be some hoops for us to jump through to ensure that the data we wish to access is secure.

Before we can execute commands against the database, we need to connect to the database server, giving us a session.

Sessions are designed to be long-lived, so you connect once, perform many requests, and eventually disconnect. There is a small overhead during connection. It may become noticeable if you connect and disconnect repeatedly, so you may wish to investigate the use of connection pools. Connection pools allow preconnected sessions to be served quickly to you when you wish to reconnect.

Getting ready

First, catch your database. If you don't know where it is, you'll probably have difficulty accessing it. There may be more than one database, and you'll need to know the right one to access and have the authority to connect to it.

How to do it...

You need to specify the following parameters to connect to PostgreSQL:

- Host or host address
- Port
- Database name
- User
- Password (or other means of authentication, if any)

To connect, there must be a PostgreSQL server running on host, listening to port number port. On that server, a database named dbname and a user named user must also exist. The host must explicitly allow connections from your client (this is explained in the next recipe), and you must also pass authentication using the method the server specifies; for example, specifying a password won't work if the server has requested a different form of authentication.
Almost all PostgreSQL interfaces use the libpq interface library. When using libpq, most of the connection parameter handling is identical, so we can discuss that just once.

If you don't specify the preceding parameters, PostgreSQL looks for values set through environment variables, which are as follows:

- PGHOST or PGHOSTADDR
- PGPORT (set this to 5432 if it is not set already)
- PGDATABASE
- PGUSER
- PGPASSWORD (this is definitely not recommended)

If you specify the first four parameters somehow but not the password, then PostgreSQL looks for a password file, discussed in the Avoiding hardcoding your password recipe.

Some PostgreSQL interfaces use the client-server protocol directly, so the way defaults are handled may differ. The information we need to supply won't vary significantly, so check the exact syntax for that interface.

Starting from PostgreSQL 9.2, connection details can also be specified using a URI format, as follows:

```
psql postgresql://myuser:mypasswd@myhost:5432/mydb
```

This specifies that we will connect to PostgreSQL using the myhost host, 5432 port, mydb database name, myuser user, and mypasswd password.

**How it works...**

PostgreSQL is a client-server database. The system it runs on is known as the host. We can access the PostgreSQL server remotely through the network. However, we must specify host, which is a hostname, or hostaddr, which is an IP address. We can specify a host as localhost if we wish to make a TCP/IP connection to the same system. It is often better to use a Unix socket connection, which is attempted if the host begins with a slash (/) and the name is presumed to be a directory name (default is /tmp).

On any system, there can be more than one database server. Each database server listens to exactly one “well-known” network port, which cannot be shared between servers on the same system. The default port number for PostgreSQL is 5432, which has been registered with IANA and is uniquely assigned to PostgreSQL (you can see it used in the /etc/services file on most *nix servers). The port number can be used to uniquely identify a specific database server if many exist. IANA is the acronym for Internet Assigned Numbers Authority (http://www.iana.org), the organization that coordinates the allocation of available numbers for various Internet protocols.
A database server is also sometimes known as a "database cluster", because the PostgreSQL server allows you to define one or more databases on each server. Each connection request must identify exactly one database identified by its `dbname`. When you connect, you will be able to see only the database objects created within that database.

A database user is used to identify the connection. By default, there is no limit on the number of connections for a particular user; in a later recipe, we will cover how to restrict that. In the more recent versions of PostgreSQL, users are referred to as login roles, though many clues remind us of the earlier nomenclature, and that still makes sense in many ways. A login role is a role that has been assigned the `CONNECT` privilege.

Each connection will typically be authenticated in some way. This is defined at server level: client authentication will not be optional at connection time, if the administrator has configured the server to require it.

Once you've connected, each connection can have one active transaction at a time and one fully active statement at any time.

The server will have a defined limit on the number of connections it can serve, so a connection request can be refused if the server is oversubscribed.

**There's more...**

If you are already connected to a database server with `psql` and you want to confirm that you've connected to the right place and in the right way, you can execute some, or all, of the following commands. Here is the command that shows the current database:

```sql
SELECT current_database();
```

The following command shows the current user ID:

```sql
SELECT current_user;
```

The next command shows the IP address and port of the current connection, unless you are using Unix sockets, in which case both values are `NULL`:

```sql
SELECT inet_server_addr(), inet_server_port();
```

A user's password is not accessible using general SQL for obvious reasons.

You may also need the following:

```sql
SELECT version();
```
From PostgreSQL version 9.1 onwards, you can also use psql's new meta-command, \conninfo. It displays most of the preceding information in a single line:

```
postgres=# \conninfo
You are connected to database "postgres" as user "postgres" via socket in "/var/run/postgresql" at port "5432".
```

See also

There are many other snippets of information required to understand connections. Some of them are mentioned in this chapter, and others are discussed in Chapter 6, Security. For further details, refer to the PostgreSQL server documentation.

Enabling access for network/remote users

PostgreSQL comes in a variety of distributions. In many of these, you will notice that remote access is initially disabled as a security measure.

How to do it...

By default, PostgreSQL gives access to clients who connect using Unix sockets, provided the database user is the same as the system's username. Here, we'll show you how to enable other connections.

In this recipe, we mention configuration files, which can be located as shown in the Finding the current configuration settings recipe in Chapter 3, Configuration.

The steps are as follows:

1. Add or edit this line in your postgresql.conf file:
   ```
   listen_addresses = '*'
   ```

2. Add the following line as the first line of pg_hba.conf to allow access to all databases for all users with an encrypted password:
   ```
   # TYPE   DATABASE   USER        CIDR-ADDRESS   METHOD
   Host      all        all          0.0.0.0/0       md5
   ```

3. After changing listen_addresses, we restart the PostgreSQL server, as explained in the Updating the parameter file recipe from Chapter 3, Configuration.
This recipe assumes that `postgresql.conf` does not include any other configuration file, which is the case in a default installation. If changing `listen_addresses` in `postgresql.conf` does not seem to work, perhaps that setting is overridden by another configuration file. Check out the `Updating the parameter file` recipe in Chapter 3, Configuration, for more details.

**How it works...**

The `listen_addresses` parameter specifies which IP addresses to listen to. This allows you to flexibly enable and disable listening on interfaces of multiple network cards (NICs) or virtual networks on the same system. In most cases, we want to accept connections on all NICs, so we use `*`, meaning "all IP addresses."

The `pg_hba.conf` file contains a set of host-based authentication rules. Each rule is considered in a sequence until one rule fires or the attempt is specifically rejected with a `reject` method.

The preceding rule means that a remote connection that specifies any user or database on any IP address will be asked to authenticate using an MD5-encrypted password. Precisely, the following:

- **Type**: For this, `host` means a remote connection.
- **Database**: For this, `all` means "for all databases". Other names match exactly, except when prefixed with a plus (`+`) symbol, in which case we mean a group role rather than a single user. You can also specify a comma-separated list of users, or use the `@` symbol to include a file with a list of users. You can even specify `sameuser`, so that the rule matches when you specify the same name for the user and database.
- **User**: For this, `all` means "for all users". Other names match exactly, except when prefixed with a plus (`+`) symbol, in which case we mean a group role rather than a single user. You can also specify a comma-separated list of users or use the `@` symbol to include a file with a list of users.
- **CIDR-ADDRESS**: This consists of two parts, IP address and subnet mask. The subnet mask is specified as the number of leading bits of the IP address that make up the mask. Thus, `/0` means 0 bits of the IP address, so that all IP addresses will be matched. For example, `192.168.0.0/24` would mean match the first 24 bits, so any IP address of the form `192.168.0.x` would match. You can also use `samenet` or `samehost`. 
Method: For this, `md5` means that PostgreSQL will ask the client to provide a password encrypted with MD5. Another common setting is `trust`, which effectively means no authentication. Other authentication methods include GSSAPI, SSPI, LDAP, RADIUS, and PAM. PostgreSQL connections can also be made using SSL, in which case client SSL certificates provide authentication. See the *Using SSL certificates to authenticate the client* recipe in Chapter 6, Security, for more details about this.

Don’t use the `password` setting, as this sends the password in plain text. This is not a real security issue if your connection is encrypted with SSL, and there are normally no downsides with MD5 anyway, and you have extra security for non-SSL connections.

There’s more...

In earlier versions of PostgreSQL, accessing through the network was enabled by adding the `-i` command-line switch when you started the server. This is still a valid option, but now it means the following:

```
listen_addresses = '*'
```

So, if you’re reading some notes about how to set things up and this is mentioned, then be warned that those notes are probably long out of date. They are not necessarily wrong, but it’s worth looking further to see whether anything else has changed.

See also

Look at installer- and/or operating-system-specific documentation to find the standard location of the files.

Using graphical administration tools

Graphical administration tools are often requested by system administrators.

PostgreSQL has a range of tool options. The two most popular options are as follows:

- `pgAdmin3`
- `phpPgAdmin`

We’re going to describe `pgAdmin3` in more detail here because it is installed by default with the PostgreSQL Windows installer. That most likely makes it the most popular interface, even though many people choose to use server software running on Linux or variants.
How to do it...

pgAdmin3 is a client application that sends and receives SQL to PostgreSQL, displaying the results for you to browse. One pgAdmin client can access many PostgreSQL servers, and a PostgreSQL server can be accessed by many pgAdmin clients.

pgAdmin3 is usually named just pgAdmin. The "3" at the end has a long history, but isn't that important. It is not the release level; the release level at the time of writing this book is 1.20.

When you start pgAdmin, you will be prompted to register a new server, as shown in the following screenshot:
**First Steps**

Note the five basic connection parameters encircled in the preceding screenshot, as well as other information.

The port number prompted is 2345, but this is deliberately not the default PostgreSQL port of 5432. Presumably, this is done to force you to think about the setting that should be used.

You should uncheck the **Store password** box.

If you have many database servers, you can group them together. Personally, I would avoid giving each server a color, as green, yellow, and red are usually taken to mean status, which can easily be misinterpreted. Just give each server a sensible name.

You will then get access to the main browser screen, with the object tree view on the left and properties on the right, as shown in the following screenshot:

![PostgreSQL Object Tree View](image.png)

pgAdmin easily displays much of the data that is available from PostgreSQL. The information is context sensitive, allowing you to navigate and see everything quickly and easily. The information is not dynamically updated; this will occur only when you click to refresh, so keep F5 in mind when using the application.

You’ll also notice that pgAdmin provides a **Tip of the Day**, though I would turn those off. Keep the **Guru Hints** option on. Luckily, there are no chirpy paperclips offering suggestions.

pgAdmin also provides an **Object Report** generator and a **Grant Wizard**. These are useful for DBAs for review and immediate maintenance.
The pgAdmin Query tool allows you to have multiple active sessions. The Query tool has a good-looking Visual Explain feature, which displays the best execution plan found for a given query, as well as Graphical Query Builder, as shown in the following screenshot:

How it works...

pgAdmin provides a wide range of features, many of which are provided by other tools as well. This gives us the opportunity to choose which of those tools we want. For many reasons, it is best to use the right tool for the right job, and that is always a matter of expertise, experience, and personal taste.

pgAdmin submits SQL to the PostgreSQL server, and displays the results quickly and easily. As a browser, it is fantastic. For performing small DBA tasks, it is ideal. As you might've guessed from these comments, I don't recommend pgAdmin for every task.
Scripting is an important technique for DBAs. You keep a copy of the task executed, and you can edit and resubmit if problems occur. It's also easy to put all the tasks in a script into a single transaction, which isn't possible using the current GUI tools. pgAdmin provides pgScript, which only works with pgAdmin, so it is more difficult to port. For scripting, I strongly recommend the psql utility, which has many additional features that you'll increasingly appreciate over time.

Although I use psql as a scripting tool, I also find it convenient as a query tool. Some people may find this strange, and assume it is a choice for experts only. Two great features of psql are the online help for SQL and the tab completion feature, which allows you to build up SQL quickly without having to remember the syntax. See the Using the psql query and scripting tool recipe for more information.

pgAdmin also provides pgAgent, a task scheduler. Again, more portable schedulers are available, and you may wish to use those instead. Schedulers aren't covered in this book.

A quick warning—when you create an object in pgAdmin, the object will be created with a mixed case name if you use capitals anywhere in the object name. If I ask for a table named MyTable, then the only way to access that table is by referring to it in double quotes—"MyTable". See the Handling objects with quoted names recipe in Chapter 5, Tables and Data.

There's more...

phpPgAdmin is available at http://phppgadmin.sourceforge.net/. There is an online demonstration of the software, so you can try it out yourself and see whether it does the job you want done. The following screenshot shows phpPgAdmin 4 displayed on the Windows Internet Explorer browser. Version 5.1 works with PostgreSQL 9:
One of the big contrasts with pgAdmin is that phpPgAdmin is browser-based, so it may be easier to provide secure access to administrators this way.

phpPgAdmin provides the familiar left-hand-side tree view of the database, and also provides a simple SQL query tool. These are the basics that you should be looking for. Many additional features in pgAdmin3 aren’t available, but if you follow my advice you will be doing much of your work using scripts, so this may not be a problem.

For more details on the meaning of the output of the Auto Explain option, refer to the Finding out what makes SQL slow recipe from Chapter 10, Performance and Concurrency.

See also

You may also be interested in commercial tools of various kinds for PostgreSQL. A full listing is given in the PostgreSQL software catalogue at http://www.postgresql.org/download/products/1.

Using the psql query and scripting tool

psql is the query tool supplied as a part of the core distribution of PostgreSQL, so it is available and works similarly in all environments. This makes it an ideal choice for developing portable applications and techniques.

psql provides features for use as both an interactive query tool and a scripting tool.

Getting ready

From here on, we will assume that psql is a sufficient command to allow you access to the PostgreSQL server. This assumes that all of your connection parameters are defaults, which may not be true.

Written in full, the connection parameters would be as follows:

```
psql -h hostname -p 5432 -d dbname -U username
```

The default value for the port (-p) is 5432. By default, dbname and username are both identical to the operating system’s username. The default hostname on Windows is localhost, while on Unix, we use the default directory for Unix socket connections. The location of such directories varies across distributions and is set at compile time. However, note that you don’t actually need to know its value because on local connections, both the server and the client are normally compiled together, so they use the same default.
How to do it...

The command that executes a single SQL command and prints the output is the easiest, as shown here:

```
$ psql -c "SELECT current_time"
  timetz
-----------------
  18:48:32.484+01
(1 row)
```

The `-c` command is noninteractive. If we want to execute multiple commands, we can write those commands in a text file and then execute them using the `-f` option. This command loads a very small and simple set of examples:

```
$ psql -f examples.sql
```

It produces the following output when successful:

```
SET
SET
SET
SET
SET
SET
SET
DROP SCHEMA
CREATE SCHEMA
SET
SET
SET
CREATE TABLE
CREATE TABLE
COPY 5
COPY 3
```

The `examples.sql` script is very similar to a dump file produced by PostgreSQL backup tools, so this type of file and the output it produces are very common. When a command is executed successfully, PostgreSQL outputs a "command tag" equal to the name of that command; this is how the preceding output was produced.
psql can also be used in interactive mode, which is the default, so it requires no option:

```
$ psql
postgres=#
```

The first interactive command you'll need is the following:

```
postgres=# help
```

You can then type in SQL or other commands. The last interactive command you'll need is this:

```
postgres=# \quit
```

Unfortunately, you cannot type `quit` on its own, nor can you type `\exit` or other options. Sorry, just `\quit` or `\q` for short!

**How it works...**

In psql, you can enter the following two types of commands:

- `psql` meta-commands
- SQL

A meta-command is a command for the psql client, whereas SQL is sent to the database server. An example of a meta-command is `\q`, which tells the client to disconnect. All lines that begin with `\` (backslash) as the first nonblank character are presumed to be meta-commands of some kind.

If it isn't a meta-command, then it's SQL. We keep reading SQL until we find a semicolon, so we can spread SQL across many lines and format it any way we find convenient.

The `help` command is the only exception. We provide this for people who are completely lost, which is a good thought; so let's start from there ourselves.

There are two types of `help` commands, which are as follows:

- `\?`: This provides help on psql meta-commands
- `\h`: This provides help on specific SQL commands

Consider the following snippet as an example:

```
postgres=# \h DELETE
Command:     DELETE
Description: delete rows of a table
Syntax:
DELET{E}E FROM { ONLY } table [ [ AS ] alias ]
```
I find this a great way to discover or remember options and syntax. You'll also like the ability to scroll back through the previous command history.

You'll get a lot of benefit from tab completion, which will fill in the next part of the syntax, just by pressing the Tab key. This also works for object names, so you can type in just the first few letters and then press Tab; all the options will be displayed. Thus, you can type in just enough letters to make the object name unique and then hit Tab to get the rest of the name.

One-line comments are used with two dashes, as follows:

```sql
-- This is a single-line comment
```

Multiline comments here are similar to those in C and Java:

```sql
/*
 * Multi-line comment
 */
```

You'll probably agree that psql looks a little daunting at first, with strange backslash commands. I do hope you'll take a few moments to understand the interface and keep digging for more information. psql is one of the most surprising parts of PostgreSQL, and it is incredibly useful for database administration tasks when used alongside other tools.

**There's more...**

psql works across releases, though in older versions you may see a message like the following, if you do so:

```
$ psql
psql (9.6.0) Type "help" for help.
```

psql on Windows can be a little problematic, but things are constantly improving. I recommend using a terminal emulator to connect to your server and accessing psql from there.
See also

Check out some other useful features of psql, which are as follows:

- Information functions
- Output formatting
- Execution timing using the \timing command
- Input/output and editing commands
- Automatic startup files, such as .psqlrc
- Substitutable parameters (variables)
- Access to the OS command line

Changing your password securely

If you are using password authentication, then you may wish to change your password from time to time.

How to do it...

The most basic method is to use psql. The \password command will prompt you once for a new password and again to confirm. Connect to psql and type the following:

\password

Enter a new password. This causes psql to send a SQL statement to the PostgreSQL server, which contains an already encrypted password string. An example of the SQL statement sent, is as follows:

ALTER USER postgres PASSWORD 'md53175bce1d3201d16594cebf9d7eb3f9d';

Whatever you do, don't use postgres as your password. This will make you vulnerable to idle hackers, so make it a little more difficult than that, please!

Make sure you don't forget your password either. It may prove difficult to maintain your database if you can't get access to it later.

How it works...

As changing the password is just a SQL statement, any interface can do this. Other tools also allow this, such as the following:

- pgAdmin3
- phpPgAdmin
If you don’t use one of the main routes to change the password, you can still do this yourself using SQL from any interface. Note that you need to encrypt your password because if you do submit a password in plain text, like the following, then it will be shipped to the server in plain text:

```
ALTER USER myuser PASSWORD 'secret'
```

Luckily the password in this case will still be stored in an encrypted form. It will also be recorded in plain text in psql’s history file, as well as in any server and application logs, depending on the actual log-level settings.

PostgreSQL doesn’t enforce a password change cycle, so you may wish to use more advanced authentication mechanisms such as GSSAPI, SSPI, LDAP, RADIUS, and so on.

## Avoiding hardcoding your password

We all agree that hardcoding your password is a bad idea. This recipe shows you how to keep your password in a secure password file.

### Getting ready

Not all database users need passwords; some databases use other means of authentication. Don’t do this step unless you know you will be using password authentication and you know your password.

First, remove the hardcoded password from where you had set it previously. Completely remove the `password = xxxx` text from the connection string in a program. Otherwise, when you test the password file, the hardcoded setting will override the details you are just about to place in the file. Keeping the password hardcoded and in the password file is not any better. Using `PGPASSWORD` is not recommended either, so remove that also.

If you think someone may have seen the password, then change your password before placing it in the secure password file.

### How to do it...

A password file contains the usual five fields that we require when connecting, as shown here:

```
host:port:dbname:user:password
```

Change this to the following:

```
myhost:5432:postgres:sriggs:moresecure
```
The password file is located using an environment variable named `PGPASSFILE`. If `PGPASSFILE` is not set, then a default filename and location must be searched for, as follows:

- On *nix systems, look for `~/.pgpass`.
- On Windows systems, look for `%APPDATA%\postgresql\pgpass.conf`, where `%APPDATA%` is the application data subdirectory in the path (for me, that would be `C:\`).

Don't forget to set the file permissions on the file, so that security is maintained. File permissions are not enforced on Windows, though the default location is secure. On *nix systems, you must issue the following:

```bash
chmod 0600 ~/.pgpass
```

If you forget to do this, the PostgreSQL client will ignore the `.pgpass` file. While `psql` will issue a clear warning, many other clients will just fail silently, so don't forget!

### How it works...

Many people name the password file `.pgpass`, whether or not they are on Windows, so don't get confused if they do this.

The password file can contain multiple lines. Each line is matched against the requested `host:port:dbname:user` combination until we find a line that matches. Then, we use that password.

Each item can be a literal value or `*`, a wildcard that matches anything. There is no support for partial matching. With appropriate permissions, a user can potentially connect to any database. Using the wildcard in the `dbname` and `port` fields makes sense, but it is less useful in other fields. Here are a few examples:

- `myhost:5432:*:sriggs:moresecurepw`
- `myhost:5432:perf:hannu:okpw`
- `myhost:*:perf:gianni:sicurissimo`

### There's more...

This looks like a good improvement if you have a small number of database servers. If you have many different database servers, you may want to think about using a connection service file instead (see the next recipe), or perhaps even storing details on an LDAP server.
Using a connection service file

As the number of connection options grows, you may want to consider using a connection service file.

The connection service file allows you to give a single name to a set of connection parameters. This can be accessed centrally to avoid the need for individual users to know the host and port of the database, and it is more resistant to future change.

You can set up a system-wide file as well as individual per-user files. The default file paths for these files are `/etc/pg_service.conf` and `~/.pg_service.conf` respectively.

A system-wide connection file controls service names for all users from a single place, while a per-user file applies only to that particular user. Keep in mind that the per-user file overrides the system-wide file: if a service is defined in both files, then the definition in the per-user file will prevail.

How to do it...

First, create a file named `pg_service.conf` with the following content:

```plaintext
[dbservice1]
host=postgres1
port=5432
dbname=postgres
```

You can then copy it either to `/etc/pg_service.conf` or another agreed central location. You can then set the `PGSYSCONFDIR` environment variable to that directory location.

Alternatively, you can copy it to `~/.pg_service.conf`. If you want to use a different name, set `PGSERVICEFILE`. Either way, you can then specify a connection string like the following:

```
service=dbservice1 user=sriggs
```

The service can also be set using an environment variable named `PGSERVICE`.

How it works...

This feature applies to libpq connections only, so it does not apply to JDBC.

The connection service file can also be used to specify the user, though that would mean that the username would be shared.
The `pg_service.conf` and `.pgpass` files can work together, or you can use just one of the two as you choose. Note that the `pg_service.conf` file is shared, so it is not a suitable place for passwords. The per-user connection service file is not shared, but in any cases, it seems best to keep things separate and confine passwords to `.pgpass`.

## Troubleshooting a failed connection

This recipe is all about what you should do when things go wrong.

Bear in mind that 90 percent of the problems are just misunderstandings, and you’ll quickly be on track again.

### How to do it...

Here we’ve made a checklist to be followed in case a connection attempt fails:

- Check whether the database name and username are accurate. You may be requesting a service on one system while the database you require is on another system. Recheck your credentials. Especially ensure that you haven’t mixed things up, so that you are using the database name as the username or vice versa. If you receive "too many connections," then you may need to disconnect another session before you can connect, or wait for the administrator to re-enable the connections.

- Check for explicit rejections. If you receive the `pg_hba.conf` rejects connection for `host` error message, it means your connection attempt has been explicitly rejected by the database administrator for that server. You will not be able to connect from the current client system using those credentials. There is little point attempting to contact the administrator, as you are violating an explicit security policy in what you are attempting to do.

- Check for implicit rejections. If the error message you receive is `no pg_hba.conf entry for`, it means there is no explicit rule that matches your credentials. This is likely an oversight on the part of the administrator and is common in very complex networks. Contact the administrator and request a ruling on whether your connection should be allowed (hopefully) or explicitly rejected in the future.

- Check whether the connection works with `psql`. If you’re trying to connect to PostgreSQL from anything other than the `psql` command-line utility, switch to that now. If you can make `psql` connect successfully but cannot make your main connection work correctly, then the problem may be in the local interface you are using.
PostgreSQL 9.3 and later versions ship the `pg_isready` utility, which checks the status of a database server, either local or remote, by establishing a minimal connection. Only the hostname and port are mandatory, which is great if you don't know the database name, username, or password. The following outcomes are possible:

- The server is running and accepting connections
- The server is running but not accepting connections (because it is starting up, shutting down, or in recovery)
- A connection attempt was made, but it failed
- No connection attempt was made because of a client problem (invalid parameters, out of memory, and so on)

Check whether the server is up. If a server is shut down, then you cannot connect. The typical problem here is simply mixing up the server to which you are connecting. You need to specify the hostname and port, so it's possible that you are mixing up those details.

Check whether the server is up and accepting new connections. A server that is shutting down will not accept new connections, apart from superusers. Also, a standby server may not have the `hot_standby` parameter enabled, preventing you from connecting.

Check whether the server is listening correctly, and check the port to which the server is actually listening. Confirm that the incoming request is arriving on the interface listed in the `listen_addresses` parameter. Check whether it is set to `*` for remote connections and `localhost` for local connections.

Check whether the database name and username exist. It's possible the database or user no longer exists.

Check the connection request, that is, check whether the connection request was successful and was somehow dropped after connection. You can confirm this by looking at the server log when the following parameters are enabled:

```sql
log_connections = on
log_disconnections = on
```

Check for other reasons for disconnection. If you are connecting to a standby server, it is possible that you have been disconnected because of Hot Standby conflicts. See Chapter 12, Replication and Upgrades for more information.
There's more...

Client authentication and security are the rapidly changing areas over subsequent major PostgreSQL releases. You will also find differences between maintenance release levels. The PostgreSQL documents on this topic can be viewed at http://www.postgresql.org/docs/current/interactive/client-authentication.html.

Always check which release levels you are using before consulting the manual or asking for support. Many problems are caused simply by confusing the capabilities between release levels.
Where to buy this book

You can buy PostgreSQL 9 Administration Cookbook Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.