PostgreSQL Cookbook

PostgreSQL is an open source database management system. It is used for a wide variety of development practices such as software and web design, as well as for handling large datasets (big data).

With the goal of teaching you the skills to master PostgreSQL, the book begins by giving you a glimpse of the unique features of PostgreSQL and how to utilize them to solve real-world problems. With the aid of practical examples, the book will then show you how to create and manage databases. You will learn how to secure PostgreSQL, perform administration and maintenance tasks, implement high availability features, and provide replication. The book will conclude by teaching you how to migrate information from other databases to PostgreSQL.

What this book will do for you...

- Perform regular maintenance tasks to keep your database steady and achieve optimal performance
- Design and implement various high availability and replication features to provide redundancy, fault tolerance, and failover
- Diagnose and troubleshoot CPU, memory, and I/O related database performance issues
- Perform database operations using languages such as Perl and Python
- Discover the different backup and recovery strategies that can be implemented in PostgreSQL
- Implement connection pooling methods to achieve load balancing
- Discover the different backup and recovery strategies that can be implemented in PostgreSQL
- Perform database operations using languages such as Perl and Python
- Implement connection pooling methods to achieve load balancing

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 Cookbook

Over 90 hands-on recipes to effectively manage, administer, and design solutions using PostgreSQL

Chitij Chauhan


$ 44.99 US
£ 27.99 UK

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Managing Databases and the PostgreSQL Server"
- A synopsis of the book’s content
- More information on PostgreSQL Cookbook

About the Author

Chitij Chauhan currently works as a senior database administrator at an IT-based MNC in Chandigarh. He has over 10 years of work experience in the field of database and system administration, with specialization in MySQL clustering, PostgreSQL, Greenplum, Informix DB2, SQL Server 2008, Sybase, and Oracle. He is a leading expert in the area of database security, with expertise in database security products such as IBM InfoSphere Guardium, Oracle Database Vault, and Imperva.
PostgreSQL Cookbook

PostgreSQL is a database server that is available on a wide range of platforms and is one of the most popular open source databases deployed in production environments worldwide.

It is also one of the most advanced databases, with a wide range of features that challenge even many proprietary databases. This book offers you an insight into the various features and implementations of these features in PostgreSQL. It is intended to be a practical guide for database administrators and developers alike, with solutions related to data migration, table partitioning, high availability and replication, database performance, and using Perl and Python languages for integration with PostgreSQL.

What This Book Covers

Chapter 1, Managing Databases and the PostgreSQL Server, helps you to create databases and understand the concept of schemas, roles, users, groups, and tablespaces in the PostgreSQL server.

Chapter 2, Controlling Security, lets you see and understand the security controls and levels of security that are present in PostgreSQL. After this chapter, you should be able to understand and configure the security controls that exist in the PostgreSQL server. You should also be able to use SSL connections in PostgreSQL in order to encrypt data.

Chapter 3, Backup and Recovery, shows the different backup and recovery scenarios that can be implemented in PostgreSQL. After this chapter, you should be familiar with logical and physical backup methods and restoring databases or database objects in a recovery-based scenario.

Chapter 4, Routine Maintenance Tasks, gives information about the regular maintenance tasks that are carried out to achieve optimal performance.

Chapter 5, Monitoring the System Using Unix Utilities, covers different Unix/Linux commands useful to troubleshoot CPU, memory, and I/O-related issues. After reading this chapter, you should be able to successfully troubleshoot CPU, memory, and disk contention issues using various Unix commands.

Chapter 6, Monitoring Database Activity and Investigating Performance Issues, teaches you different aspects related to improving PostgreSQL performance. After reading this chapter, you should be able to resolve lock conflicts, find slow-running SQL statements, collect statistics, examine index usage, and investigate and troubleshoot various PostgreSQL database issues in a real-time environment.
Chapter 7, *High Availability and Replication*, demonstrates the high availability and replication concepts in PostgreSQL. After reading this chapter, you will be able to implement high availability and replication options using different techniques including streaming replication, Slony replication, replication using Bucardo, and replication using Longdiste. Eventually, you will be able to implement a full-fl edged, active/passive, highly available PostgreSQL cluster using open source tools such as DRBD, Pacemaker, and Corosync.

Chapter 8, *Connection Pooling*, covers connection pooling methods such as pgpool and pgbouncer. They help reduce database overhead when there are a large number of concurrent connections. After reading this chapter, you should be able to configure the pgpool and pgbouncer methods.

Chapter 9, *Table Partitioning*, explains the different partitioning methods and implementing logical segregation of table data into partitions. You will also get familiar with horizontal partitioning implementation using PL/Proxy.

Chapter 10, *Accessing PostgreSQL from Perl*, makes you familiar with creating database connections, accessing data, and performing DML operations on the PostgreSQL database using Perl programming.

Chapter 11, *Accessing PostgreSQL from Python*, shows you how to create database connections, access data, and carry out DML operations on the PostgreSQL database using Python programming.

Chapter 12, *Data Migration from Other Databases and Upgrading the PostgreSQL Cluster*, covers the different mechanisms available to initiate minor and major version upgrades of PostgreSQL. You will also become familiar with the Oracle GoldenGate tool used to replicate data from other databases to PostgreSQL.
Managing Databases and the PostgreSQL Server

In this chapter, we will cover the following recipes:

- Creating databases
- Creating schemas
- Creating users
- Creating groups
- Destroying databases
- Creating and dropping tablespaces
- Moving objects between tablespaces
- Initializing a database cluster
- Starting the server
- Stopping the server
- Displaying the server status
- Reloading the server configuration files
- Terminating connections
Managing Databases and the PostgreSQL Server

Introduction

PostgreSQL is an open source, object-oriented relational database management system that was originally developed at the Berkeley Computer Science Department of the University of California.

PostgreSQL is an advanced database server available on a wide range of platforms, ranging from Unix-based operating systems such as Oracle Solaris, IBM AIX, and HP-UX; Windows; and Mac OS X to Red Hat Linux and other Linux-based platforms.

We start with showing how to create databases in PostgreSQL. During the course of this chapter, we will cover schemas, users, groups, and tablespaces, and show how to create these entities. We will also show how to start and stop the PostgreSQL server services.

Creating databases

A database is a systematic and organized collection of data which can be easily accessed, managed, and updated. It provides an efficient way of retrieving stored information. PostgreSQL is a powerful open source database. It is portable because it written in ANSI C. As a result, it is available for different platforms and is reliable. It is also ACID (short for Atomicity, Consistency, Isolation, Durability) compliant, supports transactions, is scalable as it supports multi version concurrency control (MVCC) and table partitioning, is secure as it employs host based access control and supports SSL, and provides high availability and replication by implementing features such as streaming replication and its support for point in time recovery.

Getting ready

Before you start creating databases, you would need to install PostgreSQL on your computer. For Red Hat or CentOS Linux environments, you can download the correct rpm for the PostgreSQL 9.3 version from yum.postgresql.org.

Here is the link you can use to install PostgreSQL on CentOS:


The following are the links you can use to install PostgreSQL on an Ubuntu platform:

Alternatively, you may download the graphical PostgreSQL installer available from the EnterpriseDB website, at http://www.enterprisedb.com/products-services-training/pgdownload.

For details on how to install PostgreSQL using the graphical PostgreSQL installer from the EnterpriseDB website, you can refer to the following link for further instructions:

http://www.enterprisedb.com/docs/en/9.3/pginstguide/Table%20of%20Contents.htm

Once you have downloaded and installed PostgreSQL, you will need to define the data directory, which is the storage location for all of the data files for the database. You will then need to initialize the data directory. Initialization of the data directory is covered under the recipe titled Initializing a database cluster. After this, you are ready to create the database.

To connect to a database using the psql utility, you can use the following command:

```
psql -h localhost -d postgres -p 5432
```

Here, we are basically connecting to the postgres database, which is resident on the localhost, that is the same server on which PostgreSQL was installed, and the connection is taking place on port 5432.

In the following code, we are creating a user, hr. Basically, this user is being created because in the next section, it is being used as the owner of the hrdb database:

```
CREATE USER hr with PASSWORD 'hr';
```

More details regarding creating users will be covered in the Creating users recipe.

### How to do it...

PostgreSQL provides two methods to create a new database:

- The first method relies on using the CREATE DATABASE SQL statement:
  ```
  CREATE DATABASE hrdb WITH ENCODING='UTF8' OWNER=hr
  CONNECTION LIMIT=25;
  ```

- The second method requires using the createdb command-line executable:
  ```
  createdb -h localhost -p 5432 -U postgres testdb1
  ```

### How it works...

A database is a named collection of objects such as tables, functions, and so on. In order to create a database, the user must be either a superuser or must have the special CREATEDB privilege.
Managing Databases and the PostgreSQL Server

The `createdb` command-line executable connects to the `postgres` database when triggered, and then issues the `CREATE DATABASE` command.

You can view the list of existing databases by querying the `pg_database` catalog table, as shown in the following screenshot:

```
postgres=# SELECT datname FROM pg_database WHERE datistemplate = false;
          datname
----------
         postgres
           testdb1
           hrdb
(3 rows)
```

Alternatively, you may use the `\l` switch of `psql` to view the list of existing databases.

Creating schemas

Schemas are among the most important objects within a database. A schema is a named collection of tables. A schema may also contain views, indexes, sequences, data types, operators, and functions. Schemas help organize database objects into logical groups, which helps make these objects more manageable.

How to do it...

You can use the `CREATE SCHEMA` statement to create a new schema in PostgreSQL:

```sql
CREATE SCHEMA employee;
```

Alternatively, it is also possible to create a schema for a particular user:

```sql
CREATE SCHEMA university AUTHORIZATION bob;
```

Here, a schema called `university` is created and is owned by `bob`.

How it works...

A schema is a logical entity that helps organize objects and data in the database.

By default, if you don’t create any schemas, any new objects will be created in the public schema.

In order to create a schema, the user must either be a superuser or must have the `CREATE` privilege for the current database.

Once a schema is created, it can be used to create new objects such as tables and views within that schema.
There's more...

You may use the \dn switch of psql to list all of the schemas in a database as shown in the following screenshot:

```
postgres=# \dn
List of schemas
    Name |    Owner
----------+----------
      hr  |  postgres
     hrd  |       hr
        public |  postgres
(3 rows)
```

To identify the schema in which you are currently working, you can use the following command:

```
SELECT current_schema();
```

While searching for objects in the database, you can define the search schemas preferences for where those searches should start. You can use the `search_path` parameter for this, as follows:

```
ALTER DATABASE hrd SET search_path TO hr, hrms, public, pg_catalog;
```

Creating users

A user is a login role that is allowed to log in to the PostgreSQL server. The login roles section is where you define accounts for individual users for the PostgreSQL system. Each database user should have an individual account to log in to the PostgreSQL system. Each user has an internal system identifier in PostgreSQL, which is known as a sysid. The user's system ID is used to associate objects in a database with their owner. Users may also have global rights assigned to them when they are created. These rights determine whether a user is allowed to create or drop databases and whether the existing user is a superuser or not.

How to do it...

PostgreSQL provides two methods by which database users are created:

- The first method requires using the `CREATE USER` SQL statement to create a new user in the database. You can create a new user with the `CREATE USER` SQL statement, like this:
  ```sql
  CREATE user agovil WITH PASSWORD 'Kh@rt0um';
  ```
Here, we created the agovil user and provided a password for the user to log in with.

- The second method requires executing the createuser script from the command line.

We may also use the createdb script to create a user called nchabbra on the same host (port 5432), and the -S option specifies that the created user will not have the superuser privileges:

```
$ createuser -h localhost -p 5432 -S nchabbra
```

### How it works...

The `CREATE USER` SQL statement requires one mandatory parameter which is the name of the new user. Other parameters, which are optional, however, are passwords for the user or group, the system ID, and a set of privileges that may be explicitly allocated.

The `createuser` script can be invoked without arguments. In that case, it will prompt us to provide the username and the set of rights and will attempt to make a local connection to PostgreSQL. It can also be invoked with options and the username to be created on the command line, and you will need to give the user access to a database explicitly if he/she is not the owner of the database.

### There's more...

We can use the `\du` switch of `psql` to display the list of existing users, inclusive of roles in the PostgreSQL server, as shown in this screenshot:

```
postgres=# \du
              List of roles
Role name | Attributes     | Member of
----------|----------------|---------------------
          |                | {dba_community     |
agovil    |                | }                  
          | Cannot login  | ()                  
dba_community | Cannot login  | ()                  
hr        | Superuser     | ()                  
manager   | Superuser     | ()                  
nchabbra  |                | {dba_community     |
          |                | }                  
nchabbra  |                | ()                  
postgres  | Superuser, Create role, Create DB, Replication | ()
salesuser |                | ()                  
```
Alternatively you may obtain the list of users by querying the \texttt{pg\_user} catalog table using the SQL statement, as shown in the following screenshot:


define CREATE GROUP SQL statement.


<table>
<thead>
<tr>
<th>User name</th>
<th>User ID</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>agovil</td>
<td>16399</td>
<td></td>
</tr>
<tr>
<td>manager</td>
<td>16400</td>
<td>superuser</td>
</tr>
<tr>
<td>nchabbra</td>
<td>16403</td>
<td></td>
</tr>
<tr>
<td>nchabbra</td>
<td>16402</td>
<td></td>
</tr>
<tr>
<td>postgres</td>
<td>10</td>
<td>superuser, create database</td>
</tr>
<tr>
<td>saleuser</td>
<td>16398</td>
<td></td>
</tr>
</tbody>
</table>

(6 rows)

\section*{Creating groups}

A \textit{group} in the PostgreSQL server is similar to the groups that exist in Unix and Linux. A group in PostgreSQL serves to simplify the assignment of rights. It simply requires a name and may be created empty. Once it is created, users who are intended to share common access rights are added into the group together, and are thus associated by their membership within that group. Grants on the database objects are then given to the group instead of each individual group member.

\section*{How to do it...}

Groups in the PostgreSQL server can be created by using the \texttt{CREATE GROUP} SQL statement. The following command will create a group. However, no users are currently a part of this group:

hrdb=# CREATE GROUP dept;

In order to assign members/users to the group, we can use the \texttt{ALTER GROUP} statement as follows:

hrdb=# ALTER GROUP dept ADD USER agovil,nchabbra;

It is also possible to create a group and assign users upon its creation, as shown in the following \texttt{CREATE GROUP} statement:

hrdb=# CREATE GROUP admins WITH user agovil,nchabbra;
Managing Databases and the PostgreSQL Server

How it works...

A group is a system-wide database object that can be assigned privileges and have users added to it as members. A group is a role that cannot be used to log in to any database.

It is also possible to grant membership in a group to another group, thereby allowing the member role use of privileges assigned to the group it is a member of.

Database groups are global across a database cluster installation.

There's more...

To list all of the available groups in the PostgreSQL server instance, you need to query the pg_group catalog table, as shown in the following screenshot:

```
hrdb=# SELECT * FROM pg_group;
   grolist
------
     hr | 16394 | []
dba_community | 16404 | (16399,16403)
sam | 16418 | (16399,16417)
sales | 16419 | []
data | 16420 | []
dept | 16421 | (16399,16403)
(6 rows)
```

Destroying databases

Every major RDBMS vendor offers the ability to drop databases just as it allows you to create databases. However, one should exercise caution when dealing with situations like dropping databases. Once a database is dropped, all of the information residing in it is lost forever. It is only for a valid business purpose that we should drop databases. In normal circumstances, a database is only dropped when it gets decommissioned and is no longer required for business operations.

How to do it...

There are two methods to drop a database in the PostgreSQL server instance:

- You can use the `DROP DATABASE` statement to drop a database from PostgreSQL, as follows:
  ```
  hrdb=# DROP DATABASE hrdb;
  ```
You can use the `dropdb` command line utility, which is a wrapper around the `DROP DATABASE` command:

```
$ dropdb hrdb;
```

### How it works...

The `DROP DATABASE` statement permanently deletes catalog entries and the data directory. Only the owner of the database can issue the `DROP DATABASE` statement.

Also, it is not possible to drop a database to which you are connected. In order to delete the database, the database owner will have to make a connection to another database of which he is an owner.

### There's more...

One situation that demands attention is when a user tries to drop a database that has active connections. The user will get an error when trying to drop such a database.

In order to drop a database that has active connections to it, you will have to follow these steps:

1. Identify all of the active sessions on the database. To identify all of the active sessions on the database, you need to query the `pg_stat_activity` catalog table as follows:

   ```
   SELECT * from  pg_stat_activity where datname='testdb1';
   ```

2. Terminate all of the active connections to the database. To terminate all of the active connections, you will need to use the `pg_terminate_backend` function as follows:

   ```
   SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname = 'testdb1';
   ```

3. Once all of the connections are terminated, you may proceed with dropping the database using the `DROP DATABASE` statement.

### Creating and dropping tablespaces

PostgreSQL stores data files consisting of database objects such as tables and indices on the disk. The `tablespace` is defined as the location of these objects on the disk. A tablespace is used to map a logical name to a physical location on the disk.

### Getting ready

A tablespace is a location on the disk where PostgreSQL stores data files containing database objects, for example indexes, tables, and so on.
Before you create the tablespace, the directory location must be physically created and the directory must be empty:

```
mkdir -p /var/lib/pgsql/data/dbs
```

### How to do it...

To create a tablespace in PostgreSQL, you need to use the `CREATE TABLESPACE` statement.

The following command creates a `data_tbs` tablespace, which is owned by the `agovil` user:

```
CREATE TABLESPACE data_tbs OWNER agovil LOCATION '/var/lib/pgsql/data/dbs';
```

Similarly, a tablespace in PostgreSQL can be dropped using the `DROP TABLESPACE` statement, as follows:

```
DROP TABLESPACE data_tbs;
```

### How it works...

A tablespace allows you to control the disk layout of PostgreSQL. The owner of the tablespace, by default, would be the user who executed the `CREATE TABLESPACE` statement. This statement also gives you the option of assigning the ownership of the tablespace to a new user. This option is the part of the `OWNER` clause in the `CREATE TABLESPACE` statement.

The name of the tablespace should not begin with a `pg_` prefix because this is reserved for the system tablespaces.

Before deleting a tablespace, ensure that it is empty, which means there should be no database objects inside it. If the user tries to delete the tablespace when it is not empty, the command will fail.

There are two options that will aid in deleting the tablespace when it is not empty:

- You may drop the database
- You may alter the database to move it to a different tablespace

After any of the preceding actions have been completed, then the corresponding tablespace may be dropped.

### There's more...

By default, two tablespaces exist in PostgreSQL:

- `pg_default`: This is used to store user data
- `pg_global`: This is used to store global data
You may query the `pg_tablespace` catalog table to get the list of existing tablespaces in PostgreSQL, as shown in the following screenshot:

<table>
<thead>
<tr>
<th>spcname</th>
<th>spcowner</th>
<th>spcacl</th>
<th>spcoptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_default</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pg_global</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>demo</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Moving objects between tablespaces

A tablespace can contain both permanent and temporary objects. You will need to define and create a secondary tablespace to serve as the target destination of objects that might get moved from the primary tablespace. Moving objects between tablespaces is a mechanism of copying bulk data in which copying happens sequentially, block by block. Moving a table to another tablespace locks it for the duration of the move.

### Getting ready

Here, we will first create a new tablespace, `hrms`, using the following command:

```bash
mkdir -p /var/lib/pgsql/data/hrms
```

Then we set the default tablespace for the `testdb1` database to `hrms` using the following statement:

```sql
CREATE TABLESPACE HRMS OWNER agovil LOCATION '/var/lib/pgsql/data/hrms';
```

We will also create a table, insert some records into it, and create a corresponding index for it. This is being done because the table and its index will be used in the *How to do it...* section of this recipe:

```sql
CREATE TABLE EMPLOYEES(id integer PRIMARY KEY, name varchar(40));
INSERT INTO EMPLOYEES VALUES (1, 'Mike Johansson');
INSERT INTO EMPLOYEES VALUES(2, 'Rajat Arora');
CREATE INDEX emp_idx on employees(name);
```
Managing Databases and the PostgreSQL Server

How to do it...

Moving a complete database to a different tablespace involves three steps:

1. You will change the tablespace for the given database so that new objects for the associated database are created in the new tablespace:
   
   ALTER DATABASE testdb1 SET default_tablespace='hrms';

2. You will have to then move all of the existing tables in the corresponding database to the new tablespace:
   
   ALTER TABLE employee SET TABLESPACE hrms;

3. You will also have to move any existing indexes to the new tablespace:
   
   ALTER INDEX emp_idx SET TABLESPACE hrms;

How it works...

You will have to query the pg_tables catalog table to find out which tables from the current database need to be moved to a different tablespace.

Similarly for the indexes, you will have to query the pg_indexes catalog table to find out which indexes need to be moved to a different tablespace.

Initializing a database cluster

In terms of a filesystem, a database cluster is a collection of databases that are managed by a single server instance, and it is the framework upon which PostgreSQL databases are created.

How to do it...

The initdb command is used to initialize or create the database cluster. The -D switch of the initdb command is used to specify the filesystem location for the database cluster.

To create the database cluster, use the initdb command:

$ initdb -D /var/lib/pgsql/data

Another way of initializing the database cluster is by calling the initdb command via the pg_ctl utility:

$ pg_ctl -D /var/lib/pgsql/data initdb
A database cluster is a collection of databases that are managed by a single server instance.

When the `initdb` command is triggered, the directories in which the database data will reside are created, shared catalog tables are generated, and the `template1` and `postgres` databases are created, out of which the default database is `postgres`. The `initdb` command initializes the database cluster default locale and the character set encoding.

You can refer to [http://www.postgresql.org/docs/9.3/static/creating-cluster.html](http://www.postgresql.org/docs/9.3/static/creating-cluster.html) for more information on initializing a database cluster.

### Starting the server

Before anyone can access the database, the database server must be started. Then you will be able to start all of the instances of the `postgres` database in the cluster using the different commands with options as mentioned in this recipe.

### Getting ready

The term "server" refers to the database and the associated backend processes. The term "service" refers to the operating system wrapper through which the server gets invoked. In normal circumstances, the PostgreSQL server will usually start automatically when the system boots up. However, there will be situations where you may have to start the server manually for different reasons.

### How to do it...

There are a couple of methods through which the PostgreSQL server can be started on Unix or Linux platforms:

- The first method relies on passing the start argument to the `pg_ctl` utility to get the postmaster backend process started, which effectively means starting the PostgreSQL server.
- The next method relies on using the service commands, which, if supported by the operating system, can be used as a wrapper to the installed PostgreSQL script.
- The last method involves invoking the installed PostgreSQL script directly using its complete path.

On most Unix distributions and Red Hat-based Linux distributions, the `pg_ctl` utility can be used as follows:

```
pg_ctl -D /var/lib/pgsql/data start
```
If you are using the service command, the service can be started like this:

```
    service postgresql<version> start
```

For PostgreSQL version 9.3, the service command to start the PostgreSQL server is as follows:

```
    service postgresql-9.3 start
```

You may also start the server by manually invoking the installed PostgreSQL script using its complete path:

```
    /etc/rc.d/init.d/postgresql-9.3 start
```

On Windows-based systems, the PostgreSQL service can be started using the following command:

```
    NET START postgresql-9.3
```

**How it works...**

The start argument of the `pg_ctl` utility will first start PostgreSQL's postmaster backend process using the path of the data directory.

The database system will then start up successfully, report the last time the database system was shut down, and provide various debugging statements before returning the `postgres` user to the shell prompt.

**There's more...**

In Ubuntu and Debian Linux distributions, the `pg_ctlcluster` wrapper can be used with the `start` argument to start the postmaster server for a particular cluster. A cluster is a group of one or more PostgreSQL database servers that may coexist on a single host.

**Stopping the server**

Sometimes in emergency situations, you might have to bring down the PostgreSQL server's services. There are certain situations in which you may need to stop the database services. For instance, during an operating system migration, you might need to stop the running services, take a filesystem backup, and then proceed with OS migration.

**How to do it...**

There are a couple of ways by which the PostgreSQL server can be stopped.
On Unix distributions and Red Hat-based Linux distributions, we can use the `stop` argument of the `pg_ctl` utility to stop the postmaster:

```bash
pg_ctl -D /var/lib/pgsql/data stop -m fast
```

Using the `service` command, the PostgreSQL server can be stopped like this:

```bash
service postgresql stop
```

You may also stop the server by manually invoking the installed PostgreSQL script using its complete path:

```bash
/etc/rc.d/init.d/postgresql stop
```

On Windows-based systems, you may stop the postmaster service in this manner:

```bash
NET STOP postgresql-9.3
```

### How it works...

The `pg_ctl` utility checks for the running postmaster process, and if the `stop` argument of the `pg_ctl` utility is invoked, then the server is shut down.

By default, the PostgreSQL server will wait for clients to first cancel their connections before shutting down.

However, with the use of a fast shutdown, there is no wait time involved as all of the user transactions will be aborted and all connections will be disconnected.

### There's more...

There may be situations where one needs to stop the PostgreSQL server in an emergency situation, and for this, PostgreSQL provides the immediate shutdown mode.

In case of immediate shutdown, a process will receive a harsher signal and will not be able to respond to the server anymore.

The consequence of this type of shutdown is that PostgreSQL is not able to finish its disk I/O, and therefore has to do a crash recovery the next time it is started.

The immediate shutdown mode can be invoked like this:

```bash
pg_ctl -D /var/lib/pgsql/data stop -m immediate
```
Another way to shut down the server would be to send the signal directly using the `kill` command. The PID of the `postgres` process can be found using the `ps` command or from the `postmaster.pid` file in the `data` directory. In order to initiate a fast shutdown, you can issue the following command:

```
$ kill -INT head -1 /usr/local/pgsql/data/postmaster.pid
```

**Displaying the server status**

Many a times, there will be situations where end users complain that the database performance is sluggish and they are not able to log in to the database. In such situations, it is often helpful to take a quick glance through the status of the PostgreSQL backend `postmaster` process and confirm whether the PostgreSQL server services are up and running.

**How to do it...**

There are a couple of ways by which the status of the PostgreSQL server can be checked.

On Unix and on Red Hat-based Linux distributions, the status argument of the `pg_ctl` utility can be used to check the status of a running `postmaster` backend:

```
pg_ctl -D /var/lib/pgsql/data status
```

On Unix-based and Linux-based platforms supporting the `service` command, the status of a `postgresql` process can be checked as follows:

```
service postgresql status
```

You may also check the server status by manually invoking the installed PostgreSQL script using its complete path:

```
/etc/rc.d/init.d/postgresql status
```

**How it works...**

The status mode of the `pg_ctl` utility checks whether the postmaster process is running in the specified data directory.

If the server is running, then the process ID and the command-line options that were used to invoke it are displayed.
**Reloading the server configuration files**

Changes made to certain PostgreSQL configuration parameters come into effect when the server configuration files, such as `postgresql.conf`, are reloaded. Reloading the server configuration files becomes necessary in such cases.

**How to do it...**

Some of the configuration parameters in PostgreSQL can be changed on the fly. However, changes to other configurations can only be reflected once the server configuration files are reloaded.

On most Unix-based and Linux-based platforms, the command to reload the server configuration file is as follows:

```
pg_ctl -D /var/lib/pgsql/data reload
```

It is also possible to reload the configuration file while being connected to a PostgreSQL session. However, this can be done by the superuser only:

```
postgres=# select pg_reload_conf();
```

On Red Hat and other Linux-based systems that support the `service` command, the `postgresql` command to reload the configuration file is as follows:

```
service postgresql reload
```

**How it works...**

To ensure that changes made to the parameters in the configuration file take effect, a reload of the configuration file is needed. Reloading the configuration files requires sending the `sighup` signal to the postmaster process, which in turn will forward it to the other connected backend sessions.

There are some configuration parameters whose changed values can only be reflected by a server reload. These configuration parameters have a value known as `sighup` for the attribute context in the `pg_settings` catalog table:

```
SELECT name, setting, unit , (source = 'default') as is_default FROM
pg_settings WHERE context = 'sighup'
AND (name like '%$delay' or name like '%$timeout')
AND setting != '0';
```
Output for the preceding query is as shown below:

<table>
<thead>
<tr>
<th>name</th>
<th>setting</th>
<th>unit</th>
<th>is_default</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication_timeout</td>
<td>60</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>autovacuum_vacuum_cost_delay</td>
<td>20</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>bgwriter_delay</td>
<td>200</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>checkpoint_timeout</td>
<td>300</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>max_standby_archive_delay</td>
<td>30000</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>max_standby_streaming_delay</td>
<td>30000</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>wal_receiver_timeout</td>
<td>60000</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>wal_sender_timeout</td>
<td>60000</td>
<td>ms</td>
<td>t</td>
</tr>
<tr>
<td>walwriter_delay</td>
<td>200</td>
<td>ms</td>
<td>t</td>
</tr>
</tbody>
</table>

(9 rows)

Terminating connections

Every major RDBMS, including PostgreSQL, allows simultaneous and concurrent database connections in order for users to run transactions. Due to such concurrent processing of databases, it may be during peak transaction hours that database performance becomes slow or that there are some blocking sessions. In order to deal with such situations, we might have to terminate some specific sessions or sessions coming from a particular user so that we can get database performance back to normal.

How to do it...

PostgreSQL provides the `pg_terminate_backend` function to kill a specific session. Even though the `pg_terminate_backend` function acts on a single connection at a time, we can embed `pg_terminate_backend` by wrapping it around the `SELECT` query to kill multiple connections, based on the filter criteria specified in the `WHERE` clause.

To terminate all of the connections from a particular database, we can use the `pg_terminate_backend` function as follows:

```sql
SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE datname = 'testdb1';
```

To terminate all of the connections for a particular user, we can use `pg_terminate_backend` like this:

```sql
SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE usename = 'agovil';
```
How it works...

The `pg_terminate_backend` function requires the `pid` column or process ID as input. The value of `pid` can be obtained from the `pg_stat_activity` catalog table. Once `pid` is passed as input to the `pg_terminate_backend` function, all running queries will automatically be canceled and it will terminate a specific connection corresponding to the process ID as found in the `pg_stat_activity` table.

Terminating backends is also useful to free memory from idle `postgres` processes that was not released for whatever reason and was hogging system resources.

There's more...

If the requirement is to cancel running queries and not to terminate existing sessions, then we can use the `pg_cancel_backend` function to cancel all active queries on a connection. However, with the `pg_cancel_backend` function, we can only kill runaway queries issued in a database or by a specific user. It does not have the ability to terminate connections.

To cancel all of the running queries issued against a database, we can use the `pg_cancel_backend` function as follows:

```sql
SELECT pg_cancel_backend(pid) FROM pg_stat_activity
WHERE datname = 'testdb1';
```

To cancel all of the running queries issued by a specific user, we can use the `pg_cancel_backend` function like this:

```sql
SELECT pg_cancel_backend(pid) FROM pg_stat_activity
WHERE usename = 'agovil';
```

In versions before PostgreSQL 9.2, the `procpid` column has to be passed as input to the `pg_terminate_backend` and `pg_cancel_backend` functions to terminate running sessions and cancel queries. The `pid` column replaced the `procpid` column from PostgreSQL version 9.2 onwards.

Where to buy this book

You can buy PostgreSQL Cookbook 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.