Quick answers to common problems

PostgreSQL Cookbook

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

Chitij Chauhan

[PACKT] open source*
PUBLISHING
community experience distilled

PostgreSQL Cookbook

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

Chitij Chauhan

[PACKT] open source
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

PostgreSQL Cookbook

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2015

Production reference: 1240115

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-533-8

`www.packtpub.com`

Credits

Author
Chitij Chauhan

Reviewers
Naoya Hashimoto
Sergio Martínez-Losa Del Rincón
Danny Sauer

Commissioning Editor
Akram Hussain

Acquisition Editor
Nikhil Karkal

Content Development Editor
Sumeet Sawant

Technical Editor
Ruchi Desai

Copy Editors
Dipti Kapadia
Vikrant Phadke

Project Coordinator
Purav Motiwalla

Proofreaders
Maria Gould
Paul Hindle
Linda Morris
Stephen Silk

Indexer
Monica Ajmera Mehta

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

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.

About the Reviewers

Naoya Hashimoto has been working on system design and integration with open source software for years. Recently, his career and interest have shifted toward cloud engineering on both public and hybrid clouds, such as AWS, as well as toward orchestration tools, such as Chef or CloudFormation. He has reviewed the books *Icinga Network Monitoring* and *Building a Home Security System with BeagleBone*, both by Packt Publishing. Moreover, currently he is a technical reviewer of the book *Building Networks and Servers Using Beaglebone*, which is also by Packt Publishing.

> Thanks to the author and the project coordinator, Purav, who gave me this opportunity to review the book. I am very impressed with their work and this project because it gives us a chance to learn about the latest technology of PostgreSQL 9.x.

Sergio Martínez-Losa Del Rincón is a computer engineer who loves programming languages since the time he was in high school, where he learned about programming and computer interactions. He is always learning and discovers something new to learn everyday.

He likes all kind of programming languages, but he focuses his efforts on mobile development with native languages, such as Objective-C (iPhone), Java (Android), and Xamarin (C#). He builds Google Glass applications as well as mobile applications for iPhone and Android devices at work. He also develops games for mobile devices with cocos2d-x and cocos2d. He likes cross-platform applications as well. He has reviewed *Learning Xamarin Studio*, *Packt Publishing*.

He loves challenging problems, and he is always keen to work with new technologies. More information about his experience and details can be found at `www.linkedin.com/in/sergiomtzlosa`.

Danny Sauer has been a professional Unix geek of various stripes for roughly 20 years, most recently in the flavor of security engineer. His experience with open source databases extends through most of that time period, both as DBA and as a user. He currently lives with his wife in an old house in a small town outside of a small city, which provides plenty of opportunity to restore antique houses, cars, and computers.

Table of Contents

Preface

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-fledged, 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.

What you need for this book

You'll need the following software:

- VMware Workstation Version 7 or higher / VirtualBox
- PostgreSQL 9.3 installer

- Win32 OpenSSL v1.0.1
- pgAdmin v1.18.1
- PostgreSQL v.9.3
- Oracle Solaris Version 10
- CentOS Linux Version 6 or higher

Who this book is for

This book is for system administrators, database administrators, architects, developers, and anyone with an interest in planning, managing, and designing database solutions using PostgreSQL. This book is ideal for you if you have some prior experience with any relational database or with the SQL language.

Sections

This book contains the following sections:

Getting ready

This section tells us what to expect in the recipe, and describes how to set up any software or any preliminary settings needed for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and explanations of their meanings.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The first method relies on using the `CREATE DATABASE` SQL statement."

A block of code is set as follows:

```
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';
```

Any command-line input or output is written as follows:

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In the **New Inbound Rule Wizard** dialog box, click on the **Protocol and Ports** option, then click on the radio buttons, as shown in the following screenshot, and finally click on the **Next** button."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

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

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:

`http://www.postgresonline.com/journal/archives/329-An-almost-idiots-guide-to-install-PostgreSQL-9.3,-PostGIS-2.1-and-pgRouting-with-Yum.html`

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

- `http://technobytz.com/install-postgresql-9-3-ubuntu.html`
- `http://www.cloudservers.com/installing-and-configuring-postgresql-9-3-on-hosted-linux-cloud-vps-server/`

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:

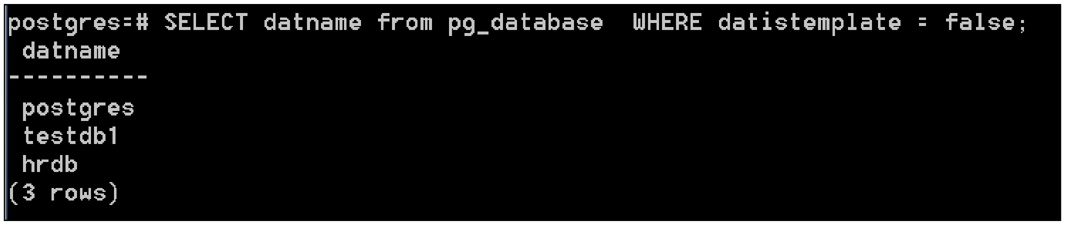
  ```
  createdb -h localhost -p 5432 -U postgres testdb1
  ```

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.

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 \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:

```
CREATE SCHEMA employee;
```

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

```
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.

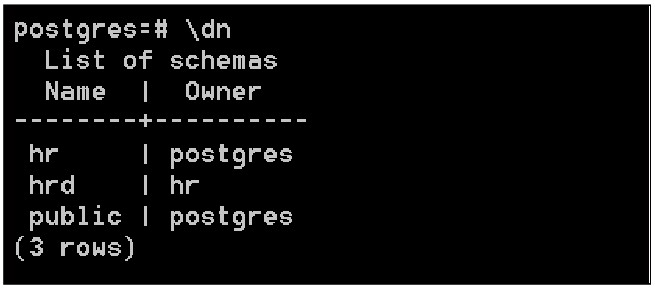
By default, if you don't create any schemas, any new objects will be created in the public schema.

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:

  ```
  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
  ```

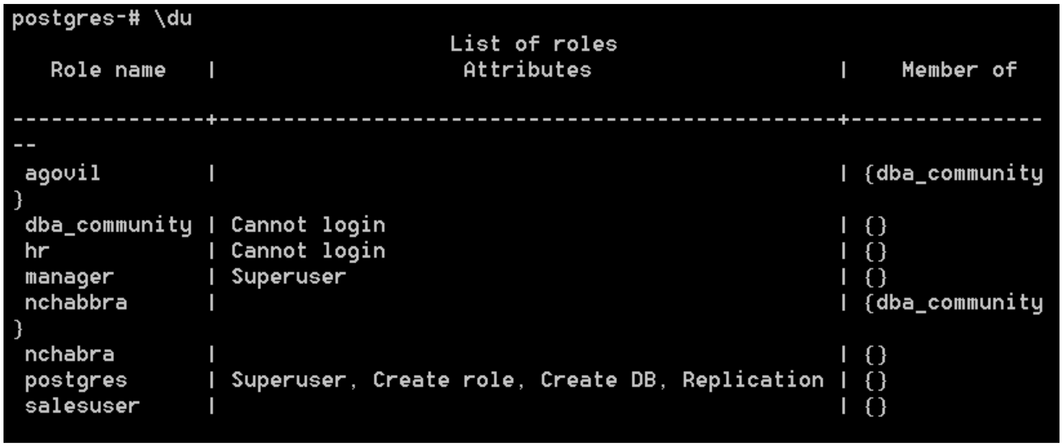
How it works...

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.

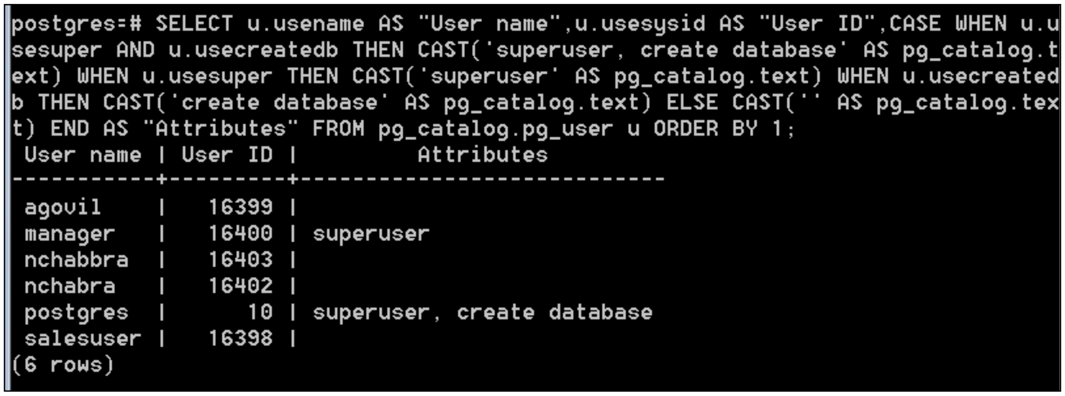
There's more...

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

---------------+------------------------------------------------+---------------
--
 agovil         |                                                | {dba_community
}
 dba_community  | Cannot login                                   | {}
 hr             | Cannot login                                   | {}
 manager        | Superuser                                      | {}
 nchabbra       |                                                | {dba_community
}
 nchabra        |                                                | {}
 postgres       | Superuser, Create role, Create DB, Replication | {}
 salesuser      |                                                | {}
```

Alternatively you may obtain the list of users by querying the `pg_user` catalog table using the SQL statement, as shown in the following screenshot:



```
postgres=# SELECT u.usename AS "User name",u.usesysid AS "User ID",CASE WHEN u.u
sesuper AND u.usecreatedb THEN CAST('superuser, create database' AS pg_catalog.t
ext) WHEN u.usesuper THEN CAST('superuser' AS pg_catalog.text) WHEN u.usecreated
b THEN CAST('create database' AS pg_catalog.text) ELSE CAST('' AS pg_catalog.tex
t) END AS "Attributes" FROM pg_catalog.pg_user u ORDER BY 1;
 User name | User ID |         Attributes
-----------+---------+---------------------------
 agovil    |   16399 |
 manager   |   16400 | superuser
 nchabbra  |   16403 |
 nchabra   |   16402 |
 postgres  |      10 | superuser, create database
 salesuser |   16398 |
(6 rows)
```

Creating groups

A **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.

How to do it...

Groups in the PostgreSQL server can be created by using the `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 `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 `CREATE GROUP` statement:

```
hrdb=# CREATE GROUP admins WITH user agovil,nchabbra;
```

How it works...

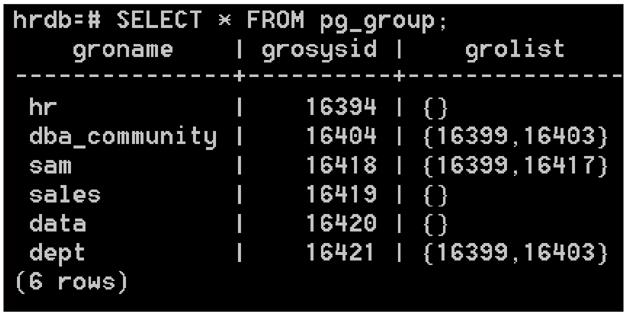
A group is a system-wide database object that can be assigned privileges and have users added to it as members. A group is a role that cannot be used to log in to any database.

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;
    groname     | grosysid |    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

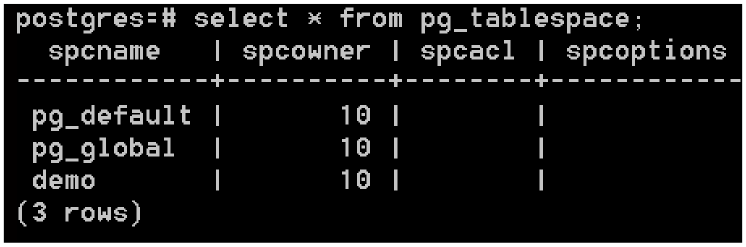
After any of the preceding actions have been completed, then the corresponding tablespace may be dropped.

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:



```
postgres=# select * from pg_tablespace;
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+-----------
 pg_default |       10 |        |
 pg_global  |       10 |        |
 demo       |       10 |        |
(3 rows)
```

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:

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

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

```
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:

```
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);
```

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
```

How it works...

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 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:

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

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

```
service postgresql stop
```

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

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

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

```
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:

```
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
```

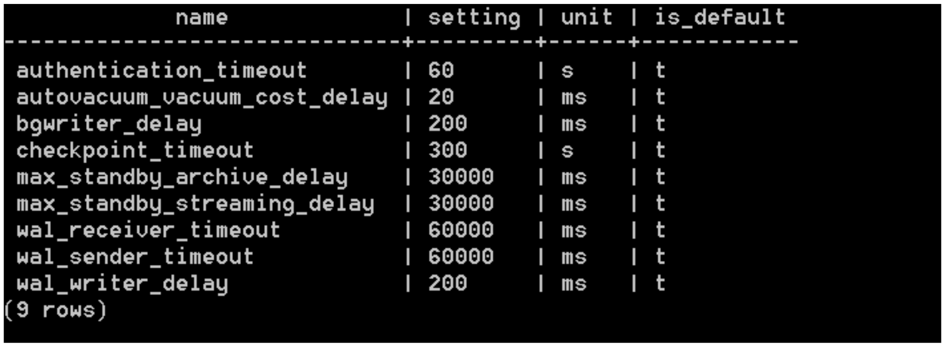
How it works...

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:



```
             name             | setting | unit | is_default
------------------------------+---------+------+------------
 authentication_timeout       | 60      | s    | t
 autovacuum_vacuum_cost_delay | 20      | ms   | t
 bgwriter_delay               | 200     | ms   | t
 checkpoint_timeout           | 300     | s    | t
 max_standby_archive_delay    | 30000   | ms   | t
 max_standby_streaming_delay  | 30000   | ms   | t
 wal_receiver_timeout         | 60000   | ms   | t
 wal_sender_timeout           | 60000   | ms   | t
 wal_writer_delay             | 200     | ms   | t
(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:

```
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:

```
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:

```
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:

```
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.

You may refer to https://blog.sleeplessbeastie.eu/2014/07/23/how-to-terminate-postgresql-sessions/ and http://www.devopsderek.com/blog/2012/11/13/list-and-disconnect-postgresql-db-sessions/ for more information regarding terminating backend connections.

2
Controlling Security

In this chapter, we will cover the following recipes:

- Securing database objects
- Controlling access via firewalls
- Controlling access via configuration files
- Testing remote connectivity
- Auditing database changes
- Enabling SSL in PostgreSQL
- Testing SSL encryption
- Encrypting confidential data
- Cracking PostgreSQL passwords

Introduction

Databases are used to store data in an organized manner. All relevant organization-related data is maintained in databases. Since all company-related information is stored in databases, it becomes imperative that controls be placed on data access and only authorized persons be allowed to access relevant data. It is in this context that database security is of utmost importance because it is important to ensure that the information stored in databases is protected against malicious attempts to view and modify data by hackers or people with malicious intent.

Database security deals with the information security measures that are undertaken to protect databases in order to ensure confidentiality, integrity, and availability of data. Databases need to be protected against various risks and threats, such as misuse by authorized database users, malicious attempts made by hackers to steal information or damage data, design flaws and software bugs in databases that lead to various security vulnerabilities that are exploited by hackers, data corruption that might be caused by wrong input and mistakes by humans, the possibility of data being sabotaged, and the administrator tendency of keeping a default schema password which might lead to unauthorized access to data by people with malicious intent.

Securing database objects

It is important to ensure that the authenticated users get access only to the data they are authorized to access. However, the pertinent question of how to keep authenticated users away from accessing unauthorized data remains. In PostgreSQL, this is implemented by maintaining a strong access control policy. The access control list governs which users are allowed to select, update, and modify objects within the database. A set of restrictions and controls are placed on every database object which determine who is allowed to access that object. Access control rights on database objects are maintained through the usage of the `GRANT` and `REVOKE` commands.

How to do it...

A database user usually has no access rights on any database objects apart from the ones that they own. As per business requirements, access to appropriate database objects is granted to other users by the owner of these objects. However, if the requirement comes to revoke a right after a user has been granted access to the object, then the `REVOKE` command can be issued.

We will discuss two cases here:

- Revoking all the permissions on a table from a specific user. Here, we show the usage of the `REVOKE` command:

  ```
  REVOKE ALL on testusers from nchhabra;
  ```

- Revoking specific permissions on a table from a user, as shown here:

  ```
  REVOKE insert,update,delete,truncate on testusers from agovil;
  ```

How it works...

Normally, all users have a set of rights, which include `SELECT`, `UPDATE`, `DELETE`, `INSERT`, `TRUNCATE`, and `TRIGGER`, on all the newly created tables through the `PUBLIC` role.

In order to ensure that a particular user is no longer able to access the table, the rights to that table must be revoked from both the `PUBLIC` role and the user. In the first step of the preceding section, we revoke all the permissions on the `testusers` table from the `nchhabra` user. In this case, the `ncchabra` user is in a way restricted from performing any operation on the `testusers` table.

In the second scenario of the preceding section, we explicitly revoke the insert, update, delete, and truncate operations on the `testusers` table from the PostgreSQL user, `agovil`, thereby permitting the user to perform read-only operations on the table via the `SELECT` table clause.

Controlling access via firewalls

The most basic way to protect network services on a server is through a **firewall**. A firewall can be in the form of both hardware and software. A firewall allows you to configure which clients are allowed to pass packets through the firewall to specific applications.

The **server firewall** is the front door of the system on the network. It can block attempts to access individual services on the server before the packets even pass to the server applications. This acts as the first line of defense in protecting PostgreSQL databases from attack and intrusion.

How to do it...

The following are a series of steps that are required to configure database port access through the firewall on Red Hat Linux and other Red Hat based distributions:

1. For the network machines to be able to access the PostgreSQL server, you must manually configure the firewall rules to allow access to the PostgreSQL server port. By default, PostgreSQL listens to the TCP port 5432. So, you need to enable port 5432 on the firewall. In Linux environments, you can enable port 5432 by modifying iptables' rules. For this, you need to open the file containing firewall rules. This file can be found at `vim/etc/sysconfig/iptables`.
2. Once you have opened the file, you need to add the following rule to enable access to port 5432:

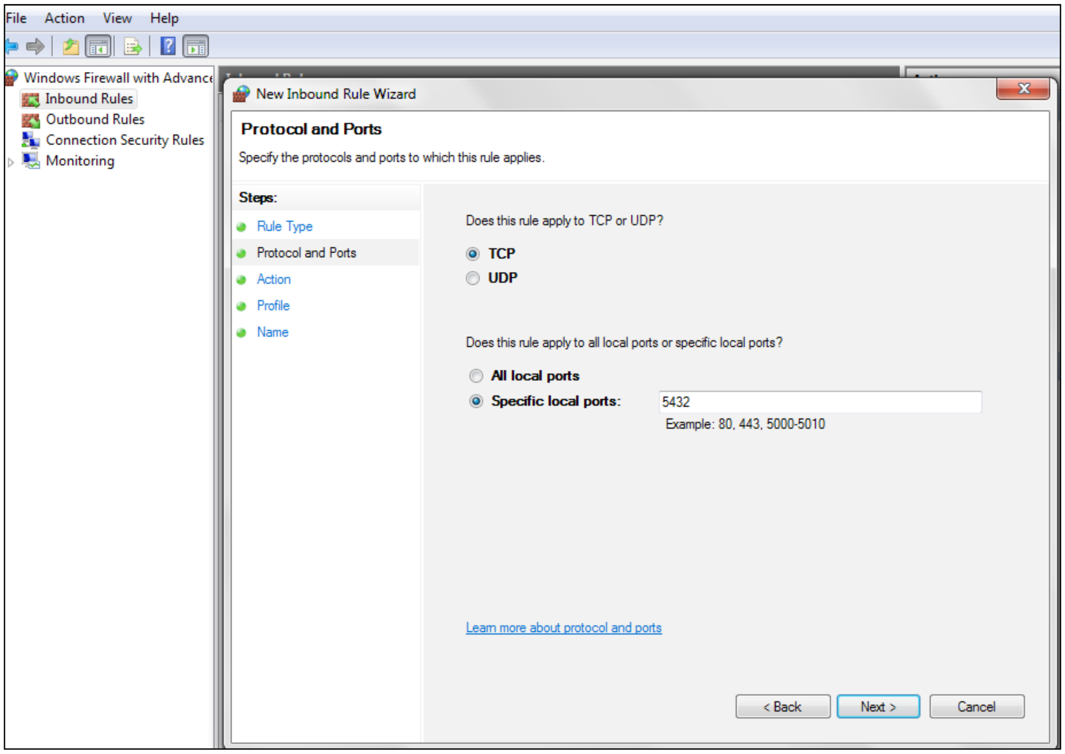
   ```
   -A INPUT -m state --state NEW -m tcp -p tcp --dport 5432
   -j ACCEPT
   ```

3. After this, you need to save the changes and reload the configuration file that contains the firewall rules in order to ensure that the new changes made come into effect, using the following command:

   ```
   service iptables restart
   ```

The next series of steps are required to configure database port access through the firewall on Windows 7. To enable port 5432 on Windows, you need to follow the sequence of steps given here:

1. Open Windows Firewall by navigating to **Start** | **Control Panel** | **Systems And Security** | **Windows Firewall**.
2. In the left-hand side pane, click on **Advanced settings**. If you're prompted for an administrator password or confirmation, type the password or provide confirmation.
3. In the **Windows Firewall with Advanced Security** dialog box, click on **Inbound Rules** in the left pane and then click on **New Rule** in the right pane.
4. In the **New Inbound Rule Wizard** dialog box, click on the **Protocol and Ports** option, then click on the radio buttons, as shown in the following screenshot, and finally click on the **Next** button.
5. In the next screen, select the default settings for all the options, enter the port number 5432 in the **Specific local ports:** text field, and then keep clicking on the **Next** button until you get to the **Finish** button. Enter a name for the rule and then click on **Finish**.



Protocol and Ports

Specify the protocols and ports to which this rule applies.

Steps:

- Rule Type
- Protocol and Ports
- Action
- Profile
- Name

Does this rule apply to TCP or UDP?

● **TCP**
○ **UDP**

Does this rule apply to all local ports or specific local ports?

○ **All local ports**
● **Specific local ports:** 5432
Example: 80, 443, 5000-5010

Learn more about protocol and ports

How it works...

Servers that host production databases have firewall policies which either allow or block a port or an IP address.

By default, firewall blocks everything. In order to enable access to applications, you need to configure rules in the firewall access control policy in order to allow the application to be accessed.

You will need to enable access on port 5432, which is the default port for PostgreSQL, in order to access the PostgreSQL server.

Controlling access via configuration files

Once the firewall is configured to allow access to the PostgreSQL server, you need to configure the PostgreSQL server to allow remote connections. This is implemented by making the necessary changes in the `postgresql.conf` and `pg_hba.conf` configuration files.

The `postgresql.conf` file contains a single entry that controls on which network interfaces PostgreSQL listens for connections.

The `pg_hba.conf` file is used to define which clients can connect to which database and using which login role.

How to do it...

1. You need to configure the `listen_addresses` configuration parameter in order to enable the remote network clients to make a connection to the PostgreSQL server:

   ```
   listen_addresses = '*'
   ```

2. Here, you use an asterisk as the value of the `listen_addresses` configuration parameter. This configuration parameter enables all network ports.
3. The next step will be to make changes in the `pg_hba.conf` configuration file. These changes define access rules in order to allow remote connections access to the PostgreSQL server.
4. Open the `pg_hba.conf` file under the data directory or under the directory defined by the `$PGDATA` environment variable and define the necessary access control rules:

   ```
   # TYPE    DATABASE    USER   CIDR-ADDRESS       METHOD    OPTION
     host    hrdb        all    192.168.12.10/32   md5
     host    all         all    192.168.54.1/32    reject
     host    all         all    192.168.1.0/24     trust
     host    hrd         all    192.168.1.10/24    crypt
   ```

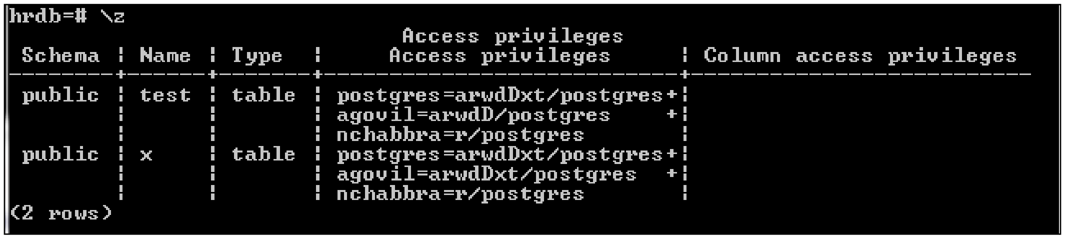
5. The first entry in the `pg_hba.conf` file signifies that any user from the host 192.168.12.10 is allowed to connect to the `hrdb` database if the user's password is supplied correctly.
6. The second entry in the `pg_hba.conf` file shows a host record that will reject all the users from the host 192.168.54.1 for any requested database.
7. The third entry in the `pg_hba.conf` file shows a host record that allows any machine on the 192.168.1.0 subnet to connect and access any database without specifying any password. Basically, with the `trust` method, we are relying on host-based authentication with the use of this method.
8. The final entry in the `pg_hba.conf` file states that any user with an IP address 192.168.1.10 and with a valid password is allowed to connect to the `hrdb` database. However, here the password will be encrypted during authentication because of the term `crypt`, which is specified as the authentication method.

How it works...

Client authentication is controlled by the `pg_hba.conf` configuration file. Entries in the `pg_hba.conf` file govern the authentication and authorization permissions for a host.

Entries in the `pg_hba.conf` file will be read for authentication whenever a connection request is received. Initially, the `pg_hba.conf` file is used to determine whether a client making a database connection request has the `CONNECT` privilege on a database object or not. Once it has been determined that a user is allowed to access the database, the next step is to ensure that all the conditions are met for the client to authenticate successfully.

Even if the user is authenticated and has permissions to connect to a database, any of the table-level permissions will still apply to the database. You can check the permissions on the database using the `\z` switch, as shown in the screenshot below:



```
hrdb=# \z
                             Access privileges
 Schema | Name | Type  |     Access privileges      | Column access privileges
--------+------+-------+----------------------------+--------------------------
 public | test | table | postgres=arwdDxt/postgres+|
        |      |       | agovil=arwdD/postgres    +|
        |      |       | nchabbra=r/postgres        |
 public | x    | table | postgres=arwdDxt/postgres+|
        |      |       | agovil=arwdDxt/postgres  +|
        |      |       | nchabbra=r/postgres        |
(2 rows)
```

During the initialization of a database connection, entries in the `pg_hba.conf` file are read from top to bottom. The moment a matching entry is found, PostgreSQL will stop the search and it will allow or reject a connection based on the mentioned rules for the found entry. The connection will fail completely if a matching entry is not located in the `pg_hba.conf` file.

There's more...

An authentication method type known as ident is defined in the pg_hba.conf configuration file. The ident authentication method works by obtaining the client's operating system username and using it as the allowed database username.

If the ident authentication method is used for a host entry in the pg_hba.conf file, then an ident map or a named mapping need to be specified. This option is defined in the pg_ident.conf configuration file, and it is used to map the identifying username, that is the client operating system username with an existing PostgreSQL database user.

The key aspect here is to obtain the client's operating system username so that it can be mapped to an existing database PostgreSQL database user.

Similar to the pg_hba.conf file, the pg_ident.conf file is also located in the data directory or in the path specified by the PGDATA environment variable.

First, the ident term must be set as the authentication method in the pg_hba.conf file, as follows:

```
# TYPE    DATABASE   USER    CIDR-ADDRESS        METHOD     OPTION
  host    hrdb        all    192.168.12.10/32    ident      hruser
```

Here, in the pga_hba.conf file, any user using the IP address 192.168.12.10 can connect to the hrdb database using an hruser mapname, which is basically a mapping of the UNIX usernames and the corresponding PostgreSQL database username. These entries are defined in the pg_ident.conf file, as follows:

```
# MAPNAME     Ident-USERNAME     PG-USERNAME
   hruser      govil_amit         agovil
   hruser      kumar_neeraj       agovil
```

The hruser identmap is now configured in the pg_ident.conf file. The entries in the pg_ident.conf file allow either of the UNIX system users, govil_amit and kumar_neeraj, to connect to the hrdb database using the PostgreSQL system user account agovil.

For more information on the entries in the pg_hba.conf file, you can refer to http://www.postgresql.org/docs/9.3/static/auth-pg-hba-conf.html.

Testing remote connectivity

After configuring the network environment in PostgreSQL, it is usually a good idea to test it out.

How to do it...

You can use the psql program to test connections to the PostgreSQL server from a remote client:

```
D:\Postgresql_Project\bin>psql -h 192.168.12.10 hrdb agovil
Password for user agovil:
psql (9.3.4)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

Hrdb=>
```

How it works...

After enabling client authentication between the PostgreSQL server and the client application, as well as after configuring access control rules in the host's pg_hba.conf configuration file, it is a good idea to test for remote connectivity.

This will help you to find out whether the access control rules are configured correctly in the pg_hba.conf file and whether the clients face any connection errors despite being allowed access based on the host configuration file's rules.

Auditing database changes

Database security remains a concern for any database application. For the purpose of audit, it is important to identify which data has been changed, who has made this change, and when and how this change was implemented in the production environment.

A change log trigger can be used as a mechanism to identify what changes have been made to data in the PostgreSQL database and to answer all the pertinent questions from the auditing perspective.

How to do it...

1. First, create a schema, and then the other objects that are required to track changes will be stored in this schema. You can create the schema as follows:

   ```
   CREATE SCHEMA logging;
   ```

2. The next step will be to create a table to store some history in order to track changes, as follows:

   ```
   CREATE TABLE logging.t_history (
     id serial,
     tstamp timestamp DEFAULT now(),
     schemaname text,
     tabname text,
     operation text,
     who text DEFAULT current_user,
     new_val json,
     old_val json
   );
   ```

   The point of using this table is to keep track of all the changes made to the table. We want to know which operation is taking place. The next important issue is when a new row is added, it will be visible by the trigger procedure. The same is true for deletion and changes.

3. Next, create a function that logs the changes, including the old changes, and values into the `t_history` table. The function is defined in such a manner that it tracks all DML operations—including inserts, updates, and deletes—and depending on the type of DML operations, it logs data—including changes—into the `t_history` table:

   ```
   CREATE FUNCTION change_trigger()
   RETURNS trigger AS $$
   BEGIN
     IF TG_OP = 'INSERT' THEN
       INSERT INTO logging.t_history (
         tabname,
         schemaname,
         operation,
         new_val
       )
       VALUES (
         TG_RELNAME,
         TG_TABLE_SCHEMA,
         TG_OP,
         row_to_json(NEW)
       );
   ```

```
  RETURN NEW;
  ELSIF TG_OP = 'UPDATE' THEN
    INSERT INTO logging.t_history (
      tabname,
      schemaname,
      operation,
      new_val,
      old_val
    )
    VALUES (
      TG_RELNAME,
      TG_TABLE_SCHEMA,
      TG_OP,
      row_to_json(NEW),
      row_to_json(OLD)
    );
  RETURN NEW;
  ELSIF TG_OP = 'DELETE' THEN
    INSERT INTO logging.t_history (
      tabname,
      schemaname,
      operation,
      old_val
    )
    VALUES (
      TG_RELNAME,
      TG_TABLE_SCHEMA,
      TG_OP,
      row_to_json(OLD)
    );
  RETURN OLD;
  END IF;
END;
$$ LANGUAGE 'plpgsql' SECURITY DEFINER;
```

4. Now, create a table with some data in it and use this table to make changes:

   ```
   CREATE TABLE t_trig (id int,name text);
   ```

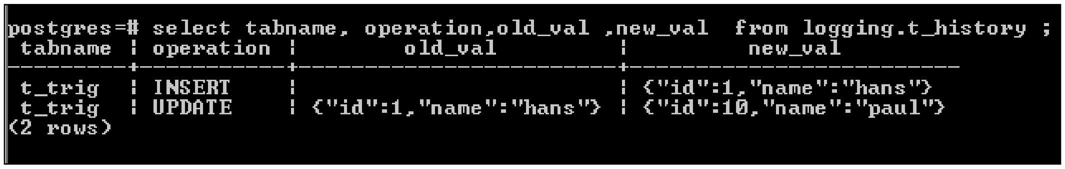
5. The next step is to create a change log trigger that will be executed before any DML event occurs on the `t_trig` table, created in the earlier step:

   ```
   CREATE TRIGGER t BEFORE INSERT OR UPDATE OR DELETE ON
   t_trig  FOR EACH ROW EXECUTE PROCEDURE change_trigger();
   ```

6. Now, make changes in the `t_trig` table and test the trigger execution, as follows:

   ```
   INSERT INTO t_trig VALUES (1, 'hans');
   UPDATE t_trig SET id = 10 * id, name ='paul';
   ```

7. Next, check whether your history tables contain changes made to the underlying tables using the query, as shown in the screenshot below:



```
postgres=# select tabname, operation,old_val ,new_val  from logging.t_history ;
 tabname | operation |         old_val          |         new_val
---------+-----------+--------------------------+--------------------------
 t_trig  | INSERT    |                          | {"id":1,"name":"hans"}
 t_trig  | UPDATE    | {"id":1,"name":"hans"} | {"id":10,"name":"paul"}
(2 rows)
```

How it works...

A generic trigger function can be used to record changes into a history table. It will record the old and new records, tables affected, users who made the change, type of DML operation, and a timestamp for each change.

It is important to ensure that the log, which is preserved to keep a track of the changes, cannot be changed by an authorized person. This can be ensured by marking the trigger function as `SECURITY DEFINER`. This will ensure that the function itself is not executed by the user who makes the change but by the user who has written the function.

This sort of a trigger-based mechanism cannot be used to track the following activities from an auditing perspective:

- It cannot audit the `SELECT` statements
- It cannot audit system tables
- It cannot audit DDL operations such as the `ALTER TABLE` statement

There's more...

There is another way to collect data changes made to PostgreSQL. These changes also includes changes made to DDL statements. We can collect the changes that are made to a PostgreSQL system from the server's logfile.

In order to collect data changes from the server log, you need to modify the `log_statement` configuration parameter and set its value to either `mod` or `all` in the `postgresql.conf` configuration file.

Once this is done, you need to reload the configuration as follows:

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

Enabling SSL in PostgreSQL

By default, the PostgreSQL server is configured to accept remote client connections using a standard TCP connection. The issue with these type of network connections is that the data is sent in clear text over the network and is clearly susceptible to sniffing. Anyone using a network sniffer can easily intercept the data sent in clear text, and in this way, the data confidentiality can be compromised.

Now, the pertinent question is what data in PostgreSQL will be susceptible to sniffing. The SQL statement sent by the psql utility to the server and the result set generated by the PostgreSQL server are some of the things that are susceptible to sniffing. Getting an interceptor to see the result set of the query means enabling the network sniffer to see your table data.

To deal with this situation, PostgreSQL supports **Secure Sockets Layer** (**SSL**) encrypted TCP sessions. SSL-based TCP encrypted sessions use an encryption key to encrypt data before it is sent out on the network. The PostgreSQL server and the client machine pass an encryption key that is used to encrypt data.

How to do it...

In order to deal with this, you need to enable SSL support in PostgreSQL. This can be done by modifying the value of the `ssl` configuration parameter in the `postgresql.conf` file:

```
ssl = on
```

When the PostgreSQL server is restarted, it will recognize the change in the configuration and enable SSL connections. The PostgreSQL server will now listen for both normal TCP connections, as well as secure SSL-based TCP connections on the same port.

However, once SSL is enabled, the PostgreSQL server will make sure that the encryption keys or certificate files are available in the PostgreSQL data directory, otherwise it will not start until it finds them.

How it works...

Now that SSL support is enabled in PostgreSQL, to support an SSL session, the PostgreSQL server must have access to both an encryption key and a certificate. The SSL protocol uses the encryption key to encrypt network data, while the remote client uses the certificate supplied by the server to validate that the encryption key came from a trusted source.

The encryption key is generated from a certificate signed by an organization that the client trusts. These two methods can be used to obtain a certificate:

- You can purchase a certificate from a certification authority such as Verisign or Thawte
- In other situations, you can create a self-signed certificate, indicating that the encryption came from your end

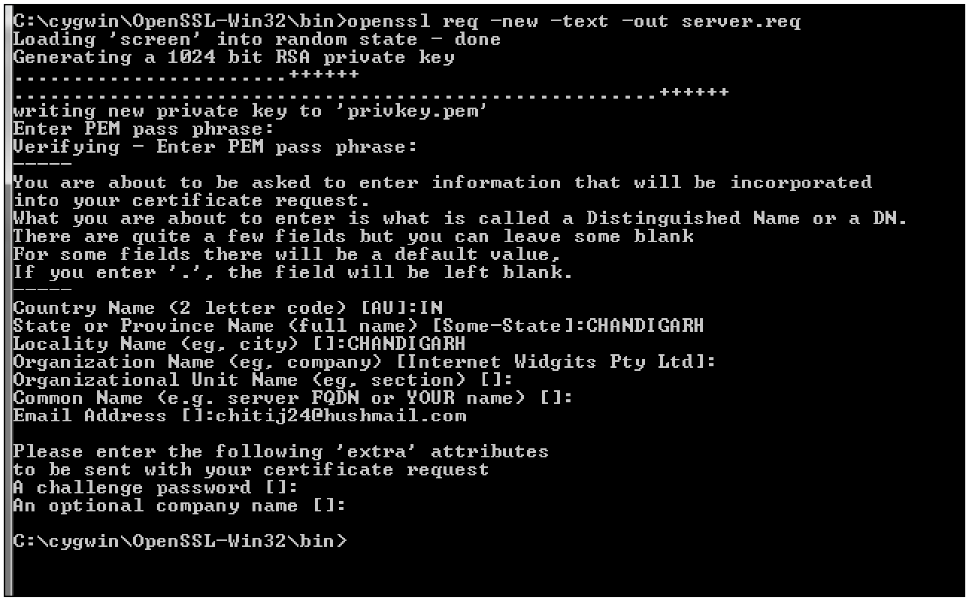
There's more...

Here we are going to create a self-signed certificate and encryption keys using an open source tool know as OpenSSL. For Windows, you need to download the latest version of the Win32OpenSSL package.

The following steps are required to create the encryption key and self-signed certificate files:

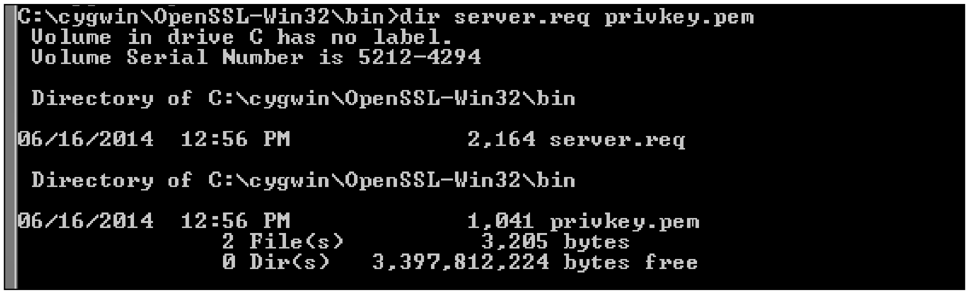
1. Create a passphrase protected encryption key.
2. Remove the key passphrase.
3. Create the self-signed certificate.

The first step is to create the encryption key used by PostgreSQL for encrypting SSL sessions. This is done using the `req` OpenSSL option:



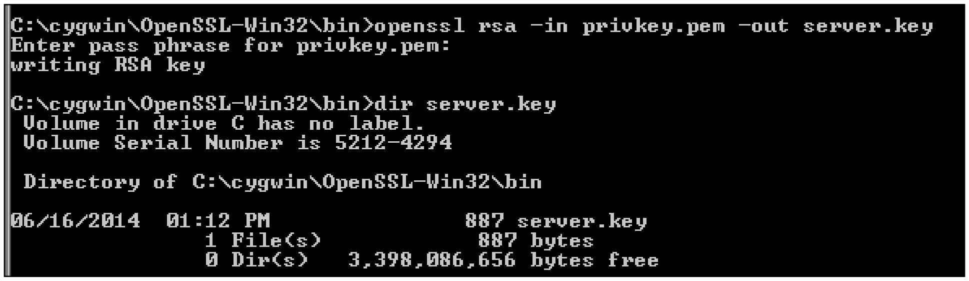
```
C:\cygwin\OpenSSL-Win32\bin>openssl req -new -text -out server.req
Loading 'screen' into random state - done
Generating a 1024 bit RSA private key
.......................++++++
......................................................++++++
writing new private key to 'privkey.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:CHANDIGARH
Locality Name (eg, city) []:CHANDIGARH
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:chitij24@hushmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

C:\cygwin\OpenSSL-Win32\bin>
```

The preceding step creates a file called `privkey.pem`, which contains the encryption key and the `server.req` file, which contains a basic certificate:



```
C:\cygwin\OpenSSL-Win32\bin>dir server.req privkey.pem
 Volume in drive C has no label.
 Volume Serial Number is 5212-4294

 Directory of C:\cygwin\OpenSSL-Win32\bin

06/16/2014  12:56 PM             2,164 server.req

 Directory of C:\cygwin\OpenSSL-Win32\bin

06/16/2014  12:56 PM             1,041 privkey.pem
               2 File(s)          3,205 bytes
               0 Dir(s)   3,397,812,224 bytes free
```

As mentioned earlier, the file that contains the `privkey.pem` encryption key is protected by a passphrase. You can remove this passphrase from the encryption key using another SSL option, as follows:



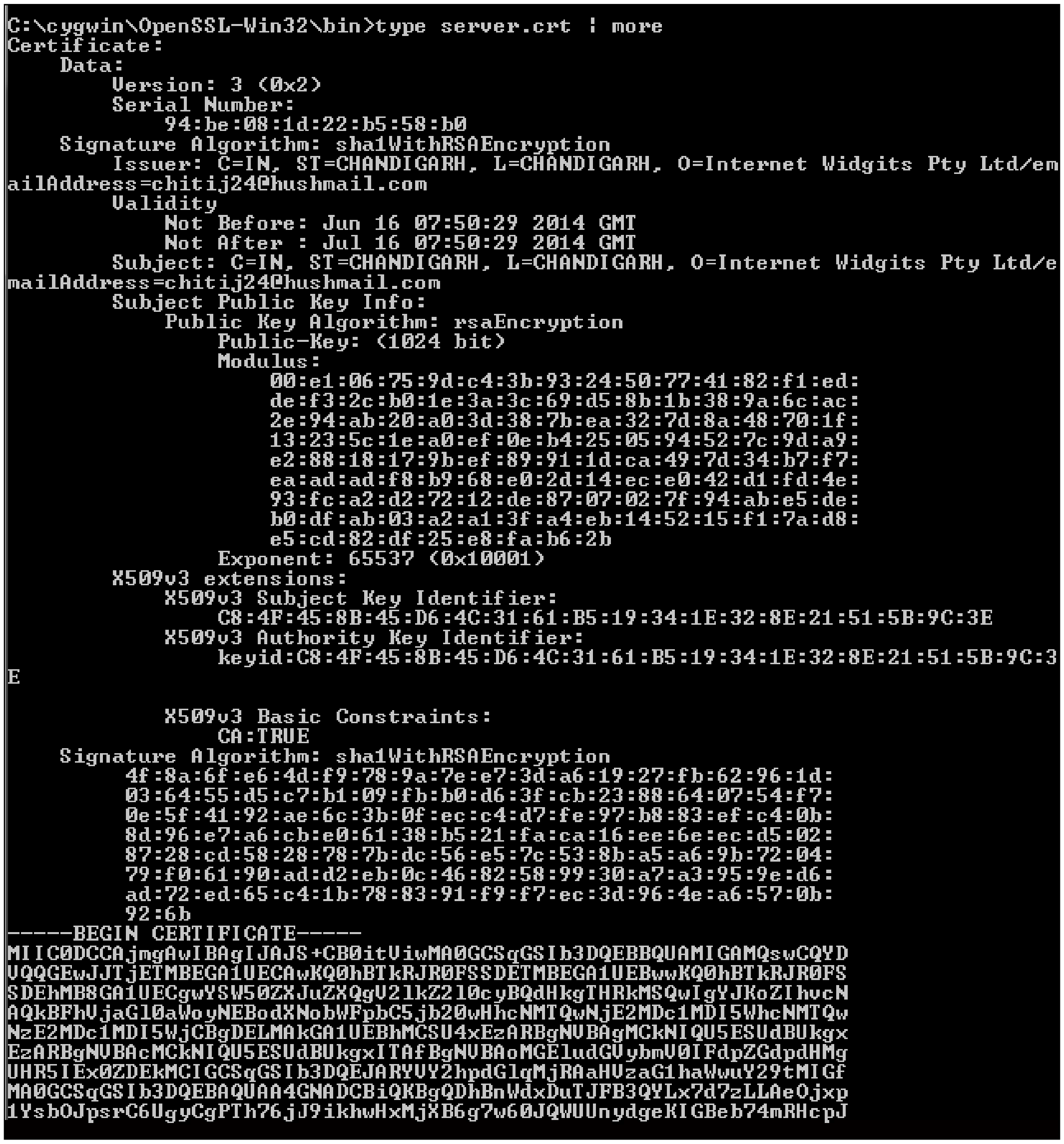
```
C:\cygwin\OpenSSL-Win32\bin>openssl rsa -in privkey.pem -out server.key
Enter pass phrase for privkey.pem:
writing RSA key

C:\cygwin\OpenSSL-Win32\bin>dir server.key
 Volume in drive C has no label.
 Volume Serial Number is 5212-4294

 Directory of C:\cygwin\OpenSSL-Win32\bin

06/16/2014  01:12 PM               887 server.key
               1 File(s)            887 bytes
               0 Dir(s)   3,398,086,656 bytes free
```

When the previously mentioned `openssl` command is executed, OpenSSL asks for the passphrase for the encryption key. It then creates a new encryption key, called `server.key`, which does not require the pass phrase to be entered.

Now that you have an encryption key without a pass phrase and a basic certificate, the next step is to convert the certificate to a standard X.509 format and self sign it using the encryption key:

```
C:\cygwin\OpenSSL-Win32\bin>openssl req -x509 -in server.req
-text -key server.key -out server.crt
Loading 'screen' into random state - done
```

The certificate is created in the text mode using the standard X.509 format and is saved in the `server.crt` file. As the certificate was created in the text mode, let's take a look at it:



```
C:\cygwin\OpenSSL-Win32\bin>type server.crt | more
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            94:be:08:1d:22:b5:58:b0
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=IN, ST=CHANDIGARH, L=CHANDIGARH, O=Internet Widgits Pty Ltd/em
ailAddress=chitij24@hushmail.com
        Validity
            Not Before: Jun 16 07:50:29 2014 GMT
            Not After : Jul 16 07:50:29 2014 GMT
        Subject: C=IN, ST=CHANDIGARH, L=CHANDIGARH, O=Internet Widgits Pty Ltd/e
mailAddress=chitij24@hushmail.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:e1:06:75:9d:c4:3b:93:24:50:77:41:82:f1:ed:
                    de:f3:2c:b0:1e:3a:3c:69:d5:8b:1b:38:9a:6c:ac:
                    2e:94:ab:20:a0:3d:38:7b:ea:32:7d:8a:48:70:1f:
                    13:23:5c:1e:a0:ef:0e:b4:25:05:94:52:7c:9d:a9:
                    e2:88:18:17:9b:ef:89:91:1d:ca:49:7d:34:b7:f7:
                    ea:ad:ad:f8:b9:68:e0:2d:14:ec:e0:42:d1:fd:4e:
                    93:fc:a2:d2:72:12:de:87:07:02:7f:94:ab:e5:de:
                    b0:df:ab:03:a2:a1:3f:a4:eb:14:52:15:f1:7a:d8:
                    e5:cd:82:df:25:e8:fa:b6:2b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                C8:4F:45:8B:45:D6:4C:31:61:B5:19:34:1E:32:8E:21:51:5B:9C:3E
            X509v3 Authority Key Identifier:
                keyid:C8:4F:45:8B:45:D6:4C:31:61:B5:19:34:1E:32:8E:21:51:5B:9C:3
E


            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha1WithRSAEncryption
         4f:8a:6f:e6:4d:f9:78:9a:7e:e7:3d:a6:19:27:fb:62:96:1d:
         03:64:55:d5:c7:b1:09:fb:b0:d6:3f:cb:23:88:64:07:54:f7:
         0e:5f:41:92:ae:6c:3b:0f:ec:c4:d7:fe:97:b8:83:ef:c4:0b:
         8d:96:e7:a6:cb:e0:61:38:b5:21:fa:ca:16:ee:6e:ec:d5:02:
         87:28:cd:58:28:78:7b:dc:56:e5:7c:53:8b:a5:a6:9b:72:04:
         79:f0:61:90:ad:d2:eb:0c:46:82:58:99:30:a7:a3:95:9e:d6:
         ad:72:ed:65:c4:1b:78:83:91:f9:f7:ec:3d:96:4e:a6:57:0b:
         92:6b
-----BEGIN CERTIFICATE-----
MIIC0DCCAjmgAwIBAgIJAJS+CB0itViwMA0GCSqGSIb3DQEBBQUAMIGAMQswCQYD
VQQGEwJJTjETMBEGA1UECAwKQ0hBTkRJR0FSSDETMBEGA1UEBwwKQ0hBTkRJR0FS
SDEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMSQwIgYJKoZIhvcN
AQkBFhVjaGl0aWoyNEBodXNobWFpbC5jb20wHhcNMTQwNjE2MDc1MDI5WhcNMTQw
NzE2MDc1MDI5WjCBgDELMAkGA1UEBhMCSU4xEzARBgNVBAgMCkNIQU5ESUdBUkgx
EzARBgNVBAcMCkNIQU5ESUdBUkgxITAfBgNVBAoMGEludGVybmV0IFdpZGdpdHMg
UHR5IEx0ZDEkMCIGCSqGSIb3DQEJARYVY2hpdGlqMjRAaHVzaG1haWwuY29tMIGf
MA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDhBnWdxDuTJFB3QYLx7d7zLLAeOjxp
1YsbOJpsrC6UgyCgPTh76jJ9ikhwHxMjXB6g7w60JQWUUnydgeKIGBeb74mRHcpJ
```

The next step will be to copy the `server.key` and `server.crt` files to the PostgreSQL data directory and then restart the PostgreSQL server service.

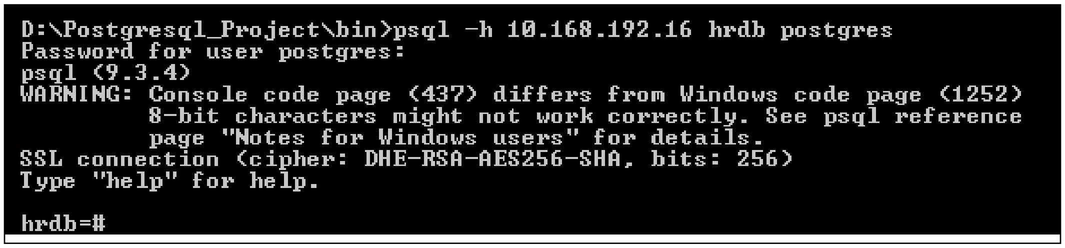
For more information on SSL in PostgreSQL, you can refer to `www.postgresql.org/docs/9.3/static/ssl-tcp.html`.

Testing SSL encryption

Usually, communication between the client application, such as psql, and the database server, that is PostgreSQL, is sent over the network as clear text and is susceptible to sniffing. To prevent any sort of eavesdropping or sniffing, ensure that the communication between the psql client and the database server is encrypted. We set up SSL and enabled encryption in the previous recipe. Once the certificate and encryption key files are moved to the data directory, the next step is to check the SSL encryption. It is now time to test the SSL encryption that we set up in the previous recipe.

How to do it...

After the PostgreSQL server is restarted, the next step is to use the psql application in order to test the SSL connection, as shown here:



```
D:\Postgresql_Project\bin>psql -h 10.168.192.16 hrdb postgres
Password for user postgres:
psql (9.3.4)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

hrdb=#
```

How it works...

The psql application attempts to connect to the PostgreSQL server in the SSL mode first and then tries to connect in the plain text mode if the SSL mode fails. While making the connection to the psql client, you can see the banner information containing keywords such as SSL connection and some cipher text. This can be seen from the screenshot in the earlier section.

Encrypting confidential data

It is important to protect confidential information stored in databases, such as credit card information, information about financial transactions, and personal information of an employee. Usual database mechanisms, such as maintaining access control lists to implement tight security controls on confidential information, to ensure that such sensitive information does not fall into the hands of malicious users is not enough. What is important is to ensure that the confidential data be kept in a format that is not understandable to unauthorized users. For authorized users, however, the information must be converted back to its original format so that it is understandable. This is where encryption comes into the picture. Encryption is the process of converting data into a format that renders the data unreadable or intangible to unauthorized users.

Encryption translates the data into cipher text or secret mode, which can only be decoded and converted into its original form with the help of a key that is kept by authorized personnel only. For this reason, encryption is considered to be one of the most effective ways to achieve data security.

There are two categories of encryption: one is the symmetric system, and the other is the asymmetric system. **Symmetric encryption** uses an identical key to both encrypt and decrypt the data. Symmetric key algorithms are much faster computationally than asymmetric algorithms, as the encryption process is less complicated and takes less time. **Asymmetric encryption** uses two related keys (public and private) for data encryption and decryption and eliminates the security risk of sharing keys. The private key is never exposed. A message that is encrypted using the public key can only be decrypted by applying the same algorithm and using a matching private key. Likewise, a message that is encrypted using the private key can only be decrypted using the matching public key.

How to do it…

PostgreSQL has various levels of encryption to choose from. PostgreSQL provides the `pgcrypto` module, which provides cryptographic functions for PostgreSQL.

In the following section, we are going to create a table and use the **Advanced Encryption Standard** (**AES**) to encrypt the table data and then decrypt the data via the `encrypt` and `decrypt` functions:

```
testdb1=# create extension pgcrypto;

testdb1=# create table demo(pw bytea);

testdb1=# insert into demo(pw) values ( encrypt( 'champion', 'key',
'aes') );

testdb1=# select * from demo;
               pw
------------------------------------
\xdf5fa25e36fd16c9e4688bcf46bf11c3
(1 row)


testdb1=# select decrypt(pw, 'key', 'aes') FROM demo;
    decrypt
--------------------
\x6368616d70696f6e
(1 row)


testdb1=# select convert_from(decrypt(pw, 'key', 'aes'), 'utf-8') FROM
demo;
```

```
 convert_from
--------------
 champion
(1 row)
```

How it works...

The pgcrypto module is a module in PostgreSQL which provides encryption in the form of database functions. It is client independent. The pgcrypto module provides support for raw encryption, **Pretty Good Privacy** (**PGP**) compatible encryption, and hashing.

PostgreSQL provides supports for both symmetric and asymmetric encryption. For stronger encryption, you can use a PGP-based encryption approach where you have a public and a private key pair, in which case, the public key is used to encrypt the data and the private key is used to decrypt the data.

We are now going to demonstrate the use of a public and private key pair to encrypt and decrypt the data. We are going to use the following four functions for the demonstration, along with an explanation of the usage of these functions:

- pgp_pub_encrypt: This is the function you will use to encrypt your data using your public key.
- pgp_pub_decrypt: This is the function you will use to decrypt your data using your private key.
- dearmor: The dearmor function is used to unwrap binary data into its native format, that is, the PGP ASCII armor format, which makes it suitable to be passed to the encrypt.
- pgp_key_id: The pgp_key_id function is used to extract the key ID of a public or secret key. This function tells you the key that was used to encrypt a given message, so that from the collection of available keys, you can use the right key to decrypt the given message.

The following are a series of steps that are used to demonstrate the usage of public and private key pairs to encrypt and decrypt data using the previously described functions:

1. First, create the table in which you are going to store data:

```
CREATE TABLE testuserscards(
  card_id SERIAL PRIMARY KEY,
  username varchar(100),
  cc bytea
);
```

2. Next, insert records in the table and encrypt the data:

```
INSERT INTO testuserscards(username, cc)
SELECT robotccs.username, pgp_pub_encrypt(robotccs.cc, keys.
pubkey) As cc
```

```
FROM (VALUES ('robby', '41111111111111111'),
    ('artoo', '41111111111111112') ) As robotccs(username, cc)
     CROSS JOIN (SELECT dearmor('
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.1 (GNU/Linux)

mQGiBELIIgoRBAC1onBpxKYgDvrgCaUWPY34947X3ogxGOfCN0p6Eqrx+2PUhm4n
vFvmczpMT4iDc0mUO+iwnwsEkXQI1eC99g8c0jnZAvzJZ5miAHL8hukMAMfDkYke
5aVvcPPc8uPDlItpszGmH0rM0V9TIt/i9QEXetpyNWhk4jj5qnohYhLeZwCgkOdO
RFAdNi4vfFPivvtAp2ffjU8D/R3x/UJCvkzi7i9rQHGo313xxmQu5BuqIjANBUij
8IE7LRPI/Qhg2hYy3sTJwImDi7VkS+fuvNVk0d6MTWplAXYU96bn12JaD21R9sKl
Fzcc+0iZI1wYA1PczisUkoTISE+dQFUsoGHfpDLhoBuesXQrhBavI8t8VPd+nkdt
J+oKA/9iRQ87FzxdYTkh2drrv69FZHc3Frsjw9nPcBq/voAvXH0MRilqyCg7HpW/
T9naeOERksa+Rj4R57IF1l4e5oiiGJo9QmaKZcsCsXrREJCycrlEtMqXfSPy+bi5
0yDZE/Qm1dwu13+OXOsRvkoNYjO8Mzo9K8wU12hMqN0a2bu6a7QjRWxnYW1hbCAy
MDQ4IDx0ZXN0MjA0OEBleGFtcGxlLm9yZz6IXgQTEQIAHgUCQsgiCgIbAwYLCQgH
AwIDFQIDAxYCAQIeAQIXgAAKCRBI6c1W/qZo29PDAKCG724enIxRog1j+aeCp/uq
or6mbwCePuKy2/1kD1FvnhkZ/R5fpm+pdm25Ag0EQsgiIhAIAJI3Gb2Ehtz1taQ9
AhPY4Avad2BsqD3S5X/R11Cm0KBE/04D29dxn3f8QfxDsexYvNIZjoJPBqqZ7iMX
MhoWyw8ZF5Zs1mLIjFGVorePrm94N3MNPWM7x9M36bHUjx0vCZKFIhcGY1g+htE/
QweaJzNVeA5z4qZmik41FbQyQSyHa3bOkTZu++/U6ghP+iDp5UDBjMTkVyqITUVN
gC+MR+da/I60irBVhue7younh4ovF+CrVDQJC06HZl6CAJJyA81SmRfi+dmKbbjZ
LF6rhz0norPjISJvkIqvdtM4VPBKI5wpgwCzpEqjuiKrAVujRT68zvBvJ4aVqb11
k5QdJscAAwUH/jVJh0HbWAoiFTe+NvohfrA8vPcD0rtU3Y+siiqrabotnxJd2NuC
bxghJYGfNtnx0KDjFbCRKJVeTFok4UnuVYhXdH/c6i0/rCTNdeW2D6pmR4GfBozR
Pw/ARf+jONawGLyUj7uq13iquwMSE7VyNuF3ycL2OxXjgOWMjkH8c+zfHHpjaZ0R
QsetMq/iNBWraayKZnWUd+eQqNzE+NUo7w1jAu7oDpy+8a1eipxzK+O0HfU5LTiF
Z1Oe4Um0P2l3Xtx8nEgj4vSeoEkl2qunfGW00ZMMTCWabg0ZgxPzMfMeIcm6525A
Yn2qL+X/qBJTInAl7/hgPz2D1Yd7d5/RdWaISQQYEQIACQUCQsgiIgIbDAAKCRBI
6c1W/qZo25ZSAJ98WTrtl2HiX8ZqZq95v1+9cHtZPQCfZDoWQPybkNescLmXC7q5
1kNTmEU=
=8QM5
-----END PGP PUBLIC KEY BLOCK-----
') As pubkey) As keys;
```

3. You might then see the records in the table:

```
SELECT username, cc FROM testuserscards;
```

4. Now, you can use `pgp_keyid` to verify which public key you used to encrypt your data:

```
SELECT pgp_key_id(dearmor('
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.1 (GNU/Linux)

mQGiBELIIgoRBAC1onBpxKYgDvrgCaUWPY34947X3ogxGOfCN0p6Eqrx+2PUhm4n
vFvmczpMT4iDc0mUO+iwnwsEkXQI1eC99g8c0jnZAvzJZ5miAHL8hukMAMfDkYke
5aVvcPPc8uPDlItpszGmH0rM0V9TIt/i9QEXetpyNWhk4jj5qnohYhLeZwCgkOdO
RFAdNi4vfFPivvtAp2ffjU8D/R3x/UJCvkzi7i9rQHGo313xxmQu5BuqIjANBUij
8IE7LRPI/Qhg2hYy3sTJwImDi7VkS+fuvNVk0d6MTWplAXYU96bn12JaD21R9sKl
Fzcc+0iZI1wYA1PczisUkoTISE+dQFUsoGHfpDLhoBuesXQrhBavI8t8VPd+nkdt
J+oKA/9iRQ87FzxdYTkh2drrv69FZHc3Frsjw9nPcBq/voAvXH0MRilqyCg7HpW/
T9naeOERksa+Rj4R57IF1l4e5oiiGJo9QmaKZcsCsXrREJCycrlEtMqXfSPy+bi5
0yDZE/Qm1dwu13+OXOsRvkoNYjO8Mzo9K8wU12hMqN0a2bu6a7QjRWxnYW1hbCAy
MDQ4IDx0ZXN0MjA0OEBleGFtcGxlLm9yZz6IXgQTEQIAHgUCQsgiCgIbAwYLCQgH
AwIDFQIDAxYCAQIeAQIXgAAKCRBI6c1W/qZo29PDAKCG724enIxRog1j+aeCp/uq
or6mbwCePuKy2/1kD1FvnhkZ/R5fpm+pdm25Ag0EQsgiIhAIAJI3Gb2Ehtz1taQ9
AhPY4Avad2BsqD3S5X/R11Cm0KBE/04D29dxn3f8QfxDsexYvNIZjoJPBqqZ7iMX
MhoWyw8ZF5Zs1mLIjFGVorePrm94N3MNPWM7x9M36bHUjx0vCZKFIhcGY1g+htE/
QweaJzNVeA5z4qZmik41FbQyQSyHa3bOkTZu++/U6ghP+iDp5UDBjMTkVyqITUVN
gC+MR+da/I60irBVhue7younh4ovF+CrVDQJC06HZl6CAJJyA81SmRfi+dmKbbjZ
LF6rhz0norPjISJvkIqvdtM4VPBKI5wpgwCzpEqjuiKrAVujRT68zvBvJ4aVqb11
k5QdJscAAwUH/jVJh0HbWAoiFTe+NvohfrA8vPcD0rtU3Y+siiqrabotnxJd2NuC
bxghJYGfNtnx0KDjFbCRKJVeTFok4UnuVYhXdH/c6i0/rCTNdeW2D6pmR4GfBozR
Pw/ARf+jONawGLyUj7uq13iquwMSE7VyNuF3ycL2OxXjgOWMjkH8c+zfHHpjaZ0R
QsetMq/iNBWraayKZnWUd+eQqNzE+NUo7w1jAu7oDpy+8a1eipxzK+O0HfU5LTiF
Z1Oe4Um0P2l3Xtx8nEgj4vSeoEkl2qunfGW00ZMMTCWabg0ZgxPzMfMeIcm6525A
Yn2qL+X/qBJTInAl7/hgPz2D1Yd7d5/RdWaISQQYEQIACQUCQsgiIgIbDAAKCRBI
6c1W/qZo25ZSAJ98WTrtl2HiX8ZqZq95v1+9cHtZPQCfZDoWQPybkNescLmXC7q5
1kNTmEU=
=8QM5
-----END PGP PUBLIC KEY BLOCK-----'));
```

The output of this query shows that the following public key was encrypting data:

```
    pgp_key_id
------------------
 2C226E1FFE5CC7D4
(1 row)
```

5. The next step is to verify whether the public key that you got was used to encrypt the data in the table:

```
hrdb=# SELECT username, pgp_key_id(cc) As keyweused FROM
testuserscards;

 username |    keyweused
----------+------------------
 robby    | 2C226E1FFE5CC7D4
 artoo    | 2C226E1FFE5CC7D4
```

6. Finally, decrypt the data using the private key that matches the public key you used to encrypt the data with:

```
SELECT username, pgp_pub_decrypt(cc, keys.privkey) As ccdecrypt
FROM testuserscards
    CROSS JOIN
    (SELECT dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: GnuPG v1.4.1 (GNU/Linux)

lQG7BELIIgoRBAC1onBpxKYgDvrgCaUWPY34947X3ogxGOfCN0p6Eqrx+2PUhm4n
vFvmczpMT4iDc0mUO+iwnwsEkXQI1eC99g8c0jnZAvzJZ5miAHL8hukMAMfDkYke
5aVvcPPc8uPDlItpszGmH0rM0V9TIt/i9QEXetpyNWhk4jj5qnohYhLeZwCgkOdO
RFAdNi4vfFPivvtAp2ffjU8D/R3x/UJCvkzi7i9rQHGo313xxmQu5BuqIjANBUij
8IE7LRPI/Qhg2hYy3sTJwImDi7VkS+fuvNVk0d6MTWplAXYU96bn12JaD21R9sKl
Fzcc+0iZI1wYA1PczisUkoTISE+dQFUsoGHfpDLhoBuesXQrhBavI8t8VPd+nkdt
J+oKA/9iRQ87FzxdYTkh2drrv69FZHc3Frsjw9nPcBq/voAvXH0MRilqyCg7HpW/
T9naeOERksa+Rj4R57IF1l4e5oiiGJo9QmaKZcsCsXrREJCycrlEtMqXfSPy+bi5
0yDZE/Qm1dwu13+OXOsRvkoNYjO8Mzo9K8wU12hMqN0a2bu6awAAn2F+iNBElfJS
8azqO/kEiIfpqu6/DQG0I0VsZ2FtYWwgMjA0OCA8dGVzdDIwNDhAZXhhbXBsZS5v
cmc+iF0EExECAB4FAkLIIgoCGwMGCwkIBwMCAxUCAwMWAgECHgECF4AACgkQSOnN
Vv6maNvTwwCYkpcJmpl3aHCQdGomz7dFohDgjgCgiThZt2xTEi6GhBB1vuhk+f55
n3+dAj0EQsgiIhAIAJI3Gb2Ehtz1taQ9AhPY4Avad2BsqD3S5X/R11Cm0KBE/04D
29dxn3f8QfxDsexYvNIZjoJPBqqZ7iMXMhoWyw8ZF5Zs1mLIjFGVorePrm94N3MN
PWM7x9M36bHUjx0vCZKFIhcGY1g+htE/QweaJzNVeA5z4qZmik41FbQyQSyHa3bO
kTZu++/U6ghP+iDp5UDBjMTkVyqITUVNgC+MR+da/I60irBVhue7younh4ovF+Cr
VDQJC06HZl6CAJJyA81SmRfi+dmKbbjZLF6rhz0norPjISJvkIqvdtM4VPBKI5wp
gwCzpEqjuiKrAVujRT68zvBvJ4aVqb11k5QdJscAAwUH/jVJh0HbWAoiFTe+Nvoh
frA8vPcD0rtU3Y+siiqrabotnxJd2NuCbxghJYGfNtnx0KDjFbCRKJVeTFok4Unu
VYhXdH/c6i0/rCTNdeW2D6pmR4GfBozRPw/ARf+jONawGLyUj7uq13iquwMSE7Vy
NuF3ycL2OxXjgOWMjkH8c+zfHHpjaZ0RQsetMq/iNBWraayKZnWUd+eQqNzE+NUo
7w1jAu7oDpy+8a1eipxzK+O0HfU5LTiFZ1Oe4Um0P2l3Xtx8nEgj4vSeoEkl2qun
fGW00ZMMTCWabg0ZgxPzMfMeIcm6525AYn2qL+X/qBJTInAl7/hgPz2D1Yd7d5/R
dWYAAVQKFPXbRaxbdArwRVXMzSD3qj/+VwwhwEDt8zmBGnlBfwVdkjQQrDUMmV1S
EwyISQQYEQIACQUCQsgiIgIbDAAKCRBI6c1W/qZo25ZSAJ4sgUfHTVsG/x3p3fcM
3b5R86qKEACggYKSwPWCs0YVRHOWqZY0pnHtLH8=
=3Dgk
-----END PGP PRIVATE KEY BLOCK-----') As privkey) As keys;
```

```
 username |     ccdecrypt
----------+-------------------
 robby    | 4111111111111111
 artoo    | 4111111111111112
(2 rows)
```

There's more...

Instead of explicitly specifying the private/public key pair, you can also use a tool called GPG to generate the public and secret keys and export it and use it in PostgreSQL.

GPG is available both for Linux and Windows platforms.

You can use the following sequence of steps to generate the `gpg` keys and export them:

1. First, generate the keys:

   ```
   gpg --gen-key
   ```

2. Next, see the list of keys that you generated:

   ```
   gpg --list-secret-keys

   sec   1024D/D9ABCD1E 2014-06-17
   uid                  aaaac
   ssb   1024g/E0B81D3A 2014-06-17
   ```

3. Finally, export both the secret and public keys:

   ```
   gpg -a --export E0B81D3A > public.key

   gpg -a --export-secret-keys D9ABCD1E > secret.key
   ```

Cracking PostgreSQL passwords

Many databases including open source databases as well as proprietary ones, come with default user accounts, and such schemas also have default passwords. These passwords are well known in today's context, and it is important that a database administrator keep nondefault passwords for these user accounts. However, administrators usually prefer to keep simple passwords or sometimes allow default passwords to be kept. This is something that needs to be avoided in a production environment because compromising here would lead to a big security loophole and something that can be exploited by hackers. For this reason, organizations have started implementing a strong password policy.

A common norm in password policies is to keep a combination of alphanumeric characters coupled with a few special characters to enforce a strong password. It is important to keep a password length of at least up to eight characters. In the following recipe, we are going to see how weak passwords can be cracked and how important it is to enforce a strong password policy.

How to do it...

Here, we are presenting a scenario where we will demonstrate how weak passwords can be easily cracked. For the purpose of demonstration, we will create two users: one user whose password contains only digits, and one user whose password contains only alphabets. Perform the following steps to create the users:

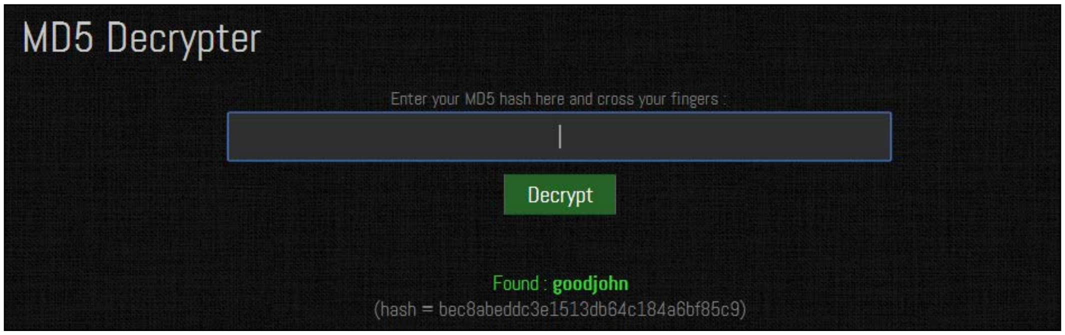
1. First, create a user and specify a password for the user:

```
create user xyz with password '123';
create user john with password 'good';
```

2. Next, get the encrypted passwords of users from the `pg_shadow` catalog table:

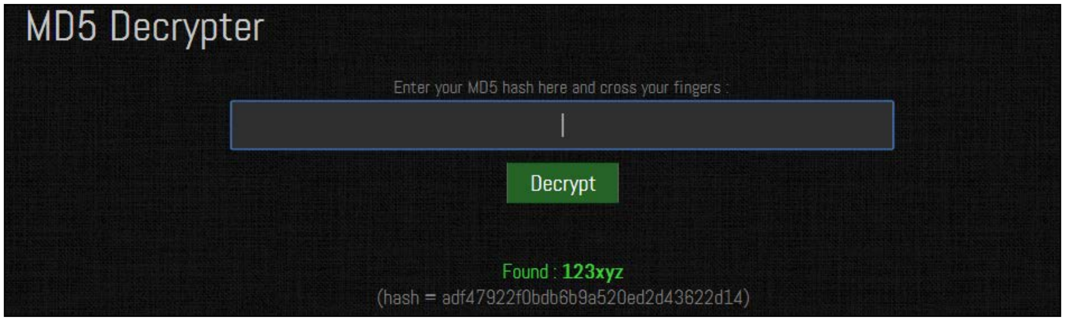
```
select usename as useraccount,passwd as "password"
from pg_shadow
where length(passwd)>1 order by usename;

 useraccount |               password
-------------+-------------------------------------
 john        | md5bec8abeddc3e1513db64c184a6bf85c9
 xyz         | md5adf47922f0bdb6b9a520ed2d43622d14
(2 rows)
```

3. The next step will be to use an MD5 password decrypter tool. You can use tools such as Cain and Abel, MD Crack, and so on. However, in our case, we will be using a website called `www.md5online.org`, and we will use its online MD5 decrypter facility, as shown here:



Found : goodjohn
(hash = bec8abeddc3e1513db64c184a6bf85c9)

In the above screenshot, we entered the MD5 hash for the user `john` and found his actual password. The convention that we see here is that the password is displayed first, followed by the name of the user. For instance, we get **goodjohn**, where the password is `good` for the user `john`.



Found : 123xyz
(hash = adf47922f0bdb6b9a520ed2d43622d14)

Similarly, in the preceding screenshot, we entered the md5 hash for the user `xyz` and found his actual password. The convention that we see here as well, is just as in the earlier case, where the password is displayed first, followed by the name of the user. For instance, we get **123xyz**, where the password is `123` for the user `xyz`.

How it works...

In the preceding scenario, you can see that any password less than six characters in length is vulnerable to being quickly cracked. Therefore, it is important to enforce a strong password policy and to educate users with the effectiveness of a strong password.

The length of the password plays a key role in enforcing a strong password. The longer the length of the password, the more is the time it requires to break it. The cracking of passwords is based on two approaches:

- **Brute force**: This method requires you to try every possible approach needed to be undertaken in order to crack a password. There are well-known password crackers, such as Cain and Abel, LophtCrack, Hydra, and so on, that can use the brute force approach to crack a password. This method is suitable only for testing short passwords.
- **Dictionary attack**: This method involves the usage of a dictionary list in order to crack a password. Here, every dictionary word is taken in sequence, converted into a hash, and then matched with the system hash. If both the hashes match, then the password is cracked, else move to the next word in the dictionary and so on.

3
Backup and Recovery

In this chapter, we will cover the following recipes:

- A logical backup of a single PostgreSQL database
- A logical backup of all PostgreSQL databases
- A logical backup of specific objects
- File system level backup
- Taking a base backup
- Hot physical backup and continuous archiving
- Point-in-time recovery
- Restoring databases and specific database objects

Introduction

Backup and recovery usually refers to protecting the database against the loss of data and enables the restoration of data in the event of a data loss. A backup, in simple terms, is a copy of your database data.

Backups are divided into two components:

- **Logical backups**: A logical backup refers to the dump file that is created by the pg_dump utility and which might be used to restore the database in the case of a data loss or an accidental deletion of a database object, such as a table. The pg_dump utility is a PostgreSQL specific utility that can be run on the command line, which makes a connection to the database and initiates the logical backup.
- **Physical backups**: A physical backup refers to the OS level backup of a database directory and its associated files.

It is essential to have a planning strategy in order to implement backups. This is desirable from the point of view of a recovery scenario, and in the event of such a situation arising, the type of backups that we initiate will influence the type of recovery that is possible.

A logical backup of a single PostgreSQL database

The `pg_dump` utility is used to back up a PostgreSQL database. It does make consistent backups even if the database is being used by other transactions. Dumps can be created in script or in archive file formats. Script dumps are usually plain text files that contain the SQL commands required to reconstruct the database to the state it was in at the time it was saved. Script dumps can also be used to reconstruct the database on other machines and architectures.

Getting ready

Please note that the dump keyword is evenly used here as a synonym for backup.

The `pg_dump` utility is considered to be a logical backup because it makes a copy of the data in the database by dumping out the contents of each table.

The basic syntax to take a logical backup of a single database is mentioned here:

```
pg_dump -U username -W -F t database_name > [Backup Location Path]
```

The usage of the options used with the `pg_dump` command is explained here:

- **U switch**: The `-U` switch specifies the database user initiating the connection. As `pg_dump` is a command-line utility, we need to specify the username via which the `pg_dump` utility can make a database connection.
- **W switch**: This option is not mandatory. This option forces `pg_dump` to prompt for the password before connecting to the PostgreSQL database server. After you press *Enter*, `pg_dump` will prompt for the password of the database user from which the connection is initiated.
- **F switch**: The `-F` switch specifies the output file format that will be used. We specified the `t` option with the `-F` switch because the output file will be implemented as a tar format archive file.

There are plenty of other options available with the `pg_dump` command; however, for our purpose, we are going to the use the preceding options.

How to do it...

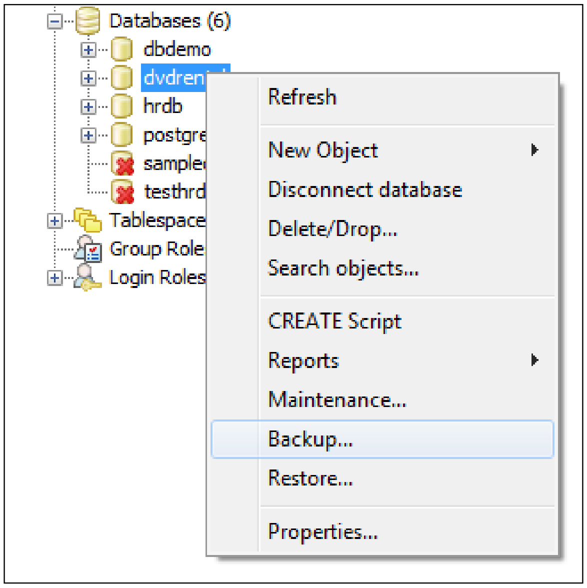
Here, in our situation, we have a database named dvdrental for which we need to generate a logical dump.

There are two ways in which a logical dump can be initiated in PostgreSQL:

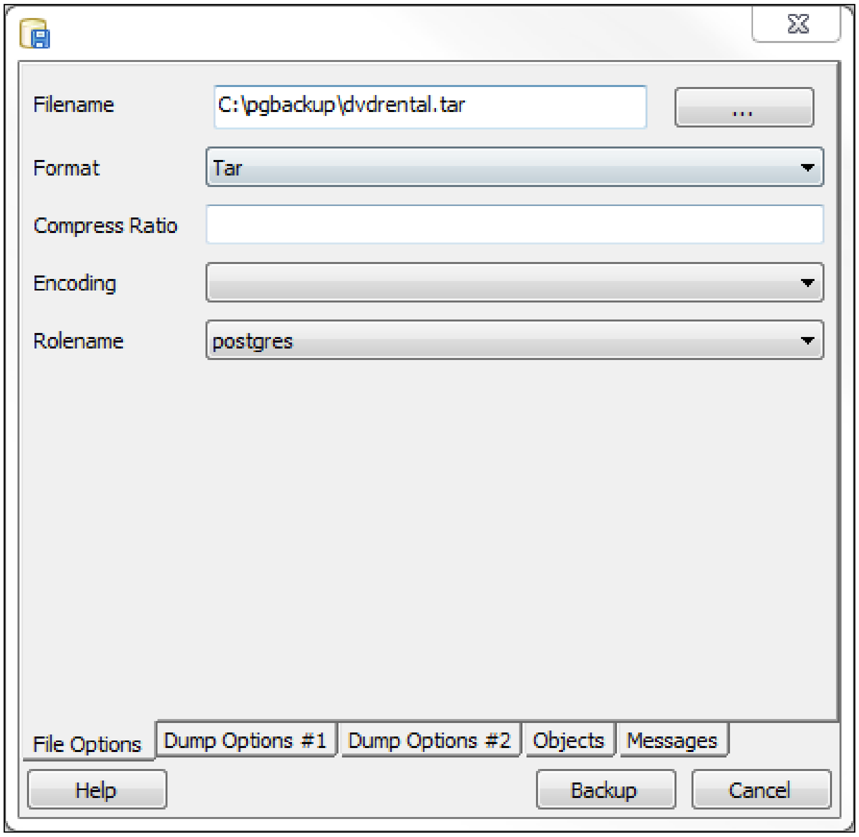
- The first approach is to use the command-line utility pg_dump to make a logical dump of a database. Here, we use the pg_dump utility to back up the dvdrental database in an output file named dvdrental.tar, which is saved in the abcd subdirectory of the home directory:

  ```
  pg_dump -U postgres -W -F t dvdrental >
  /home/abcd/dvdrental.tar
  ```

- The second option is to use the pgAdmin GUI tool to back up an individual database. Here, we will show you how to backup the dvdrental database using the pgAdmin tool:
  1. First, launch the pgAdmin GUI tool.
  2. Click on the **Databases** menu under **Object browser** in the left pane of the window, select the dvdrental database, and right-click on it.
  3. Then, select the **Backup...** option, as shown in the following screenshot:



4. Once the **Backup...** option is selected, a dialog box will open, as shown in the next screenshot, and you will have to enter the name of the logical dump that will contain the command necessary to restore the database or a specific table in the event of a failure. Here, we name the logical dump as `dvdrental.tar` and store it in the `pgbackup` directory under the `C` drive.



Filename: C:\pgbackup\dvdrental.tar

Format: Tar

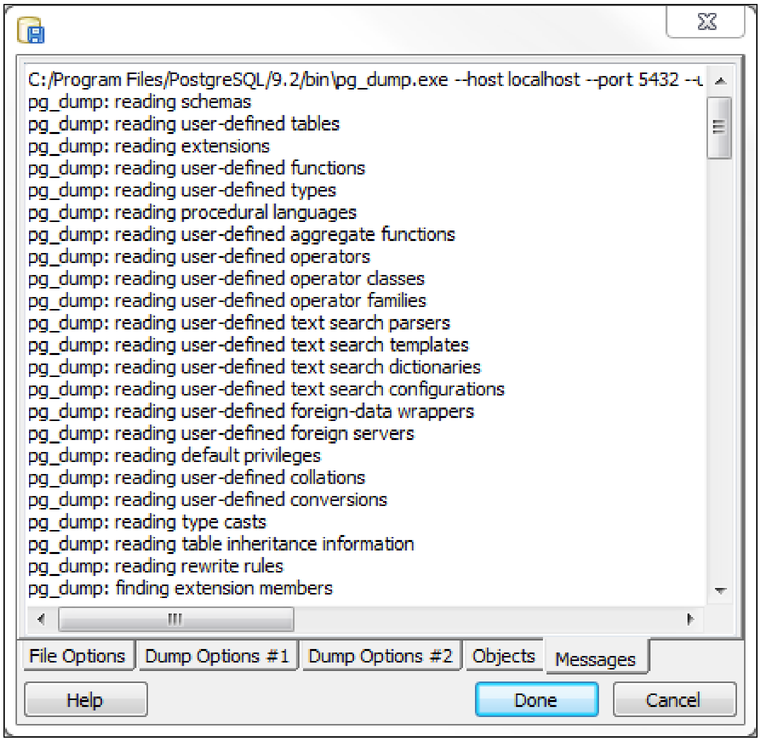
Compress Ratio:

Encoding:

Rolename: postgres

5. Click on the **Backup** button to generate a logical dump of the `dvdrental` database.

6. On the click of the **Backup** button, the creation of the logical dump will start, and the backup-related messages that are generated can be seen in the next screenshot:



```
C:/Program Files/PostgreSQL/9.2/bin\pg_dump.exe --host localhost --port 5432 --u
pg_dump: reading schemas
pg_dump: reading user-defined tables
pg_dump: reading extensions
pg_dump: reading user-defined functions
pg_dump: reading user-defined types
pg_dump: reading procedural languages
pg_dump: reading user-defined aggregate functions
pg_dump: reading user-defined operators
pg_dump: reading user-defined operator classes
pg_dump: reading user-defined operator families
pg_dump: reading user-defined text search parsers
pg_dump: reading user-defined text search templates
pg_dump: reading user-defined text search dictionaries
pg_dump: reading user-defined text search configurations
pg_dump: reading user-defined foreign-data wrappers
pg_dump: reading user-defined foreign servers
pg_dump: reading default privileges
pg_dump: reading user-defined collations
pg_dump: reading user-defined conversions
pg_dump: reading type casts
pg_dump: reading table inheritance information
pg_dump: reading rewrite rules
pg_dump: finding extension members
```

How it works...

The pg_dump command runs by executing SQL statements against the database to unload data. While the pg_dump command is running, it acquires locks on the tables that are being dumped. This is done in order to ensure that DDL operations are restricted against the tables while the dump is running in order to ensure data consistency.

Dumps created by pg_dump are internally consistent; that is, the dump represents a snapshot of the database at the time pg_dump begins running. The pg_dump utility does not block any other database operations while it is executing. In this case, the only exceptions are those operations that require an exclusive lock to operate.

Any system user might run pg_dump by default, but the user with which you connect to PostgreSQL must have SELECT rights for every database object being dumped.

Since pg_dump also provides standard connection options to specify a host connection, it can also be used to perform remote backups from any host that is allowed to make a remote connection, as defined in the pg_hba.conf file:

```
pg_dump -u postgres -h 192.168.16.54   -F c -f
dvdrental.sql.tar.gz dvdrental
```

Here, in the preceding scenario, we connect to the dvdrental database located at the host with the IP address 192.168.16.54 and initiate a remote backup for the dvdrental database. The dvdrental.sql.tar.gz dump file is generated at the current location from where the pg_dump command is executed.

You can refer to the following links for more detailed information on the pg_dump utility:

- http://www.postgresql.org/docs/9.3/static/app-pgdump.html
- http://www.commandprompt.com/blogs/joshua_drake/2010/07/a_better_backup_with_postgresql_using_pg_dump/
- http://www.postgresonline.com/special_feature.php?sf_name=postgresql83_pg_dumprestore_cheatsheet

A logical backup of all PostgreSQL databases

To backup all databases, you can run the individual pg_dump command for each database sequentially or in parallel if you want to speed up the backup process:

- First, from the psql client, use the \l command to list all the available databases in your cluster.
- Second, back up each individual database using the pg_dump command, as described in the previous recipe.

The other approach is to use the pg_dumpall tool to back up all the databases in one single go.

How to do it...

You can use the pg_dump command to back up each database in the server; however, pg_dump does not dump information about the role definition and tablespaces. To dump the global information, use the following command:

```
pg_dumpall -g
```

To back up all the databases in one go, you can use the pg_dumpall utility, as follows, in Windows:

```
pg_dumpall -U postgres > c:\pgbackup\all.sql
```

Similarly, to back up all the databases in one go in Linux, use the pg_dumpall command, as follows:

```
pg_dumpall -U postgres >   /home/pgbackup/all.sql
```

How it works...

The pg_dumpall command exports all the databases, one after another, into a single script file that prevents you from performing a parallel restore. If you are going to back up all the databases this way, the restore process will take a lot more time.

The processing time of dumping all databases takes longer than the time required to dump each database individually, so we don't know which dump of each database relates to a specific point-in-time.

For this reason, you should use the pg_dump command to dump each individual database and then use the -g switch of the pg_dumpall command to keep a backup of all the user and group data.

The pg_dumpall command generally requires the user executing the script to be a PostgreSQL superuser. This is because the pg_dumpall command requires access to the PostgreSQL system catalogs, as it dumps global objects as well as the database objects.

There's more...

Sometimes, you want to back up only the database object definitions, so that you can restore the schema only. This is useful for comparing what is stored in the database against the definitions in a data or object modeling tool.

It is also useful to make sure that you can recreate objects in exactly the correct schema, tablespace, and database with the correct ownership and permissions.

To back up all object definitions in all the databases, including roles, tablespaces, databases, schemas, tables, indexes, triggers, functions, constraints, views, ownership, and privileges, you can use the following command in Windows:

```
pg_dumpall --schema-only > c:\pgdump\definitiononly.sql
```

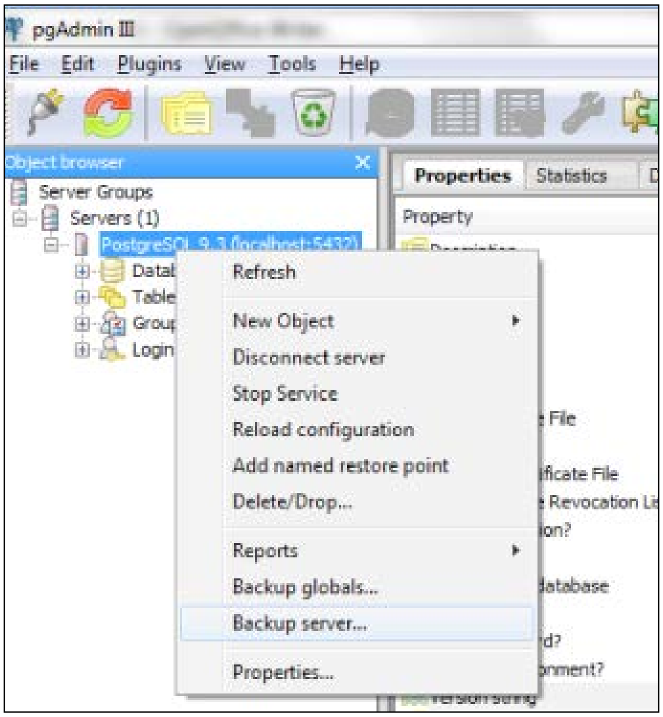
If you want to back up the role definition only, use the following command:

```
pg_dumpall --roles-only > c:\pgdump\myroles.sql
```

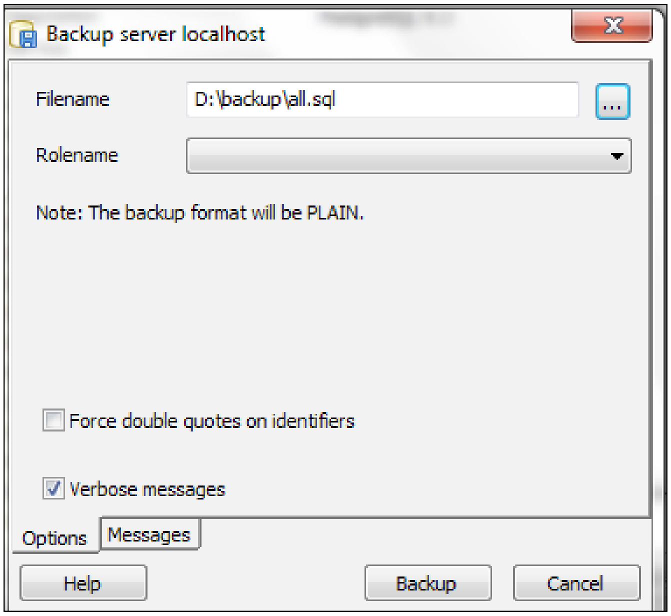
If you want to back up tablespace definitions, use the following command:

```
pg_dumpall --tablespaces-only > c:\pgdump\mytablespaces.sql
```

You can also use the pgAdmin tool to backup all the databases on the server, including the roles, users, groups, and tablespaces. This option can be selected by launching the pgAdmin tool, expanding the **Servers** menu option in the left pane, right clicking on the **PostgreSQL 9.3** option, and then selecting the **Backup server...** option, as shown in the following screenshot:



After the **Backup server...** option is clicked on, a dialog box will open, as shown in the next screenshot, and it will ask you to name the logical dump file that will contain the definition of all the objects and databases that are being backed up. Here, `all.sql` is the name of the logical dump file that can be used later on to restore data and object definitions in all the databases in the event of data loss due to hardware or disk issues.



Backup server localhost

Filename: D:\backup\all.sql

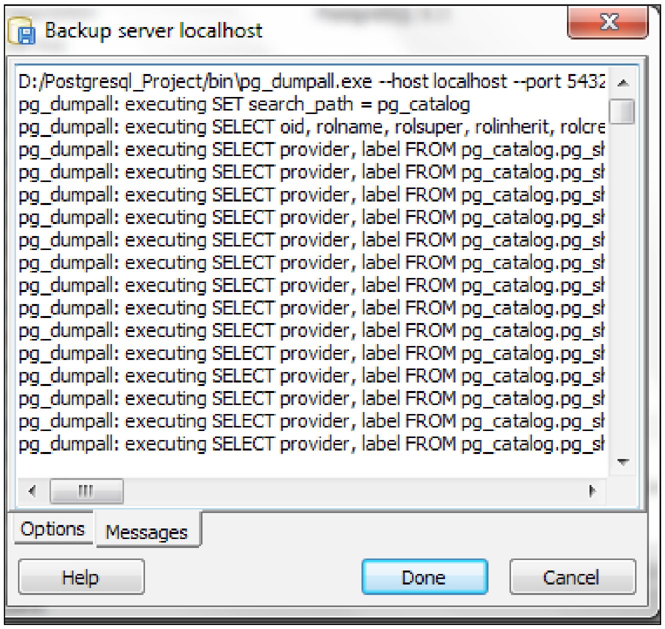
Rolename:

Note: The backup format will be PLAIN.

□ Force double quotes on identifiers

☑ Verbose messages

Once you click on the **Backup** button, as shown in the previous screenshot, a logical backup of all the databases will start. This is equivalent to issuing the `pg_dumpall` command, and the effect of this can be seen in the next screenshot, which opens when you click on the **Backup** Button:



Backup server localhost

```
D:/Postgresql_Project/bin\pg_dumpall.exe --host localhost --port 5432
pg_dumpall: executing SET search_path = pg_catalog
pg_dumpall: executing SELECT oid, rolname, rolsuper, rolinherit, rolcre
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
pg_dumpall: executing SELECT provider, label FROM pg_catalog.pg_sl
```

A logical backup of specific objects

Sometimes, there are situations where you wish to back up only specific database objects, such as tables. The pg_dump utility provides options to back up specific database objects, such as tables.

How to do it...

If you wish to back up some specific tables in a certain schema, you can use the pg_dump command, as follows:

```
pg_dump -h localhost -p 5432 -U agovil -F c -b -v -f
"C:\pgbak\testdb_test.backup" -t case.test postgres
```

In the previous command, we are backing up a table called test, which resides in the case schema in the PostgreSQL database.

How it works...

There are many situations that require you to back up certain tables only. Some of these situations are mentioned here:

- You wish to back up all the tables that are part of a particular tablespace. In this situation, it is possible for a tablespace to contain objects from more than one database; hence, you have to identify the databases from which those tables need to be dumped.

  The following procedure allows you to dump all the tables that reside on one tablespace and one database only:

  1. Create a file named table_tablespace_dump.sql, which contains the following SQL command that extracts the list of tables in a tablespace.

     ```
     SELECT 'pg_dump' UNION ALL
     SELECT '-t ' || spcname || '.' || relname FROM pg_class t
     JOIN pg_tablespace ts
     ON reltablespace = ts.oid AND spcname = :TSNAME
     JOIN pg_namespace n ON n.oid = t.relnamespace
     WHERE relkind = 'r'
     UNION ALL
     SELECT '-F c > dumpfile'; -- dumpfile is the name of the
     output file
     ```

2. Execute the following to build the pg_dump script:

   ```
   psql -t -v TSNAME="'my_tablespace'" -f
   table_tablespace_dump.sql >  myts
   ```

3. From the database server, dump the tables in the tablespace, including the data and definitions:

   ```
   chmod 700 myts
   ```

   ```
   ./myts
   ```

- Another situation is where there are multiple schemas that have similarly named important tables. In this situation, you wish to back up the tables having the same names from different schemas.

  The following scenario is where you can back up named tables from different schemas. You can use the following query, which generates a pg_dump command to back up any table that is not present in public, or pg_catalog that has the rent keyword as a suffix as a part of the table name:

  ```
  SELECT 'pg_dump ' || ' -h localhost -p 5432 -U postgres -F c -b
    -v -f "/pgbak/dvdrental_keytbls.backup" ' ||
    array_to_string(ARRAY(SELECT '-t ' || table_schema || '.' ||
    table_name FROM information_schema.tables
  WHERE table_name LIKE '%_rent' AND table_schema NOT IN
  ('pg_catalog','public' ) ), ' ') || 'dvdrental';
  ```

- Another situation is where you want to back up recently changed tables in the database.

  We are going to utilize vacuum statistics for this. The vacuum statistics uses the assumption that the vacuum process will try to go around the database and vacuum tables where enough data has changed since the last vacuum run. This mechanism will only work if autovacuum has been enabled. The following query will generate a pg_dump command to back up all the tables in the dvdrental database that has been autoanalyzed in the past one day:

  ```
  SELECT 'pg_dump ' || ' -h localhost -p 5432 -U postgres -F c -b
    -v -f "C:/pgbak/dvdrental_changed_keytbls.backup" ' ||
    array_to_string(ARRAY(SELECT '-t ' || schemaname || '.' ||
  relname
  FROM pg_  stat_user_tables
  WHERE last_autoanalyze > ( CURRENT_TIMESTAMP - (INTERVAL '1
    day') ) ) , ' ') || 'dvdrental';
  ```

File system level backup

Another possible option to backup is to utilize the operating system commands and make a file system backup of the files that PostgreSQL uses to store the data in the database.

How to do it...

The easiest way to do this is to make an archive of the PostgreSQL data directory or the directory defined by the `$PGDATA` environment variable, as follows:

```
tar -cvf backup.tar /home/abcd/psql/data
```

Here, we have created an archive file named `backup.tar` that contains a backup of the data directory.

How it works...

The primary benefit of making a file system backup is that this procedure is simple and straightforward. You need to simply back up the data directory with any of the available Unix backup utilities, such as tar, which creates an archive file that can be further used for restoration if the database crashes.

There are, however, a couple of restrictions that use the preceding method to make an archive of the data directory:

- The database must be shut down completely in order to get a useful backup. A file system backup is meaningful only when the database is in a consistent state. For this reason, you need to shut down the database, and as a result, all the data files will be in sync and in a consistent state and that is when the file system backup should be taken.
- With a file system backup, it is not feasible to back up specific databases or individual tables. The entire data directory must be backed up for a complete restoration of the file system. This is due to the reason that many files are associated with a specific database and it becomes difficult to correlate which files belong to which database.

There's more...

Here, we will talk about how to take backups for PostgreSQL using an **LVM** (short for **logical volume manager**) snapshot. This involves taking a frozen snapshot of the volume containing the database, then making a copy of the database directory from the snapshot to a backup device, and finally releasing the frozen snapshot. This will work even when the database server is running. Before you proceed to take the snapshot, you must perform a `CHECKPOINT` command in PostgreSQL via which you can ensure that the backup will be consistent until the time of the `CHECKPOINT`.

Before we begin, we assume that we have a formatted and active XFS file system. We also assume that `VG_POSTGRES/RV_DATA` is our primary data volume.

A root user has to perform the following steps in order to create and use an LVM snapshot:

1. First, issue the `CHECKPOINT` command as the superuser on the PostgreSQL database, as follows:

   ```
   postgres=# CHECKPOINT  ;
   ```

2. The rest of the commands mentioned here need to be executed by the root user. In this step, we will create a snapshot, as follows:

   ```
   lvcreate -l 100%FREE -s -n snap VG_POSTGRES/RV_DATA
   ```

3. Create the directory on which you wish to mount the snapshot, as follows:

   ```
   mkdir /mnt/pg_snap
   ```

4. The next step will be to mount the snapshot as an XFS file system, as follows:

   ```
   mount -t xfs -o nouuid /dev/VG_POSTGRES/snap /mnt/pg_snap
   ```

5. Enter the snapshot directory, as shown here:

   ```
   cd /mnt/pg_snap/
   ```

6. Next, back up the snapshot using the following command:

   ```
   tar –czvf /backup/ pgsql.$(date +"%m-%d%-%Y).tar.gz  /mnt/pg_snap/
   ```

7. Once the snapshot backup has been moved to a different server, the next step will be to unmount it and delete it from the source server as the root user:

   ```
   umount /mnt/pg_snap
   lvremove VG_POSTGRES/snap
   ```

An alternative approach is to initiate a consistent use of a snapshot of the data directory. This involves taking a frozen snapshot of the volume that contains the database, making a copy of the database directory from the snapshot to a backup device, and then releasing the frozen snapshot. This will work even when the database server is running. Before you proceed to take the snapshot, you must perform a `CHECKPOINT` command in PostgreSQL via which you can ensure that the backup would be consistent until the time of the `CHECKPOINT`.

Taking a base backup

The `pg_basebackup` tool takes base backups of a running PostgreSQL database server. These backups are initiated without affecting other PostgreSQL database clients and can be used for both point-in-time recovery, as well as the start point for log shipping or to stream replication standby servers.

How to do it...

You can use the pg_basebackup command in the following manner:

```
$ pg_basebackup -h 192.168.10.14 -D /home/abcd/pgsql/data
```

Here, we take a base backup of the server located at 192.168.10.14 and store it in the /home/abcd/pgsql/data local directory.

How it works...

The pg_basebackup tool makes a binary copy of the database cluster files while ensuring that the system is put in and out of the backup mode automatically. Note that backups are always made up of the entire database cluster. It is not possible to back up individual databases or databases with the pg_basebackup tool.

The pg_basebackup utility initiates a backup that is made over a regular PostgreSQL database connection and utilizes the replication protocol for this purpose. The connection must be made either as a superuser or a user having a REPLICATION privilege. The server must also be configured with enough max_wal_senders to leave at least one session available for backup.

Hot physical backup and continuous archiving

In this recipe, we are going to talk about taking a hot physical backup with continuous archiving in place. A **hot physical backup** is an online backup that is taken while the transactions are running against the database. Even though we have the relevant online physical backup through which we can restore the database, it can restore data only until the time of backup. Any subsequent transactions that may have been recorded in the database after the backup got completed will be missed out. In order to be able to restore the database up to its current state, we will need to apply the archives generated after the backup got completed. For this reason, we need to have continuous archiving enabled.

How to do it...

There are a series of steps that need to be carried out in order to have a hot physical backup and continuous archiving in place:

1. The first step is to enable a continuous **write-ahead log** (**WAL**) archiving. This can be done by making the following parameter changes in the `postgresql.conf` configuration file, which resides in the `data` directory, that is the directory defined by the `$PGDATA` environment variable:

   ```
   wal_level = hot_standby
   archive_mode = on
   archive_command = 'test ! -f
   /home/abcd/pgsql/backup_in_progress || (test ! -f
   /home/abcd/pgsql/archive/%f && cp %p
   /home/abcd/pgsql/archive/%f)'
   ```

2. Once these changes have been implemented, you need to bounce the PostgreSQL server for the changes made to the above mentioned parameters to come into effect:

   ```
   pg_ctl -d $PGDATA stop
   pg_ctl -d $PGDATA start
   ```

3. The next step will be to create the `archive` directory, as follows:

   ```
   mkdir -p /home/abcd/pgsql/archive/
   touch /home/abcd/pgsql/backup_in_progress
   ```

4. Once this is done, the next step will be to start the backup process using the following command:

   ```
   psql -c "select pg_start_backup('hot_backup');"
   ```

5. Next, perform a file system backup of the PostgreSQL data directory, as follows:

   ```
   tar -cvf /home/abcd/pgsql/backup.tar /home/abcd/pgsql/data
   ```

6. When the file system backup is completed, the next step will be to connect to the database and stop the backup process using the following command:

   ```
   psql -c "select pg_stop_backup();"
   ```

7. Now that the backup has been completed, the next step will be to go the `archive` location and confirm the archives that were generated. These archives, along with the base backup, will help you recover data to the last checkpoint or any point-in-time after the base backup happened and to the point where you have the archive logs:

   ```
   cd /home/abcd/pgsql/archive
   [postgres@localhost archive]$ ls -ltrh
   total 49M
   ```

```
-rw------- 1 postgres postgres 16M Jun 30 23:53
000000010000000000000009

-rw------- 1 postgres postgres 16M Jun 30 23:53
000000010000000000000008

-rw------- 1 postgres postgres 294 Jun 30 23:54
00000001000000000000000A.00000024.backup

-rw------- 1 postgres postgres 16M Jun 30 23:54
00000001000000000000000A
```

How it works...

A physical backup or base backup takes a copy of all the files in the database or the data directory; however, this alone is not sufficient as a backup and there are other steps that need to be performed as well. A simple file system backup of the database, while the server is running, produces a time-inconsistent copy of the database files. However, in the current context, production databases need to be available 24/7, and to take backups, it is not possible to bring down the database every time and then take a file system backup. Such a strategy is not feasible. Moreover, the backup process should ensure that all the changes made from the time the backup starts until the time it ends are tracked and recorded. These changes are tracked and recorded in WAL logs, and once the changes recorded in WAL logs are archived, the WAL logs can be used later.

Now that it has become a business need to take online backups for production databases, you need to ensure that the backups that are taken online are consistent. To make the backup consistent, you need to add to it all the changes that took place from the start to the finish of the backup process. That's why we have steps 4 and 6 to bracket our backup step.

The changes that are made are put into the archive directory as a set of archived transaction log/WAL files. In step 3, we have created the `archive` directory. Enabling the archive mode, as mentioned in step 1, requires a database restart and this was done in step 2. In step 3, you can also see that we have created a file named `backup_in_progress`. The presence of this file enables or disables the archiving process.

Point-in-time recovery

Many a times, DBAs will encounter situations where they might need to restore the database from an existing backup. This might be due to a business requirement or a critical table might have been dropped, or else the hard disk on which the database was mounted crashed and became corrupt. For whatever reason, you might have to go for a database recovery scenario. In this recipe, we are going to discuss the steps required to recover the database in the event of a failure and how to use the archive logs to do a point-in-time recovery.

How to do it...

There are a series of steps that need to be carried out if you need to recover a database from the backup:

1. First, check the status of the database server. If the server is running, then stop the server, using the following command:

   ```
   pg_ctl -d $PGDATA stop
   ```

2. Next, copy the existing data directory and any existing tablespaces to a temporary location, if anything is required from the existing structure later on. In case there is a space crunch, you should at least consider keeping a copy of the content of the `pg_xlog` subdirectory. This is essential because `pg_xlog` might contain logs that were not archived before the system went down:

   ```
   mv $PGDATA /tmp
   ```

3. Next, restore the database files from your file system backup, which was taken earlier. Please ensure that the restoration is done with the right ownership and right permissions. If you are using tablespaces, you must confirm whether the symbolic links in the `pg_tblspc` directory are correctly restored:

   ```
   tar  -xvf /home/abcd/pgsql/backup.tar
   ```

4. Remove any existing files from the `pg_xlog` directory, as these appear to have come from the file system backup and are most probably obsolete, rather than being current. If `pg_xlog/` was not archived earlier, then you need to recreate it with proper permissions, being careful to ensure that you re-establish it as a symbolic link, if it was set up in that manner before:

   ```
   rm -rf  /home/abcd/pgsql/data/pg_xlog/*
   ```

   > If there were any unarchived WAL segment files that were saved in step 2, then you need to copy them to the `pg_xlog/` location.

5. The next step will be to configure the `recovery.conf` file in the data directory. You can copy the `recovery.conf.sample` file from the `share` directory, which is located under the installation directory of `postgres`, and once you have copied the `recovery.conf.sample` file to the data directory, you will need to rename it as the `recovery.conf` file:

   ```
   cp /home/abcd/pgsql/share/recovery.conf.sample $PGDATA
   cd $PGDATA
   mv recovery.conf.sample recovery.conf
   ```

6. The only parameter that needs to be configured in the `recovery.conf` file is the `restore_command` parameter. This parameter tells PostgreSQL how to retrieve archived WAL file segments:

   ```
   restore_command = 'cp /home/abcd/pgsql/archive/%f %p'
   ```

7. Once this is done, you are ready to start the server. The server launches into the recovery mode and will proceed through the archived WAL files it needs. Once the recovery is complete, the server will rename `recovery.conf` to `recovery.done`, and with this, the database stands recovered and you are ready to launch normal operations against the database:

   ```
   pg_ctl -D $PGDATA start
   ```

The following is an excerpt from the log:

```
LOG:  starting archive recovery
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

You can also see that the `recovery.conf` file is now renamed to `recovery.done` once the recovery is completed:

```
[postgres@localhost data]$ ls -ltrh |tail -7
-rw-r--r-- 1 postgres postgres 4.7K Jul  4 04:13 recovery.done
-rw------- 1 postgres postgres   78 Jul  4 04:14 postmaster.pid
-rw------- 1 postgres postgres   59 Jul  4 04:15 postmaster.opts
drwx------ 2 postgres postgres 4.0K Jul  4 04:15 pg_notify
drwx------ 2 postgres postgres 4.0K Jul  4 04:15 global
drwx------ 3 postgres postgres 4.0K Jul  4 04:15 pg_xlog
drwx------ 2 postgres postgres 4.0K Jul  4 04:24 pg_stat_tmp
```

How it works...

A point-in-time recovery works in this way.

First, you need to restore the `data` directory from the backup file. The database at this stage is in an inconsistent state because it is restored at the time of backup. It still needs to account for any changes made by transactions that were persistent during the backup. For this, you need to apply archives. Setting the `restore_command` value in step 6, in the *How to do it...* section, ensures that the changes recorded in the archived WAL segments are applied to the database. Once the server is started, it launches into recovery mode in order to restore all the data properly. After a few minutes, the database will be successfully restored to the last checkpoint that the archived logs recorded.

There's more...

If the business requirement is to restore the database to a previous point-in-time, then we will need to specify the required stop point in the `recovery.conf` configuration file. The stop point, also referred to as the recovery target, can be specified in terms of data and time, a name restoration point, or by the completion of a specific transaction ID.

The parameters that can control a point-in-time recovery in the `recovery.conf` file are `recovery_target_time` and `recovery_target_xid`. Either of these options might be configured to lead a previous point-in-time recovery.

Restoring databases and specific database objects

In this recipe, we are going to talk about how to restore a single database, all databases, and specific objects, such as tables.

How to do it...

Here, we are going to talk about three scenarios: what needs to be followed when you need to restore all the databases on the server, a specific database, or a specific table only. We will cover these scenarios in a series of steps, shown as follows:

1. Restoring all databases: In the *A logical backup of all PostgreSQL databases* recipe, we created a logical dump for all the databases on a server. The dump file, `all.sql`, will be used here to restore all the databases on the server, assuming that the database files are corrupted and the server has crashed. The command to restore all the databases is given here:

   ```
   psql -U postgres -f c:\pgbackup\all.sql
   ```

2. Restoring a single database: In the *A logical backup of a single PostgreSQL database* recipe, we created a backup using the `pg_dump` tool and the dump was named `dvdrental.tar`.

   Now, we are going to restore the `dvdrental` database by assuming that it has already been dropped, as follows:

   ```
   pg_restore --dbname=dvdrental --create --verbose
   /home/abcd/dvdrental.tar
   ```

   The `--create` option of the `pg_restore` command first creates an empty database before restoring all the objects from the `dvdrental.tar` dump file.

3. Restoring a single table: We can also restore individual objects such as tables. Here, we are going to drop a table named store and then recover the table using the logical dump that was used to restore the dvdrental database in the previous step:

    1. First, drop the store table:

       ```
       dvdrental#drop table store cascade;
       ```

    2. In the next step, we are going to extract the table definition from the available dump and then dump the table, its definition, as well as the data of the dropped table in a new file:

       ```
       pg_restore -t store dvdrental.tar > droppedtable.sql
       ```

       Here, the droppedtable.sql file contains the table definitions, along with data that is necessary to restore the store table that was dropped in the earlier step.

    3. The final step will be to use the newly created file that contains the table definition and data to restore the table into the dvdrental database:

       ```
       psql -f droppedtable.sql dvdrental
       ```

How it works...

The pg_restore utility enables you to restore databases that have been backed up by either pg_dump or pg_dumpall. It is a utility for restoring PostgreSQL databases from an archive created by pg_dump in one of the non-plain-text formats.

The pg_restore utility will issue the commands necessary to restore the database to the state it was in at the time of being saved. The archive files allow pg_restore to be selective about what is restored.

There's more...

To speed up the restore process, it is possible to perform parallel restore operations in PostgreSQL. The -j switch is used to specify the number of threads required for restoration. Each thread restores a separate table simultaneously, which speeds up the restore process.

4

Routine Maintenance Tasks

In this chapter, we will cover the following recipes:

- Controlling automatic database maintenance
- Preventing auto freeze and page corruption
- Preventing transaction ID wraparound failures
- Updating planner statistics
- Dealing with bloating tables and indexes
- Monitoring data and index pages
- Routine reindexing
- Maintaining log files

Introduction

It is important to carry out regular maintenance operations at scheduled intervals for a PostgreSQL database to achieve optimal performance. Heavy database transactions can leave behind a significant amount of data, which can lead to a drop in database performance. Thus, a database administrator needs to carry out maintenance operations in order to clean up the database and improve database performance. In this chapter, we are going to discuss how to deal with bloating tables and indexes, transaction ID wraparound failures, and maintenance tasks, such as vacuuming.

Controlling automatic database maintenance

PostgreSQL has a feature known as **autovacuum**, which although optional, is enabled by default in the major PostgreSQL release versions, starting with PostgreSQL 9.0 onwards. The job of the autovacuum daemon is to automate the execution of the VACUUM and ANALYZE commands and to perform these maintenance tasks.

How to do it...

Even though autovacuum is enabled by default in PostgreSQL, you need to ensure that autovacuum is switched on. Enabling the autovacuum daemon requires you to configure and enable the following parameters in the postgresql.conf configuration file:

```
autovacuum = on
track_counts = on
```

As the name suggests, the autovacuum parameter controls whether the server should launch the autovacuum daemon.

The track_counts parameter enables statistics collection on database activity. Usually, this parameter is enabled by default because most of the checks that autovacuum performs require the usage of a statistics collection, and unless the statistics collection facility is enabled, autovacuum cannot be used.

The prior mentioned setting of autovacuum enablement happens on a global level, as it is defined in the postgresql.conf configuration file. It is also possible to enable autovacuum at the table level, as follows:

```
ALTER TABLE hrms SET (
  autovacuum_enabled = TRUE, toast.autovacuum_enabled = TRUE
);
```

Here we enabled autovacuum for TOAST tables as well. Usually, long data values are placed in a secondary table known as the TOAST table. Hence, for each actual table, there will be a corresponding toast table that contains long data values, and thereby a corresponding toast index will be defined as well.

How it works...

Initially, autovacuum checks for tables that are eligible candidates for vacuuming. It does this by checking for tables that have a large number of inserted, updated, or deleted rows; that is, fragmented rows. Once autovacuum has figured out the fragmented tables, all the autovacuum workers are assigned the task of vacuuming the fragmented tables.

Having earlier discussed the job that autovacuum performs, let's now discuss the autovacuum process itself. The autovacuum daemon basically consists of multiple processes. There is a persistent daemon process known as the **autovacuum launcher**, whose job is to start the autovacuum worker for all the databases residing on the PostgreSQL server. The autovacuum launcher will attempt to start one worker within each database after the value specified, in seconds, in the `autovacuum_naptime` parameter has elapsed; the launcher will distribute the work accordingly to each worker. The job of the worker process is to find fragmented tables in its database and execute the `VACUUM` or `ANALYZE` commands as and when needed automatically.

For more information on automatic vacuuming, visit `http://www.postgresql.org/docs/9.3/static/runtime-config-autovacuum.html`.

There's more...

There are a good number of tunable autovacuum related parameters that control the behavior of the autovacuum feature. Some of these parameters are discussed as follows:

- `log_autovacuum_min_duration`: This parameter helps to monitor autovacuuming activity. This parameter specifies that each action that is executed by autovacuum is logged if it executed for at least the time specified in milliseconds in this parameter.
- `autovacuum_max_workers`: This parameter states the maximum number of worker processes that might be executing at any particular time. The autovacuum launcher process is an exception to this; hence, it is not included or counted in the list of vacuum max workers.
- `autovacuum_vacuum_threshold`: This parameter specifies the number of updated and deleted rows to initiate `VACUUM` in the associated table.
- `autovacuum_analyze_threshold`: This parameter specifies the number of updated and deleted rows to initiate `ANALYZE` in the associated table.
- `autovacuum_vacuum_scale_factor`: The value of this parameter specifies the fraction of the table size that needs to be added to `autovacuum_vacuum_threshold` when deciding whether to trigger a `VACUUM`.
- `autovacuum_analyze_scale_factor`: The value of this parameter specifies the fraction of the table size that needs to be added to `autovacuum_vacuum_threshold` when deciding whether to trigger an `ANALYZE`.

- autovacuum_freeze_max_age: The value of this parameter specifies the maximum value that the table's pg_class.relfrozenxid field can attain before the VACUUM operation is forced to prevent a transaction ID (XID) wraparound within the table. This parameter puts a limit on how far autovacuum will let you go before it starts to kick in and goes around exhaustively vacuum freezing the old XIDs in your tables with old rows. It is a not a good idea to hit the threshold set by the value of this parameter because it can generate a lot of I/O umpteen times, causing performance issues, and freeze autovacuums are not cancellable.
- autovacuum_vacuum_cost_delay: This parameter defines the cost delay value that is to be used in the VACUUM operations.

Preventing auto freeze and page corruption

In an OLTP environment, we usually expect and normally see that there are lots of DML operations on tables. Because of frequent DML operations on tables, we can see rows that have been deleted or have become obsolete due to an update operation; however, they haven't been physically removed from their tables. Such rows are referred to as **dead rows**.

It is also quite possible that a row version might become old enough for it to become a candidate for being frozen. Such rows are referred to as **frozen rows**.

Vacuuming deals with both dead rows, by reclaiming space from dead rows, and old row versions, by freezing them so that they are preserved until they are deleted.

How to do it...

Freezing occurs when the XID, that is the transaction identifier, on a row becomes more than the vacuum_freeze__min_age transactions older than the next current value. To ensure that all old transaction identifiers have been replaced by FrozenXID, a table scan is performed. The vacuum_freeze_table_age parameter controls when a scan on the whole table is performed. Setting the value of the vacuum_freeze_table_age parameter to zero forces VACUUM to always scan all the pages. Scanning all the pages block by block for the entire database while VACUUM is being run is also an effective way to confirm the absence of page corruptions. This can be initiated on the database level as follows:

```
SET vacuum_freeze_table_age = 0;
VACUUM;
```

This can be initiated at a table level as follows:

```
VACUUM demo;
```

Here, demo is the name of the table being vacuumed.

How it works...

Vacuuming deals with performing the following functions:

- Reclaiming or reusing disk space occupied by dead rows
- To keep the statistics collection up to date, which is used by the PostgreSQL query planner
- To protect against the loss of old transaction data due to transaction ID wraparound issues, which is discussed in the next recipe

If any page corruptions are detected, then you can use the `pageinspect` utility to examine the contents of the database pages at a low level, which is useful from a debugging perspective. It can also be used to examine index pages.

There are two situations where there might be huge I/O generation during freezing while `VACUUM` is being run:

- When there are many rows with the same transaction identifier during freeze time
- When a table scan is being performed and you encounter a large number of rows that need freezing

Preventing transaction ID wraparound failures

For MVCC, PostgreSQL uses a transaction ID which is 32 bits long.

It is not feasible to have a larger transaction ID because that will increase the size of each row by a significant amount. A 32-bit value can take over four billion transactions; however, it can handle a range of about two billion transactions before rolling over to zero. When this range is exceeded, past transactions will now appear to be from the future. That is, their outputs become invisible; this will result in a catastrophic data loss, and the database will fail to operate in a sane manner.

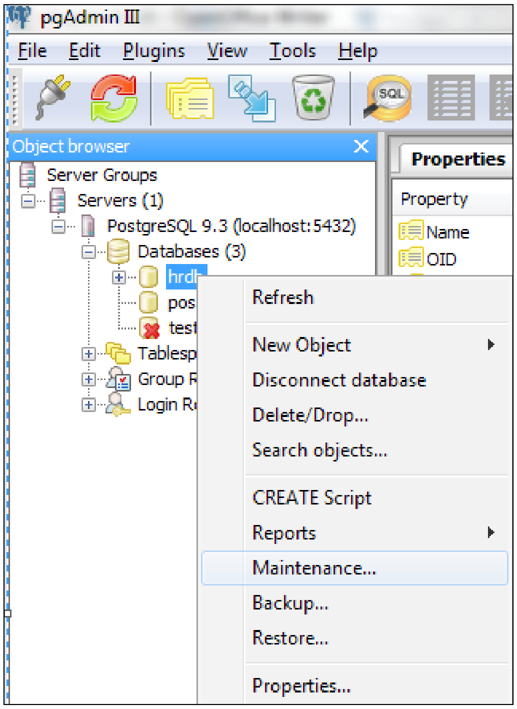
To prevent data loss, old rows must be assigned a transaction ID (XID), FrozenXID (frozen transaction ID), sometime before they reach the two billion transactions mark. Once these rows are assigned a FrozenXID, they will appear to be in the past to all normal transactions regardless of the wraparound issues; so, such rows will be good until they are deleted, no matter how long that is. This reassignment of XID is handled by `VACUUM`.

How to do it...

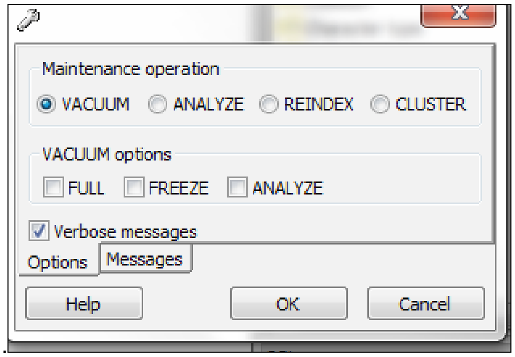
In the previous recipe, you saw the usage of the VACUUM command at the database level as well as at the table level. As an alternative to using the VACUUM command, you can also use the vacuumdb utility to clean the PostgreSQL database. Like VACUUM, the vacuumdb utility will also generate statistics to be used by the query optimizer. The vacuumdb utility is just a wrapper around the VACUUM command. We can use the vacuumdb utility to clean the hrdb database, as follows:

```
$ vacuumdb hrdb
```

You can also use the pgAdmin tool to vacuum a database. To vacuum a database in pgAdmin, under **Object browser** in the left-hand side pane, right-click on the specific database under the **Databases** menu, and click on **Maintenance...**, as shown in the following screenshot:



This will open a dialog box where you need to select the **VACUUM** radio button and then click on **OK** in order to vacuum a database, as shown here:



Maintenance operation

● VACUUM ○ ANALYZE ○ REINDEX ○ CLUSTER

VACUUM options

□ FULL □ FREEZE □ ANALYZE

☑ Verbose messages

Updating planner statistics

In order to generate a good plan for the queries, the PostgreSQL query planner relies on available statistical information about the contents of the tables. It is, therefore, essential to ensure that the statistics are accurate and up to date. If the statistics are stale, then it will result in a poor plan being generated for the queries, which will end up further degrading database performance.

How to do it...

There are two ways in which statistics can be gathered:

- You can run the ANALYZE command to generate statistics on tables.
- The ANALYZE command can be invoked as an optional step while using VACUUM. If autovacuum is enabled, it will automatically invoke the ANALYZE command when the contents of a table have changed substantially.

Details about the autovacuum daemon and the VACUUM command have already been covered in the first two recipes, so we are now going to focus on the ANALYZE command here:

```
ANALYZE demo;
```

In the preceding case, we used the ANALYZE command to generate statistics on the table named demo and stored the results in the pg_statistic system catalog table.

Statistics collection is fine-grained and can even be done at the column level.

How it works...

The statistics that are collected by the ANALYZE command include some of the most common values in each column along with a histogram that depicts the approximate data distribution in each column.

For larger tables, instead of examining each row, ANALYZE takes a random sample of table contents. The benefit of using this approach is that by using the random sampling method, even larger tables can be analyzed in a short span of time.

The amount of statistics collected by ANALYZE can be controlled by adjusting the value of the default_statistics_target configuration parameter.

The ANALYZE command acquires a read-only lock on the target table. This way, it can be run in parallel with any other activity on the target table.

Dealing with bloating tables and indexes

It is common to find a database where vacuuming has been turned off for either a table or for the entire database. The reason for turning off vacuuming is that vacuuming creates too much disk I/O. This might help temporarily, but in the longer run, it is not recommended that you turn off vacuuming or abandon it. On the other hand, if vacuuming is performed too frequently, the system's performance can become slow because vacuuming creates a lot I/O traffic.

If the database has been maintained without vacuuming or if the data is badly structured, we might experience bloating tables and indexes. The problem with bloating tables and indexes is that they occupy more storage space than required, which often causes performance issues when these are used by queries. In this recipe, we are going to see how to detect bloating tables and indexes and what the best time is to run a VACUUM command. If there are lots of dead rows in a table, the bloat percentage is higher.

How to do it...

Here, we are going to see when a table become bloated and how to deal with it:

1. First, we are going to activate the pgstattuple module, which is used to detect a table bloat, as follows:

```
hrdb=# create schema stats;
hrdb=# create extension pgstattuple with schema stats;
```

2. Next, we are going to create a table and add some rows into it:

```
hrdb=# CREATE TABLE num_test AS SELECT *
FROM generate_series(1, 10000);
```

3. Now, we are going to use the pgstattuple function, provided by the pgstattuple extension, to examine row-level statistics for the num_test table:

   ```
   hrdb=# SELECT * FROM stats.pgstattuple('num_test');
   -[ RECORD 1 ]------+-------
   table_len          | 368640
   tuple_count        | 10000
   tuple_len          | 280000
   tuple_percent      | 75.95
   dead_tuple_count   | 0
   dead_tuple_len     | 0
   dead_tuple_percent | 0
   free_space         | 7380
   free_percent       | 2
   ```

4. At this point, we don't see any dead rows, so we will delete some data from the num_test table:

   ```
   hrdb=#  DELETE FROM num_test WHERE generate_series % 2 = 0;
   ```

5. Now we will reuse the pgstattuple module to examine the table bloat in the num_test table:

   ```
   hrdb=# SELECT * FROM stats.pgstattuple('num_test');
   -[ RECORD 1 ]------+-------
   table_len          | 368640
   tuple_count        | 5000
   tuple_len          | 140000
   tuple_percent      | 37.98
   dead_tuple_count   | 5000
   dead_tuple_len     | 140000
   dead_tuple_percent | 37.98
   free_space         | 7380
   free_percent       | 2
   ```

   In this output, you can see that the percentage of dead rows is approximately 38 percent. So, we are now going to vacuum the table in order to remove the table bloat. Also, observe that the percentage of free storage space is around 2 percent, as seen in the free_percent column:

   ```
   hrdb=# VACUUM num_test;
   ```

6. Now that the table has been vacuumed, we will reexamine the row-level statistics for the num_test table:

   ```
   hrdb=# SELECT * FROM stats.pgstattuple('num_test');
   -[ RECORD 1 ]------+-------
   table_len          | 368640
   tuple_count        | 5000
   ```

```
tuple_len          | 140000
tuple_percent      | 37.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 167380
free_percent       | 45.4
```

In the preceding output, you can see that the `dead_tuple_percent` column value is zero, which means there are no dead rows. Also, the storage space has increased; we can now see that the free storage space is around 45 percent, which shows that more storage space has become available after vacuuming. Before vacuuming, the free space percent was around 2 percent. Thus, with vacuuming, we have successfully managed to remove the bloat that existed in the tables.

Now that we have spoken about bloating tables, let's move toward bloating indexes. The following query can help identify whether there are any bloating indexes for a particular table:

```
hrdb=# SELECT relname, pg_table_size(oid) as index_size,
  100-(stats.pgstatindex(relname)).avg_leaf_density AS bloat_ratio
FROM pg_class WHERE relname ~ 'casedemo' AND relkind = 'i';
                      relname                      | index_size |
bloat_ratio
---------------------------------------------------+------------+----
---------
 casedemo_inventory_id_idx                         |     507904 |
34.11
 casedemo_rental_date_inventory_id_customer_id_idx |     630784 |
26.14
 casedemo_pkey                                     |     376832 |
10.25
(3 rows)
```

In the preceding output, you can see the index bloat ratio for all of the indexes belonging to the `casedemo` table.

How it works...

PostgreSQL has a feature known as MVCC, that is Multi Version Concurrency Control that allows you to read data at the same time as writers make changes. Due to the MVCC feature being implemented, we often encounter situations where the `UPDATE` command can cause tables and indexes to grow in size because they leave behind dead row versions. Similarly, the `DELETE` and `INSERT` operations take up space that must be reclaimed by vacuuming. Also, some deletion patterns can cause large chunks of the index to be filled with empty entries, which creates a bloating-index scenario. To overcome the problem of a bloating index, you need to rebuild indexes. Rebuilding indexes is covered in the *Routine reindexing* recipe further on in the chapter.

Thus, it is important to examine the dead_tuple_len and dead_tuple_percent columns from the pgstattuple package for a given table, and if both of these columns have high values, then it it best to VACUUM these tables at a time when the transaction activity on the database is low so as not to impact database performance.

There's more...

If you want to identify the estimated amount of bloat in your tables and indexes, you can use the following query. This query is based on the check_postgres script available at http://bucardo.org/wiki/Check_postgres:

```
SELECT
  current_database(), schemaname, tablename, /*reltuples::bigint, 
relpages::bigint, otta,*/
  ROUND(CASE WHEN otta=0 THEN 0.0 ELSE sml.relpages/otta::numeric 
END,1) AS tbloat,
  CASE WHEN relpages < otta THEN 0 ELSE bs*(sml.relpages-otta)::bigint 
END AS wastedbytes,
  iname, /*ituples::bigint, ipages::bigint, iotta,*/
  ROUND(CASE WHEN iotta=0 OR ipages=0 THEN 0.0 ELSE ipages/
iotta::numeric END,1) AS ibloat,
  CASE WHEN ipages < iotta THEN 0 ELSE bs*(ipages-iotta) END AS 
wastedibytes
FROM (
  SELECT
    schemaname, tablename, cc.reltuples, cc.relpages, bs,
    CEIL((cc.reltuples*((datahdr+ma-
      (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma 
END))+nullhdr2+4))/(bs-20::float)) AS otta,
    COALESCE(c2.relname,'?') AS iname, COALESCE(c2.reltuples,0) AS 
ituples, COALESCE(c2.relpages,0) AS ipages,
    COALESCE(CEIL((c2.reltuples*(datahdr-12))/(bs-20::float)),0) AS 
iotta -- very rough approximation, assumes all cols
  FROM (
    SELECT
      ma,bs,schemaname,tablename,
      (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%ma 
END)))::numeric AS datahdr,
      (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma ELSE 
nullhdr%ma END))) AS nullhdr2
    FROM (
      SELECT
        schemaname, tablename, hdr, ma, bs,
        SUM((1-null_frac)*avg_width) AS datawidth,
        MAX(null_frac) AS maxfracsum,
        hdr+(
          SELECT 1+count(*)/8
          FROM pg_stats s2
          WHERE null_frac<>0 AND s2.schemaname = s.schemaname AND 
s2.tablename = s.tablename
        ) AS nullhdr
      FROM pg_stats s, (
        SELECT
          (SELECT current_setting('block_size')::numeric) AS bs,
          CASE WHEN substring(v,12,3) IN ('8.0','8.1','8.2') THEN 27 
ELSE 23 END AS hdr,
          CASE WHEN v ~ 'mingw32' THEN 8 ELSE 4 END AS ma
        FROM (SELECT version() AS v) AS foo
      ) AS constants
      GROUP BY 1,2,3,4,5
    ) AS foo
  ) AS rs
  JOIN pg_class cc ON cc.relname = rs.tablename
  JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.nspname = 
rs.schemaname AND nn.nspname <> 'information_schema'
  LEFT JOIN pg_index i ON indrelid = cc.oid
  LEFT JOIN pg_class c2 ON c2.oid = i.indexrelid
) AS sml
ORDER BY wastedbytes DESC;
```

Monitoring data and index pages

In the earlier recipes, you saw that frequent updates to data result in dead rows across both tables and indexes. These dead rows consume storage space; hence, it is important to monitor tables and indexes in order to identify the amount of bloat present in these objects.

Apart from bloating, there are other aspects of a table and index that need to be monitored. For instance, if there are any unused indexes, then they should be identified and removed. Hence, you need to monitor for unused indexes too.

How to do it...

A DBA usually requires some statistical information about the tables that is stored in the PostgreSQL database, as follows:

- Information on the total count of the number of rows in the table and table length
- Information on the number of dead rows and the dead row percentage on a given table

- Information on the amount and percentage of free space available in the table
- Information regarding the number of update, insert, and delete operations on the table
- Information regarding the last time the table was vacuumed, manually or by the autovacuum daemon, and the last time the table was analyzed for statistics collection

The preceding statistical information can be obtained from the `pgstattuple` module, which provides row-level statistics for a given table and the `pg_stat_all_tables` view, which shows statistics about accesses to that specific table.

In the following output, from the `pgstattuple` module, you can see the number of rows, row length, number of dead rows, and the percentage and amount of free space available:

```
hrdb=# SELECT * FROM stats.pgstattuple('casedemo');
-[ RECORD 1 ]------+--------
table_len          | 1228800
tuple_count        | 16044
tuple_len          | 1152240
tuple_percent      | 93.77
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 8184
free_percent       | 0.67
```

You can also use the `pg_stat_all_tables` table to pull up some interesting details, such as the number of rows updated, deleted, and inserted as well as the timestamp of the last time this table was autovacuumed, the timestamp of the last time the table was autoanalyzed, and so on:

```
hrdb=# SELECT schemaname,relname,n_tup_ins,n_tup_upd,n_tup_del,
  n_live_tup,n_dead_tup,last_autovacuum,last_analyze
from pg_stat_all_tables where relname='casedemo';

-[ RECORD 1 ]---+-----------------------------
schemaname      | public
relname         | casedemo
n_tup_ins       | 16044
n_tup_upd       | 0
n_tup_del       | 0
n_live_tup      | 16044
n_dead_tup      | 0
last_autovacuum |
last_analyze    | 2014-07-13 20:21:11.636+05:30
```

Statistical information regarding indexes includes the following:

- Information regarding the space occupied by the indexes and whether there are any bloated indexes.
- Information regarding the number of times the index was used by the query planner.
- Information regarding the number of rows read by the index.
- Information regarding the number of rows fetched by the index.
- Information regarding the leaf fragmentation in the index. **Leaf fragmentation** occurs when rows are deleted, thus creating partially or completely empty blocks in the index binary tree. Because of row deletion, these leaf-level index pages have free space; as a result, the index uses more data pages to store data on disk and in memory, thereby affecting the performance of scan operations even when data pages are cached due to extra pages that need to be processed.

Statistics about indexes can be retrieved from the `pg_stat_user_indexes` and `pg_index` tables. You can use these two tables to find the unused indexes:

```
SELECT
    relid::regclass AS table,
    indexrelid::regclass AS index,
    pg_size_pretty(pg_relation_size(indexrelid::regclass))
AS index_size,
    idx_tup_read,
    idx_tup_fetch,
    idx_scan
FROM pg_stat_user_indexes
  JOIN pg_index USING (indexrelid)
 WHERE idx_scan = 0
AND indisunique IS FALSE;
```

In the preceding query output, the `idx_tup_read`, `idx_tup_fetch`, and `idx_scan` columns indicate the usage of the index:

- `idx_tup_read`: This column indicates the number of rows that have been read using the index
- `idx_tup_fetch`: This column indicates the number of rows that have been fetched using the index
- `idx_scan`: This column indicates the number of times the index was used by the query planner

Routine reindexing

In some scenarios, it is worth rebuilding indexes periodically with the REINDEX command. Indexes can become an issue in database applications that involve a high proportion of repeated inserts and deletes, and this might cause indexes to become bloated. The potential for bloat is not indefinite; that is, at worst there will be one key per page, but it might still be worthwhile to schedule periodic reindexing for indexes that have such usage patterns. With the help of the REINDEX command, index pages that have become completely empty are reclaimed for reuse.

How to do it...

Indexes can be rebuilt at various levels, as follows:

- You can recreate the index at the individual index level, whereby a single index can be rebuilt. You can recreate a single index, as given here:

  ```
  REINDEX INDEX customer_pkey;
  ```

- You can recreate the indexes at the table level, whereby all of the indexes for a given table are rebuilt:

  Here, in the following code, we are rebuiliding all of the indexes for a table customer:

  ```
  REINDEX TABLE CUSTOMER;
  ```

- You can recreate the indexes at the system level, whereby you can recreate all of the indexes on system catalogs within the current database. Here, we are rebuilding all of the indexes on the system catalog for the hrdb database:

  ```
  REINDEX SYSTEM hrdb;
  ```

- You can recreate the indexes at the database level, whereby you recreate all of the indexes within the current database, which is hrdb in the code:

  ```
  REINDEX DATABASE hrdb;
  ```

How it works...

The REINDEX command is used to rebuild an index using the data stored in the index's table, thereby replacing the old copy of the index.

`REINDEX` is used in the following situations:

- `REINDEX` is to be used when an index becomes corrupted and does not contain any valid data. Indexes can become corrupted due to software bugs or hardware failures. `REINDEX` provides a recovery method.
- `REINDEX` is to be used when an index becomes bloated, that is, when it contains many empty pages. `REINDEX` reduces the space consumption of the index by writing a new version of the index without dead pages.
- `REINDEX` needs to be used when a storage parameter for an index has been altered and you wish to ensure that the changes come into effect.
- `REINDEX` locks out write operations on the index's parent table but does not block read operations on the table. `REINDEX` also acquires an exclusive lock on the specific index being processed, which will block reads that attempt to use the index.

There's more...

There is an option through which the `REINDEX` command can rebuild an index without locking out write operations on the index's parent table. For this, you can use the `CREATE INDEX CONCURRENTLY` command, which will build the index without taking any locks that prevent concurrent inserts, updates, and deletes on the table. So, instead of rebuilding the index, you have to perform the following three steps:

1. First, create an index identical to the one you wish to rebuild using the `CREATE INDEX CONCURRENTLY` option.
2. Next, drop the old index.
3. The final step is to rename the new index to the same name as the one that the old index had.

The following code demonstrates the preceding steps:

```
CREATE INDEX CONCURRENTLY card_index ON creditcard (cardno);
BEGIN;
DROP INDEX credit_card_idx;
ALTER INDEX card_index RENAME TO credit_card_idx;
COMMIT;
```

Maintaining log files

The information stored in log files can prove invaluable when diagnosing or troubleshooting problems. With the help of the information stored in the log files, you can identify the sources of the problems in the underlying database. For this very reason, it is important to preserve log files rather than discarding them. However, the information in the log files tends to be voluminous, so it important that a rotation policy be implemented in order to preserve certain log files and to discard log files that are no longer required. Log files need to be rotated so that new log files are started and old ones are removed after a reasonable period of time.

How to do it...

There are various mechanisms through which logging information is maintained and preserved in log files. These are discussed as follows:

- One way to deal with this is to send the server's stderr output to some kind of log rotation program. PostgreSQL has a built-in log rotation facility, which can be used by setting the `logging_collector` configuration parameter in the `postgresql.conf` file:

  ```
  logging_collector=true
  ```

- Another approach is to use an external log rotation program that you might be using with some other server software. For instance, the Apache distribution includes a tool known as rotatelogs that can be used with PostgreSQL. This can be done by piping the stderr output of the server to the desired external program. If you are using the `pg_ctl` command to start the PostgreSQL server, then the stderr is already redirected to the output, so you just need a pipe command, as shown here:

  ```
  pg_ctl start | rotatelogs /var/log/pgsql_log 86400
  ```

- The third approach to managing the log file output is to send the log output to the syslog and letting syslog deal with file rotation. To do this, you need to set the `log_destination` parameter to syslog in the `postgresql.conf` file.

5

Monitoring the System Using Unix Utilities

In this chapter, we will cover the following recipes:

- Monitoring CPU usage
- Monitoring paging and swapping
- Finding the worst user on the system
- Monitoring system load
- Identifying CPU bottlenecks
- Identifying disk I/O bottlenecks
- Monitoring system performance
- Examining historical CPU load
- Examining historical memory load
- Monitoring disk space usage
- Monitoring network status

Introduction

In order to be able to solve performance problems, we should be able to effectively use operating system utilities. We should also be able to use the right operating system tools and commands to identify performance problems that may be due to CPU, memory or disk I/O issues. Many times, a DBA's duties often overlap with certain system administration related functions, and it is important for a DBA to be effective in using the related operating system utilities in order to correctly identify where the underlying issue on the server could be. In this chapter, we are going to discuss various Unix/Linux-related operating system utilities that can help the DBA in performance analysis and troubleshooting issues.

Monitoring CPU usage

In this recipe, we are going to use the `sar` command to monitor CPU usage on the system.

Getting ready

The commands used in this recipe have been performed on an Oracle Solaris machine. Hence, the command output may vary on different Unix- and Linux-related systems.

How to do it...

We can use the `sar` command with the `-u` switch to monitor CPU utilization:

```
bash-3.2$ sar -u 10 8
SunOS usstlz-pinfsi09 5.10 Generic_150400-04 sun4u    08/06/2014

23:32:17    %usr    %sys    %wio   %idle
23:32:27      80      14       3       3
23:32:37      70      14      12       4
23:32:47      72      13      21       4
23:32:57      76      14       6       3
23:33:07      73      10      13       4
23:33:17      71       8      17       4
23:33:27      67       9      20       4
23:33:37      69      10      17       4

Average       73      11      13       4
```

In the preceding command, with the `-u` switch, two values are passed as the input. The first value, which is `10`, displays the number of seconds between `sar` readings, and the second value, which is `8`, indicates the number of times you want `sar` to run.

How it works...

The `sar` command provides a quick snapshot of how much of the CPU is bogged down or utilized. The `sar` output reports values in the following columns:

- `%usr`: This indicates the percentage of CPU running in user mode
- `%sys`: This indicates the percentage of CPU running in system mode

- %wio: This indicates the percentage of CPU running idle, with a process waiting for block I/O
- %idle: This indicates the percentage of CPU that is idle

Often, a low percentage of idle time points to a CPU-intensive job or an underpowered CPU. You could use the ps or prstat command in Solaris to find a CPU-intensive job.

The following are general indicators of performance problems:

- If you see an abnormally high value in the %usr column, this would mean that applications are not tuned properly or are overutilizing the CPU
- If you see a high value in the %sys column, it probably indicates a bottleneck that could be due to swapping or paging and needs to be investigated further

Monitoring paging and swapping

In this recipe, we are going to use the sar and vmstat commands with options to monitor paging and swapping operations.

Getting ready

It is necessary to monitor the amount of paging and swapping happening on the operating system. **Paging** occurs when a part of the operating system process gets transferred from the physical memory to the disk or is read back from the physical memory to the disk. **Swapping** occurs when an entire process gets transferred to the disk from the physical memory or is read back to the physical memory from the disk. Depending on the system, either paging or swapping could be an issue. If paging occurs normally and you see a trend of heavy swapping, then the issue could be related to insufficient memory, or sometimes the issue could be related to disk as well. If the system is heavily paging and not swapping, the issue could be related to either the CPU or the memory. The commands in this section are performed in an Oracle Solaris environment.

How to do it...

We could use the vmstat and sar commands with options in the following manner to monitor the paging and swapping operations:

1. The vmstat command can be used with the -S switch to monitor swapping and paging operations, as follows:

```
bash$ vmstat -S


 kthr      memory                  page            disk
faults      cpu
```

```
 r b w   swap      free      si so   pi  po  fr de sr s0 s2 s3 s4
in      sy      cs     us    sy   id

 6 14 0  453537392 170151696 0  0   2444 186 183 0 0  1  1  1  1
77696   687853  72596  13    4    83
```

In the aforementioned commands, the `si` and `so` columns represent swap-in and swap-out operations, respectively.

Similarly, `pi` and `po` represent page-in and page-out operations, respectively. However, the `sar` command provides more in-depth analysis of paging and swapping operations when used with options.

2. We can also use the `sar` command with the `-p` switch to report paging operations, as follows:

```
bash-3.2$  sar -p 5 4


SunOS usmtnz-sinfsi17 5.10 Generic_150400-04 sun4u    08/08/2014


05:45:18  atch/s  pgin/s ppgin/s  pflt/s  vflt/s slock/s

05:45:23 4391.18    0.80    2.20 12019.44 30956.92    0.60

05:45:28 2172.26    1.80    2.40 5417.76 15499.80    0.20

05:45:33 2765.60    0.20    0.20 9893.20 20556.60    0.00

05:45:38 2194.80    2.00    2.00 7494.80 19018.60    0.00


Average  2879.85    1.20    1.70 8703.00 21500.25    0.20
```

The preceding output reports the following columns:

- `atch/s`: These are the page faults per second that are satisfied by reclaiming a page currently in memory.
- `pgin/s`: The number of times per second that the filesystem receives page in requests.
- `ppgin/s`: These are the pages paged in per second.
- `pflt/s`: This is the number of page faults from protection errors.
- `vflt/s`: This is the number that addresses translation page faults per second. This happens when a valid process table entry does not exist for a given virtual address.
- `slock/s`: These are the faults per second caused by software lock requests requiring physical I/O.

3. Similarly, we can use the `sar` command with the `-w` switch to report swapping activities and identify if there are any swap related issues:

   ```
   bash-3.2$ sar -w 5 4

   SunOS usmtnz-sinfsi17 5.10 Generic_150400-04 sun4u    08/08/2014

   06:20:55 swpin/s bswin/s swpot/s bswot/s pswch/s
   06:21:00    0.00     0.0    0.00     0.0   53143
   06:21:05    0.00     0.0    0.00     0.0   60949
   06:21:10    0.00     0.0    0.00     0.0   55149
   06:21:15    0.00     0.0    0.00     0.0   64075

   Average     0.00     0.0    0.00     0.0   58349
   ```

   The aforementioned output reports the following columns:

   - `swpin/s`: This indicates the number of LWP transfers in the memory per second
   - `bswin/s`: This indicates the number of blocks transferred for swap-ins per second
   - `swpot/s`: This reports the average number of processes that are swapped out of the memory per second
   - `bswot/s`: This reports the number of blocks that are transferred for swap-outs per second
   - `pswch/s`: This indicates the number of kernel thread switches per second

How it works…

If the `si` and `so` columns of the `vmstat -S` output have nonzero values, then this serves as a good indicator of a possible performance issue related to swapping. This must be further investigated using the more detailed analysis provided by the `sar` command with the `-p` and `-w` switches respectively.

For paging, the key is to look for an inordinate amount of page faults of any kind. This would indicate a high degree of paging. The concern is not with paging but with swapping because as paging increases, it would be followed by swapping. We can look at the values in the `atch/s`, `pflt/s`, `vflt/s`, and `slock/s` columns of the `sar -p` command output to review the number of page faults of any type and see the paging statistics to observe whether the paging activity remains steady or increases during a specific timeframe.

For the output of the `sar -w` command, the key column to observe is the `swpot/s` column. This column indicates the average number of processes that are swapped out of the memory per second. If the value in this column is greater than 1, it is an indicator of memory deficiency, and to correct this you would have to increase the memory.

Finding the worst user on the system

In this recipe, we are going to use the `top` command to find the worst performing user on the system at a given point in time.

Getting ready

The `top` command is a Linux-based utility that also works on Unix-based systems.
The commands in this section have been performed on a CentOS Linux machine.

How to do it...

The usage of the `top` command is shown as follows:

```
bash-3.2$top

Cpu states: 0.0% idle, 82.0% user, 18.7% kernel, 0.8% wait, 0.5% swap
Memory: 795M real, 12M free, 318M swap, 1586M free swap
PID USERNAME PRI NICE SIZE RES STATE TIME WCPU CPU COMMAND
23624 postgres -25 2 208M 4980K cpu 1:20 22.47% 94.43% postgres
15811 root -15 4 2372K 716K sleep 22:19 0.61% 3.81% java
20435 admin 33 0 207M 2340K sleep 2:47 0.23% 1.14% postgres
20440 admin 33 0 93M 2300K sleep 2:28 0.23% 1.14% postgres
23698 root 33 0 2052K 1584K cpu 0:00 0.23% 0.95% top
23621 admin 27 2 5080K 3420K sleep 0:17 1.59% 0.38% postgres
23544 root 27 2 2288K 1500K sleep 0:01 0.06% 0.38% br2.1.adm
15855 root 21 4 6160K 2416K sleep 2:05 0.04% 0.19% proctool
897 root 34 0 8140K 1620K sleep 55:46 0.00% 0.00% Xsun
20855 admin -9 2 7856K 2748K sleep 7:14 0.67% 0.00% PSRUN
208534 admin -8 2 208M 4664K sleep 4:21 0.52% 0.00% postgres
755 admin 23 0 3844K 1756K sleep 2:56 0.00% 0.00% postgres
2788 root 28 0 1512K 736K sleep 1:03 0.00% 0.00% lpNet
18598 root 14 10 2232K 1136K sleep 0:56 0.00% 0.00% xlock
1 root 33 0 412K 100K sleep 0:55 0.00% 0.00% ini
```

The first two lines in the preceding output give general system information, whereas the rest of the display is arranged in order of decreasing current CPU usage.

How it works...

The top command provides statistics on CPU activity. It displays a list of CPU-intensive tasks on the system and also provides an interface for manipulating processes.

In the preceding output, we can see the top user to be postgres with a process ID of 23624. We can see the CPU consumption of this user to be 94.43%, which is too high and needs to be investigated, or the corresponding operating system process needs to be killed if it is causing performance issues on the system.

Monitoring system load

In this recipe, we are going to use the uptime command to monitor overall system load.

How to do it...

The uptime command gives us the following information:

- Current system time
- How long the system has been running
- Number of currently logged-on users in the system
- System load average for the past 1, 5, and 15 minutes

The uptime command can be used as follows:

```
bash-3.2$ uptime
11:44pm  up 20 day(s), 20 hr(s),  10 users,  load average: 27.80,
30.46, 33.77
```

In the preceding output, we can see that the current system time is 11:44pm (GMT) and the system has been up and running for the last 20 days and 20 hours without requiring a reboot. The output also tells us that there are ten concurrently logged-on users in the system. Finally, we get the load average during the past 1, 5, and 15 minutes as 27.80, 30.46, and 33.77, respectively.

How it works...

The basic purpose of running the `uptime` command is to take a quick look at the current CPU load on the system. This provides a peek at the current system performance. System load refers to the average number or processes that are either in a runnable or uninterruptable state. A process enters the **runnable** state when it starts to utilize the CPU resources or is waiting to acquire them. It enters the **uninterruptable** state when it spends time waiting for an I/O operation. **Load average** is categorized over the three time intervals, that is, the 1-, 5-, and 15-minute periods. Load averages are not categorized for the number of CPUs on the system. So for a system with a single CPU, a load average of 1 indicates 100 percent busy time period with zero idle time, whereas for a system with 5 CPUs a load average of 1 would indicate an idle time of 80 percent and a busy time period of only 20 percent.

Identifying CPU bottlenecks

In this recipe, we are going to use the `mpstat` command to identify CPU bottlenecks.

Getting ready

The commands in this section have been performed on a Solaris server.

How to do it...

The `mpstat` command is used to report per processor statistics in a tabular format.

The usage of the `mpstat` command is shown as follows:

```
bash-3.2$ mpstat 1 1

CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt
idl

  0  672   0 2457   681   12  539   17   57  119    0  4303   18  10   0
73

  1   90   0 1551   368   22  344    6   37  104    0  3775   17   4   0
79

  2   68   0 1026   274   14  217    4   24   83    0  2393   11   3   0
86

  3   50   0  568   218    9  128    3   17   56    0  1319    7   2   0
92

  4   27   0  907   340   12  233    3   22   72    0  2034    9   2   0
88
```

```
  5   75   0 1777   426   25  370    5   33  111    0  4820  22   4   0
74

  6   69   0 1395   421   15  337    4   27   96    0  2948  14   3   0
83

  7   29   0  888   394    9  273    3   23   74    0  1873   9   3   0
89

  8   10   0  344   177    9   80    2   13   44    0  1007   5   1   0
94

  9   63   0 1275   288   17  268    4   22   90    0  4337  20   3   0
77

 10   72   0 1875   324   28  330    5   30  110    0  5514  25   4   0
71

 11   17   0  438   183   10   94    2   17   50    0  1048   5   2   0
93

 12   10   0  351   175    9   79    2   13   44    0  1047   5   1   0
94

 13   60   0 1207   267   17  245    4   21   87    0  4243  19   3   0
78

 14   72   0 1859   323   29  331    4   30  109    0  5347  24   4   0
72

 15   16   0  434   184   10   94    2   17   50    0  1031   5   2   0
94

 16   20   0  638   197   12  127    2   16   57    0  1810   9   2   0
90

 17   16   0  621   215   12  141    2   16   59    0  2062   9   2   0
89

 18   19   0  785   214   14  151    2   18   63    0  2384  11   2   0
87

 19   66   0 1584   293   24  288    4   26   96    0  5681  25   3   0
71

 20   23   0  826   207   14  142    2   17   64    0  2172  10   2   0
88

 21   14   0  543   190   12  115    2   15   54    0  1896   9   2   0
90

 22   19   0  772   210   14  147    2   18   62    0  2347  11   2   0
88

 23   66   0 1591   294   24  293    4   26   96    0  5398  25   4   0
72

 24   66   0 1820   305   27  311    4   27  100    0  4941  22   4   0
74

 25   20   0  672   210   11  135    2   18   62    0  1783   8   2   0
90
```

```
 26   16   0  645   192   12  116    2   15   59    0  2184   10   2   0
89

 27   20   0  821   213   15  152    2   17   62    0  3016   13   2   0
85

 28   66   0 1698   305   28  308    4   27   98    0  5106   23   4   0
74

 29   20   0  641   194   13  121    2   17   59    0  1731    8   2   0
90

 30   17   0  633   190   11  118    2   15   57    0  2164    9   2   0
89

 31   22   0  798   215   15  161    2   17   61    0  3044   13   2   0
85

 32  637   0 2183   507   21  672   17   61  114    0  4939   20   9   0
70

 33   98   0 1998   383   24  431    7   39   94    0  4076   19   4   0
77

 34   74   0 1217   273   14  265    4   25   73    0  2589   13   3   0
85

 35   54   0  661   216    9  168    3   18   51    0  1624    8   2   0
90

 36   17   0  925   311  117  144    2   18   49    0  1610    8   2   0
90

 37   69   0 2146   302   23  312    4   28   86    0  4624   22   3   0
75

 38   61   0 1856   910  665  254    3   22   71    0  2734   13   5   0
82

 39   12   0 1006   848  661  138    2   15   40    0  1099    5   4   0
91

 40    9   0  402   168    9   82    2   12   32    0   986    5   1   0
94

 41   59   0 1490   288   16  285    4   21   73    0  4233   20   3   0
77

 42   70   0 2486   329   26  356    5   29   91    0  5326   25   4   0
71

 43   16   0  541   180   10   99    2   16   39    0  1052    5   1   0
93

 44   10   0  438   169    8   83    2   13   34    0  1051    5   1   0
94

 45   57   0 1436   264   16  257    3   20   69    0  4137   19   3   0
78
```

How it works...

In the preceding output of the `mpstat` command, each row of the table represents the activity of one processor. The first table shows a summary of activity since the last boot time. The important value from a DBA's perspective is the value in the `smtx` column. The `smtx` measurement indicates the number of times the CPU failed to obtain the **mutual exclusion lock** (**mutex**). Mutex stalls waste CPU time and degrade multiprocessor scaling.

A general rule of thumb is that if the value in the `smtx` column is greater than 200, then it is a symptom and indication of CPU bottleneck issues, which need to be investigated.

Identifying disk I/O bottlenecks

In this recipe, we are going to use the `iostat` command to identify disk-related bottlenecks.

Getting ready

The commands in this section have been performed on a Solaris server.

How to do it...

There are various switches available with the `iostat` command. The following are the most important switches used along with `iostat`:

- `-d`: This switch reports the number of kilobytes transferred per second for specific disks, the number of transfers per second, and the average service time in milliseconds. The following is the usage of the `iostat -d` command:

  ```
  bash-3.2$iostat -d 5  5

  sd0              sd2          sd3           sd4
  Kps  tps  serv  Kps  tps  serv Kps  tps  serv Kps  tps  serv
  1    0     53   57   5   145    19 1    89     0    0   14
  140  14    16   0    0    0     785  31   21     0    0   0
  8    1     15   0    0    0     814  36   18     0    0   0
  11   1     82   0    0    26    818  36   19     0    0   0
  0    0     0    1    0    22    856  37   20     0    0   0
  ```

- `-D`: This switch lists the reads per second, writes per second, and percentage of disk utilization:

  ```
  bash-3.2$ iostat -D 5 5

  sd0             sd2          sd3            sd4
  rps  wps  util  rps  wps  util  rps  wps  util  rps  wps  util
  0     0  0.3     4    0    6.2   1    1    1.8   0    0   0.0
  0     0  0.0     0   35   90.6  237   0   97.8   0    0   0.0
  0     0  0.0     0   34   84.7  218   0   98.2   0    0   0.0
  0     0  0.0     0   34   88.3  230   0   98.2   0    0   0.0
  0     2  4.4     0   37   91.3  225   0   97.7   0    0   0.0
  ```

- `-x`: This switch will report extended disk statistics for all disks:

  ```
  bash-3.2$ iostat -x
                    extended device statistics
  device      r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
  fd0         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
  sd0         0.0    0.0    0.4    0.4  0.0  0.0   49.5   0   0
  sd2         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
  sd3         0.0    4.6    0.0  257.6  0.0  0.1   26.4   0  12
  sd4        69.3    3.6  996.9  180.5  0.0  7.6  102.4   0 100
  nfs10       0.0    0.0    0.4    0.0  0.0  0.0   24.5   0   0
  nfs14       0.0    0.0    0.0    0.0  0.0  0.0    6.3   0   0
  nfs16       0.0    0.0    0.0    0.0  0.0  0.0    4.9   0   0
  ```

How it works...

The `iostat` command reports statistics about disk input and output operations to produce measurements of throughput, utilization, queue lengths, transaction rate, and service time. The first line of the `iostat` output shows everything since booting the system, whereas each subsequent line shows only the prior interval specified.

If we observe the preceding output of the `iostat -d` command in the *How to do it...* section, we can clearly see that the `sd3` disk drive is heavily overloaded. The values in the `kps` (short for kilobytes transferred per second), `tps` (short for the number of transfers per second), and `serv` (short for average service time in milliseconds) columns for the `sd3` disk drive are consistently high over the specified interval. This leads us to a conclusion that moving information from `sd3` to any other drive might be a good idea if this information is representative of disk I/O on a consistent basis. This would reduce the load on `sd3`.

Also, if you observe the output of the `iostat -D` command in the *How to do it...* section, you can conclude that `sd3` has high read activity, which is indicated by the high values in the `rps` (short for reads per second) column. Similarly, we can see that the `sd2` disk drive has a high write activity, which is indicated by the high values in the `wps` (short for writes per second) column. Both of the disk drives, `sd2` and `sd3`, are at the peak level of utilization, which can be seen from the high percentage in the `util` (short for utilization) column. The high values in the `util` column are an indication of I/O problems, which should be investigated by the system administrator.

Similarly, if we take a look at the preceding output of the `iostat -x` command in the *How to do it...* section, we can easily come to the conclusion that the `sd4` disk drive is experiencing I/O problems as seen from the `%b` column, which indicates the percentage of time the disk is busy. For `sd4`, the disk utilization is at 100 percent, which would need a system administrator's immediate attention.

Monitoring system performance

Many times, there are situations when the application users start complaining about the database performance being slow, and as a DBA, you need to determine whether there are system resource bottlenecks on the PostgreSQL server. Running the `vmstat` command can help us to quickly locate and identify any bottlenecks on the server.

Getting ready

The commands in this section have been performed on a CentOS Linux machine.

How to do it...

The `vmstat` command is used to report real-time performance statistics about processes, memory, paging, disk I/O, and CPU consumption. The following is the usage of the `vmstat` command:

```
$ vmstat
procs --------memory--------  --swap-- --io-- -system- ---cpu---
r  b   swpd  free  buff cache   si  so   bi  bo   in  cs   us  sy id  wa
14 0   52340 25272 3068 1662704 0   0    63  76   9   31   15  1   84  0
```

In the preceding output, the first line divides the columns on the second line into six different categories, which are discussed as follows:

- The first category is the process (procs) and it contains the following columns:
    - r: This column indicates the total number of processes waiting for runtime
    - b: This column reports the total number of processes in uninterruptible sleep
- The second category is memory and it contains the following columns:
    - swpd: This column indicates the total amount of virtual memory in use
    - free: This column reports the amount of idle memory available for use
    - buff: This column indicates the amount of memory used for buffers
    - cache: This column indicates the amount of memory used as page cache
- The third category is swap, which contains the following columns:
    - si: This column reports the amount of memory swapped in from the disk
    - so: This column indicates the amount of memory swapped out from the disk
- The fourth category is I/O (io) and it contains the following columns:
    - bi: This column indicates the blocks that are read in from a block device
    - bo: This column reports the blocks that are written out to a block device
- The fifth category is system, which contains the following columns:
    - in: This column reports the number of interrupts per second
    - cs: This column reports the number of context switches per second
- The final category is the CPU (cpu) and it contains the following columns:
    - us: This column reports the percentage of the time the CPU ran user-level code
    - sy: This column reports the percentage of the time the CPU ran system-level code
    - id: This column reports the percentage of time the CPU was idle
    - wa: This column reports the amount of time spent waiting for I/O to complete

How it works...

The following are the general rules of thumb used while interpreting the vmstat command output.

- If the value in the wa column is high, it is an indication that the storage system is probably overloaded and that action needs to be taken to address that issue

- If the value in the b column is greater than zero consistently, then it is an indication that the system does not have enough processing power to service the currently running and scheduled jobs
- If the values in the so and si columns are greater than zero when monitored for a period of time, then it is an indication and symptom of a memory bottleneck

Examining historical CPU load

In this recipe, we are going to show how to use the sar command in combination with various switches to analyze historical CPU load at some time in the past.

Getting ready

The commands used in this recipe have been performed on an Ubuntu Linux machine.

How to do it...

The sar command when used with the -u switch is used to display CPU statistics. When used this way, the sar command will report the current day's activities.

If we are looking to analyze the CPU statistics from some time in the past, we would need to use the -f switch in conjunction with the -u switch of the sar command. The -f option is followed by the files that sar uses to report statistics for different days of the month. These files are usually located in the /var/log/sa directory and they usually have a naming convention of sadd, where dd represents the numeric day of the month, whose values are in the range 01 to 31.

The following is the usage of the sar command to view CPU statistics for the eighth day of the month:

```
$ sar -u -f /var/log/sa/sa08


03:50:10 PM CPU %user %nice %system  %iowait  %idle
04:00:10 PM all 0.42   0.00    0.24    0.00   96.41
04:10:10 PM all 0.22   0.00    1.96    0.00   95.53
04:20:10 PM all 0.22   0.00    1.22    0.01   99.55
04:30:10 PM all 0.22   0.00    0.24    2.11   99.54
04:40:10 PM all 0.24   0.00    0.23    0.00   92.54
Average:    all 0.19   0.00    0.19    0.07   99.55
```

How it works...

Generally the rules of thumb are that if the `%idle` value is low, it serves as an indication that either the CPUs are underpowered or the application load is high. Similarly, if we see nonzero values in the `iowait` time column, it serves as a reminder that the I/O subsystem could be a potential bottleneck.

If we observe the preceding output of the `sar` command, we can see that the `%idle` time is high, which clearly indicates that the CPU is probably not overburdened, and we do not see many nonzero values in the `%iowait` column, which tells us that there is not much contention for disk I/O either.

There's more...

When the `sysstat` package is installed, a few `cron` jobs are scheduled to create files used by the `sar` utility to report historical server statistics. We can observe these `cron` jobs by taking a look at the `/etc/cron.d/sysstat` file.

Examining historical memory load

In this recipe, we are going to see how to analyze the memory load for a previous day of the month.

Getting ready

The commands used in this recipe have been performed on an Ubuntu Linux machine. The command output may vary in other Linux- and Unix-based operating systems.

How to do it...

When it comes to analyzing memory statistics, we need to check out both paging statistics and swapping statistics.

We can use the `sar` command in conjunction with the `-B` switch to report paging statistics along with the `-f` switch to report statistics for different days of the month. As mentioned in the previous recipe, the files that the `sar` command uses to report statistics for different days of the month are located in the `/var/log/sa` directory, and they have a naming convention of `sadd`, where `dd` represent the numeric date of the month, with values ranging from `01` to `31`.

For instance, to report the paging statistics for the fifth day of the month, we can use the `sar` command as follows:

```
$ sar -B -f /var/log/sa/sa05

06:10:05 AM pgpgin/s  pgpgout/s  fault/s   majflt/s
06:20:05 AM 0.02        18.17      19.37     0.00
06:30:05 AM 4.49        26.68      76.15     0.05
06:40:05 AM 4512.43   419.24     380.14      0.65
06:50:06 AM 4850.03  1055.79    4364.73      0.51
07:00:06 PM 4172.68  1096.96    6650.51      0.16
```

Similarly, to report the swapping statistics for different days of the month, we can use the `sar` command in conjunction with the `-W` switch and the `-f` switch.

For instance, to report on the swapping statistics for the fifth day of the month, we can use the `sar` command as follows:

```
$ sar -W -f /var/log/sa/sa05

06:10:05 AM  pswpin/s pswpout/s
06:20:05 AM   0.00     0.00
06:30:05 AM   0.02     0.00
06:40:05 AM   1.15     1.45
06:50:06 AM   0.94     2.99
07:00:06 PM   0.67     6.95
```

How it works...

In the preceding output of the `sar -B -f /var/log/sa/sa05` command, we can clearly see that at about `6.40 AM`, there was a substantial increase in paging from the disk (`pgpgin/s`), pages paged out to disk (`pgpgout/s`), and page faults per second (`fault/s`).

Similarly, when swapping statistics are being reported with the `sar -W -f /var/log/sa/sa05` command, we can clearly see that the swapping started at about `06.40 AM`, which can be seen from the values in the `pswpin/s` column and the `pswpout/s` column. If we see high values in the `pswpin/s` (pages swapped into the memory per second) column and the `pgpgout/s` (pages swapped out per second) column, it means that the current memory is inadequate and needs to be either increased or, for certain application components, optimally resized.

Monitoring disk space usage

In this recipe, we are going to show the commands that are used to monitor disk space.

Getting ready

The commands in this section have been performed on a Solaris server.

How to do it...

We can use the `df` command with various switches to monitor disk space. To make the output more understandable, we often use the `-h` switch with the `df` command:

```
bash-3.2$ df -h


Filesystem                                           size   used  avail
capacity  Mounted on

                                                    132G    80G    50G
62%     /

/devices                                             0K     0K     0K
0%     /devices

ctfs                                                 0K     0K     0K
0%    /system/contract

proc                                                 0K     0K     0K
0%     /proc

mnttab                                               0K     0K     0K
0%     /etc/mnttab

swap                                                418G   488K   418G
1%    /etc/svc/volatile

swap                                                418G    38M   418G
1%    /tmp

swap                                                418G   152K   418G
1%    /var/run

/dev/dsk/c20t60000970000192602156533030374242d0s0   236G   240M   234G
1%    /peterdata/cm_new

/dev/dsk/c20t60000970000192602156533032353441d0s0    30G    30M    29G
1%    /peterdata/native

/dev/dsk/c20t60000970000192602156533033313441d0s0   236G    60G   174G
26%    /peterdata/db_new

/dev/dsk/c20t60000970000195701036533032454646d0s0    30G   6.9G    22G
24%    /peterdata/native
```

```
/dev/dsk/c20t60000970000195701036533032444137d0s0   236G   224G    12G
95%    /peterdata/db

/dev/dsk/c20t60000970000192602156533032333232d0s2   709G   316G   386G
45%    /peterdata/cm

usmtnnas4106-epmnfs.emrsn.org:/peterb1ap_2156   276G   239G    36G    87%
/peterb1ap

usmtnnas4106-epmnfs.emrsn.org:/peterdata_data_2156    98G    53G    45G
54%    /peterdata/data

usmtnnas4106-epmnfs.emrsn.org:/peterdata_uc4appmgr   9.8G   3.6G   6.3G
37%    /peterdata/uc4/
```

How it works...

If we observe the preceding output, we can see that the `/peterdata/db` mount point is nearing its full capacity (it has reached a capacity of `95%`) and only another 12 GB of free disk space is available on the device. This is an indication that the administrator needs to either clean up some old files on the existing mount point to release more free space, or allocate additional space to the given mount point before it reaches its full capacity.

Monitoring network status

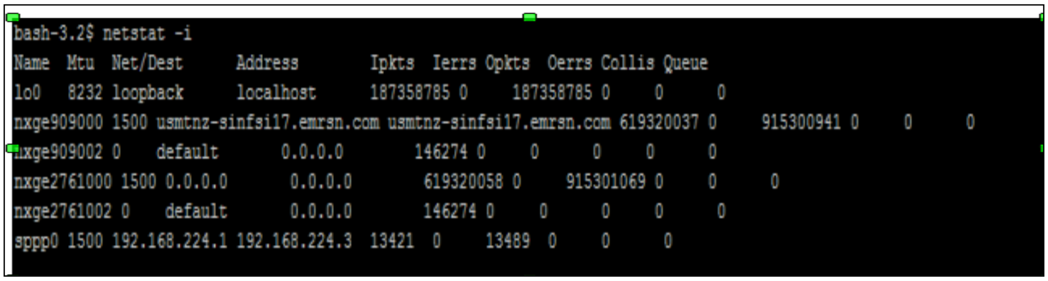
In this recipe, we are going to show how to monitor the status of network interfaces.

Getting ready

The commands used in this recipe have been performed on a CentOS Linux machine. The command output may vary in other Linux- and Unix-based operating systems.

How to do it...

We are going to use the `netstat` command with the `-i` switch to display the status of the network interfaces that are configured on the system. Here is a screenshot that shows the usage of the `netstat` command:



```
bash-3.2$ netstat -i
Name  Mtu  Net/Dest      Address       Ipkts  Ierrs Opkts  Oerrs Collis Queue
lo0   8232 loopback      localhost     187358785 0     187358785 0     0      0
nxge909000 1500 usmtnz-sinfsi17.emrsn.com usmtnz-sinfsi17.emrsn.com 619320037 0     915300941 0     0      0
nxge909002 0    default       0.0.0.0       146274 0     0      0     0      0
nxge2761000 1500 0.0.0.0       0.0.0.0       619320058 0     915301069 0     0      0
nxge2761002 0    default       0.0.0.0       146274 0     0      0     0      0
sppp0 1500 192.168.224.1 192.168.224.3  13421  0     13489  0     0      0
```

How it works...

In the preceding output of the `netstat -i` command, we can determine the number of packets a system transmits and receives on each network interface. The `Ipkts` column determines the input packet count and the `Okpts` column determines the output packet count. If the input packet count remains steady over a period of time, it means that the machine is not receiving network packets at all, and the outcome suggests that it is possibly a hardware failure on the network interface. If the output packet count remains steady over a period of time, then it could possibly mean problems that may be caused due to an incorrect address entry in the host's or the ethers database.

6

Monitoring Database Activity and Investigating Performance Issues

In this chapter, we will cover the following recipes:

- Checking active sessions
- Finding out what queries users are currently running
- Getting the execution plan for a statement
- Logging slow statements
- Collecting statistics
- Monitoring database load
- Finding blocking sessions
- Table access statistics
- Finding unused indexes
- Forcing a query to use an index
- Determining disk usage

Introduction

Monitoring databases and troubleshooting performance issues is one of the major duties of a database administrator. Ensuring a healthy database with optimal performance is what a DBA is employed for. Database monitoring should be done regularly in a proactive manner to resolve any known issues before they reach a critical state and lead to performance issues. Troubleshooting performance issues is a reactive approach because it is only after an issue is reported that the DBA starts troubleshooting it. If proactive alerts are set in a timely manner, a DBA can ensure that the database is in a healthy state, and this can lead to a reduction in performance issues.

We have used the `dvdrental` sample database for all of the code in this chapter. The `dvdrental` sample is available in the code bundle. Details about the installation and working of the `dvdrental` database are also in the code bundle.

Checking active sessions

In this recipe, we are going to learn how to check for active sessions in a database.

Getting ready

We are going to query the `pg_stat_activity` table to check for active sessions in a database. The query used in this recipe works in PostgreSQL version 9.2 onwards.

How to do it...

We can use the following SQL query to find the active sessions in the `hrdb` database:

```
SELECT pid , usename, application_name, client_addr,
  client_hostname, query, state from pg_stat_activity
where datname='dvdrental';
```

How it works...

We use the preceding query to find all of the client connections made to the `hrdb` database. Here is an explanation of the columns in the `pg_stat_activity` table to find information regarding active sessions in the `hrdb` database:

- The `pid` column: The value in this column indicates the process ID of the currently connected user to the database, the `hrdb` database in our case.
- The `datname` column: The value in this column indicates the name of the database to which the user is currently connected.

- The application_name column: The value in this column provides the application name that is being used by the user currently connected to the database.
- The client_addr column: The value in this column gives the IP address of the user that is currently connected to the database.
- The client_hostname column: The value in this column gives the hostname of the connected client.
- The query column: The value in this column provides the full text of the SQL query that is being executed by the client.
- The preceding output also includes the state column, which indicates the status of the pid column of the currently connected user. The state column can have the following possible values:
    - active: This value indicates that the user session is currently executing a query at the backend.
    - idle: This value indicates that the backend is waiting for a new client command.
    - idle in transaction: This value indicates that the backend process is currently involved in a transaction but it is not executing a query.
    - fastpath function call: This value indicates that the backend process is executing a fast-path function.
    - disabled: This value is reported if the value of the track_activities configuration parameter is disabled for the currently connected backend. If the value of the state column is disabled, it means that information is not being collected on the currently executing command for each session.

There's more...

If you are using a PostgreSQL version earlier than 9.2, then you can use the following query to find the active sessions in a PostgreSQL database:

```
SELECT datname , procpid, usename,application_name,client_addr,
  client_hostname,current_query FROM pg_stat_activity;
```

Finding out what queries users are currently running

In this recipe, we are going to show the most recent, or currently executing, queries executed by users in a PostgreSQL database.

Getting ready

Before finding out what queries the users are issuing against the database, the first thing we need to do is to enable the `track_activities` configuration parameter in the `postgresql.conf` configuration file, as follows:

```
track_activities =  on
```

Once this parameter is enabled, we would need to reload the configuration file to ensure that the changes made come into effect:

```
pg_ctl -D $PGDATA reload
```

How to do it...

We can use the following query to view the text of the query that is being executed by the user currently connected to the database:

```
SELECT datname, pid, usename, query_start, state, query
FROM pg_stat_activity
```

This query can also be used in the SQL editor of the pgAdmin tool.

How it works...

PostgreSQL will collect data about all of the running queries whenever the `track_activities` configuration parameter is enabled. We can see the most recent query executed by a user in a specific PostgreSQL database by referring to the SQL statement retrieved from the query column in the `pg_stat_activity` table. The `query_start` column indicates the time on the server that the client executed the query.

Getting the execution plan for a statement

In this recipe, we are going to see how to get the execution plan for a SQL statement.

Getting ready

The `EXPLAIN` command is used to get the execution plan for a SQL statement.

How to do it...

Every query that is triggered in PostgreSQL has an execution plan. The `EXPLAIN` command can be run in any of the three given modes:

- **Generic Mode**: In this mode, we just need to specify the `EXPLAIN` command followed by the SQL statement. The PostgreSQL planner will display the execution plan that it generated for the specified SQL statement. The execution plan will show the scan method used to access the table referenced in the query. Other details included could be the estimated execution cost of the SQL statement, which is the planner's estimation of how long it will take to execute the SQL statement. The `EXPLAIN` command can be invoked as follows:

```
dvdrental=# EXPLAIN  select * from payment where amount > 4.99;
                         QUERY PLAN
--------------------------------------------------------------
 Seq Scan on payment  (cost=0.00..290.45 rows=3616 width=26)
   Filter: (amount > 4.99)
(2 rows)
```

- **Analyze Mode**: The SQL statement can also be executed in analyze mode. This provides the actual runtime statistics such as the total time it took to execute the query and the actual number of rows returned. With the help of this option we can determine whether the PostgreSQL planner's estimates are close to the actual numbers or not. We can run the `EXPLAIN ANALYZE` mode as follows:

```
dvdrental=# EXPLAIN ANALYZE select * from payment where amount >
4.99;
                         QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on payment  (cost=0.00..290.45 rows=3616 width=26)
(actual time=0.024.
.7.117 rows=3618 loops=1)
   Filter: (amount > 4.99)
   Rows Removed by Filter: 10978
 Total runtime: 7.457 ms
(4 rows)
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

- **Verbose Mode**: The benefit of running the `EXPLAIN` command in verbose mode is that the EXPLAIN plan output will also display the columns that are passed by the query. This information can be valuable when the underlying query is complicated. We can run the `EXPLAIN` command in verbose mode as follows:

```
dvdrental=# EXPLAIN VERBOSE select * from payment where amount >
4.99;
                         QUERY PLAN
------------------------------------------------------------------
 Seq Scan on public.payment  (cost=0.00..290.45 rows=3616
width=26)
   Output: payment_id, customer_id, staff_id, rental_id, amount,
payment_date
   Filter: (payment.amount > 4.99)
(3 rows)
```

How it works...

The output of the `EXPLAIN` command is organized in a series of plan nodes. It is analyzed with a bottom-to-top approach. At the bottom, there are nodes that look at tables, scan them, or look at things through an index. Each line in the `EXPLAIN` command output is a plan node.

There are several numeric measurements that are associated with a node. For instance, if we look at the output of the `EXPLAIN ANALYZE` command, we can see the following details:

- `Seq Scan`: The first thing that we observe is that the plan has one node, which is Sequential Scan node.
- `cost=0.00..290.45`: The first cost is the start-up cost of this node. The value here determines how much work is estimated to be done before the node produces its first row of output. Here, the value is zero because a `Seq Scan` node instantly produces rows. The second estimated cost is the cost of running the entire node until it completes.
- `rows=3616`: The number of rows to output if the node runs to completion.
- `width=26`: This value provides an estimate of the average number of bytes each row output for the node will contain.

The points that were just discussed are related to the estimated values. The actual figures tell the response time details for the query. The actual time consists of the actual start-up cost and the cost of running the entire node. The `rows` column displays the actual number of rows returned by the query.

If the difference between estimated rows and actual rows is huge, it is an indication that the query optimizer has made a bad decision based on the current execution plan.

For more detailed information, refer to `http://www.postgresql.org/docs/9.2/static/sql-explain.html`, which describes execution plans in PostgreSQL.

Logging slow statements

In this recipe, we are going to cover how to log slow queries in the PostgreSQL server.

Getting ready

We would need to make changes to some of the configuration parameters in the `postgresql.conf` file that enable logging, and then restart the PostgreSQL server in order to ensure that the changes made to those configuration parameters come into effect.

How to do it...

Here is the sequence of steps that needs to be followed in order to log slow-running SQL statements:

1. The following parameters would need to be set in the `postgresql.conf` file:

```
logging_collector = on
log_directory = 'pg_log'
log_min_duration_statement = 100
```

2. Once these parameters are set in the `postgresql.conf` file, we would need to restart the PostgreSQL server as follows:

```
pg_ctl  -D $PGDATA restart
```

How it works...

Here is the explanation of the sequence of steps done in the preceding section:

- Setting `log_min_duration_statement` to `100`, as seen in the preceding section, means that any SQL statements that run for 100 milliseconds or longer will be logged in the PostgreSQL server. This is a useful parameter to enable because it can help in tracking down unoptimized queries in client applications.
- Setting the `logging_collector` parameter enables the logging collector, a background process whose function is to capture log messages sent to stderr and redirect them to logfiles. Setting this parameter is useful because log messages captured this way may contain more information than syslog.
- Setting the `log_directory` parameter would determine the directory in which the log files will be created.

Collecting statistics

In this recipe, we are going to cover the parameters that need to be enabled in order to collect statistics.

Getting ready

The PostgreSQL server comes with set of predefined statistics access functions and a set of predefined statistics views. These views use the predefined statistics functions to collect statistics in PostgreSQL. By default, only a small number of statistics are collected. In the next section, we will cover the configuration parameters that control the collection of statistics.

How to do it...

This is the sequence of steps that needs to be followed in order to enable statistics collection in PostgreSQL:

1. The following configuration parameters would need to be set in the `postgresql.conf` file:

   ```
   track_activities = on
   track_counts = on
   track_functions = all
   track_io_timing = on
   ```

2. Once these configuration parameters have been set, we would need to reload the configuration file in order to ensure that parameter changes come into effect:

   ```
   pg_ctl -D $PGDATA reload
   ```

How it works...

Here is the explanation for the steps done in the preceding section.

- Setting `track_activities` enables monitoring of the command currently being executed by the server process, along with the time the command began execution.
- Setting the `track_counts` configuration parameter enables collection of statistics on database activity, which includes statistics collection for table and index accesses.
- Setting the value of the `track_functions` configuration parameter to `all` enables tracking of user-defined functions, which includes tracking procedural language functions along with SQL and C language functions.

- Setting the track_io_timing configuration parameter enables the timing of database I/O calls. Enabling this parameter may cause some performance overhead because when this parameter is enabled, PostgreSQL repeatedly probes the operating system for the current time. I/O timing is captured in the pg_stat_database view. Major pieces of information that are captured are the number of disk blocks and the time spent on reading and writing the database blocks for the given PostgreSQL database.

For more details on this topic, please refer to http://www.postgresql.org/docs/9.3/static/monitoring-stats.html

and http://www.postgresql.org/docs/9.3/static/runtime-config-statistics.html#GUC-TRACK-ACTIVITIES.

Monitoring database load

In this recipe, we are going to use queries that can be used to monitor the database load.

Getting ready

We can use the pg_stat_database view to monitor the current database load.

How to do it...

In order to identify the existing database load, we would need to know the following:

- information such as the number of active database connections
- number of commits and rollbacks issued
- total blocks read and the percentage of buffer hits for a given database

We can use the following query to identify the existing database load for the dvdrental database:

```
dvdrental=# SELECT numbackends as CONN, xact_commit as TX_COMM,
xact_rollback as
TX_RLBCK, blks_read + blks_hit as READ_TOTAL,
blks_hit * 100 / (blks_read + blks_hit)
as BUFFER FROM pg_stat_database WHERE datname = 'dvdrental';

 conn | tx_comm | tx_rlbck | read_total | buffer
------+---------+----------+------------+--------
    9 |      45 |        1 |       1456 |     99
(1 row)
```

How it works...

The following columns are retrieved by the preceding query:

- `numbackends`: This column represents the total number of active connections
- `xact_commit`: This column represents the total number of commits
- `xact_rollback`: This column represents the total number of rollbacks
- `blks_read`: This column represents the total blocks read
- `blks_hit`: This column represents the total number of buffer hits

Here is the sequence of steps that are required in order to determine the current database load:

1. First, we would need to reset the statistics by using the `pg_stat_reset()` function, like this:

   ```
   dvdrental=# SELECT pg_stat_reset();
   pg_stat_reset
   ---------------
   (1 row)
   ```

2. The next step would be to wait for a period of time to ensure that sufficient statistics have been collected.
3. The final step would be to invoke the statistics query on the `pg_stat_database` view, as shown in the previous section.

Finding blocking sessions

In this recipe, we are going to see the queries that can help us to find out which user sessions are getting blocked and who is blocking them.

Getting ready

To run these queries, you will need to use the superuser account.

How to do it...

We can use the following query to find information regarding the blocking and blocked sessions:

```
SELECT bl.pid AS blocked_pid,
  a.usename AS blocked_user,
  ka.query AS blocking_statement,
  now() - ka.query_start AS blocking_duration,
```

```
  kl.pid AS blocking_pid,
  ka.usename AS blocking_user,
  a.query AS blocked_statement,
  now() - a.query_start AS blocked_duration
FROM  pg_catalog.pg_locks bl
JOIN pg_catalog.pg_stat_activity a  ON a.pid = bl.pid
JOIN pg_catalog.pg_locks kl ON kl.transactionid =
  bl.transactionid AND kl.pid != bl.pid
JOIN pg_catalog.pg_stat_activity ka ON ka.pid = kl.pid
WHERE NOT bl.granted;
```

The aforementioned query works in PostgreSQL version 9.2 and subsequent versions.

How it works...

The query in the preceding section finds the process ID, the username, and the queries that are being run by the blocked and the blocking sessions. Here, we are using the `JOIN` condition on the `pid` column of the `pg_locks` and `pg_stat_activity` tables twice: once for the blocking sessions and then for the blocked sessions. We are also doing a join of the `pg_lock` table to itself, on the `transactionid` column, and the filter condition here is that the `pid` (process ID) column should be unique to each other when the same table `pg_locks` is joined to itself.

If you are using a version of PostgreSQL older than PostgreSQL version 9.2, then you may use this query to identify blocking sessions:

```
SELECT bl.pid AS blocked_pid,
  a.usename AS blocked_user,
  ka.current_query AS blocking_statement,
  now() - ka.query_start AS blocking_duration,
  kl.pid AS blocking_pid,
  ka.usename AS blocking_user,
  a.current_query AS blocked_statement,
  now() - a.query_start AS blocked_duration
FROM  pg_catalog.pg_locks bl
JOIN pg_catalog.pg_stat_activity a  ON a.procpid = bl.pid
JOIN pg_catalog.pg_locks kl ON kl.transactionid = bl.transactionid
AND kl.pid != bl.pid
JOIN pg_catalog.pg_stat_activity ka ON ka.procpid = kl.pid
WHERE NOT bl.granted;
```

Table access statistics

In this recipe, we are going to see the details of how the tables are being accessed.

Getting ready

Statistical values about user tables are available in the `pg_stat_user_tables` view. This table can be used to get details such as the estimated number of live and dead rows and the timestamps for the time when the table was last vacuumed or autovacuumed. Similarly, we can use the `pg_stat_user_tables` to find details about table's access.

How to do it...

We can use the following query to determine whether the tables are being accessed by sequential or index scans:

```
dvdrental=# SELECT schemaname,relname,seq_scan,idx_scan,cast(idx_scan
AS numeric) / (idx_scan + seq_scan) AS idx_scan_pct
FROM pg_stat_user_tables WHERE (idx_scan + seq_scan)>0
ORDER BY idx_scan_pct;
 schemaname |    relname    | seq_scan | idx_scan |
idx_scan_pct
------------+---------------+----------+----------+-------------------
 public     | category      |        2 |        0 |
0.00000000000000000000
 public     | actor         |        3 |        0 |
0.00000000000000000000
 public     | customer      |        7 |        0 |
0.00000000000000000000
 public     | country       |        2 |        0 |
0.00000000000000000000
 public     | film_category |        3 |        0 |
0.00000000000000000000
 public     | payment       |        7 |        0 |
0.00000000000000000000
 public     | inventory     |        4 |        0 |
0.00000000000000000000
 public     | language      |        2 |        0 |
0.00000000000000000000
 public     | store         |        4 |        0 |
0.00000000000000000000
 public     | film_actor    |        4 |        0 |
0.00000000000000000000
 public     | city          |        4 |        0 |
0.00000000000000000000
```

```
 public      | rental        |        7 |        0 |
0.00000000000000000000
 public      | staff         |        5 |        0 |
0.00000000000000000000
 public      | film          |        8 |        0 |
0.00000000000000000000
 public      | address       |        4 |        4 |
0.50000000000000000000
(15 rows)
```

How it works...

In the preceding output, we can see that for a majority of tables starting from the `category` table until the `film` table, access is done by sequential scans because all of the data fits into a single data page. For the table address, we can see that for some of the queries, access is done by sequential scans, and for some statements, PostgreSQL is using indexes to look up values.

Another interesting aspect is to find out how many rows were processed by these scans. We can use the following query to get this information:

```
dvdrental=# SELECT relname,seq_tup_read,idx_tup_fetch,cast(
idx_tup_fetch AS numeric) / (idx_tup_fetch + seq_tup_read)
AS idx_tup_pct FROM pg_stat_user_tables WHERE (idx_tup_fetch +
seq_tup_read)>0 ORDER BY idx_tup_pct;
    relname    | seq_tup_read | idx_tup_fetch |        idx_tup_pct
---------------+--------------+---------------+----------------------
------
 category      |           32 |             0 |
0.00000000000000000000
 actor         |          600 |             0 |
0.00000000000000000000
 customer      |         4193 |             0 |
0.00000000000000000000
 country       |          218 |             0 |
0.00000000000000000000
 film_category |         3000 |             0 |
0.00000000000000000000
 payment       |       102172 |             0 |
0.000000000000000000000000
 inventory     |        18324 |             0 |
0.000000000000000000000000
 language      |           12 |             0 |
0.00000000000000000000
 store         |            8 |             0 |
0.00000000000000000000
 film_actor    |        21848 |             0 |
0.000000000000000000000000
```

```
 city          |         2400 |             0 |
0.00000000000000000000
 rental        |       112308 |             0 |
0.000000000000000000000000
 staff         |           10 |             0 |
0.00000000000000000000
 film          |         8000 |             0 |
0.00000000000000000000
 address       |         2412 |             4 |
0.00165562913907284768
(15 rows)
```

In the preceding output, we can see that for all of the tables, most of the rows were processed by sequential scan. Only for the addresses table, four rows were accessed using an index lookup.

Finding unused indexes

It becomes necessary to check for unused indexes because indexes end up consuming a significant chunk of disk space, and if not monitored closely, they can consume unnecessary CPU cycles, more so in the case of them becoming fragmented.

Getting ready

In order to be able to find unused indexes in PostgreSQL, we need to ensure that the `track_activities` and `track_counts` configuration parameters are enabled in the `postgresql.conf` file. It is only when statistics are collected that we will be able to identify the unused indexes.

How to do it...

We can use the following query to identify unused indexes in PostgreSQL:

```
SELECT
    relid::regclass AS table,
    indexrelid::regclass AS index,
    pg_size_pretty(pg_relation_size(indexrelid::regclass))
AS index_size,
    idx_tup_read,
    idx_tup_fetch,
    idx_scan
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE idx_scan = 0
```

```
AND indisunique IS FALSE;

   table    |         index         | index_size | idx_tup_read | idx_
tup_fetch | idx_scan
------------+-----------------------+------------+--------------+------
---------+----------
 film       | film_fulltext_idx     | 88 kB      |            0 |
0 |        0
 actor      | idx_actor_last_name   | 16 kB      |            0 |
0 |        0
 customer   | idx_fk_address_id     | 32 kB      |            0 |
0 |        0
 address    | idx_fk_city_id        | 32 kB      |            0 |
0 |        0
 city       | idx_fk_country_id     | 32 kB      |            0 |
0 |        0
 payment    | idx_fk_customer_id    | 336 kB     |            0 |
0 |        0
 film_actor | idx_fk_film_id        | 136 kB     |            0 |
0 |        0
 rental     | idx_fk_inventory_id   | 368 kB     |            0 |
0 |        0
 film       | idx_fk_language_id    | 40 kB      |            0 |
0 |        0
 payment    | idx_fk_rental_id      | 336 kB     |            0 |
0 |        0
 payment    | idx_fk_staff_id       | 336 kB     |            0 |
0 |        0
 customer   | idx_fk_store_id       | 32 kB      |            0 |
0 |        0
 customer   | idx_last_name         | 32 kB      |            0 |
0 |        0
 inventory  | idx_store_id_film_id  | 120 kB     |            0 |
0 |        0
 film       | idx_title             | 56 kB      |            0 |
0 |        0
(15 rows)
```

How it works...

If we take a look at the preceding output, we can conclude that wherever the entry for idx_scan is zero, it clearly means that either the given index has never been used or most likely not used since the time pg_stat_reset() function was run, which basically resets all of the statistics counters for the current database to zero. In the preceding section, we are doing a join on the pg_stat_user_indexes and pg_index tables, on the indexrelid column.

In the preceding query output, the idx_tup_read, idx_tup_fetch, and idx_scan columns indicate the usage of the index:

- The idx_tup_read column indicates how many rows have been read using the index
- The idx_tup_fetch column indicates the number of rows that have been fetched using the index
- The idx_scan column indicates the number of times the index was used by the query planner

There's more…

Just as with unused indexes, we also need to find out whether there are any duplicate indexes because duplicate indexes also consume unnecessary space. Quite often, there are instances of indexes defined on a column of a table with a unique key and the same column is also defined as the primary key. This situation would result in a duplicate index since the primary key itself is unique, and in that situation, there is no need to define an additional index on the same column as a unique index. We can use the following query to identify duplicate indexes in PostgreSQL:

```
SELECT pg_size_pretty(sum(pg_relation_size(idx))::bigint) AS size,
  (array_agg(idx))[1] AS idx1, (array_agg(idx))[2] AS idx2,
  (array_agg(idx))[3] AS idx3, (array_agg(idx))[4] AS idx4
FROM (SELECT indexrelid::regclass AS idx, (indrelid::text ||E'\n'||
  indclass::text ||E'\n'|| indkey::text ||E'\n'||
  coalesce(indexprs::text,'')||E'\n' || coalesce(indpred::text,''))
AS KEY
FROM pg_index) sub
GROUP BY KEY HAVING count(*)>1
ORDER BY sum(pg_relation_size(idx)) DESC;
```

Once the duplicate indexes have been identified, they can then be dropped to reclaim lost space.

You can refer to https://gist.github.com/jberkus/6b1bcaf7724dfc2a54f3 and http://www.databasesoup.com/2014/05/new-finding-unused-indexes-query.html, which contain more information related to unused indexes.

Forcing a query to use an index

In this recipe, we show different methods that can be used to force the database to use an index.

Getting ready

Usually, it is the job of the PostgreSQL optimizer to determine whether a sequential scan or an index lookup is going to be more efficient when the table is being accessed by a query to fetch results. However, if we decide that it is worth gambling on an index, then we must confirm our results by testing the query execution in the development environment before moving the results over to production.

How to do it...

There are two ways by which we can force the database to use an index:

- The first is by setting `enable_seqscan` to `false`. This can be demonstrated by a scenario given as follows:

```
dvdrental=# create table test_no_index(id int);
CREATE TABLE
dvdrental=# set enable_seqscan to false;
SET
dvdrental=# explain select * from test_no_index where id > 12;
                         QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on test_no_index  (cost=10000000000.00..10000000040.00
rows=800 width=
4)
   Filter: (id > 12)
(2 rows)
```

  - Next, we create an index on the given table so as to give the optimizer one or more access paths:

```
dvdrental=# create index new_idx_test_no_index on test_no_
index(id);
CREATE INDEX
```

  - If we now check the execution plan for the query, we will see that instead of a sequential scan, the query plan uses an index lookup to access the table to fetch the query result:

```
dvdrental=# explain select * from test_no_index where id >
12;
                         QUERY PLAN
-------------------------------------------------------------
```

```
 Bitmap Heap Scan on test_no_index  (cost=10.35..30.35
rows=800 width=4)
   Recheck Cond: (id > 12)
   ->  Bitmap Index Scan on new_idx_test_no_index
(cost=0.00..10.15 rows=800 wi
dth=0)
         Index Cond: (id > 12)
(4 rows)
```

- Another way is to set the value of the random_page_cost configuration parameter to a lower or equivalent value to seq_page_cost. By doing this, PostgreSQL will prefer index scans for some of the SQL queries. This can be done as follows:

  ```
  dvdrental=# set random_page_cost = 2;
  SET
  ```

How it works...

In the preceding section, setting enable_seqscan to false will disable sequential scans and force the optimizer to try and use a different plan. In our scenario, we disabled sequential scans and created an index, new_idx_test_no_index, on the test_no_index table. By doing this, we are providing the optimizer with another access path for the test_no_index table.

Similarly, lowering the value of the random_page_cost configuration parameter will cause the system to prefer index scans. By default, the value of randon_page_cost is 4, which is higher than the default value of the seq_page_cost configuration parameter, which is 1, therefore causing a preference for sequential scans over index scans. Lowering the value of random_page_cost might help some queries whereby the optimizer might prefer to use an index lookup.

Determining disk usage

In this recipe, we are going to display the amount of disk usage for a specific database and its associated tables and indexes.

How to do it...

We can use the following SQL query to find the total size of an existing database, the dvdrental database in this case:

```
dvdrental=#  SELECT pg_size_pretty(pg_database_size('dvdrental')) As
fulldbsize;
 fulldbsize
```

```
------------
 14 MB
(1 row)
```

In this query output, we can see that the total size of the `dvdrental` database is approximately 14 MB.

Similarly, to display the size of the existing tables and their associated indexes in the `dvdrental` database, we can use the following SQL query:

```
SELECT relname as "Table",
  pg_size_pretty(pg_relation_size(relid)) As " Table Size",
  pg_size_pretty(pg_total_relation_size(relid) -
  pg_relation_size(relid)) as "Index Size"
FROM pg_catalog.pg_statio_user_tables ORDER BY
pg_total_relation_size(relid) DESC;
```

```
        Table        |  Table Size | Index Size
---------------------+-------------+------------
 rental              | 1200 kB     | 1272 kB
 payment             | 864 kB      | 1368 kB
 film                | 432 kB      | 256 kB
 film_actor          | 240 kB      | 296 kB
 inventory           | 200 kB      | 264 kB
 customer            | 72 kB       | 152 kB
 keytbl              | 40 kB       | 144 kB
 address             | 64 kB       | 88 kB
 city                | 40 kB       | 88 kB
 film_category       | 48 kB       | 64 kB
 actor               | 16 kB       | 56 kB
 encdata             | 24 kB       | 32 kB
 store               | 8192 bytes  | 32 kB
 staff               | 8192 bytes  | 24 kB
 category            | 8192 bytes  | 16 kB
 country             | 8192 bytes  | 16 kB
 language            | 8192 bytes  | 16 kB
 test                | 0 bytes     | 8192 bytes
 test_no_index       | 0 bytes     | 8192 bytes
 table_with_no_index | 0 bytes     | 8192 bytes
(20 rows)
```

How it works...

If we examine the preceding output, we can see the names of the tables along with the respective table and index sizes. In the preceding query, we used two functions, `pg_relation_size()` and `pg_total_relation_size()`. The `pg_relation_size()` function reports the table size in kilobytes and the `pg_total_relation_size()` function reports the total size of the table used on the disk inclusive of the space used by TOAST data and indexes. So in order to get the correct index sizes for all of the indexes for a specific table, we subtracted the value of `pg_relation_size()` from `pg_total_relation_size()` using the `relid` column as a parameter in both of the functions.

If you need more information on determining disk usage, you can refer to `http://wiki.postgresql.org/wiki/Disk_Usage` and `https://wiki.postgresql.org/wiki/Index_Maintenance`.

There's more...

In this section, we will provide some links you can refer to to get advice on dealing with performance issues related to PostgreSQL.

You can check out the performance mailing list at `http://archives.postgresql.org/pgsql-performance/`.

You can also refer to some of the PostgreSQL Wiki links that explain what to include in your performance problem report and some useful troubleshooting information, at `http://wiki.postgresql.org/wiki/Guide_to_reporting_problems` and `http://wiki.postgresql.org/wiki/Performance_Optimization`.

If you have purchased Premium support from vendors such as 2ndQuadrant and EnterpriseDB, you can log tickets with their support team concerning PostgreSQL issues.

7

High Availability and Replication

In this chapter, we will cover the following recipes:

- Setting up hot streaming replication
- Replication using Slony-I
- Replication using Londiste
- Replication using Bucardo
- Replication using DRBD
- Setting up the Postgres-XC cluster

Introduction

The important components for any production database is to achieve fault tolerance, 24/7 availability, and redundancy. It is for this purpose that we have different high availability and replication solutions available for PostgreSQL.

From a business perspective, it is important to ensure 24/7 data availability in the event of a disaster situation or a database crash due to disk or hardware failure. In such situations, it becomes critical to ensure that a duplicate copy of the data is available on a different server or a different database, so that seamless failover can be achieved even when the primary server/database is unavailable.

In this chapter, we will talk about various high availability and replication solutions, including some popular third-party replication tools such as Slony-I, Londiste, and Bucardo. We will also discuss block-level replication using DRBD, and finally, set up a PostgreSQL extensible cluster, that is, Postgres-XC.

Setting up hot streaming replication

In this recipe, we are going to set up a master-slave streaming replication.

Getting ready

For this exercise, you will need two Linux machines, each with the latest version of PostgreSQL installed. We will be using the following IP addresses for the master and slave servers:

- Master IP address: 192.168.0.4
- Slave IP address: 192.168.0.5

Before you start with the master-slave streaming setup, it is important that the SSH connectivity between the master and slave is setup.

How to do it...

Perform the following sequence of steps to set up a master-slave streaming replication:

1. First, we are going to create a user on the master, which will be used by the slave server to connect to the PostgreSQL database on the master server:

   ```
   psql -c "CREATE USER repuser REPLICATION LOGIN ENCRYPTED
   PASSWORD 'charlie';"
   ```

2. Next, we will allow the replication user that was created in the previous step to allow access to the master PostgreSQL server.

   This is done by making the necessary changes as mentioned in the `pg_hba.conf` file:

   ```
   Vi pg_hba.conf

   host    replication    repuser    192.168.0.5/32    md5
   ```

3. In the next step, we are going to configure parameters in the `postgresql.conf` file. These parameters need to be set in order to get the streaming replication working:

   ```
   Vi /var/lib/pgsql/9.3/data/postgresql.conf

   listen_addresses = '*'
   wal_level = hot_standby
   max_wal_senders = 3
   wal_keep_segments = 8
   archive_mode = on
   ```

```
archive_command = 'cp %p /var/lib/pgsql/archive/%f && scp %p
postgres@192.168.0.5:/var/lib/pgsql/archive/%f'

checkpoint_segments = 8
```

4. Once the parameter changes have been made in the `postgresql.conf` file in the previous step, the next step will be to restart the PostgreSQL server on the master server, in order to let the changes take effect:

   ```
   pg_ctl -D /var/lib/pgsql/9.3/data restart
   ```

5. Before the slave can replicate the master, we will need to give it the initial database to build off. For this purpose, we will make a base backup by copying the primary server's data directory to the standby. The `rsync` command needs to be run as a root user:

   ```
   psql -U postgres -h 192.168.0.4 -c "SELECT pg_start_
   backup('label', true)"


   rsync -a /var/lib/pgsql/9.3/data/ 192.168.0.5:/var/lib/pgsql/9.3/
   data/  --exclude postmaster.pid


   psql -U postgres -h 192.168.0.4 -c "SELECT pg_stop_backup()"
   ```

6. Once the data directory, mentioned in the previous step, is populated, the next step is to enable the following parameter in the `postgresql.conf` file on the slave server:

   ```
   hot_standby = on
   ```

7. The next step will be to copy the `recovery.conf.sample` file in the `$PGDATA` location on the slave server and then configure the following parameters:

   ```
   cp /usr/pgsql-9.3/share/recovery.conf.sample
   /var/lib/pgsql/9.3/data/recovery.conf
   ```

   ```
   standby_mode = on
   primary_conninfo = 'host=192.168.0.4 port=5432 user=repuser
   password=charlie'
   trigger_file = '/tmp/trigger.replication'
   restore_command = 'cp /var/lib/pgsql/archive/%f "%p"'
   ```

8. The next step will be to start the slave server:

   ```
   service postgresql-9.3 start
   ```

9. Now that the above mentioned replication steps are set up, we will test for replication. On the master server, log in and issue the following SQL commands:

   ```
   psql -h 192.168.0.4  -d postgres -U postgres -W


   postgres=# create database test;

   postgres=# \c test;

   test=# create table testtable ( testint int, testchar varchar(40)
   );

   CREATE TABLE
   test=# insert into testtable values ( 1, 'What A Sight.' );
   INSERT 0 1
   ```

10. On the slave server, we will now check whether the newly created database and the corresponding table, created in the previous step, are replicated:

    ```
    psql -h 192.168.0.5  -d test -U postgres -W

    test=# select * from testtable;
    testint | testchar
    ---------+--------------------------
    1 | What  A Sight.
    (1 row)
    ```

How it works...

The following is the explanation for the steps performed in the preceding section.

In the initial step of the preceding section, we create a user called repuser, which will be used by the slave server to make a connection to the primary server. In the second step of the preceding section, we make the necessary changes in the pg_hba.conf file to allow the master server to be accessed by the slave server using the repuser user ID that was created in step 1. We then make the necessary parameter changes on the master in step 3 of the preceding section to configure a streaming replication. The following is a description of these parameters:

- listen_addresses: This parameter is used to provide the IP address associated with the interface that you want to have PostgreSQL listen to. A value of * indicates all available IP addresses.
- wal_level: This parameter determines the level of WAL logging done. Specify hot_standby for streaming replication.

- wal_keep_segments: This parameter specifies the number of 16 MB WAL files to be retained in the pg_xlog directory. The rule of thumb is that more such files might be required to handle a large checkpoint.
- archive_mode: Setting this parameter enables completed WAL segments to be sent to the archive storage.
- archive_command: This parameter is basically a shell command that is executed whenever a WAL segment is completed. In our case, we are basically copying the file to the local machine and then using the secure copy command to send it across to the slave.
- max_wal_senders: This parameter specifies the total number of concurrent connections allowed from the slave servers.
- checkpoint_segments: This parameter specifies the maximum number of logfile segments between automatic WAL checkpoints. Once the necessary configuration changes have been made on the master server, we then restart the PostgreSQL server on the master in order to let the new configuration changes take effect. This is done in step 4 of the preceding section. In step 5 of the preceding section, we are basically building the slave by copying the primary server's data directory to the slave.

Now, with the data directory available on the slave, the next step is to configure it. We will now make the necessary parameter replication related parameter changes on the slave in the postgresql.conf directory on the slave server. We set the following parameters on the slave:

- hot_standby: This parameter determines whether you can connect and run queries when the server is in the archive recovery or standby mode. In the next step, we are configuring the recovery.conf file. This is required to be set up so that the slave can start receiving logs from the master. The parameters explained next are configured in the recovery.conf file on the slave.
- standby_mode: This parameter, when enabled, causes PostgreSQL to work as a standby in a replication configuration.
- primary_conninfo: This parameter specifies the connection information used by the slave to connect to the master. For our scenario, our master server is set as 192.168.0.4 on port 5432 and we are using the repuser userid with the password charlie to make a connection to the master. Remember that repuser was the userid which was created in the initial step of the preceding section for this purpose, that is, connecting to the master from the slave.

- trigger_file: When a slave is configured as a standby, it will continue to restore the XLOG records from the master. The trigger_file parameter specifies what is used to trigger a slave, in order to switch over its duties from standby and take over as master or primary server. At this stage, the slave is fully configured now and we can start the slave server; then, the replication process begins. This is shown in step 8 of the preceding section. In steps 9 and 10 of the preceding section, we are simply testing our replication. We first begin by creating a test database, then we log in to the test database and create a table by the name testtable, and then we begin inserting some records into the testtable table. Now, our purpose is to see whether these changes are replicated across the slave. To test this, we log in to the slave on the test database and then query the records from the testtable table, as seen in step 10 of the preceding section. The final result that we see is that all the records that are changed/inserted on the primary server are visible on the slave. This completes our streaming replication's setup and configuration.

You can refer to the following links for more detailed information on streaming replication:

- https://www.digitalocean.com/community/tutorials/how-to-set-up-master-slave-replication-on-postgresql-on-an-ubuntu-12-04-vps
- http://www.rassoc.com/gregr/weblog/2013/02/16/zero-to-postgresql-streaming-replication-in-10-mins/

Replication using Slony-I

Here, we are going to set up replication using Slony-I. We will be setting up the replication of table data between two databases on the same server.

Getting ready

The steps performed in this recipe are carried out on a CentOS Version 6 machine. It is also important to remove the directives related to hot streaming replication prior to setting up replication using Slony-I.

We will first need to install Slony-I. The following steps need to be performed in order to install Slony-I:

1. First, go to http://slony.info/downloads/2.2/source/and download the given software.

2. Once you have downloaded the Slony-I software, the next step is to unzip the .tar file and then go the newly created directory. Before doing this, please ensure that you have the postgresql-devel package for the corresponding PostgreSQL version installed before you install Slony-I:

   ```
   tar xvfj slony1-2.2.3.tar.bz2


   cd slony1-2.2.3
   ```

3. In the next step, we are going to configure, compile, and build the software:

   ```
   ./configure --with-pgconfigdir=/usr/pgsql-9.3/bin/


   make


   make install
   ```

How to do it...

You need to perform the following sequence of steps, in order to replicate data between two tables using Slony-I replication:

1. First, start the PostgreSQL server if you have not already started it:

   ```
   pg_ctl -D $PGDATA start
   ```

2. In the next step, we will be creating two databases, test1 and test2, which will be used as the source and target databases respectively:

   ```
   createdb test1
   createdb test2
   ```

3. In the next step, we will create the t_test table on the source database, test1, and insert some records into it:

   ```
   psql -d test1


   test1=# create table t_test (id numeric primary key, name
   varchar);

   test1=# insert into t_test values(1,'A'),(2,'B'), (3,'C');
   ```

4. We will now set up the target database by copying the table definitions from the `test1` source database:

   ```
   pg_dump -s -p 5432 -h localhost test1 | psql -h localhost -p 5432
   test2
   ```

5. We will now connect to the target database, `test2`, and verify that there is no data in the tables of the `test2` database:

   ```
   psql -d  test2


   test2=# select * from t_test;
   ```

6. We will now set up a `slonik` script for the master-slave, that is source/target, setup. In this scenario, since we are replicating between two different databases on the same server, the only different connection string option will be the database name:

   ```
   cd /usr/pgsql-9.3/bin
   vi init_master.slonik


     #!/bin/sh
     cluster name = mycluster;
     node 1 admin conninfo = 'dbname=test1 host=localhost
   port=5432 user=postgres password=postgres';
     node 2 admin conninfo = 'dbname=test2 host=localhost
   port=5432 user=postgres password=postgres';
     init cluster ( id=1);
     create set (id=1, origin=1);
     set add table(set id=1, origin=1, id=1, fully qualified
   name = 'public.t_test');
     store node (id=2, event node = 1);
     store path (server=1, client=2, conninfo='dbname=test1
   host=localhost port=5432 user=postgres password=postgres');
     store path (server=2, client=1, conninfo='dbname=test2
   host=localhost port=5432 user=postgres password=postgres');
     store listen (origin=1, provider = 1, receiver = 2);
     store listen (origin=2, provider = 2, receiver = 1);
   ```

7. We will now create a `slonik` script for subscription to the slave, that is, target:

   ```
   cd /usr/pgsql-9.3/bin
   vi init_slave.slonik
    #!/bin/sh
    cluster name = mycluster;
    node 1 admin conninfo = 'dbname=test1 host=localhost
   port=5432 user=postgres password=postgres';
    node 2 admin conninfo = 'dbname=test2 host=localhost
   port=5432 user=postgres password=postgres';
    subscribe set ( id = 1, provider = 1, receiver = 2, forward
   = no);
   ```

8. We will now run the `init_master.slonik` script created in step 6 and run this on the master, as follows:

   ```
   cd /usr/pgsql-9.3/bin

   slonik init_master.slonik
   ```

9. We will now run the `init_slave.slonik` script created in step 7 and run this on the slave, that is, target:

   ```
   cd /usr/pgsql-9.3/bin

   slonik init_slave.slonik
   ```

10. In the next step, we will start the master `slon` daemon:

    ```
    nohup slon mycluster "dbname=test1 host=localhost port=5432
    user=postgres password=postgres" &
    ```

11. In the next step, we will start the slave `slon` daemon:

    ```
    nohup slon mycluster "dbname=test2 host=localhost port=5432
    user=postgres password=postgres" &
    ```

12. Next, we will connect to the master, that is, the `test1` source database, and insert some records in the `t_test` table:

    ```
    psql -d test1

    test1=# insert into t_test values (5,'E');
    ```

13. We will now test for the replication by logging on to the slave, that is, the test2 target database, and see whether the inserted records in the t_test table are visible:

    ```
    psql -d test2
    ```

    ```
    test2=# select * from t_test;
     id | name
    ----+------
      1 | A
      2 | B
      3 | C
      5 | E
    (4 rows)
    ```

How it works...

We will now discuss the steps performed in the preceding section:

- In step 1, we first start the PostgreSQL server if it is not already started. In step 2, we create two databases, namely test1 and test2, that will serve as our source (master) and target (slave) databases.
- In step 3, we log in to the test1 source database, create a t_test table, and insert some records into the table.
- In step 4, we set up the target database, test2, by copying the table definitions present in the source database and loading them into test2 using the pg_dump utility.
- In step 5, we log in to the target database, test2, and verify that there are no records present in the t_test table because in step 4, we only extracted the table definitions into the test2 database from the test1 database.
- In step 6, we set up a slonik script for the master-slave replication setup. In the init_master.slonik file, we first define the cluster name as mycluster. We then define the nodes in the cluster. Each node will have a number associated with a connection string, which contains database connection information. The node entry is defined both for the source and target databases. The store_path commands are necessary, so that each node knows how to communicate with the other.
- In step 7, we set up a slonik script for the subscription of the slave, that is, the test2 target database. Once again, the script contains information such as the cluster name and the node entries that are designated a unique number related to connection string information. It also contains a subscriber set.
- In step 8, we run the init_master.slonik file on the master. Similarly, in step 9, we run the init_slave.slonik file on the slave.

- In step 10, we start the master slon daemon. In step 11, we start the slave slon daemon.
- The subsequent steps, 12 and 13, are used to test for replication. For this purpose, in step 12 of the preceding section, we first log in to the test1 source database and insert some records into the t_test table. To check whether the newly inserted records have been replicated in the target database, test2, we log in to the test2 database in step 13. The result set obtained from the output of the query confirms that the changed/inserted records on the t_test table in the test1 database are successfully replicated across the target database, test2.

For more information on Slony-I replication, go to http://slony.info/documentation/tutorial.html.

There's more...

If you are using Slony-I for replication between two different servers, in addition to the steps mentioned in the *How to do it...* section, you will also have to enable authentication information in the pg_hba.conf file existing on both the source and target servers. For example, let's assume that the source server's IP is 192.168.16.44 and the target server's IP is 192.168.16.56 and we are using a user named super to replicate the data.

If this is the situation, then in the source server's pg_hba.conf file, we will have to enter the information, as follows:

```
host          postgres          super      192.168.16.44/32            md5
```

Similarly, in the target server's pg_hba.conf file, we will have to enter the authentication information, as follows:

```
host          postgres          super      192.168.16.56/32            md5
```

Also, in the shell scripts that were used for Slony-I, wherever the connection information for the host is localhost that entry will need to be replaced by the source and target server's IP addresses.

Replication using Londiste

In this recipe, we are going to show you how to replicate data using Londiste.

Getting ready

For this setup, we are using the same host CentOS Linux machine to replicate data between two databases. This can also be set up using two separate Linux machines running on VMware, VirtualBox, or any other virtualization software. It is assumed that the latest version of PostgreSQL, version 9.3, is installed. We used CentOS Version 6 as the Linux operating system for this exercise.

To set up Londiste replication on the Linux machine, perform the following steps:

1. Go to `http://pgfoundry.org/projects/skytools/` and download the latest version of Skytools 3.2, that is, `tarball skytools-3.2.tar.gz`.
2. Extract the tarball file, as follows:

   ```
   tar -xvzf skytools-3.2.tar.gz
   ```

3. Go to the new location and build and compile the software:

   ```
   cd skytools-3.2
   ./configure  --prefix=/var/lib/pgsql/9.3/Sky –with-pgconfig=/usr/
   pgsql-9.3/bin/pg_config


   make


   make install
   ```

4. Also, set the `PYTHONPATH` environment variable, as shown here. Alternatively, you can also set it in the `.bash_profile` script:

   ```
   export PYTHONPATH=/opt/PostgreSQL/9.2/Sky/lib64/python2.6/site-
   packages/
   ```

How to do it...

1. We are going to perform the following sequence of steps to set up replication between two different databases using Londiste. First, create the two databases between which replication has to occur:

   ```
    createdb node1
    createdb node2
   ```

2. Populate the `node1` database with data using the `pgbench` utility:

   ```
   pgbench -i -s 2 -F 80 node1
   ```

3. Add any primary key and foreign keys to the tables in the node1 database that are needed for replication. Create the following .sql file and add the following lines to it:

   ```
   Vi /tmp/prepare_pgbenchdb_for_londiste.sql -- add primary key to
   history table
   ```

   ```
   ALTER TABLE pgbench_history ADD COLUMN hid SERIAL PRIMARY KEY;

   -- add foreign keys
   ALTER TABLE pgbench_tellers ADD CONSTRAINT pgbench_tellers_
   branches_fk FOREIGN KEY(bid) REFERENCES pgbench_branches;
   ALTER TABLE pgbench_accounts ADD CONSTRAINT pgbench_accounts_
   branches_fk FOREIGN KEY(bid) REFERENCES pgbench_branches;
   ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_
   branches_fk FOREIGN KEY(bid) REFERENCES pgbench_branches;
   ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_
   tellers_fk FOREIGN KEY(tid) REFERENCES pgbench_tellers;
   ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_
   accounts_fk FOREIGN KEY(aid) REFERENCES pgbench_accounts;
   ```

4. We will now load the .sql file created in the previous step and load it into the database:

   ```
   psql node1 -f /tmp/prepare_pgbenchdb_for_londiste.sql
   ```

5. We will now populate the node2 database with table definitions from the tables in the node1 database:

   ```
   pg_dump -s -t 'pgbench*' node1 > /tmp/tables.sql
   psql  -f /tmp/tables.sql node2
   ```

6. Now starts the process of replication. We will first create the londiste.ini configuration file with the following parameters in order to set up the root node for the source database, node1:

   ```
   Vi londiste.ini

   [londiste3]
   job_name = first_table
   db = dbname=node1
   queue_name = replication_queue
   logfile = /home/postgres/log/londiste.log
   pidfile = /home/postgres/pid/londiste.pid
   ```

7. In the next step, we are going to use the londiste.ini configuration file created in the previous step to set up the root node for the node1 database, as shown here:

   ```
   [postgres@localhost bin]$ ./londiste3 londiste3.ini create-root
   node1 dbname=node1


   2014-12-09 18:54:34,723 2335 WARNING No host= in public connect
   string, bad idea
   2014-12-09 18:54:35,210 2335 INFO plpgsql is installed
   2014-12-09 18:54:35,217 2335 INFO pgq is installed
   2014-12-09 18:54:35,225 2335 INFO pgq.get_batch_cursor is
   installed
   2014-12-09 18:54:35,227 2335 INFO pgq_ext is installed
   2014-12-09 18:54:35,228 2335 INFO pgq_node is installed
   2014-12-09 18:54:35,230 2335 INFO londiste is installed
   2014-12-09 18:54:35,232 2335 INFO londiste.global_add_table is
   installed
   2014-12-09 18:54:35,281 2335 INFO Initializing node
   2014-12-09 18:54:35,285 2335 INFO Location registered
   2014-12-09 18:54:35,447 2335 INFO Node "node1" initialized for
   queue "replication_queue" with type "root"
   2014-12-09 18:54:35,465 2335 INFO Don
   ```

8. We will now run the worker daemon for the root node:

   ```
   [postgres@localhost bin]$ ./londiste3 londiste3.ini worker
   2014-12-09 18:55:17,008 2342 INFO Consumer uptodate = 1
   ```

9. In the next step, we will create a slave.ini configuration file in order to make a leaf node for the node2 target database:

   ```
   Vi slave.ini
   [londiste3]
   job_name = first_table_slave
   db = dbname=node2
   queue_name = replication_queue
   logfile = /home/postgres/log/londiste_slave.log
   pidfile = /home/postgres/pid/londiste_slave.pid
   ```

10. We will now initialize the node in the target database:

    ```
     ./londiste3 slave.ini create-leaf node2 dbname=node2 -
    provider=dbname=node1
    2014-12-09 18:57:22,769 2408 WARNING No host= in public connect
    string, bad idea
    2014-12-09 18:57:22,778 2408 INFO plpgsql is installed
    2014-12-09 18:57:22,778 2408 INFO Installing pgq
    2014-12-09 18:57:22,778 2408 INFO   Reading from /var/lib/
    pgsql/9.3/Sky/share/skytools3/pgq.sql
    2014-12-09 18:57:23,211 2408 INFO pgq.get_batch_cursor is
    installed
    2014-12-09 18:57:23,212 2408 INFO Installing pgq_ext
    2014-12-09 18:57:23,213 2408 INFO   Reading from /var/lib/
    pgsql/9.3/Sky/share/skytools3/pgq_ext.sql
    2014-12-09 18:57:23,454 2408 INFO Installing pgq_node
    2014-12-09 18:57:23,455 2408 INFO   Reading from /var/lib/
    pgsql/9.3/Sky/share/skytools3/pgq_node.sql
    2014-12-09 18:57:23,729 2408 INFO Installing londiste
    2014-12-09 18:57:23,730 2408 INFO   Reading from /var/lib/
    pgsql/9.3/Sky/share/skytools3/londiste.sql
    2014-12-09 18:57:24,391 2408 INFO londiste.global_add_table is
    installed
    2014-12-09 18:57:24,575 2408 INFO Initializing node
    2014-12-09 18:57:24,705 2408 INFO Location registered
    2014-12-09 18:57:24,715 2408 INFO Location registered
    2014-12-09 18:57:24,744 2408 INFO Subscriber registered: node2
    2014-12-09 18:57:24,748 2408 INFO Location registered
    2014-12-09 18:57:24,750 2408 INFO Location registered
    2014-12-09 18:57:24,757 2408 INFO Node "node2" initialized for
    queue "replication_queue" with type "leaf"
    2014-12-09 18:57:24,761 2408 INFO Done
    ```

11. We will now launch the `worker` daemon for the target database, that is, `node2`:

    ```
    [postgres@localhost bin]$ ./londiste3 slave.ini worker
    2014-12-09 18:58:53,411 2423 INFO Consumer uptodate = 1
    ```

12. We will now create the configuration file, that is `pgqd.ini`, for the ticker daemon:

```
vi pgqd.ini

[pgqd]
logfile = /home/postgres/log/pgqd.log
pidfile = /home/postgres/pid/pgqd.pid
```

13. Using the configuration file created in the previous step, we will launch the ticker daemon:

```
[postgres@localhost bin]$ ./pgqd pgqd.ini
2014-12-09 19:05:56.843 2542 LOG Starting pgqd 3.2
2014-12-09 19:05:56.844 2542 LOG auto-detecting dbs ...
2014-12-09 19:05:57.257 2542 LOG node1: pgq version ok: 3.2
2014-12-09 19:05:58.130 2542 LOG node2: pgq version ok: 3.2
```

14. We will now add all the tables to the replication on the root node:

```
[postgres@localhost bin]$ ./londiste3 londiste3.ini add-table
--all
2014-12-09 19:07:26,064 2614 INFO Table added: public.pgbench_
accounts
2014-12-09 19:07:26,161 2614 INFO Table added: public.pgbench_
branches
2014-12-09 19:07:26,238 2614 INFO Table added: public.pgbench_
history
2014-12-09 19:07:26,287 2614 INFO Table added: public.pgbench_
tellers
```

15. Similarly, add all the tables to the replication on the leaf node:

```
[postgres@localhost bin]$ ./londiste3 slave.ini add-table -all
```

16. We will now generate some traffic on the `node1` source database:

```
pgbench -T 10 -c 5 node1
```

17. We will now use the `compare` utility available with the `londiste3` command to check the tables in both the nodes; that is, both the source database (`node1`) and destination database (`node2`) have the same amount of data:

```
[postgres@localhost bin]$ ./londiste3 slave.ini compare

2014-12-09 19:26:16,421 2982 INFO Checking if node1 can be used
for copy
```

```
2014-12-09 19:26:16,424 2982 INFO Node node1 seems good source,
using it
2014-12-09 19:26:16,425 2982 INFO public.pgbench_accounts: Using
node node1 as provider
2014-12-09 19:26:16,441 2982 INFO Provider: node1 (root)
2014-12-09 19:26:16,446 2982 INFO Locking public.pgbench_accounts
2014-12-09 19:26:16,447 2982 INFO Syncing public.pgbench_accounts
2014-12-09 19:26:18,975 2982 INFO Counting public.pgbench_accounts
2014-12-09 19:26:19,401 2982 INFO srcdb: 200000 rows,
checksum=167607238449
2014-12-09 19:26:19,706 2982 INFO dstdb: 200000 rows,
checksum=167607238449
2014-12-09 19:26:19,715 2982 INFO Checking if node1 can be used
for copy
2014-12-09 19:26:19,716 2982 INFO Node node1 seems good source,
using it
2014-12-09 19:26:19,716 2982 INFO public.pgbench_branches: Using
node node1 as provider
2014-12-09 19:26:19,730 2982 INFO Provider: node1 (root)
2014-12-09 19:26:19,734 2982 INFO Locking public.pgbench_branches
2014-12-09 19:26:19,734 2982 INFO Syncing public.pgbench_branches
2014-12-09 19:26:22,772 2982 INFO Counting public.pgbench_branches
2014-12-09 19:26:22,804 2982 INFO srcdb: 2 rows,
checksum=-3078609798
2014-12-09 19:26:22,812 2982 INFO dstdb: 2 rows,
checksum=-3078609798
2014-12-09 19:26:22,866 2982 INFO Checking if node1 can be used
for copy
2014-12-09 19:26:22,877 2982 INFO Node node1 seems good source,
using it
2014-12-09 19:26:22,878 2982 INFO public.pgbench_history: Using
node node1 as provider
2014-12-09 19:26:22,919 2982 INFO Provider: node1 (root)
2014-12-09 19:26:22,931 2982 INFO Locking public.pgbench_history
2014-12-09 19:26:22,932 2982 INFO Syncing public.pgbench_history
2014-12-09 19:26:25,963 2982 INFO Counting public.pgbench_history
2014-12-09 19:26:26,008 2982 INFO srcdb: 715 rows,
checksum=9467587272
```

```
2014-12-09 19:26:26,020 2982 INFO dstdb: 715 rows,
checksum=9467587272

2014-12-09 19:26:26,056 2982 INFO Checking if node1 can be used
for copy

2014-12-09 19:26:26,063 2982 INFO Node node1 seems good source,
using it

2014-12-09 19:26:26,064 2982 INFO public.pgbench_tellers: Using
node node1 as provider

2014-12-09 19:26:26,100 2982 INFO Provider: node1 (root)

2014-12-09 19:26:26,108 2982 INFO Locking public.pgbench_tellers

2014-12-09 19:26:26,109 2982 INFO Syncing public.pgbench_tellers

2014-12-09 19:26:29,144 2982 INFO Counting public.pgbench_tellers

2014-12-09 19:26:29,176 2982 INFO srcdb: 20 rows,
checksum=4814381032

2014-12-09 19:26:29,182 2982 INFO dstdb: 20 rows,
checksum=4814381032
```

How it works…

The following is an explanation of the steps performed in the preceding section:

- Initially, in step 1, we create two databases, that is `node1` and `node2`, that are used as the source and target databases, respectively, from a replication perspective.
- In step 2, we populate the `node1` database using the `pgbench` utility.
- In step 3 of the preceding section, we add and define the respective primary key and foreign key relationships on different tables and put these DDL commands in a `.sql` file.
- In step 4, we execute these DDL commands stated in step 3 on the `node1` database; thus, in this way, we force the primary key and foreign key definitions on the tables in the `pgbench` schema in the `node1` database.
- In step 5, we extract the table definitions from the tables in the `pgbench` schema in the `node1` database and load these definitions in the `node2` database. We will now discuss steps 6 to 8 of the preceding section.
- In step 6, we create the configuration file, which is then used in step 7 to create the root node for the `node1` source database.
- In step 8, we will launch the `worker` daemon for the root node. Regarding the entries mentioned in the configuration file in step 6, we first define a job that must have a name, so that distinguished processes can be easily identified. Then, we define a connect string with information to connect to the source database, that is `node1`, and then we define the name of the replication queue involved. Finally, we define the location of the `log` and `pid` files.

- We will now discuss steps 9 to 11 of the preceding section. In step 9, we define the configuration file, which is then used in step 10 to create the leaf node for the target database, that is node2.
- In step 11, we launch the worker daemon for the leaf node. The entries in the configuration file in step 9 contain the job_name connect string in order to connect to the target database, that is node2, the name of the replication queue involved, and the location of log and pid involved. The key part in step 11 is played by the slave, that is the target database—to find the master or provider, that is source database node1.
- We will now talk about steps 12 and 13 of the preceding section. In step 12, we define the ticker configuration, with the help of which we launch the ticker process mentioned in step 13. Once the ticker daemon has started successfully, we have all the components and processes setup and needed for replication; however, we have not yet defined what the system needs to replicate.
- In step 14 and 15, we define the tables to the replication that is set on both the source and target databases, that is node1 and node2, respectively.
- Finally, we will talk about steps 16 and 17 of the preceding section. Here, at this stage, we are testing the replication that was set up between the node1 source database and the node2 target database.
- In step 16, we generate some traffic on the node1 source database by running pgbench with five parallel database connections and generating traffic for 10 seconds.
- In step 17, we check whether the tables on both the source and target databases have the same data. For this purpose, we use the compare command on the provider and subscriber nodes and then count and checksum the rows on both sides. A partial output from the preceding section tells you that the data has been successfully replicated between all the tables that are part of the replication set up between the node1 source database and the node2 destination database, as the count and checksum of rows for all the tables on the source and target destination databases are matching:

```
2014-12-09 19:26:18,975 2982 INFO Counting public.pgbench_accounts
2014-12-09 19:26:19,401 2982 INFO srcdb: 200000 rows,
checksum=167607238449
2014-12-09 19:26:19,706 2982 INFO dstdb: 200000 rows,
checksum=167607238449


2014-12-09 19:26:22,772 2982 INFO Counting public.pgbench_branches
2014-12-09 19:26:22,804 2982 INFO srcdb: 2 rows,
checksum=-3078609798
```

```
2014-12-09 19:26:22,812 2982 INFO dstdb: 2 rows,
checksum=-3078609798


2014-12-09 19:26:25,963 2982 INFO Counting public.pgbench_history
2014-12-09 19:26:26,008 2982 INFO srcdb: 715 rows,
checksum=9467587272
2014-12-09 19:26:26,020 2982 INFO dstdb: 715 rows,
checksum=9467587272


2014-12-09 19:26:29,144 2982 INFO Counting public.pgbench_tellers
2014-12-09 19:26:29,176 2982 INFO srcdb: 20 rows,
checksum=4814381032
2014-12-09 19:26:29,182 2982 INFO dstdb: 20 rows,
checksum=4814381032
```

Check out the following links for more information on Londiste replication:

https://wiki.postgresql.org/wiki/Londiste_Tutorial_(Skytools_2)

http://manojadinesh.blogspot.in/2012/11/skytools-londiste-replication.html

Replication using Bucardo

In this recipe, we are going to show you the replication between two databases using Bucardo.

Getting ready

This exercise is carried out on a Red Hat Linux machine.

Install the EPEL package for your Red Hat platform from https://fedoraproject.org/wiki/EPEL.

Then, install these RPMs with the following yum command:

```
yum install perl-DBI perl-DBD-Pg perl-DBIx-Safe
```

If it is not already installed, download the PostgreSQL repository from http://yum.pgrpms.org/repopackages.php.

After this, install the following package; this is required because Bucardo is written in Perl:

```
yum install postgresql93-plperl
```

- To install Bucardo, download the latest version of Bucardo, which is Bucardo Version 5.2.0, from `http://bucardo.org/wiki/Bucardo`.
- Extract from the tarball file, go to the newly downloaded location, and compile and build the software:

  ```
  tar xvfz Bucardo-5.2.0.tar.gz

  cd Bucardo-5.2.0

  perl Makefile.PL

  make

  make install
  ```

How to do it...

The following is the complete sequence of steps that are used to configure replication between two databases using Bucardo:

1. The first step is to install `bucardo`; that is, create the main `bucardo` database containing the information that the Bucardo daemon will need:

   ```
   [postgres@localhost ~]$ bucardo install --batch --quiet
   ```

2. Create the `bucardo` superuser. In the next step, we create the source and target databases, that is, `gamma1` and `gamma2` respectively, between which the replication needs to be set up:

   ```
   [postgres@localhost ~]$ psql -qc 'create database gamma1'

   psql -d gamma1-qc 'create table t1 (id serial primary key, email
   text)'

   [postgres@localhost ~]$ psql -qc 'create database gamma2 template
   gamma1'
   ```

3. In the next step, we inform Bucardo about the databases that will be involved in the replication:

   ```
   postgres@localhost ~]$ bucardo add db db1 dbname=gamma1
   Added database "db1"

   [postgres@localhost ~]$ bucardo add db db2 dbname=gamma2
   Added database "db2"
   ```

4. Next, we create a herd `myherd` and include those tables from the source databases that will be part of the replication setup:

   ```
   [postgres@localhost ~]$ bucardo add herd myherd t1

   Created relgroup "myherd"
   Added the following tables or sequences:
     public.t1 (DB: db1)
   The following tables or sequences are now part of the relgroup
   "myherd":
     public.t1
   ```

5. In the next step, we create a `source` sync:

   ```
   [postgres@localhost ~]$ bucardo add sync beta herd=myherd
   dbs=db1:source
   Added sync "beta"
   Created a new dbgroup named "beta"
   ```

6. Then, we create a `target` sync:

   ```
   [postgres@localhost ~]$ bucardo add sync charlie herd=myherd
   dbs=db1:source,db2:target
   Added sync "charlie"
   Created a new dbgroup named "charlie"
   ```

7. At this stage, we have the replication procedure set up, so the next step is to start the Bucardo service:

   ```
   [postgres@localhost ~]$ bucardo start
   Checking for existing processes
   Removing file "pid/fullstopbucardo"
   Starting Bucardo
   ```

8. The next step is to test the replication setup. For this purpose, we are going to insert some records in the `t1` table on the `gamma1` source database:

   ```
   psql  -d  gamma1


   gamma1=# insert into t1 values (1,'wallsingh@gmail.com');
   INSERT 0 1
   gamma1=# insert into t1 values (2,'neha.verma@gmail.com');
   INSERT 0 1
   ```

9. Now that we have inserted some records in the source database in the previous step, we need to check whether these changes have been replicated in the `gamma2` target database:

   ```
   psql  -d  gamma2


   gamma2=# select * from t1;
    id |        email
   ----+----------------------
     1 | wallsingh@gmail.com
     2 | neha.verma@gmail.com
   (2 rows)
   ```

How it works...

The following is a description of the steps mentioned in the preceding section:

- In step 1 of the preceding section, we first create the `bucardo` database that will contain information about the `bucardo` daemon and will also create a superuser by the name `bucardo`.
- In step 2, we create our source and target databases for replication, that is, `gamma1` and `gamma2`, respectively. We also create the `t1` table on the `gamma1` database that will be used for replication.
- In step 3, we tell Bucardo about the source and target databases, that is, `gamma1` and `gamma2`, respectively that will be involved in the replication.
- In step 4, we create a herd by the name `myherd` and include the `t1` table from the `gamma1` source database that will be part of the the replication setup. Any changes made to this table should be replicated from the source to the target databases.

- In steps 5 and 6 of the preceding section, we basically create a source and a target sync, which will replicate the t1 table in the myherd herd and replicate it from the source database db1, that is gamma1, to the target database db2, that is gamma2. With the replication set up configured, we then start the Bucardo service in step 7 of the preceding section.
- We test the replication setup in steps 8 and 9 of the preceding section. In step 8, we insert some records in the t1 table on the gamma1 database and in step 9, we login to the gamma2 database and check whether the newly inserted records in the t1 table on the gamma1 database are replicated across the gamma2 database. The result set of the SELECT query from the t1 table in the gamma2 database confirms that the inserted records in the gamma1 database have been successfully replicated in the gamma2 database.

You can refer to the following links for information on Bucardo replication:

- http://blog.pscs.co.uk/postgresql-replication-and-bucardo/
- http://blog.endpoint.com/2014/06/bucardo-5-multimaster-postgres-released.html

Replication using DRBD

In this recipe, we are going to cover block-level replication using DRBD for PostgreSQL.

Getting ready

A working Linux machine is required for this setup. This setup requires network interfaces and a cluster IP. These steps are carried out in a CentOS Version 6 machine. Having covered the PostgreSQL setup in the previous chapters, it is assumed that the necessary packages and prerequisites are already installed.

We will be using the following setup in our hierarchy:

- Node1.example.org uses the LAN's IP address 10.0.0.181 and uses 172.16.0.1 for crossovers
- Node2.example.org uses the LAN's IP address 10.0.0.182 and the IP address 172.16.0.2 for crossovers
- dbip.example.org uses the cluster IP address 10.0.0.180

How to do it...

Perform the following sequence of steps for block-level replication using DRBD:

1. First, temporarily disable SELINUX, set SELINUX to disabled, and then save the file:

   ```
   vi /etc/selinux/config


   SELINUX=disabled
   ```

2. In this step, change the hostname and gateway for both the nodes, that is, network interfaces:

   ```
   vi /etc/sysconfig/network

   # For node  1

   NETWORKING=yes
   NETWORKING_IPV6=no
   HOSTNAME=node1.example.org
   GATEWAY=10.0.0.2


   #For node 2

   NETWORKING=yes
   NETWORKING_IPV6=no
   HOSTNAME=node2.example.org
   GATEWAY=10.0.0.2
   ```

3. In this step, we need to configure the network interfaces for the first node, that is, node1:

   - We first configure the first node1 database:

     ```
     vi /etc/sysconfig/network-scripts/ifcfg-eth0

     DEVICE=eth0
     BOOTPROTO=static
     IPADDR=10.0.0.181
     NETMASK=255.255.255.0
     ONBOOT=yes
     HWADDR=a2:4e:7f:64:61:24
     ```

- We then configure the crossover/DRBD interface for `node1`:

  ```
  vi /etc/sysconfig/network-scripts/ifcfg-eth1
  DEVICE=eth1
  BOOTPROTO=static
  IPADDR=172.16.0.1
  NETMASK=255.255.255.0
  ONBOOT=yes
  HWADDR=ee:df:ff:4a:5f:68
  ```

4. In this step, we configure the network interfaces for the second node, that is, `node2`:

   ```
   vi /etc/sysconfig/network-scripts/ifcfg-eth0
   DEVICE=eth0
   BOOTPROTO=static
   IPADDR=10.0.0.182
   NETMASK=255.255.255.0
   ONBOOT=yes
   HWADDR=22:42:b1:5a:42:6f
   ```

   - We then configure the crossover/DRBD interface for `node2`:

     ```
     vi /etc/sysconfig/network-scripts/ifcfg-eth1
     DEVICE=eth1
     BOOTPROTO=static
     IPADDR=172.16.0.2
     NETMASK=255.255.255.0
     ONBOOT=yes
     HWADDR=6a:48:d2:70:26:5e
     ```

5. In this step, we will configure DNS:

   ```
   vi /etc/resolv.conf
   search example.org
   nameserver 10.0.0.2
   ```

   - Also, configure a basic hostname resolution:

     ```
     vi /etc/hosts
     127.0.0.1             localhost.localdomain localhost
     10.0.0.181            node1.example.org     node1
     10.0.0.182            node2.example.org     node2
     10.0.0.180            dbip.example.org      node2
     ```

6. In this step, we will check the network connectivity between the nodes:

   - First, we will ping `node2` from `node1`, first through the LAN interface and then through the crossover IP:

     ```
     root@node1 ~]# ping -c 2 node2
     PING node2 (10.0.0.182) 56(84) bytes of data.
     64 bytes from node2 (10.0.0.182): icmp_seq=1 ttl=64 time=0.089 ms
     64 bytes from node2 (10.0.0.182): icmp_seq=2 ttl=64 time=0.082 ms
     --- node2 ping statistics ---
     2 packets transmitted, 2 received, 0% packet loss, time 999ms
     rtt min/avg/max/mdev = 0.082/0.085/0.089/0.009 ms
     [root@node1 ~]# ping -c 2 172.16.0.2
     PING 172.16.0.2 (172.16.0.2) 56(84) bytes of data.
     64 bytes from 172.16.0.2: icmp_seq=1 ttl=64 time=0.083 ms
     64 bytes from 172.16.0.2: icmp_seq=2 ttl=64 time=0.083 ms
     --- 172.16.0.2 ping statistics ---
     2 packets transmitted, 2 received, 0% packet loss, time 999ms
     rtt min/avg/max/mdev = 0.083/0.083/0.083/0.000 ms
     ```

   - Now, we will ping `node1` from `node2`, first via the LAN interfaces and then through the crossover IP:

     ```
     [root@node2 ~]# ping -c 2 node1
     PING node1 (10.0.0.181) 56(84) bytes of data.
     64 bytes from node1 (10.0.0.181): icmp_seq=1 ttl=64 time=0.068 ms
     64 bytes from node1 (10.0.0.181): icmp_seq=2 ttl=64 time=0.063 ms
     --- node1 ping statistics ---
     2 packets transmitted, 2 received, 0% packet loss, time 999ms
     rtt min/avg/max/mdev = 0.063/0.065/0.068/0.008 ms
     ```

- Next, we will ping `node1` through the crossover interface:

  ```
  [root@node2 ~]# ping -c 2 172.16.0.1
  PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.
  64 bytes from 172.16.0.1: icmp_seq=1 ttl=64 time=1.36 ms
  64 bytes from 172.16.0.1: icmp_seq=2 ttl=64 time=0.075 ms
  --- 172.16.0.1 ping statistics ---
  2 packets transmitted, 2 received, 0% packet loss, time 
  1001ms
  rtt min/avg/max/mdev = 0.075/0.722/1.369/0.647 ms
  ```

7. Install the necessary packages:

   ```
   yum install -y  drbd83 kmod-drbd83
   ```

8. In this step, configure DRBD on both the nodes:

   ```
   vi /etc/drbd.conf
   ```

   ```
   global {
       usage-count no;
   }
   common {
       syncer { rate 100M; }
       protocol      C;
   }
   resource postgres {
       startup {
          wfc-timeout 0;
          degr-wfc-timeout
          120;
       }
       disk { on-io-error detach; }
       on node1.example.org {
          device      /dev/drbd0;
          disk        /dev/sda5;
          address     172.16.0.1:7791;
          meta-disk   internal;
       }
       on node2.example.org {
          device      /dev/drbd0;
          disk        /dev/sda5;
          address     172.16.0.2:7791;
          meta-disk   internal;
       }
   }
   ```

9. Once the `drbd.conf` file is set up for both the nodes, we then write metadata on the `postgres` resource. Execute the following step on both the nodes:

```
[root@node1 ~]# drbdadm create-md postgres
Writing meta data...
initializing activity log
NOT initialized bitmap
New drbd meta data block successfully created.

root@node2 ~]# drbdadm create-md postgres
Writing meta data...
initializing activity log
NOT initialized bitmap
New drbd meta data block successfully created.
```

10. In this step, we will bring up the resource. Execute the following command on both the nodes:

```
drbdadm up postgres
```

11. In this step, we can make the initial sync between the nodes. This step can be performed on the primary node, and we set `node1` as the primary node:

```
drbdadm -- --overwrite-data-of-peer primary postgres
```

12. To monitor the progress of the sync and the status of the DRBD resource, take a look at the `/proc/drbd` file:

```
[root@node1 ~]# cat /proc/drbd
version: 8.3.8 (api:88/proto:86-94)
GIT-hash: d78846e52224fd00562f7c225bcc25b2d422321d build by
mockbuild@builder10.centos.org, 2014-10-04 14:04:09
0: cs:SyncSource ro:Primary/Secondary ds:UpToDate/Inconsistent C
r----
ns:48128 nr:0 dw:0 dr:48128 al:0 bm:2 lo:0 pe:0 ua:0 ap:0 ep:1
wo:b oos:8340188
[>....................] sync'ed:  0.6% (8144/8188)M delay_probe: 7
finish: 0:11:29 speed: 12,032 (12,032) K/sec
```

13. Once the sync process is complete, we can take a look at both the statuses of the `postgres` resource on both the nodes:

```
[root@node1 ~]# cat /proc/drbd
version: 8.3.8 (api:88/proto:86-94)
```

```
GIT-hash: d78846e52224fd00562f7c225bcc25b2d422321d build by
mockbuild@builder10.centos.org, 2014-10-04 14:04:09
0: cs:Connected ro:Primary/Secondary ds:UpToDate/UpToDate C r----
ns:8388316 nr:0 dw:0 dr:8388316 al:0 bm:512 lo:0 pe:0 ua:0 ap:0
ep:1 wo:b oos:0


[root@node2 ~]# cat /proc/drbd
version: 8.3.8 (api:88/proto:86-94)
GIT-hash: d78846e52224fd00562f7c225bcc25b2d422321d build by
mockbuild@builder10.centos.org, 2014-10-04 14:04:09
0: cs:Connected ro:Secondary/Primary ds:UpToDate/UpToDate C r----
ns:0 nr:8388316 dw:8388316 dr:0 al:0 bm:512 lo:0 pe:0 ua:0 ap:0
ep:1 wo:b oos:0
```

14. In this step, we are going to initiate DRBD services. On both the nodes, issue the following command:

    ```
    /etc/init.d/drbd start
    ```

15. In order to initialize the data directory and set up using DRBD, we will have to format and mount the DRBD device. Then, we initialize the data directory:

    - Issue the following commands on `node1`:

      ```
      mkfs.ext4 /dev/drbd0


      mount -t ext4 /dev/drbd0 /var/lib/pgsql/9.3


      chown postgres.postgres /var/lib/pgsql/9.3
      ```

    - Next, log in as the `postgres` user on `node1` and initialize the database:

      ```
      su - postgres
      initdb /var/lib/pgsql/9.3/data
      exit
      ```

16. In this step, we enable trusted authentication, and we will configure the parameters required to set up PostgreSQL in the `postgresql.conf` file.

    - On `node1`, execute the following steps:

      ```
      echo "host  all   all   10.0.0.181/32   trust" >> /var/lib/
      pgsql/9.3/data/pg_hba.conf
      echo "host  all   all   10.0.0.182/32   trust" >> /var/lib/
      pgsql/9.3/data/pg_hba.conf
      echo "host  all   all   10.0.0.180/32   trust" >> /var/lib/
      pgsql/9.3/data/pg_hba.conf
      ```

- Then, we configure the necessary parameters in the `postgresql.conf` file.

  ```
  vi /var/lib/pgsql/9.3/data/postgresql.conf
  listen_addresses = '*'
  ```

17. Once the previously mentioned parameters have been changed in the `postgresql.conf` file, the next step will be to start PostgreSQL. Execute the following command on `node1`:

    ```
    service postgresql-9.3 start
    ```

18. We will then create an admin user to manage PostgreSQL. On `node1`, execute the following command and when prompted for a password, you can choose any. However, for the sake of clarity of this exercise, we will use the `admin` keyword itself as the password:

    ```
    su - postgres
    createuser --superuser admin  --pwprompt
    ```

19. In this step, we will create a database and populate it with data. On `node1`, execute the following steps and then access the database:

    ```
    su – postgres

    createdb test

    pgbench -test

    pgbench -i test

    psql -U admin -d test


    test=# select * from pgbench_tellers;

    tid | bid | tbalance | filler
    -----+-----+----------+--------
      1 |   1 |        0 |
      2 |   1 |        0 |
      3 |   1 |        0 |
      4 |   1 |        0 |
      5 |   1 |        0 |
      6 |   1 |        0 |
      7 |   1 |        0 |
    ```

```
   8 |    1 |          0 |
   9 |    1 |          0 |
  10 |    1 |          0 |
(10 registros)
```

20. In this step, we will test the block-level replication and see whether PostgreSQL works on `node2`. On `node1`, execute the following commands:

    - We will first stop PostgreSQL on `node1`:

      ```
      service postgresql-9.3 stop
      ```

    - Then, we will unmount the DRBD device on `node1`:

      ```
      umount /dev/drbd0
      ```

    - Now, we will set up `node1` as the secondary node:

      ```
      drbdadm secondary postgres
      ```

    - Next, we will configure `node2` as the primary node:

      ```
      drbdadm primary postgres
      ```

    - In this step, mount the DRBD device:

      ```
      mount -t ext3 /dev/drbd0 /var/lib/pgsql/9.3
      ```

    - Then, we start the `postgresql` service on `node2`:

      ```
      service postgresql-9.3 start
      ```

    - Now, we will see whether we are able to access the `test` database on `node2`:

      ```
      psql -u admin -d test
      ```

      ```
      test=# select * from pgbench_tellers;

      tid | bid | tbalance | filler
      -----+-----+----------+--------
        1 |   1 |        0 |
        2 |   1 |        0 |
        3 |   1 |        0 |
        4 |   1 |        0 |
        5 |   1 |        0 |
        6 |   1 |        0 |
        7 |   1 |        0 |
        8 |   1 |        0 |
        9 |   1 |        0 |
       10 |   1 |        0 |
      (10 registros)
      ```

How it works...

In the initial steps, from steps 1 to 6, we configure the nodes, that is node1 and node2, set up the network connectivity, and configure DNS. In step 6, we do the network connectivity test between node1 and node2 on the LAN interface as well on the crossover interface. We receive successful echo response messages after doing the ping request tests. This shows that the network connectivity is successfully configured.

In step 8, we set up the drbd.conf file on both the nodes. Here is an extract from the drbd.conf file:

```
global {
    usage-count no;
}
common {
    syncer { rate 100M; }
    protocol      C;
}
resource postgres {
    startup {
       wfc-timeout 0;
       degr-wfc-timeout
       120;
    }
    disk { on-io-error detach; }
    on node1.example.org {
       device      /dev/drbd0;
       disk        /dev/sdb;
       address     172.16.0.1:7791;
       meta-disk   internal;
    }
    on node2.example.org {
       device      /dev/drbd0;
       disk        /dev/sdb;
       address     172.16.0.2:7791;
       meta-disk   internal;
    }
}
```

Basically, using the previously mentioned configuration, we are setting up a postgres resource and configuring a DRBD interface, /dev/drbd0, which is set up on two nodes, node1 and node2. This is basically what causes the block-level replication to be successful. In step 11 of the preceding section, you can see that we have initially set up node1 as the primary node and node2 serves as the secondary node at this stage. Then, we set up PostgreSQL on node1 from step 15 onwards. From step 20 onwards, we perform failover testing. We first reset node1 as the secondary node, unmount the filesystem, and then set up node2 as the primary node; then, mount the file system and bring up the PostgreSQL server. After this, we are testing for record visibility in node2. The database test that was created in step 19 of the preceding section is accessible in node2 and so are the tables in the pgbench schema in step 20. Thus, DRBD provides block-level replication, and if one of the nodes is not available, we can then configure and continue to run PostgreSQL on the secondary node, where it is going to take the role of the primary server.

Setting up the Postgres-XC cluster

In this recipe, we are going to set up a Postgres-XC cluster.

Getting ready

Here, we need to install and set up Postgres-XC. These steps are carried out on a CentOS Version 6 Linux machine.

Perform the following set of steps:

1. First, go to http://sourceforge.net/projects/postgres-xc/ in order to download the Postgres-XC software.
2. In this step, extract from the tarball file and go to the newly created directory:

   ```
   tar -zxvf pgxc-v1.0.4.tar.gz
   cd   pgxc-v1.0.4
   ```

3. Before you build and compile the software, the next step will be to install the following prerequisite packages:

   ```
   yum -y install readline*
   yum -y install bison*
   yum -y install flex*
   ```

4. Now, we are going to build and compile the software. We will also define a location to be used as the prefix:

   ```
   mkdir -p /opt/Postgres-xc
   chown -R postgres:postgres /opt/Postgres-xc/
   ./configure --prefix=/opt/Postgres-xc/
   make
   make install
   ```

How to do it...

Now, with the installation completed, we need to configure the Postgres-XC setup.

Perform the following steps:

1. We will now set up **GTM** (short for **global transaction manager**). For this purpose, we will first create a directory for GTM, set permissions, and then initialize the GTM:

   ```
   mkdir -p /usr/local/pgsql/data_gtm
   chmod -R 700 /usr/local/pgsql/data_gtm
   /opt/Postgres-xc/bin/initgtm -Z gtm -D /usr/local/pgsql/data_gtm
   ```

2. We will now configure the parameters in the `gtm.conf` file, which was created as a part of the previous step where GTM was initialized, and start the GTM:

   ```
   nodename = 'GTM_Node'
   listen_addresses = '*'
   port = 7777
   ```

   Once these parameters have been changed, we can then set up the GTM:

   ```
   /opt/Postgres-xc/bin/gtm_ctl -Z gtm start -D /opt/Postgres-xc/
   data_gtm
   Server Started
   ```

3. With the GTM set up and started, we will now set up the coordinator node. For this purpose, we will first create a directory for the coordinator, assign permissions, and then initialize the coordinator:

   ```
   mkdir -p /opt/Postgres-xc/data_coord1
   chmod  -R 700 /opt/Postgres-xc/data_coord1
   /opt/Postgres-xc/bin/pg_ctl -D /opt/Postgres-xc/data_coord1/ -o
   '--nodename coord1' initdb
   ```

4. In the next step, we will configure the necessary parameters in the `postgresql.conf` file. This will be set up in such a way that the coordinator is used as a node to connect to the GTM. Also, once the necessary parameters have been configured, we will start the coordinator:

   ```
   listen_addresses = '*'
   port = 2345
   gtm_host = 'localhost'
   gtm_port = 7777
   pgxc_node_name = 'coord1'
   pooler_port = 2344
   min_pool_size = 1
   max_pool_size = 100
   persistent_datanode_
   connections = on
   max_coordinators = 16
   max_datanodes = 16
   ```

   Once these parameters have been changed, we can start the coordinator, as follows:

   ```
   /opt/Postgres-xc/bin/pg_ctl start -D /opt/Postgres-xc/data_coord1/
   -Z coordinator -l /tmp/coord
   ```

5. We will now set up the first data node. For this purpose, we are going to create a directory, assign the respective permissions to it, and then initialize it:

   ```
   mkdir -p /opt/Postgres-xc/data_node1


   chmod -R 700  /opt/Postgres-xc/data_node1


   /opt/Postgres-xc/bin/pg_ctl -D /opt/Postgres-xc/data_node1/ -o
   '--nodename datanode1' initdb
   ```

6. In the next step, we will configure the necessary parameters for the first data node and then start the data node. These parameter changes are made in the `postgresql.conf` file:

   ```
   vi postgresql.conf


   listen_addresses = '*'
   port = 1234
   gtm_host = 'localhost'
   gtm_port = 7777
   pgxc_node_name = 'datanode1'
   ```

Once these changes have been made, we can launch the first data node:

```
/opt/Postgres-xc/bin/pg_ctl start -D /opt/Postgres-xc/data_node1
-Z datanode -l /tmp/datanode1_log
```

7. In this step, we will set up the second data node. For this purpose, we will configure the directory for the second data node, assign permissions, and then initialize it:

```
mkdir -p /opt/Postgres-xc/data_node2/


chmod -R  700  /opt/Postgres-xc/data_node2/


/opt/Postgres-xc/bin/pg_ctl -D /opt/Postgres-xc/data_node2/ -o
'--nodename datanode2' initdb
```

8. In this step, we will configure the respective parameters for the second data node, and once we are done, we will start the second data node:

```
vi postgresql.conf


listen_addresses = '*'
port = 1233
gtm_host = 'localhost'
gtm_port = 7777
pgxc_node_name = 'datanode2'
```

Once these necessary parameter changes have been made, we will start the second data node:

```
/opt/Postgres-xc/bin/pg_ctl start -D /opt/Postgres-xc/data_node2
-Z datanode -l /tmp/datanode2_log
```

9. In this step, we are going to register the first and second data nodes on the coordinator node:

```
cd /opt/Postgres-xc/bin/



psql -p 2345


postgres=# CREATE NODE datanode1 WITH ( TYPE = DATANODE , HOST =
LOCALHOST , PORT = 1234 );
```

```
CREATE NODE

postgres=# CREATE NODE datanode2 WITH ( TYPE = DATANODE , HOST =
LOCALHOST , PORT = 1233 );
CREATE NODE
```

10. Now, with the Postgres-XC architecture setup complete, we will start distributing the data by replication:

    ```
    psql -p 2345


    postgres=# CREATE TABLE DIST (T INT) DISTRIBUTE BY REPLICATION TO
    NODE datanode1,datanode2;
    CREATE TABLE

    postgres=#INSERT INTO DIST SELECT * FROM generate_series(1, 100);
    INSERT 0 100

    postgres=# EXPLAIN ANALYZE SELECT * FROM DIST;

                               QUERY PLAN
    ------------------------------------------------------------------
     Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0
    width=0) (act
    ual time=0.880..1.010 rows=100 loops=1)
       Node/s: datanode1
     Total runtime: 1.076 ms
    (3 rows)
    ```

    We will now log in to `datanode1` and `datanode2`, to see whether these records have been replicated on the `DIST` table:

    ```
    psql -p 1234


    postgres=# select count(*) from DIST;
     count
    -------
       100
    (1 row)
    ```

```
psql -p 1233

postgres=# select count(*) from DIST;
 count
-------
   100
(1 row)
```

11. Now, we will test for distribution by hash:

    Log in to the coordinator node:

    `psql -p 2345`

    ```
    CREATE TABLE t_test (id int4) DISTRIBUTE BY HASH (id);

    INSERT INTO t_test SELECT * FROM generate_series(1, 1000);
    ```

    We will now log in to `datanode1` and `datanode2` and see how many records are replicated there:

    `psql -p 1233`

    ```
    postgres=# select count(*) from t_test;
     count
    -------
       508
    (1 row)

    psql -p 1234

    postgres=# select count(*) from t_test;
     count
    -------
       492
    (1 row)
    ```

How it works...

In the entire Postgres-XC architecture, we have used the following setup. We are using a GTM, a coordinator, and two data nodes. We will discuss the functionality for each one of them:

- **GTM**: GTM is used to provide a consistent view of the data. A consistent view is basically provided through a cluster-wide snapshot. GTM is also responsible for creating global transaction IDs, which are necessary because transactions need to be coordinated cluster wide.
- **Coordinator**: This serves as an entry point for applications and is used by the application to connect to the coordinator. A coordinator is responsible for SQL analysis, the creation of a global SQL execution plan, and global SQL execution.
- **Data node**: A data node is used to hold data for a PostgreSQL cluster. One or more data nodes hold all or a part of the data inside the cluster.

We will now discuss the various steps performed in the preceding section:

- Here, we will discuss steps 1 and 2 of the preceding section. This setup is all about GTM configuration. We initially configure a directory for GTM, set permissions, initialize the directory, and then start GTM later.
- Steps 3 and 4 of the preceding section are all about the coordinator node's configuration. We initially configure a directory for the coordinator, set permissions, initialize the directory, and then start the coordinator.
- Next, we will discuss steps 5 and 6 of the preceding section. This setup is all about the first data node's configuration. We initially configure a directory for `datanode1`, set permissions, initialize the directory, and then start the first data node.
- Steps 7 and 8 of the preceding section discuss the second data node's configuration. We initially configure a directory for `datanode2`, set permissions, initialize the directory, and then start the second data node.
- In step 9, we first log in to the coordinator node, and then we register the `datanode1` and `datanode2` nodes with the coordinator node.

- In steps 10 and 11, we basically test the Postgres-XC cluster. In step 10, we log in to the coordinator node, create a DIST table, and then distribute this table by replication to datanode1 and datanode2. We then generate a series and insert about 100 records in the DIST table. Distribution by replication to the data nodes means that the data should be replicated in both the nodes for the DIST table. We then log in to the datanode1 node and then count the number of records from the DIST table; the count is 100. The same observation is obtained for the DIST table when we log in to the datanode2 node. This is effectively demonstrated in step 10 of the preceding section; thus, all the records of the DIST table are replicated across both the data nodes. In step 11, we log in to the coordinator node and create a t_test table; then, we distribute the table by hash and insert 1000 records into the table. We then log in to the first data node, and we can see 508 records here. We then log in to the second data node, and we can see 492 records in the t_test table in the datanode2 node. What we see is an even distribution and splitting of the storage of records of the t_test table between data nodes 1 and 2.

8
Connection Pooling

In this chapter, we will cover the following recipes:

- Installing pgpool
- Configuring pgpool and testing the setup
- Starting and stopping pgpool
- Setting up pgbouncer
- Connection pooling using pgbouncer
- Managing pgbouncer

Introduction

The **pgpool-II** utility is basically a middleware solution that works as an interface between a PostgreSQL server and a PostgreSQL client application. The pgpool-II utility serves as a proxy between PostgreSQL's backend and frontend protocols and relays a connection between the two. The pgpool-II utility caches connections to PostgreSQL servers and reuses them whenever a new connection with the same properties comes in, thereby reducing connection negotiation overhead such as authentication and encryption, and improving overall throughput.

In fact, the pgpool-II utility offers a lot more features than just connection pooling. It offers load balancing and replication modes along with the parallel query feature.

The **pgbouncer** utility is a lightweight connection pooler for PostgreSQL. Applications connect to the pgbouncer port just as they would connect to a PostgreSQL database on the database port. Using the pgbouncer utility, we can lower the connection overload impact on PostgreSQL server. The pgbouncer utility provides connection pooling by reusing existing connections.

The difference between pgbouncer and pgpool-II is that pgbouncer is lightweight and is dedicated to the purpose of connection pooling, whereas pgpool-II offers more features such as replication, load balancing, and the parallel query feature in addition to connection pooling.

In this chapter, we will be referring to pgpool-II as pgpool for the purpose of simplicity.

Installing pgpool

Here is the recipe to install pgpool and configure it.

Getting ready

Installing pgpool from source requires gcc 2.9 or higher and GNU make. Since the pgpool links with the libpq library, the libpq library and its development headers must also be installed prior to installing pgpool. Also, the OpenSSL library must be present in order to enable OpenSSL support in pgpool.

If you are building from source, then follow these steps:

1. Download the latest tarball of pgpool from the following website:

   `http://www.pgpool.net/mediawiki/index.php/Downloads`

2. The next step would be to extract the pgpool tarball and enter the source directory:

   ```
   tar -xzf pgpool-II-3.4.0.tar.gz
   cd pgpool-II-3.4.0
   ```

3. Build, compile, and install the pgpool software:

   ```
   ./configure –prefix=/usr/local --sysconfdir=/etc/pgpool/
   make
   make install
   ```

How to do it...

To install pgpool in a Debian or Ubuntu-based distribution, we can execute this command:

```
apt-get install pgpool2
```

On Red Hat, Fedora, CentOS, or any other RHEL-based Linux distributions use the following command. It should be noted that the package name used is what is existing now for pgpool, that is, the `93` keyword used at the end relates to a minor release of PostgreSQL. It may change later as updates are released:

```
yum install pgpool-II-93
```

The following steps are applicable to when you use the operating system package manager such as `yum` to install pgpool, and when you are downloading and compiling from source. Basically, in the following steps, we are creating a directory for pgpool where it can maintain the activity logs and its service lock files:

1. In this step, we will create the location where pgpool can maintain activity logs:

   ```
   mkdir /var/log/pgpool
   chown -R postgres:postgres /var/log/pgpool
   ```

2. The next step will be to create a directory where pgpool can store its service lock files:

   ```
   mkdir /var/run/pgpool
   chown -R postgres:postgres  /var/run/pgpool
   ```

How it works...

If you are using an operating-system-specific package manager to install pgpool, then the respective configuration files and logfiles required are automatically created. However, if you are proceeding with a full source-based pgpool installation, then there are some additional steps required. The first step is to run the configure script and then build and compile pgpool. After pgpool is installed, you will be required to create the directories where pgpool can maintain activity logs and service lock files. All of these steps need to be performed manually as can be seen in steps 1, 2, and 3, respectively, in the *Getting ready* section.

Configuring pgpool and testing the setup

In this recipe, we are going to configure pgpool and show how to make connections.

Getting ready

Before running pgpool, if you are downloading the source tarball, then the pgpool software needs to be built and compiled. These steps are shown in the first recipe of this chapter.

Also, we will be testing for replication using pgpool. For this purpose, we are setting up two data directories on the same server. They will act as two nodes.

Assuming that the default data directory, `/var/lib/pgsql/9.3/data`, which will serve as node 0, has already been set up, we will now set up another data directory that will serve as node 1:

```
initdb -D /var/lib/pgsql/9.3/data1
```

Once the data directory has been set up, the next step is to change the port number for the new data directory. This is being done because two data directories cannot have the same port number. Since port number 5432 has already been used for the initial data directory, we will set the port number as `5433` in the `postgresql.conf` file for the new data directory and then start the server using this setup:

```
cd  /var/lib/pgsql/9.3/data1
```

```
vi postgresql.conf
port=5433
```

Once the file is saved, then start the new server:

```
pg_ctl -D  /var/lib/pgsql/9.3/data1  start
```

How to do it...

We are going to follow this sequence of steps to configure pgpool and run the setup:

1. After pgpool is installed as shown in the first recipe of the chapter, the next step would be to copy the configuration files from the sample directory with the default settings. They will be later edited according to our requirements:

   ```
   cd /etc/pgpool-II-93
   cp pgpool.conf.sample /etc/pgpool.conf
   cp pcp.conf.sample /etc/pcp.conf
   ```

2. The next step is to define a username and password in the `pcp.conf` file, which is an authentication file for pgpool. Basically, to use PCP commands, user authentication is required. This mechanism is different from PostgreSQL's user authentication. Passwords are encrypted in the MD5 hash format. To obtain the MD5 hash for a user, we have to use the `pg_md5` utility as shown in the following command. Once the MD5 hash is generated, it can be used to store the MD5 password in the `pcp.conf` file:

   ```
   pg_md5 postgres
   e8a48653851e28c69d0506508fb27fc5
   vi /etc/pcp.conf
    postgres:e8a48653851e28c69d0506508fb27fc5
   ```

3. Now we edit the `pgpool.conf` configuration file to configure our pgpool settings:

   ```
   listen_addresses = 'localhost'
   port = 9999
   socket_dir = '/tmp'
   pcp_port = 9898
   pcp_socket_dir = '/tmp'
   backend_hostname0 = 'localhost'
   ```

```
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/var/lig/pgsql/9.3/data'
backend_flag0 = 'ALLOW_TO_FAILOVER'
backend_hostname1 = 'localhost'
backend_port1 = 5433
backend_weight1 = 1
backend_data_directory1 = '/var/lib/pgsql/9.3/data1'
backend_flag1 = 'ALLOW_TO_FAILOVER'
enable_pool_hba = off
pool_passwd = 'pool_passwd'
authentication_timeout = 60
ssl = off
num_init_children = 32

max_pool = 4
child_life_time = 300
child_max_connections = 0
connection_life_time = 0
client_idle_limit = 0
connection_cache = on
reset_query_list = 'ABORT; DISCARD ALL'
replication_mode = on
master_slave_mode = off
replicate_select = off
insert_lock = on
load_balance_mode = on
ignore_leading_white_space = on
white_function_list = ''
black_function_list = 'nextval,setval'
```

4. Once the preceding parameters have been configured and saved in the `pgpool.conf` file, the next step is to launch pgpool and start accepting connections to the PostgreSQL cluster using pgpool:

   ```
   pgpool -f  /etc/pgpool.conf -F /etc/pcp.conf
   psql -p 9999 postgres postgres
   ```

5. Now that pgpool has been started, we should see a handful of processes:

   ```
   -bash-4.1$ ps ax |grep pool
   28778 ?        Ss     0:00 pgpool -f /etc/pgpool.conf -F /etc/pcp.
   conf
   28779 ?        S      0:00 pgpool: wait for connection request
   28780 ?        S      0:00 pgpool: wait for connection request
   ```

```
28781 ?        S      0:00 pgpool: wait for connection request
28782 ?        S      0:00 pgpool: wait for connection request
28783 ?        S      0:00 pgpool: wait for connection request
28784 ?        S      0:00 pgpool: wait for connection request
28785 ?        S      0:00 pgpool: wait for connection request
28786 ?        S      0:00 pgpool: wait for connection request
28787 ?        S      0:00 pgpool: wait for connection request
28788 ?        S      0:00 pgpool: wait for connection request
28789 ?        S      0:00 pgpool: wait for connection request
28790 ?        S      0:00 pgpool: wait for connection request
28811 ?        S      0:00 pgpool: PCP: wait for connection
request
28812 ?        S      0:00 pgpool: worker process
28849 pts/2    S+     0:00 grep pool
```

6. Before we connect to pgpool and start executing queries, we should check the status of the nodes participating in the cluster. For this purpose we will use a tool called `pcp_node_info`. Since we are using the same server for setup, node 0 and node 1 are more specifically data directories located at `/var/lib/pgsql/9.3/data` and `/var/lib/pgsql/9.3/data1`:

   ```
   -bash-4.1$ pcp_node_info 5 localhost 9898 postgres postgres 0
   localhost 5432 1 0.500000
   -bash-4.1$ pcp_node_info 5 localhost 9898 postgres postgres 1
   localhost 5433 1 0.500000
   ```

7. Now that both nodes are participating in the cluster, the next step is to connect to pgpool, create a table, and insert some records into that table:

   ```
   psql -p 9999
   ```

   ```
   postgres=# create table emp(age int);
   CREATE TABLE
   postgres=# insert into emp values (1);
   INSERT 0 1
   postgres=# insert into emp values (2);
   INSERT 0 1
   postgres=# insert into emp values (3);
   INSERT 0 1
   postgres=# insert into emp values (4);
   INSERT 0 1
   ```

```
postgres=# insert into emp values (5);
INSERT 0 1
postgres=# insert into emp values (6);
INSERT 0 1
postgres=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | emp  | table | postgres
(1 row)

postgres=# select * from emp;
 age
-----
   1
   2
   3
   4
   5
   6
(6 rows)
```

8. Now we will test for replication by connecting to ports 5432 and 5433, and see the table and the corresponding records that were inserted into it while being connected to pgpool:

   `psql -p 5433`

```
postgres=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | emp  | table | postgres
(1 row)

postgres=# select * from emp;
 age
-----
```

```
   1
   2
   3
   4
   5
   6
(6 rows)


-bash-4.1$ psql -p 5432

postgres=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | emp  | table | postgres
(1 row)

postgres=# select * from emp;
 age
-----
   1
   2
   3
   4
   5
   6
(6 rows)
```

How it works...

Let's now discuss some of the parameters that were configured in the earlier section:

- `listen_addresses`: We configure `listen_addresses` to `*` because we want to listen to all IP addresses and not a particular IP address.
- `port`: This defines the pgpool port that the system will listen to when accepting database connections.
- `backend_hostname0`: This refers to the hostname of the first database in our setup. Similarly, we set up `backend_hostname1` for the second node.

- backend_port0: This is the TCP port of the system, that is, the system identified by the backend_hostname0 value on which the database is hosted. Similarly, we set backend_port1 for the the second node.
- backend_weight0: This is the weight assigned to the node identified by the hostname obtained from backend_hostname0. Basically in pgpool, weights are assigned to individual nodes. More requests will be dispatched to the node with a higher weight value. Similarly, we set up backend_weight1 for the second node.
- backend_data_directory0: This represents the data directory, that is, PGDATA for the host identified by the backend_hostname0 value. Similarly, we set up the backend_data_directory1 value for the second node.
- connection_cache: To enable the connection pool mode, we need to turn on connection_cache.
- max_pool: This value determines the maximum size of the pool per child. The number of connections from pgpool to the backends may reach a limit at *num_init_children*max_pool*.

In our case, we configured max_pool to 4 and num_init_children to 32, both being the default value. So when multiplied, the total number of connections from pgpool to the backend may reach a limit of 128. Remember that the *max_pool * num_init_children* value should always be less than the max_connections parameter value.

The other parameters that were discussed in the preceding steps are as follows:

- replication_mode: This parameter turns replication explicitly on. By default, it is set to off.
- load_balance_on: Enabling this parameter ensures that pgpool splits the load to all of the hosts or nodes attached to the system.
- master_slave_mode: This parameter enables the master/slave mode. This parameter must be set to off when the replication_mode is set to on.

The other parameters take default values, and you may refer to the following pgpool links follows for more information regarding them:

http://www.pgpool.net/docs/latest/pgpool-en.html#config

http://www.pgpool.net/pgpool-web/contrib_docs/simple_sr_setting_3.1/pgpool.conf

Finally, once the parameters are configured, it is time to launch pgpool and make connections on the pgpool port 9999.

Once pgpool is launched, we can see that some processes in the background have already started, as seen in step 5 of the preceding section.

Before we make connections using pgpool, we are basically utilizing a tool called pcp_node_info to check the status of the nodes, as seen in step 6 of the preceding section.

The pcp_node_info command has the following syntax:

```
pcp_node_info  <timeout>  <hostname>  <port>  <username> <password>
<nodeid>.
```

Here is the excerpt from step 6 of the preceding section:

```
-bash-4.1$ pcp_node_info 5 localhost 9898 postgres postgres 0


localhost 5432 1 0.500000


-bash-4.1$ pcp_node_info 5 localhost 9898 postgres postgres 1


localhost 5433 1 0.500000
```

As per step 6 of the preceding section, we are specifying the connection timeout value to be 5 .The hostname refers to the localhost, followed by port number 5432 and the username/password combination, which is set to the postgres user along with password as postgres. The final parameter is nodeid, which is set to 0 for the first node. Node 0 in our case refers to the /var/lib/pgsql/9.3/data data directory. Similarly, we use pcp_node_info to specify the port 5433 and the nodeid value as 1 for the /var/lib/pgsql/9.3/data1 data directory.

Whenever the pcp_node_info command is triggered, the system will respond with the following output: the hostname, port number, status, and weight of the node.

Among all of these values, the third column, which refers to the status of the node, is the most important. If the value of the status column is 1, it means that the node is up but connections are yet to be made. If the value of the status column is 2, it means that the node is up and connections are pooled. If the value of the status column is 3, it means that the node is down and some action needs to be taken.

In our scenario, the value of the status column is 1 for both the nodes, which means we are good to go and we can start making connections to pgpool. If the value of the status column is 3, then you need to enable the node using the pcp_attach_node tool. The pcp_attach_node command has the same syntax as the pcp_node_info command and can be used as shown in the following line, assuming that the value of the status column of a node is 3. Let us assume this value for the status column of node 1:

```
pcp_attach_node 5  localhost 5433 postgres postgres 1
```

In step 7 of the preceding section, we are connecting to pgpool on port 9999, creating a table named emp and inserting some records into it.

In step 8, we are testing for replication. We can clearly see that the `emp` table and the corresponding records are available there when making connections to port 5433 and port 5432. This confirms successful replication using pgpool.

> Refer to the following web links for more details on pgpool:
>
> - `http://www.pgpool.net/mediawiki/index.php/Relationship_between_max_pool,_num_init_children,_and_max_connections`
> - `http://www.pgpool.net/docs/latest/pgpool-en.html#connection_pool_mode`

Starting and stopping pgpool

In this recipe, we are going to show the commands that can be used to start and stop pgpool.

Getting ready

Before pgpool can be started, we need to configure the pgpool settings in the `pgpool.conf` configuration. This is covered in the previous recipe.

How to do it...

The pgpool utility can be started in two ways:

- By starting the `pgpool` service at the command line as the root user:

  ```
  service pgpool start
  ```

- By executing the `pgpool` command on the terminal:

  ```
  pgpool
  ```

Similarly, pgpool can be stopped in two ways:

- By stopping the `pgpool` service at the command line as the root user:

  ```
  service pgpool stop
  ```

- By executing the `pgpool` command with the `stop` option:

  ```
  pgpool stop
  ```

How it works...

Starting and stopping pgpool is relatively simple, as was seen in the preceding section. However, pgpool comes with a lot of options, and the following is the most commonly used syntax to start pgpool:

```
pgpool [-c][-f config_file][-a hba_file][-F pcp_config_file]
```

These options are discussed as follows:

- `-c`: The `-c` switch is used to clear the query cache
- `-f` config_file: This option specifies the `pgpool.conf` configuration file, and pgpool obtains its configuration from this file when starting itself
- `-a` hba_file: This option specifies the authentication file that is used when starting pgpool
- `-F` pcp_config_file: This option specifies the password file, `pcp.conf`, to be used when starting pgpool

> For the full syntax of pgpool, refer to the following web link:
>
> `http://www.pgpool.net/docs/latest/pgpool-en.html#start`

To stop pgpool, the same options that were used earlier to start pgpool can be used. However, along with these switches, we can also specify the mode that needs to be used while stopping pgpool.

There are two modes in which pgpool can be stopped:

- **Smart mode**: This option is specified using the `-m s` (smart) option. In this mode, we first wait for the clients to disconnect and then shut down pgpool.
- **Fast mode**: This mode can be set by specifying the `–m f` (fast) option. In this mode, pgpool does not wait for clients to disconnect and shuts down pgpool immediately.

The complete syntax of the pgpool `stop` command is as follows:

```
pgpool [-f config_file][-F pcp_config_file] [-m {s[mart]|f[ast]|]}]
stop
```

Usually, if there are any clients connected, pgpool waits for them to disconnect and will then terminate itself. However, if you want to shutdown pgpool forcibly without waiting for clients to disconnect, you can use the following command:

```
pgpool -m fast stop
```

Setting up pgbouncer

In this recipe, we are going to show the steps that are required to install pgbouncer.

Getting ready

We can either do a full source-based installation or use the operating-system-specific package manager to install pgbouncer.

How to do it...

On an Ubuntu or Debian-based system, we need to execute the following command to install pgbouncer:

```
apt-get install pgbouncer.
```

On CentOS, Fedora, or Red Hat-based Linux distributions we can execute the following command:

```
yum install pgbouncer
```

If you are doing a full source-based installation, then the sequence of commands is as follows:

1. Download the archive installation file from the following link:

   http://pgfoundry.org/projects/pgbouncer

2. Extract the downloaded archive and enter the source directory:

   ```
   tar -xzf pgbouncer-1.5.4.tar.gz
   cd pgbouncer-1.5.4
   ```

3. The next step is to build and proceed with the software installation:

   ```
   ./configure –prefix=/usr/local
   make & make install
   ```

4. Now create a configuration directory to hold a pgbouncer configuration file. This file can be used later on to make parameter changes:

   ```
   mkdir /etc/pgbouncer
   chown -R postgres:postgres /etc/pgbouncer
   ```

How it works...

If you are using an operating-system-specific package manager to install pgbouncer, then the respective configuration files and logfiles required by pgbouncer are automatically created. However, if you are proceeding with a full source-based pgbouncer installation, then there are some additional steps required. You will be required to create the directories where pgbouncer can maintain activity logs and service lock files. You will also be required to create the configuration directory where the configuration file for pgbouncer will be stored. All of these steps need to be performed manually as shown in steps 2, 3, and 4 in the prior section.

Connection pooling using pgbouncer

In this recipe, we are going to implement pgbouncer and benchmark the results for database connections made to the database via pgbouncer against normal database connections.

Getting ready

Before we configure and implement connection pooling, the pgbouncer utility must be installed. Installing pgbouncer is covered in the previous recipe.

How to do it...

1. First, we are going to tweak some of the configuration settings in the `pgbouncer.ini` configuration file, as follows. The first two entries are for the databases that will be passed through pgbouncer. Next, we configure the `listen_addr` parameter to `*`, which means that it is going to listen to all IP addresses. Finally, we set the last two parameters, which are `auth_file`, the location of the authentication file and `auth_type`, which indicates the type of authentication used. We use `plain` as the authentication type, which indicates that the we are using the password-based mechanism here for authentication:

   ```
   vi  /etc/pgbouncer/pgbouncer.ini


   postgres = host=localhost dbname=postgres
   pgtest = host=localhost dbname=pgtest
   listen_addr = *
   auth_file = /etc/pgbouncer/userlist.txt
   auth_type = md5
   ```

2. The next step is to create a user list that contains the users who will be allowed to access the databases through pgbouncer. The format of the entries in the user list would be supplied as username followed by the user's password as shown in the following command, where the first entry is for the username, whose value is `author`, and the second entry is for the password, whose value is `password`. Since we have set the authentication type as MD5, we have to use the MD5 password entry in the user list. Had we set the authentication type as plain, then the actual password would have been supplied in the user list:

```
postgres=# CREATE role author LOGIN PASSWORD 'author' SUPERUSER;
CREATE ROLE

postgres=# select rolname ,rolpassword from pg_authid where
rolname='author';
 rolname |             rolpassword
---------+-------------------------------------
 author  | md5d50afb6ec7b2501164b80a0480596ded
(1 row)
```

   The MD5 password obtained can then be defined in the `userlist` file for the corresponding user:

```
vi /etc/pgbouncer/userlist.txt


"author" "md5d50afb6ec7b2501164b80a0480596ded"
```

3. Once we have configured the `pgbouncer.ini` configuration file and created the `userlist` file, the next step would be to start the pgbouncer service:

```
service pgbouncer start
```

4. Once the pgbouncer service is up and running, the next step will be to make connections to it. By default, the pgbouncer service runs on port 6432, so any connections made to the pgbouncer service need to be made on port 6432:

```
psql -h localhost -p 6432 -d postgres -U author  -W
```

5. Now that we have made connections using pgbouncer, the next logical step is to find out whether there are any performance improvements using pgbouncer. For this purpose, we are going to create a temporary database—the one that was initially defined in the `pgbouncer.ini` file—and insert records into it, and then benchmark connections made against this database:

```
createdb pgtest

pgbench -i -s 10 pgtest
```

6. Then we benchmark the results against the `pgtest` database:

   ```
   -bash-3.2$ pgbench -t 1000 -c 20 -C -S  pgtest
   starting vacuum...end.
   transaction type: SELECT only
   scaling factor: 10
   query mode: simple
   number of clients: 20
   number of threads: 1
   number of transactions per client: 1000
   number of transactions actually processed: 20000/20000
   tps = 217.374571 (including connections establishing)
   tps = 1235.875488 (excluding connections establishing)
   ```

7. The final step would be to benchmark the results against the `pgtest` database on pgbouncer port 6432:

   ```
   -bash-3.2$ pgbench -t 1000 -c 20 -C -S pgtest -p 6432 -U author
   Password:
   starting vacuum...end.
   transaction type: SELECT only
   scaling factor: 10
   query mode: simple
   number of clients: 20
   number of threads: 1
   number of transactions per client: 1000
   number of transactions actually processed: 20000/20000
   tps = 2033.768075 (including connections establishing)
   tps = 53124.095230 (excluding connections establishing)
   ```

How it works...

Here we can see a couple of things. Initially, we have to configure the `pgbouncer.ini` configuration file along with the `userlist` file, which will be used for accessing databases via pgbouncer. The effectiveness of pgbouncer can be seen in steps 6 and 7 in the preceding section. We can see that the throughput increases to 2033.768 transactions per second when pgbouncer is used, whereas when pgbouncer is not used, the throughput decreases to a mere 217.37 transactions per second. In effect, using pgbouncer increases throughput by approximately 10 times.

There's more...

In the *How it works...* section, we configured a couple of parameters in the pgbouncer.ini configuration file. However, there are many more parameters that can be configured, and if not configured, they will take the default settings. Refer to the following link to get more details on pgbouncer parameters:

http://pgbouncer.projects.pgfoundry.org/doc/config.html

Managing pgbouncer

The pgbouncer utility provides an administrative console to view pool status and client connections. In this recipe, we are going to view information regarding pgbouncer connections (client connections), view pool status, and obtain connection pooling statistics.

Getting ready

Before we issue any commands, we first need to connect to the pgbouncer's administrative console. For this purpose, we need to set the admin_users parameter in the pgbouncer.ini configuration file:

```
vi /etc/pgbouncer/pgbouncer.ini


admin_users = author
```

Once the preceding changes are saved in the pgbouncer.ini configuration file, the pgbouncer service needs to be restarted in order to ensure that the parameter changes come into effect:

```
service pgbouncer restart
```

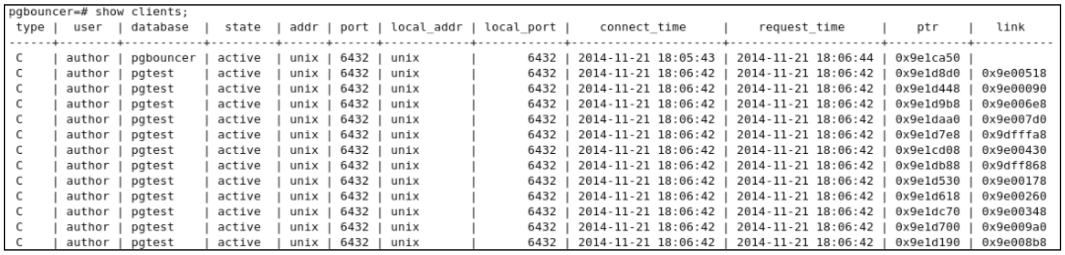
Once this is done, we can make connections to the pgbouncer administration console with the following command:

```
psql -p 6432 -U author pgbouncer
```

How to do it...

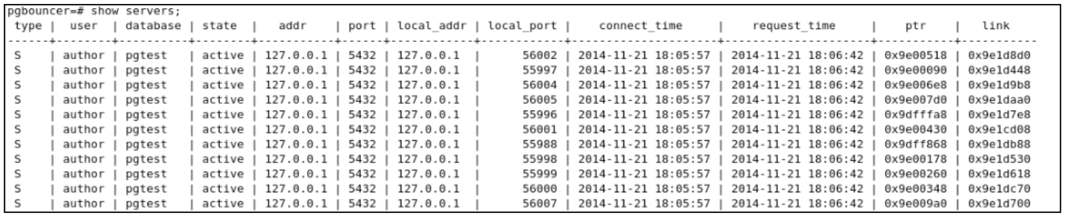
With the help of the pgbouncer administration console, we can get information regarding the clients, servers, and pool health:

1. To get information regarding the clients, issue the `SHOW CLIENTS` command, as shown in the following screenshot, on the pgbouncer admin interface:



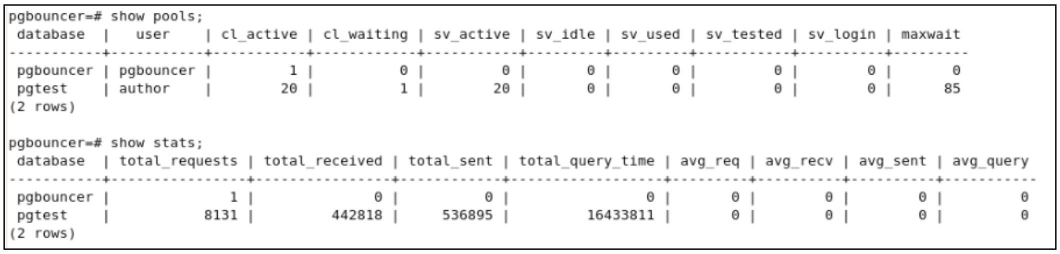
```
pgbouncer=# show clients;
 type |  user  | database  |  state   | addr | port | local_addr | local_port |    connect_time     |    request_time     |    ptr    |   link
------+--------+-----------+----------+------+------+------------+------------+---------------------+---------------------+-----------+-----------
 C    | author | pgbouncer | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:05:43 | 2014-11-21 18:06:44 | 0x9e1ca50 |
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d8d0 | 0x9e00518
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d448 | 0x9e00090
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d9b8 | 0x9e006e8
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1daa0 | 0x9e007d0
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d7e8 | 0x9dfffa8
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1cd08 | 0x9e00430
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1db88 | 0x9dff868
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d530 | 0x9e00178
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d618 | 0x9e00260
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1dc70 | 0x9e00348
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d700 | 0x9e009a0
 C    | author | pgtest    | active   | unix | 6432 | unix       |       6432 | 2014-11-21 18:06:42 | 2014-11-21 18:06:42 | 0x9e1d190 | 0x9e008b8
```

2. To get information regarding server connections, issue the `SHOW SERVERS` command, as shown in the following screenshot, on the pgbouncer administrative console:



```
pgbouncer=# show servers;
 type |  user  | database | state  |   addr    | port | local_addr | local_port |    connect_time     |    request_time     |    ptr    |   link
------+--------+----------+--------+-----------+------+------------+------------+---------------------+---------------------+-----------+-----------
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56002 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00518 | 0x9e1d8d0
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      55997 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00090 | 0x9e1d448
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56004 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e006e8 | 0x9e1d9b8
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56005 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e007d0 | 0x9e1daa0
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      55996 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9dfffa8 | 0x9e1d7e8
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56001 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00430 | 0x9e1cd08
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      55988 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9dff868 | 0x9e1db88
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      55998 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00178 | 0x9e1d530
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      55999 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00260 | 0x9e1d618
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56000 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e00348 | 0x9e1dc70
 S    | author | pgtest   | active | 127.0.0.1 | 5432 | 127.0.0.1  |      56007 | 2014-11-21 18:05:57 | 2014-11-21 18:06:42 | 0x9e009a0 | 0x9e1d700
```

3. Similarly, you can issue the `SHOW POOLS` and `SHOW STATS` commands to get information regarding pool status and pool statistics respectively, as shown in the following screenshot:



```
pgbouncer=# show pools;
 database  |   user    | cl_active | cl_waiting | sv_active | sv_idle | sv_used | sv_tested | sv_login | maxwait
-----------+-----------+-----------+------------+-----------+---------+---------+-----------+----------+---------
 pgbouncer | pgbouncer |         1 |          0 |         0 |       0 |       0 |         0 |        0 |       0
 pgtest    | author    |        20 |          1 |        20 |       0 |       0 |         0 |        0 |      85
(2 rows)

pgbouncer=# show stats;
 database  | total_requests | total_received | total_sent | total_query_time | avg_req | avg_recv | avg_sent | avg_query
-----------+----------------+----------------+------------+------------------+---------+----------+----------+-----------
 pgbouncer |              1 |              0 |          0 |                0 |       0 |        0 |        0 |         0
 pgtest    |           8131 |         442818 |     536895 |         16433811 |       0 |        0 |        0 |         0
(2 rows)
```

How it works...

As we saw in the preceding section, we can get information regarding the clients, servers, and pool health and statistics. When you issue the `SHOW CLIENTS` command on the pgbouncer administrative console, pgbouncer provides you with a list of clients that have been either using a PostgreSQL connection or waiting for it. Some of the important columns displayed in the output of the `SHOW CLIENTS` command are discussed here:

- `user`: The value in this column displays the user that is connected to the database.
- `database`: The value in this column displays the database name to which the client is connected.
- `state`: The value here displays the session state of the currently connected user. The client connection can be in active, used, waiting, or idle state.
- `connect_time`: The value in this column indicates the time at which pgbouncer initiated the client connection to PostgreSQL.
- `request_time`: This column's value shows the timestamp of the latest client request.
- `port`: The value in this column indicates the port to which the client is connected.

We have the `SHOW SERVERS` command, which is used to display information about every connection that is being used to fulfill client requests. The `SHOW SERVERS` output contains similar columns, which were discussed for `SHOW CLIENTS`. The only difference is for the `type` column. If the value for the type column is `S`, it means that it is a server entry. If the value for the type column is `C`, it means that it is a client entry. Some of the other important columns for the `SHOW SERVERS` output are discussed as follows:

- `user`: The value in this column displays the user that is connected to the database.
- `database`: The value in this column displays the name of the database to which the connection is attached.
- `state`: The value here displays the state of the pgbouncer server connection. The server state could be active, used, or idle.
- `connect_time`: The value in this column indicates the time at which the connection was made.
- `request_time`: This column's value shows the timestamp for when the most recent request was issued.
- `port`: The value in this column indicates the port number of the PostgreSQL server.

The SHOW POOLS command displays a row for every database for which pgbouncer acts as a proxy. Some of the important columns in the SHOW POOLS output are as follows:

- cl_active: The value in this column displays the number of clients that are currently active and assigned a server connection.
- cl_waiting: The value in this column displays the number of clients waiting for a server connection.
- sl_active: The value in this column displays the number of server connections that are assigned to pgbouncer clients.
- sl_idle: The value here displays the number of idle server connections, including the ones that are not in use.
- sl_used: The value in this column displays the number of used server connections. In effect, these connections are actually idle but they have not been marked by pgbouncer for reuse yet.

The SHOW STATS command displays the relevant connection pool statistics related to pgbouncer for the databases for which pgbouncer is acting as a proxy. Some of the important columns in the SHOW STATS output are as follows:

- total_requests: The value in this column displays the total number of SQL requests pooled by pgbouncer
- total_received: The value in this column displays the total volume of network traffic (measured in bytes) that has been received by pgbouncer
- total_sent: This column's value displays the total volume of network traffic (measured in bytes) that has been sent by pgbouncer
- total_query_time: The value in this column displays the amount of time in microseconds that pgbouncer spent communicating with a client in this pool

9

Table Partitioning

In this chapter, we will cover the following recipes:

- Implementing partitioning
- Managing partitions
- Partition and constraint exclusion
- Alternate partitioning methods
- Installing PL/Proxy
- Partitioning with PL/Proxy

Introduction

Partitioning is defined as splitting up a large table into smaller chunks. PostgreSQL supports basic table partitioning. Partitioning is entirely transparent to the applications if it is implemented correctly. Partitioning has a lot of benefits, which are discussed, as follows:

- The query performance can be improved significantly for certain types of queries.
- Partitioning can lead to an improved update performance. Whenever queries update a big chunk of a single partition, performance can be improved by performing a sequential scan for that partition, instead of using random reads and writes that are dispersed across the entire table.
- Bulk loads and deletes can be accomplished by adding or removing partitions if the requirement is incorporated in the partition design. The `ALTER TABLE NO INHERIT` and `DROP TABLE` operations perform faster than a bulk operation. Also, these commands avoid the `VACUUM` overhead caused by a bulk delete.
- Infrequently used data can be shipped to cheaper and slower media.

Implementing partitioning

Here, we are going to cover table partitioning and show the steps that need to be performed in order to partition a table.

Getting ready

Exposure to database design and normalization is the only requirement.

How to do it...

The following series of steps need to be carried out in order to set up table partitioning:

1. First, create a master table with all the fields. A master table is the table that will be used as a base to partition data into other tables, that is, partitions. An index is optional for a master table; however, since there are performance benefits of using an index, we will create an index from a performance perspective here:

   ```
   CREATE TABLE country_log (
       created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
       country_code char(2),
       content text
   );

   CREATE INDEX country_code_idx ON country_log USING btree
   (country_code);
   ```

2. The next step is to create child tables that will inherit from the master table:

   ```
   CREATE TABLE country_log_ru ( CHECK ( country_code = 'ru') )
   INHERITS (country_log);
   CREATE TABLE country_log_sa ( CHECK ( country_code = 'sa' ) )
   INHERITS (country_log);
   ```

3. Next, create an index for each child table:

   ```
   CREATE INDEX country_code_ru_idx ON country_log_ru USING btree
   (country_code);
   CREATE INDEX country_code_sa_idx ON country_log_sa USING btree
   (country_code);
   ```

4. Then, create a trigger function with the help of which data will be redirected to the appropriate partition table, as follows:

   ```
   CREATE OR REPLACE FUNCTION country_insert_trig() RETURNS TRIGGER
   AS $$
   BEGIN
       IF ( NEW.country_code = 'ru' ) THEN
   ```

```
        INSERT INTO country_log_ru VALUES (NEW.*);
    ELSIF ( NEW.country_code = 'sa' ) THEN
        INSERT INTO country_log_sa VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Country unknown';
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

5. Now, create a trigger and attach the trigger function to the master table:

```
CREATE TRIGGER country_insert
BEFORE INSERT ON country_log
FOR EACH ROW EXECUTE PROCEDURE country_insert_trig();
```

6. Next, insert data into the master table, as shown here:

```
postgres=# INSERT INTO country_log (country_code, content) VALUES
('ru', 'content-ru');
postgres=# INSERT INTO country_log (country_code, content) VALUES
('sa', 'content-sa');
```

7. The final step is to select the data from both the master and child tables to confirm the partitioning of data in the child tables, as follows:

```
postgres=# SELECT * from country_log;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 12:10:06.123189-08 | ru           | content-ru
 2014-11-30 12:10:14.22666-08  | sa           | content-sa
(2 rows)

postgres=# select * from country_log_ru;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 12:10:06.123189-08 | ru           | content-ru
(1 row)

postgres=# select * from country_log_sa;
          created_at          | country_code |  content
------------------------------+--------------+------------
 2014-11-30 12:10:14.22666-08 | sa           | content-sa
(1 row)
```

How it works...

The following is a detailed explanation of the steps carried out in the preceding section:

- PostgreSQL basically supports partitioning via table inheritance. Hence, partitioning is set up in such a way that every child table inherits from the parent table. For this purpose, we create two child tables, that is, country_log_ru and country_log_sa, in step 2 of the previous section. These child tables inherit from the parent or the master table, country_log, using the INHERITS keyword against the master table for the CREATE TABLE DDL statement for both the child tables. This was the initial setup.
- The next step, from our scenario, is to build partitioning in such a way that the logs by country are stored in a country-specific table. The case that we used in the previous section was to ensure that all the logs for Russia go in the country_log_ru table and all the logs for South Africa go in the country_log_sa table. To achieve this objective, we define a country_insert_trig trigger function, which helps partition the data into a country-specific table whenever an INSERT statement is triggered on the country_log master table. The moment the INSERT statement gets triggered on country_log master table, the country_log trigger gets fired upon which it calls country_insert_trig(). The country_insert_trig() trigger function checks the inserted records, and if it finds records for Russia (checked by the NEW.country_code = 'ru' condition) in the country_log table, then it inserts the said record in the country_log_ru child table. If the inserted record in the country_log master table is for South Africa (NEW.country_code = 'sa'), then it logs the same record in the country_log_sa child table. The trigger function partitions the data in this way. The following section of code, in the country_insert_trig() trigger function, uses the logic defined in the IF condition to partition the data into the child tables:

```
IF ( NEW.country_code = 'ru' ) THEN
  INSERT INTO country_log_ru VALUES (NEW.*);
ELSIF ( NEW.country_code = 'sa' ) THEN
  INSERT INTO country_log_sa VALUES (NEW.*);
ELSE
  RAISE EXCEPTION 'Country unknown';
END IF;
```

- Finally, once the data has been partitioned into the child tables, the final step is to verify the same by comparing the records from the child tables and the master table, as shown in step 7.

Initially, two records were inserted in the country_log master table. This can be confirmed by running the SELECT query against the country_log table. Here, we can see two log records in the country_log table, one for Russia, identified by the country code ru, and one for South Africa, identified by the country code sa:

```
postgres=# SELECT * from country_log;
          created_at           | country_code |   content
-------------------------------+--------------+------------
 2014-11-30 12:10:06.123189-08 | ru           | content-ru
 2014-11-30 12:10:14.22666-08  | sa           | content-sa
(2 rows)
```

The next step is to run the SELECT queries against the respective child tables, country_log_ru and country_log_sa:

```
postgres=# select * from country_log_ru;
          created_at           | country_code |   content
-------------------------------+--------------+------------
 2014-11-30 12:10:06.123189-08 | ru           | content-ru
(1 row)

postgres=# select * from country_log_sa;
          created_at          | country_code |   content
------------------------------+--------------+------------
 2014-11-30 12:10:14.22666-08 | sa           | content-sa
(1 row)
```

From the preceding output, you can see that there is only one record in each child table country_log_ru and country_log_sa. In effect, the country_insert_trig() trigger function has partitioned the log data in country-specific tables. Entries for the country_code column with the ru value, that is, for Russia, go into the country_log_ru table, and entries for the country_code column with the value sa, that is, South Africa, go into the country_log_sa child table.

There's more

You can refer to the following links for more detailed explanation on how to implement partitioning:

- https://blog.engineyard.com/2013/scaling-postgresql-performance-table-partitioning
- http://www.postgresql.org/docs/9.3/static/ddl-partitioning.html

Managing partitions

Here, we are going to show you how the partitioning scheme remains intact when an existing partition is dropped or a new partition is added.

Getting ready

Please refer to the first recipe, *Implementing partitioning*, before you read the steps outlined in this recipe.

How to do it…

There are two scenarios here. One scenario shows what happens when you delete an existing partition, and another shows what happens when a new partition is added. Let's discuss both the cases.

- In the first scenario, we will drop an existing partition table. Here, the country_code_sa table will be dropped, as follows:

  1. Before dropping the country_log_sa child table, see the records in the country_log master table:

     ```
     postgres=# SELECT * from country_log;
               created_at           | country_code |  content
     -------------------------------+--------------+------------
      2014-11-30 12:10:06.123189-08 | ru           | content-ru
      2014-11-30 12:10:14.22666-08  | sa           | content-sa
     (2 rows)
     ```

  2. Next, drop the country_log_sa child table, as shown here:

     ```
     postgres=# drop table country_log_sa;
     DROP TABLE
     ```

  3. Again, as a final step, recheck the data in the master table, country_log, once country_log_sa is dropped:

     ```
      postgres=# select * from country_log;
                created_at          | country_code |  content
      -----------------------------+--------------+------------
     2014-11-30 14:41:40.742878-08 | ru           | content-ru
     ```

- In the second scenario, we will add a partition. Let's add a new partition, `country_log_default`. The idea of creating this partition is that if there are tables for which the trigger function does not define any country codes, those records should go into a default table partition, as follows:

  1. Before we create the child table, let's see the existing records in the `country_log` master table:

     ```
     postgres=# select * from country_log;
               created_at           | country_code |  content
     -------------------------------+--------------+------------
      2014-11-30 14:41:40.742878-08 | ru           | content-ru
     ```

  2. Next, create a `country_log_sa` child table and create an index on the child table, as shown here:

     ```
     postgres=# CREATE TABLE country_log_default ( ) INHERITS
     (country_log);
     CREATE TABLE
     postgres=# CREATE INDEX country_code_default_idx ON
     country_log_default USING btree (country_code);
     CREATE INDEX;
     ```

  3. Modify your existing trigger function in order to define a condition to insert log records for those countries whose country codes are not explicitly defined in order to go into a country code specific log table:

     ```
     CREATE OR REPLACE FUNCTION country_insert_trig() RETURNS
     TRIGGER AS $$
     BEGIN
         IF ( NEW.country_code = 'ru' ) THEN
             INSERT INTO country_log_ru VALUES (NEW.*);
         ELSIF ( NEW.country_code = 'sa' ) THEN
             INSERT INTO country_log_sa VALUES (NEW.*);

         ELSE
             INSERT INTO country_log_default VALUES (NEW.*);
         END IF;

         RETURN NULL;
     END;
     $$ LANGUAGE plpgsql;
     ```

4. Now, insert records into the master table:

```
postgres=# INSERT INTO country_log (country_code, content)
VALUES ('dk', 'content-dk');
INSERT 0 0
postgres=# INSERT INTO country_log (country_code, content)
VALUES ('us', 'content-us');
INSERT 0 0
```

5. Let's check the newly created records in the `country_log` master table and see if these records have been partitioned into the `country_log_default` child table:

```
postgres=# select * from country_log;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 14:41:40.742878-08 | ru           | content-ru
 2014-11-30 15:10:28.921124-08 | dk           | content-dk
 2014-11-30 15:10:42.97714-08  | us           | content-us

postgres=# select * from country_log_default;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 15:10:28.921124-08 | dk           | content-dk
 2014-11-30 15:10:42.97714-08  | us           | content-us
(2 rows)
```

How it works...

First, let's discuss the first scenario where we drop the child partition table, `country_log_sa`. Here's the code snippet that was shown in the previous section:

```
postgres=# SELECT * from country_log;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 12:10:06.123189-08 | ru           | content-ru
 2014-11-30 12:10:14.22666-08  | sa           | content-sa
(2 rows)

postgres=# drop table country_log_sa;
DROP TABLE

postgres=# select * from country_log;
          created_at           | country_code |  content
-------------------------------+--------------+------------
 2014-11-30 14:41:40.742878-08 | ru           | content-ru
```

If you refer to the sequence of events in the preceding output, you can clearly see that once the child table, country_log_sa, got dropped, its corresponding entry from the country_log master table also got removed. This technique really helps if there are a large number of records to be pruned from the master table once the corresponding child table is dropped. This procedure is automatic and does not require DBA intervention. This way, the partition structure and data can be easily managed and handled if any existing partition is dropped.

Similarly, data can be easily managed when a new partition is added. If you refer to step 2 of the second scenario in the *How to do it…* section, you can see that we create a new child table, country_log_default, which inherits from the country_log master table. Once the existing trigger function, country_insert_trig(), is modified to include the condition-based insert for partitioning the data into the newly created partition, country_log_default, an INSERT statement is triggered on the country_log master table, and if the prevalent condition to insert records into the country_log_default child table is fulfilled, then the records are inserted into the country_log_default child table. This can be seen from steps 2 to 5 of the second scenario in the *How to do it…* section when we add a partition.

There's more

For a more detailed explanation on partitioning, go to http://www.postgresql.org/docs/9.3/static/ddl-partitioning.html.

Partitioning and constraint exclusion

In this recipe, we are going to talk about constraint exclusion and how it helps to improve performance.

Getting ready

Familiarity with table partitioning is required for this recipe.

How to do it…

Constraint exclusion can be enabled with the following command:

```
SET constraint_exclusion = ON ;
```

How it works…

Now, let's discuss constraint exclusion.

Constraint exclusion is basically a query optimization technique that helps to improve the performance of partitioned tables.

Let's just analyze the query plan for the following query:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM country_log WHERE
country_code = 'ru';
                            QUERY PLAN
---------------------------------------------------------------------
 Result  (cost=0.00..38.29 rows=16 width=52) (actual
time=26.442..27.298 rows=1 loops=1)
   ->  Append  (cost=0.00..38.29 rows=16 width=52) (actual
time=26.437..27.289 rows=1 loops=1)
         ->  Seq Scan on country_log  (cost=0.00..0.00 rows=1
width=52) (actual time=0.002..0.002 rows=0 loops=1)
               Filter: (country_code = 'ru'::bpchar)
         ->  Bitmap Heap Scan on country_log_ru country_log
(cost=4.29..12.76 rows=5 width=52) (actual time=26.431..26.433 rows=1
loops=1)
               Recheck Cond: (country_code = 'ru'::bpchar)
               ->  Bitmap Index Scan on country_code_ru_idx
(cost=0.00..4.29 rows=5 width=0) (actual time=26.413..26.413 rows=1
loops=1)
                     Index Cond: (country_code = 'ru'::bpchar)
         ->  Bitmap Heap Scan on country_log_au country_log
(cost=4.29..12.76 rows=5 width=52) (actual time=0.822..0.822 rows=0
loops=1)
               Recheck Cond: (country_code = 'ru'::bpchar)
               ->  Bitmap Index Scan on country_code_au_idx
(cost=0.00..4.29 rows=5 width=0) (actual time=0.817..0.817 rows=0
loops=1)
                     Index Cond: (country_code = 'ru'::bpchar)
         ->  Bitmap Heap Scan on country_log_default country_log
(cost=4.29..12.76 rows=5 width=52) (actual time=0.023..0.023 rows=0
loops=1)
               Recheck Cond: (country_code = 'ru'::bpchar)
               ->  Bitmap Index Scan on country_code_default_idx
(cost=0.00..4.29 rows=5 width=0) (actual time=0.013..0.013 rows=0
loops=1)
                     Index Cond: (country_code = 'ru'::bpchar)
 Total runtime: 27.442 ms
(17 rows)
```

If you analyze the preceding query plan for the preceding `SELECT` query, you will find that the query scans each of the partitions of the `country_log` table. This behavior is suboptimal from a query-performance perspective.

To deal with this scenario, we can enable constraint exclusion. By doing so, the query planner will examine the contents of each partition; however, the planner will try to prove that scanning is not required for the partitions that do not meet the query's criteria defined in the `WHERE` clause. When the planner can prove this, it excludes such partitions from the query plan. This can be seen in the query plan generated for the query after the constraint exclusion is enabled, as shown here:

```
postgres=# SET constraint_exclusion = ON;
SET

postgres=# EXPLAIN ANALYZE SELECT * FROM country_log WHERE
country_code = 'ru';
                           QUERY PLAN
---------------------------------------------------------------------
 Result  (cost=0.00..25.52 rows=11 width=52) (actual time=0.036..0.147
rows=1 loops=1)
   ->  Append  (cost=0.00..25.52 rows=11 width=52) (actual
time=0.031..0.138 rows=1 loops=1)
         ->  Seq Scan on country_log  (cost=0.00..0.00 rows=1
width=52) (actual time=0.003..0.003 rows=0 loops=1)
               Filter: (country_code = 'ru'::bpchar)
         ->  Bitmap Heap Scan on country_log_ru country_log
(cost=4.29..12.76 rows=5 width=52) (actual time=0.025..0.027 rows=1
loops=1)
               Recheck Cond: (country_code = 'ru'::bpchar)
               ->  Bitmap Index Scan on country_code_ru_idx
(cost=0.00..4.29 rows=5 width=0) (actual time=0.017..0.017 rows=1
loops=1)
                     Index Cond: (country_code = 'ru'::bpchar)
         ->  Bitmap Heap Scan on country_log_default country_log
(cost=4.29..12.76 rows=5 width=52) (actual time=0.102..0.102 rows=0
loops=1)
               Recheck Cond: (country_code = 'ru'::bpchar)
               ->  Bitmap Index Scan on country_code_default_idx
(cost=0.00..4.29 rows=5 width=0) (actual time=0.096..0.096 rows=0
loops=1)
                     Index Cond: (country_code = 'ru'::bpchar)
 Total runtime: 0.230 ms
(13 rows)
```

We can see an improved performance in the query plan, as this one shows a total runtime of 0.230 milliseconds, whereas the preceding query plan shows a total runtime of 27.442 milliseconds. Thus, you can see the performance benefits by enabling constraint exclusion.

Alternate partitioning methods

In this recipe, we are going to talk about another mechanism that can be used to redirect INSERTS into the appropriate partitions. Here, we are going to talk about using the rule-based approach instead of the trigger-based approach, in order to redirect INSERTS into the appropriate partitions.

Getting ready

Familiarity with table partitioning is required for this recipe.

How to do it...

What we are going to do now is to use a rule-based approach. To do this, perform the following steps:

1. To avoid any conflicts with the previously used trigger-based approach, proceed by dropping the existing trigger function, using the following command:

   ```
   postgres=# drop function country_insert_trig() cascade;
   ```

2. The next step will be to subsequently create rules for each of the child tables, so that whenever a new record is inserted in the master table (country_log), the rules get invoked to redirect the INSERT commands to the appropriate partition table, as shown here:

   ```
   CREATE RULE country_code_check_ru AS
   ON INSERT TO country_log WHERE
       ( NEW.country_code = 'ru' )
   DO INSTEAD
       INSERT INTO country_log_ru VALUES (NEW.*);


   CREATE RULE country_code_check_sa AS
   ON INSERT TO country_log WHERE
       ( NEW.country_code = 'sa' )
   DO INSTEAD
       INSERT INTO country_log_sa VALUES (NEW.*);



   CREATE RULE country_code_check_default AS
   ```

```
ON INSERT TO country_log WHERE
    ( NEW.country_code != 'ru' OR NEW.country_code != 'sa' )
DO INSTEAD
    INSERT INTO country_log_default VALUES (NEW.*);
```

3. Next, insert the record into the master table, that is, `country_log`:

    ```
    INSERT INTO country_log (country_code, content) VALUES ('ca',
    'content-ca');
    ```

4. Finally, use the `SELECT` query against the respective partition table to check whether the `INSERT` commands used in the previous step are redirected to the appropriate partition table using the rule-based approach, as follows:

    ```
    postgres=# select * from country_log_default;
              created_at           | country_code |  content
    -------------------------------+--------------+------------
     2014-11-30 15:10:28.921124-08 | dk           | content-dk
     2014-11-30 15:10:42.97714-08  | us           | content-us
     2014-12-01 14:36:27.746601-08 | ca           | content-ca
    (3 rows)
    ```

How it works...

What we are basically doing here is to create rules for all the partitions. The condition defined in the rules is the same as the one defined in the trigger function, `country_insert_trig()`. Let's show the trigger function's code from which the conditions defined for the rules were derived:

```
CREATE OR REPLACE FUNCTION country_insert_trig() RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.country_code = 'ru' ) THEN
        INSERT INTO country_log_ru VALUES (NEW.*);
    ELSIF ( NEW.country_code = 'sa' ) THEN
        INSERT INTO country_log_sa VALUES (NEW.*);

    ELSE
        INSERT INTO country_log_default VALUES (NEW.*);
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

If you take a look at the preceding trigger function, it is clear that when the *inserted* data into the country_log master table has the values for the country_code column as ru or sa, then the corresponding inserted row will either go to the country_log_ru or country_log_sa table, depending on what the inserted country_code entry is. If the value for the country_code column inserted is anything else other than these two values, then the corresponding row entry is directed to the country_log_default table. Based on these conditions, we define the rules for all the child tables, as follows:

```
CREATE RULE country_code_check_ru AS
ON INSERT TO country_log WHERE
    ( NEW.country_code = 'ru' )
DO INSTEAD
    INSERT INTO country_log_ru VALUES (NEW.*);

CREATE RULE country_code_check_sa AS
ON INSERT TO country_log WHERE
    ( NEW.country_code = 'sa' )
DO INSTEAD
    INSERT INTO country_log_sa VALUES (NEW.*);

CREATE RULE country_code_check_default AS
ON INSERT TO country_log WHERE
    ( NEW.country_code != 'ru' OR NEW.country_code != 'sa' )
DO INSTEAD
    INSERT INTO country_log_default VALUES (NEW.*);
```

Once the rules are defined for the child tables, the next step is to remove the previously used trigger function in order to avoid any conflict with the trigger- and rule-based approaches. Finally, we add the data to the master table, and we can see in step 4 of the *How to do it…* section, that based on the rule-based approach, the corresponding entry goes to the country_log_default partition.

Installing PL/Proxy

PL/Proxy is a database partitioning system that is implemented as a PL language. PL/Proxy makes it straightforward to split large independent tables among multiple nodes in a way that almost allows unbounded scalability. PL/Proxy scaling works on both read and write workloads. The main idea is that the proxy function will be set up with the same signature as the remote function to be called, so only the destination information needs to be specified inside the proxy function's body.

Getting Ready

Here, we are going to show the steps required to install PL/Proxy.

How to do it...

Perform the following steps to install PL/Proxy:

1. Go to `http://pgfoundry.org/projects/plproxy/` and download the latest tarball of PL/Proxy.
2. Once the latest version of PL/Proxy is downloaded, the next step is to unpack the tar archive:

   ```
   tar xvfz plproxy-2.5.tar.gz
   ```

3. Once the tar archive has been unpacked, the next step is to enter the newly created directory and start the compilation process:

   ```
   cd   plproxy-2.5
   make  &&  make install
   ```

How it works...

Installing PL/Proxy is an easy task. Here, we download the source code from the website provided in step 1 of the preceding section. The latest version of PL/Proxy at this stage is 2.5. We need to download the tarball file containing version 2.5 of PL/Proxy, and once it is downloaded, we need to compile and build it. This completes the installation of PL/Proxy.

You can also install PL/Proxy from binary packages, if prebuilt packages are available for your operating system.

Partitioning with PL/Proxy

In this recipe, we are going to cover horizontal partitioning with PL/Proxy.

Getting ready

PL/Proxy needs to be installed on the host machine. Refer to the previous recipe for more details on how to install PL/Proxy.

How to do it...

Perform the following sequence of steps to perform horizontal partitioning using PL/Proxy:

1. Create three new databases, that is one proxy database named `nodes` and two partitioned databases named `nodes_0000` and `nodes_0001`, respectively:

   ```
   postgres=# create database nodes;
   postgres=# create database nodes_0000;
   postgres=# create database nodes_0001;
   ```

2. Once you've created these databases, the next step is to create a `plproxy` extension:

   ```
   psql  -d  nodes
   ```

   ```
   nodes=# create extension plproxy;
   ```

3. The next step is to create the `plproxy` schema in the proxy database's nodes:

   ```
   nodes=# create schema plproxy;
   ```

4. Next, execute the following file, `plproxy--2.5.0.sql`, on the proxy database nodes:

   ```
   cd  /usr/pgsql-9.3/share/extension
    psql -f plproxy--2.5.0.sql  nodes


   CREATE FUNCTION
   CREATE LANGUAGE
   CREATE FUNCTION
   CREATE FOREIGN DATA WRAPPER
   ```

5. Then, configure PL/Proxy using the configuration functions on the proxy database nodes:

   ```
   psql -d   nodes


   CREATE OR REPLACE FUNCTION plproxy.get_cluster_version
   (cluster_name text) RETURNS int AS $$
   BEGIN
       IF cluster_name = 'nodes' THEN
           RETURN 1;
       END IF;
   END;
   $$ LANGUAGE plpgsql;


   CREATE OR REPLACE FUNCTION plproxy.get_cluster_partitions
   (cluster_name text) RETURNS SETOF text AS $$
   ```

```
BEGIN
    IF cluster_name = 'nodes' THEN
        RETURN NEXT 'host=127.0.0.1 dbname=nodes_0000';
        RETURN NEXT 'host=127.0.0.1 dbname=nodes_0001';
        RETURN;
    END IF;
    RAISE EXCEPTION 'no such cluster: %', cluster_name;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;


CREATE OR REPLACE FUNCTION plproxy.get_cluster_config
(cluster_name text, out key text, out val text)
RETURNS SETOF record AS $$
BEGIN
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

6. Next, log in to the partitioned databases and create the users table in both of these:

   psql -d nodes_0000

   nodes_0000=# CREATE TABLE users (username text PRIMARY KEY);

   psql -d nodes_0001

   nodes_0001=# CREATE TABLE users (username text PRIMARY KEY);

7. Now, create the following function, `insert_user()`, which will be used to insert usernames in the users table:

```
psql -d   nodes_0000

CREATE OR REPLACE FUNCTION insert_user(i_username text) RETURNS
text AS $$
BEGIN
    PERFORM 1 FROM users WHERE username = i_username;
    IF NOT FOUND THEN
        INSERT INTO users (username) VALUES (i_username);
        RETURN 'user created';
    ELSE
        RETURN 'user already exists';
    END IF;
END;
```

```
$$ LANGUAGE plpgsql SECURITY DEFINER;



psql  -d  nodes_0001

CREATE OR REPLACE FUNCTION insert_user(i_username text) RETURNS
text AS $$
BEGIN
    PERFORM 1 FROM users WHERE username = i_username;
    IF NOT FOUND THEN
        INSERT INTO users (username) VALUES (i_username);
        RETURN 'user created';
    ELSE
        RETURN 'user already exists';
    END IF;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

8. Next, create a proxy function called `insert_user()` on the proxy database nodes:

```
psql -d nodes

CREATE OR REPLACE FUNCTION insert_user(i_username text) RETURNS
TEXT AS $$
    CLUSTER 'nodes'; RUN ON hashtext(i_username);
$$ LANGUAGE plproxy;
```

9. Check the `pg_hba.conf` file; you will need to set the authentication to `trust`, as shown here, and then restart the `postgresql` service:

```
host    all             all             127.0.0.1/32
trust
```

10. The next step will be to fill the partitions by executing the following query on the proxy database nodes:

```
nodes=#SELECT insert_user('user_number_'||generate_series::text)
FROM generate_series(1,10000);
```

11. Once the data is inserted, verify the corresponding records in the partitioned databases, `nodes_0000` and `nodes_0001`, as follows:

```
nodes_0000=# select count(*) from users;
 count
-------
  5106
(1 row)

nodes_0001=# select count(*) from users;
```

```
 count
-------
  4894
(1 row)
```

How it works...

The following is an explanation of the preceding code output:

- Initially, we create three databases—one as a proxy database named `nodes` and two other databases named `nodes_0000` and `nodes_0001`—across which the data will be partitioned.
- Once the preceding step is performed, the next step will be to create the `plproxy` extension.
- As can be seen in step 5 of the preceding section, we are configuring PL/Proxy using the configuration functions on the proxy database nodes. The `plproxy.get_cluster_partitions()` function is invoked when a query needs to be forwarded to a remote database, and it is used by PL/Proxy to obtain the connection string to be used for each partition. We also use the `plproxy.get_cluster_version()` function, which is called upon each request, and it is used to determine whether the output from a cached result from `plproxy.get_cluster_partitions` can be reused. We also use the the `plproxy.get_cluster_config()` function, which enables us to configure the behavior of PL/Proxy.
- Once we are done with defining the configuration functions on the proxy database nodes, the next step is to create the table users in both the partitioned databases, across which the data will be partitioned.
- Then, we created an `insert_user()` function that will be used to insert usernames into the users table. The `insert_user()`function will be defined on both the partitioned databases, `nodes_0000` and `nodes_0001`. This is shown in step 7 of the preceding section.
- In the next step, we create a proxy function, `insert_user()`, inside the proxy database nodes. The proxy function will be used to send the `INSERT` result to the appropriate partition. This is shown in step 8 of the preceding section.
- Finally, we will be filling the partitions with random data by executing the `insert_user()` proxy function in the proxy database named nodes. This is seen in step 10 of the preceding section.

There's more

For more details on how to use PL/Proxy in order to proxy queries across a set of remote databases, check out `http://plproxy.projects.pgfoundry.org/doc/tutorial.html`.

10
Accessing PostgreSQL from Perl

In this chapter, we will cover the following recipes:

- Making a connection to a PostgreSQL database using Perl
- Creating tables using Perl
- Inserting records using Perl
- Accessing data using Perl
- Updating records using Perl
- Deleting records using Perl

Introduction

Perl is a general-purpose, high-level, interpreted, and dynamic programming language. Generally, communicating with PostgreSQL involves a lot of string manipulation, and this is where Perl excels as a language. In Perl, database interfaces are implemented by Perl DBI modules. A DBI module presents a database-independent interface to Perl applications. On the other hand, the database driver module handles the details of accessing different databases.

There are three ways to access PostgreSQL from Perl, stated as follows:

- Low-level access, which is done by the Perl mapping of the `libpq` C interface
- High-level access, with the help of a database-independent layer
- Access by embedding a Perl interpreter

Making a connection to a PostgreSQL database using Perl

Here, we are going to make connections to a PostgreSQL database using Perl.

Getting ready

The following instructions are performed on a CentOS Linux machine, and it is assumed that the Perl language is already installed.

A PostgreSQL database can be accessed by using the Perl DBI module, which is a database access module for the Perl programming language. The Perl DBI module defines a set of methods, variables, and conventions that provide a standard database interface.

The DBI module, by itself, does not have the ability to communicate with PostgreSQL. For the DBI module to communicate with PostgreSQL, it is necessary to install the appropriate backend module, which in this case is DBD::Pg.

On Red Hat, CentOS, Scientific Linux, as well as other Red Hat based Linux distributions, the package that provides this module is `perl-DBD-Pg` and it can be installed as follows:

```
yum install perl-DBD-Pg
```

On Debian-based systems, the package that provides this module is `libdbd-pg-perl`, and it can be installed as follows on Ubuntu- or Debian-based distributions:

```
apt-get install libdbd-pg-perl
```

Before we start using the Perl PostgreSQL interface, we will need to enter the following authentication and access control mechanism entry in the `pg_hba.conf` file:

```
# IPv4 local connections:
host    all         all         127.0.0.1/32          md5
```

Once these changes are done, we will need to restart the PostgreSQL server:

```
$ pg_ctl -D  $PGDATA  restart
```

How to do it...

We can use the following Perl code to make a connection to an existing PostgreSQL database, that is, the `dvdrental` database, which resides on the same machine and uses port 5432.

1. First, the following Perl code can be saved in a file called `connect.pl`:

```
#!/usr/bin/perl

use DBI;
use strict;
   my $driver   = "Pg";
my $database = "dvdrental";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
})
                      or die $DBI::errstr;

print "CONNECTION TO THE  DVDRENTAL DATABASE MADE SUCCESSFULLY\n";
```

2. The next step will be to change permissions, as follows:

```
chmod 755 connect.pl
```

3. The Perl program can then be executed at the command line, as follows:

```
bash-3.2$ perl connect.pl
CONNECTION MADE TO THE  DVDRENTAL DATABASE MADE SUCCESSFULLY
```

   As can be seen from the preceding output, while the program is being executed, the output message indicates that the connection to the `dvdrental` database has been made successfully.

How it works...

The connection to the database is made using the `connect` function. It returns a connection handle that is needed when calls are made to the DBI module. The `connect` function requires the following argument:

```
connect($data_source, "userid", "password", \%attr);
```

The first argument to the connect function is the data source name, which is a single entity that comprises of the database name and the host name or IP address and optionally, a port number. The data source also comprises of the prefix Pg, which is the PostgreSQL database driver for the DBI module.

The second argument is userid or the username by which a connection to the PostgreSQL database is made.

The third argument is the password, which is the password of the user who initiates the database connection. If an empty string is provided for the password, Perl will then look for a password value in the environment variables, DBI_USER and DBI_PASS, which could possibly cause the code to fail while it is being executed. So, we need to exercise caution in such scenarios.

The final argument is optional, and it refers to any attributes that might be used.

In the preceding code, in the *How to do it…* section, we used the connect function, as follows:

```
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
```

The first argument used here is the dsn variable, which was initially defined, as follows:

```
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
```

Here, we can see that the dsn value is a single entity, which comprises of a driver, the Pg driver; dbname, which is defined in the variable database and its value is dvdrental; and further consists of the hostname and port number being explicitly defined here.

The second argument is the userid variable, which uses the value postgres defined previously in the code.

The third argument is the password variable, whose value is postgres.

The final argument used is the RaiseError attribute, which causes the DBI module to call the HandleError condition or to die if the HandleError condition is not defined when a database error is detected.

> If you need more information on the connect function, you can use the following links for a more detailed explanation:
>
> - http://oreilly.com/catalog/perldbi/chapter/ch04.html
> - http://search.cpan.org/~rudy/DBD-Pg-1.32/Pg.pm

Creating tables using Perl

In this recipe, we are going to show you how to create tables in the PostgreSQL database using Perl.

Getting ready

We will be the using the qq operator, and the parameter passed to the operator will contain the CREATE TABLE SQL statement. The qq operator is used to return a double-quoted string. Before creating the table, we must first use the connect function to connect to the PostgreSQL database.

How to do it...

We can use the following code to create a table by the name EMPLOYEES. This table will be stored in the dvdrental database because the connection made by the PostgreSQL adapter is to the dvdrental database. The following code is saved in a file called createtable.pl, which will be executed later:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "Pg";
my $database = "dvdrental";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Dvdrental database opened \n";

my $stmt = qq(CREATE TABLE EMPLOYEES
      (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(60),
       SALARY         REAL););
my $rv = $dbh->do($stmt);
```

```
 print "EMPLOYEES table created successfully\n\n";

$dbh->disconnect();

bash-3.2$ perl createtable.pl

Dvdrental database opened
EMPLOYEES table created successfully
```

In the preceding output, you can see that when the file containing the preceding code is executed, a connection to the `dvdrental` database is made and an `EMPLOYEES` table is created. This can be seen from the command-line console message, `EMPLOYEES table created successfully`.

How it works...

From the point of view of table creation, it is the following part of the preceding code that needs an explanation:

```
my $rv = $dbh->do($stmt);
```

Here, we are using the handler returned by the `connect` function in conjunction with the `do` function to execute the `CREATE TABLE` statement passed in the `$stmt` variable, as follows:

```
my $stmt = qq(CREATE TABLE EMPLOYEES
      (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(60),

       SALARY         REAL););
```

The `do()` method is a fusion of `prepare()` and `execute()`. It can only be used for non-SELECT statements, where you don't need the statement handle to access the results of the query. `do()` returns the number of affected rows.

The `disconnect()` method, in the preceding section, is used to terminate the existing database session and disconnect from the database.

> For more details on tables in Perl, check out `http://www.postgresql.org/docs/9.3/interactive/plperl-builtins.html`.

Inserting records using Perl

In this recipe, we are going to insert new records in the EMPLOYEES table in the dvdrental database.

Getting ready

Before inserting records in the table, we first need to use the connect function, in order to connect to the database. The connect function was discussed in the first recipe of the chapter.

How to do it...

We are going to use the following code to insert new records in the EMPLOYEES table:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "Pg";
my $database = "dvdrental";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $irows = 0;
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (5, 'SandeepSingh', 37, 'Saharanpur', 90000.00 ));
my $rv = $dbh->do($stmt);


$irows = $rv + $irows;

$stmt = qq(INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (6, 'AmitGovil', 37, 'Aligarh', 85000.00 ));
```

```
   $rv = $dbh->do($stmt);


   $irows = $rv + $irows;

   $stmt = qq(INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY)
         VALUES (7, 'NeerajKumar', 38, 'Rohtak', 90000.00 ));
   $rv = $dbh->do($stmt);


   $irows = $rv + $irows;

   $stmt = qq(INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY)
         VALUES (8, 'SandeepSharma', 36, 'Gurgaon ', 75000.00 ););
   $rv = $dbh->do($stmt);


   $irows = $rv + $irows;

   print "Number of rows inserted : $irows\n";
   print "New Records created successfully\n";
   $dbh->disconnect();
```

The preceding code is saved in a file called `insert.pl`, and we get the following command-line output once the new records are inserted successfully:

```
bash-3.2$ perl  insert.pl
Opened database successfully
New Records created successfully
```

How it works...

For an explanation of the preceding code, we are taking an excerpt of the code that will demonstrate how the records are getting inserted into the `EMPLOYEES` table:

```
   my $stmt = qq(INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY)
         VALUES (5, 'SandeepSingh', 37, 'Saharanpur', 90000.00 ));
   my $rv = $dbh->do($stmt);
```

If you take a look at the preceding code, you can see that first, we used the `INSERT` statement, defined records, and passed the `INSERT` SQL statement to a variable. After this is done, we used the `do()` function to return the result of the `INSERT` statement into the table. The same steps are performed sequentially for the other `INSERT` statements used in the code.

In the preceding section, we are also using the irows variable to track the number of rows that were inserted into the table. Every time we insert a record into the table, we set the condition as shown in the following line of code. Initially, the value of the irows variable is set to zero, and whenever we insert a record into the EMPLOYEES table, the value of the rv variable is set to 1. So, as per the following condition, every time there is a change or a record is inserted, the irows variable's value will increment by 1 and so on, until all the records are inserted and eventually it stops at 4 to indicate that four records were inserted in total:

```
$irows = $rv + $irows;
```

Accessing table data using Perl

In this recipe, we are going to see how to access the table data from a PostgreSQL database using Perl.

Getting ready

A database connection is mandatory before we can select data. Hence, for this reason, the connect() function is the first one that should be invoked to make a database connection before accessing data.

How to do it...

We can use the following code to access data from the EMPLOYEES table present in the dvdrental database. The following code is saved in a file called select.pl, which will be later executed from the command line:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "Pg";
my $database = "dvdrental;";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 })
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(SELECT id, name, address, salary  from EMPLOYEES;);
my $sth = $dbh->prepare($stmt) or die "Cannot prepare: " . $dbh-
>errstr();

my $rv = $sth->execute() or die "Cannot execute: " . $sth->errstr();

while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
$sth->finish();
print "select Operation done successfully\n";
$dbh->disconnect();
```

The following is the output of the preceding code:

```
bash-3.2$ perl select.pl
Opened database successfully
ID = 5
NAME = SandeepSingh
ADDRESS = Saharanpur
SALARY =  90000

ID = 6
NAME = AmitGovil
ADDRESS = Aligarh
SALARY =  85000

ID = 7
NAME = NeerajKumar
ADDRESS = Rohtak
SALARY =  90000

ID = 8
NAME = SandeepSharma
ADDRESS = Gurgaon
SALARY =  75000

select Operation done successfully
```

How it works...

The following is the subpart of the code that mainly deals with selecting records from a table:

```
my $stmt = qq(SELECT id, name, address, salary  from EMPLOYEES;);
my $sth = $dbh->prepare( $stmt ) or die "Cannot prepare: " . $dbh-
>errstr();

my $rv = $sth->execute() or die "Cannot prepare: " . $dbh->errstr();



while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
 $sth->finish();
```

In the preceding code, the first thing that we do is write down our `SELECT` query and pass it to the `$stmt` variable.

The next step is to use the `prepare()` function in order to prepare the SQL statement, which can be executed at a later time by the database engine and returns a reference to the statement handle object.

The next step is to use the `execute()` function in order to execute the prepared statement.

Finally, in order to fetch the results of the `SELECT` command, we use the `fetchrow_array()` function. The `fetchrow_array()` function gets the next row and returns it as a list of field values. We use the `while` loop to iterate through each of the field values in a given row across the array named `row` and then move on to the next row. The same sequence of events are repeated until we have iterated through the last field value in the final row returned.

Updating records using Perl

Here, we are going to see how to update existing records in a table in the PostgreSQL database using Perl.

Getting ready

In this recipe, first we are going to show the number of existing records in the table. Then, we are going to update some records, see the number of records updated, and then see the changed records being made visible in the table when the table records are accessed again.

How to do it...

In this section we will update the existing records of the EMPLOYEES table.

1. First, we check the existing records in the `EMPLOYEES` table, as follows:

```
dvdrental=# select * from EMPLOYEES;

 id |      name      | age | address        | salary
----+---------------+-----+---------------+--------
  5 | SandeepSingh  |  37 | Saharanpur     |  90000
  6 | AmitGovil     |  37 | Aligarh        |  85000
  7 | NeerajKumar   |  38 | Rohtak         |  90000
  8 | SandeepSharma |  36 | Gurgaon        |  75000
```

2. Next, we use the following Perl code to update some of the existing records in the `EMPLOYEES` table and save the following code in a file called `update.pl`:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "Pg";
my $database = "dvdrental";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 
})
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(UPDATE EMPLOYEES set SALARY = 55000.00 where ID=5;);
my $rv = $dbh->do($stmt);



   print "Number of rows updated : $rv\n";


$stmt = qq(SELECT id, name, address, salary  from EMPLOYEES;);
my $sth = $dbh->prepare( $stmt ) or die "Check again: " . $dbh-
>errstr();

$rv = $sth->execute() or die "Cannot execute: " . $sth->errstr();


while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
$sth->finish();
print "Operation Completed successfully\n";
$dbh->disconnect();
```

In the preceding code, we use the `UPDATE` statement to set the `SALARY` value to `55000`, where the value of the `ID` column is `5`.

3. Next, we are going to see the updated records and see the changed records being made visible in the `EMPLOYEES` table, as shown in the following code output:

```
bash-3.2$ perl update.pl
Opened database successfully
Number of rows updated : 1
ID = 6
NAME = AmitGovil
ADDRESS = Aligarh
SALARY =  85000

ID = 7
NAME = NeerajKumar
ADDRESS = Rohtak
```

```
SALARY =  90000

ID = 8
NAME = SandeepSharma
ADDRESS = Gurgaon
SALARY =  75000

ID = 5
NAME = SandeepSingh
ADDRESS = Saharanpur
SALARY =  55000

Operation Completed successfully
```

How it works...

The following is an excerpt of the code that was used to update existing records using Perl into the `EMPLOYEES` table:

```
my $stmt = qq(UPDATE EMPLOYEES set SALARY = 55000.00 where ID=5;);
my $rv = $dbh->do($stmt);
print "Number of rows updated : $rv\n";
```

The first initial requirement is to connect to the database using the `connect()` function, which was explained in the first recipe of the chapter.

Once the connection to the `dvdrental` database is made, we use the `UPDATE` statement and pass the `UPDATE` SQL statement in a `$stmt` variable. After this is done, the next step is to use the `do()` function to return the result of the `UPDATE` statement contained in the `$stmt` variable. We also use a variable called `$rv`, which is used to track the number of records updated, if any. Once this is done, the next step is to fetch the records from the table in order to validate the changes done as part of using the `UPDATE` statement.

Deleting records using Perl

In this recipe, we are going to show you how to delete records in a table using Perl.

Getting ready

In this recipe, we will first display the number of existing records in the table. Then, we will delete some records, see the number of records deleted, and then see the number of available records present in the table after deletion.

How to do it...

In this section we will delete the existing records of the `EMPLOYEES` table.

1. First, we are going to check the existing records in the `EMPLOYEES` table, as shown here:

```
dvdrental=# select * from EMPLOYEES;
 id |      name      | age | address          | salary
----+----------------+-----+------------------+--------
  6 | AmitGovil      |  37 | Aligarh          |  85000
  7 | NeerajKumar    |  38 | Rohtak           |  90000
  8 | SandeepSharma  |  36 | Gurgaon          |  75000
  5 | SandeepSingh   |  37 | Saharanpur       |  55000
(4 rows)
```

2. Next, we are going to use the following Perl code to delete some records from the `EMPLOYEES` table and save the code in a file called `delete.pl`, which we are going to execute from the command line:

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "Pg";
my $database = "dvdrental";
my $dsn = "DBI:$driver:dbname=$database;host=127.0.0.1;port=5432";
my $userid = "postgres";
my $password = "postgres";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1 
})
                      or die $DBI::errstr;
```

```
print "Opened database successfully\n";

my $stmt = qq(DELETE from EMPLOYEES where ID=6;);
my $rv = $dbh->do($stmt);


   print "Number of rows deleted : $rv\n";


$stmt = qq(SELECT id, name, address, salary  from EMPLOYEES;);
my $sth = $dbh->prepare( $stmt ) or die "Cannot prepare: " . $dbh-
>errstr();

$rv = $sth->execute() or die "Cannot execute: " . $sth->errstr();


while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
$sth->finish();
print "Operation done successfully\n";
$dbh->disconnect();
```

Here, in the preceding code, the `DELETE` statement that we issue is used to delete a record from the table where the value of the `ID` column is `6`.

3. Next we check the command-line output of the preceding code:

```
bash-3.2$ perl delete.pl
Opened database successfully
Total number of rows deleted : 1
ID = 7
NAME = NeerajKumar
ADDRESS = Rohtak
SALARY =  90000

ID = 8
NAME = SandeepSharma
ADDRESS = Gurgaon
```

```
SALARY =  75000

ID = 5
NAME = SandeepSingh
ADDRESS = Saharanpur
SALARY =  55000

Operation done successfully
```

How it works...

The following is an excerpt from the code that was used to delete records from the EMPLOYEES table:

```
my $stmt = qq(DELETE from EMPLOYEES where ID=6;);
my $rv = $dbh->do($stmt);
print "Number of rows deleted : $rv\n";
```

The first initial requirement is to connect to the database using the connect() function, which was explained in the first recipe of the chapter.

Once the connection to the dvdrental database is made, we use the DELETE statement and pass the DELETE SQL statement in a $stmt variable. After this is done, the next step is to use the do() function in order to return the result of the DELETE statement contained in the $stmt variable. We also use a variable called $rv, which is used to track the number of rows deleted from the table. Once this is done, the next step is to fetch the records from the table to see the list of available records in the EMPLOYEES table.

11

Accessing PostgreSQL from Python

In this chapter, we will cover the following recipes:

- Making connections to a PostgreSQL database using Python
- Creating tables using Python
- Inserting records using Python
- Accessing data using Python
- Updating records using Python
- Deleting records using Python

Introduction

Python is a general purpose, dynamic object oriented and a high level programming language. Python is an open source, well designed, robust and portable programming language. Python has an easy to learn syntax and with its advanced programming features it is widely used by developers and administrators worldwide. Python provides an easy way for database access via the DB API which provides a minimal standard for working with databases using Python syntax and semantics.

The steps involved in using the Python API are given as follows:

- Importing the API module
- Establishing a database session
- Executing SQL statements
- Closing the database session

Making connections to a PostgreSQL database using Python

Here, in this recipe, we are going to make connections to a PostgreSQL database using Python language.

Getting ready

The following instructions are performed on a CentOS Linux machine and it is assumed that python language is already installed.

PostgreSQL database can be accessed using `psycopg2` module which is a database adapter for Python language. This can be installed, as follows, on a CentOS machine:

```
sudo yum install python-psycopg2
```

How to do it…

We can use the following Python code to make connections to an existing PostgreSQL database, that is the `dvdrental` database which resides on the same machine and uses port 5432.

The following Python code can be saved in a file called `connect.py`:

```
#!/usr/bin/python

import psycopg2

conn = psycopg2.connect(database="dvdrental", user="postgres",
password="postgres", host="127.0.0.1", port="5432")

print "Opened DVD Rental Database Successfully Using Python"
```

The preceding Python code can then be executed at the command line, as follows:

```
-bash-3.2$ python connect.py
```

The following is the output of the preceding code:

```
Opened DVD Rental Database Successfully Using Python
```

As can be seen from the preceding output, while the program is being executed the output message indicates that the connection to the database `dvdrental` has been made successfully.

How it works...

The connection to the database is made using the `connect` function which returns a connection object. The `connect` function, used here in the preceding code, consists of the following parameters:

```
connect("db", "userid", "password", host,port);
```

The first argument to the `connect` function, is the database to which the connection is to be made to.

The second argument is the `userid` or the username by which a connection to the PostgreSQL database is made.

The third argument is the password which is the password of the user who initiates the database connection.

The fourth argument refers to the hostname or the IP address of the server hosting the database.

The next argument refers to the port number of the database on which the client can initiate the database connection.

In the preceding code, in the *How to do it...* section, we used the `connect` function, as follows:

```
    conn = psycopg2.connect(database="dvdrental", user="postgres",
    password="postgres", host="127.0.0.1", port="5432")
```

Here, we are connecting to the `dvdrental` database using the `postgres` user and password `postgres`. The machine to which we are connecting is the localhost and the database server listens on port 5432 for connections.

If you need more information on the `connect` function you can use the following links for a more detailed explanation:

- `http://blogs.wrox.com/article/using-the-python-database-apis/`
- `http://initd.org/psycopg/docs/module.html#psycopg2.connect`

Creating tables using Python

Here, in this recipe, we are going to show how to create tables in the PostgreSQL database using Python language.

Getting ready

Before creating a table, we first need to make a connection to the PostgreSQL database using the `connect` function, and once the database is opened then we can use DDL statements to create the table.

How to do it...

We can use the following code to create a table by the name `EMPLOYEES`. This table will be stored in the `dvdrental` database because the connection made by the PostgreSQL adapter is to the `dvdrental` database. The following code is saved in a file called `createtable.py` which will be executed later:

```
#!/usr/bin/python

import psycopg2

conn = psycopg2.connect(database="dvdrental", user="postgres",
password="postgres", host="127.0.0.1", port="5432")
print "Opened DVD Rental Database Successfully"

cur = conn.cursor()
cur.execute('''CREATE TABLE EMPLOYEES
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully"

conn.commit()
conn.close()
```

In the following output, we can see that when the `createtable.py` file, which contains the preceding code is executed, a connection to the `dvdrental` database is made and the `EMPLOYEES` table is created. This can be seen from the command line console message `Table created successfully`:

```
bash-3.2$ python createtable.py
Opened DVD Rental Database Successfully
Table created successfully
```

How it works...

From the point of view of table creation, it is the following part of the preceding code that needs explanation:

```
cur = conn.cursor()
cur.execute('''CREATE TABLE EMPLOYEES
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully"
conn.commit()
conn.close()
```

First we create a cursor object by invoking the connection object's `cursor()` function. Once this is done, we use the cursor object's `execute()` function to execute the `CREATE TABLE` DDL statement to create a table. Then, we call the connection object's `commit` function to save the changes and finally we call the connection object's `close()` function to close the database connection.

Inserting records using Python

In this recipe we are going to insert new records in the `EMPLOYEES` table in the `dvdrental` database.

Getting ready

Before inserting records in the table, we first need to use the `connect` function to connect to the database first. The `connect` function was discussed in the first recipe of the chapter.

How to do it...

We are going to use the following code to insert new records in the `EMPLOYEES` table:

```
#!/usr/bin/python

import psycopg2

conn = psycopg2.connect(database="dvdrental", user="postgres",
password="postgres", host="127.0.0.1", port="5432")
```

```
print "Opened database successfully"

cur = conn.cursor()

cur.execute("INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'SandeepSingh', 39, 'Saharanpur', 90000.00 )");

cur.execute("INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'NeerajKumar', 42, 'Rohtak', 90000.00 )");

cur.execute("INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'AmitGovil', 37, 'Aligarh', 88000.00 )");

cur.execute("INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'SandeepSharma', 36, 'Haridwar ', 75000.00 )");

conn.commit()
print "Records created successfully in EMPLOYEES Table";
conn.close()
```

The preceding code is saved in a file called `insert.py` and the following is the command line output that we get once the new records are inserted successfully:

```
bash-3.2$ python insert.py
Opened database successfully
Records created successfully in Employees Table
```

How it works...

For the explanation of the preceding code, we are taking an excerpt of the code that will demonstrate how the records are getting inserted into the `EMPLOYEES` table.

```
cur = conn.cursor()

cur.execute("INSERT INTO EMPLOYEES (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'SandeepSingh', 39, 'Saharanpur', 90000.00 )");
```

If we see the preceding code first, then we can see that once the connection is made to the database, we use the underlying connection object's `cursor()` function which creates the cursor object which we are going to utilize in executing the respective SQL statements. Here, in the preceding code, we can see that the cursor object is stored in the `cur` variable and then we call the cursor object's `execute()` function to execute the `INSERT` statements and so for other SQL statements. Eventually we call the `commit()` to ensure that the changes made / records inserted are saved in the database.

Accessing table data using Python

In this recipe, we are going to see how to access table data from a PostgreSQL database using Python.

Getting ready

Database connection is mandatory before we can select data. Henceforth for this reason the connect() function is the first one that should be invoked to first make a database connection before accessing data.

How to do it...

We can use the following code to access data from the EMPLOYEES table present in the dvdrental database. The following code is saved in a file called select.py, which will be later executed from the command line:

```
#!/usr/bin/python

import psycopg2

conn = psycopg2.connect(database="dvdrental", user="postgres",
password="postgres", host="127.0.0.1", port="5432")
print "Opened database successfully"

cur = conn.cursor()

cur.execute("SELECT id, name, address, salary  from EMPLOYEES")
rows = cur.fetchall()
for row in rows:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Select Operation done successfully";
conn.close()
```

The following is the output of the preceding code:

```
bash-3.2$ python select.py
Opened database successfully
ID =  1
```

```
NAME =  SandeepSingh
ADDRESS =  Saharanpur
SALARY =  90000.0

ID =  2
NAME =  NeerajKumar
ADDRESS =  Rohtak
SALARY =  90000.0

ID =  3
NAME =  AmitGovil
ADDRESS =  Aligarh
SALARY =  88000.0

ID =  4
NAME =  SandeepSharma
ADDRESS =  Haridwar
SALARY =  75000.0

Select Operation done successfully
```

How it works...

The following is the sub part of the code that mainly deals with selecting records from a table:

```
cur.execute("SELECT id, name, address, salary  from EMPLOYEES")
rows = cur.fetchall()
for row in rows:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Select Operation done successfully";
conn.close()
```

In the preceding code, we first call the cursor object's `execute()` function to execute the `SELECT` statement. However, here the situation is that a given table may consist of multiple records and our objective is to fetch those multiple rows by integrating through each row, that is one record at a time. To achieve the target of fetching multiple rows from a table we use the cursor object's `fetchall()` function which returns all the rows of a resultset, thereby returning a list of rows from a table. To iterate through each row we use the `for` loop to iterate through the rows of a table printing each row of the table on the console during each iteration.

You may refer to the following web link for more information:

`https://wiki.postgresql.org/wiki/Psycopg2_Tutorial`

Updating records using Python

Here, in this recipe we are to update existing records in a table in the PostgreSQL database using Python language.

Getting ready

In this recipe, first we are going to show the number of existing records in the table, then we are going to update some records, see the number of records updated and then see the changed records being made visible in the table when the table records are accessed again.

How to do it...

First we check the existing records in the `EMPLOYEES` table.

```
dvdrental=# select * from employees;
 id |     name      | age | address                        | salary
----+---------------+-----+--------------------------------+--------
  1 | SandeepSingh  |  39 | Saharanpur                     |  90000
  2 | NeerajKumar   |  42 | Rohtak                         |  90000
  3 | AmitGovil     |  37 | Aligarh                        |  88000
  4 | SandeepSharma |  36 | Haridwar                       |  75000
```

Next we use the following Python code to update some of the existing records in the `EMPLOYEES` table and save the following code in a file called `update.py`:

```
#!/usr/bin/python

import psycopg2

conn = psycopg2.connect(database="dvdrental", user="postgres",
password="postgres", host="127.0.0.1", port="5432")
print "Opened database successfully"

cur = conn.cursor()

cur.execute("UPDATE EMPLOYEES SET  SALARY = 105000.00 WHERE ID=1")
conn.commit()
print "Total number of rows updated :", cur.rowcount

cur.execute("SELECT id, name, address, salary  FROM  EMPLOYEES")
rows = cur.fetchall()
for row in rows:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Update Operation done successfully";
conn.close()
```

In the preceding code, we are using the `UPDATE` statement to set the `salary` to `55000` where the value of `id` column is `1`. Next we are going to see the updated records and see the changed records being visible in the `EMPLOYEES` table, as shown in the following code output:

```
--bash-3.2$ python update.py
Opened database successfully
Total number of rows updated : 1
ID =  2
NAME =  NeerajKumar
ADDRESS =  Rohtak
SALARY =  90000.0

ID =  3
NAME =  AmitGovil
ADDRESS =  Aligarh
```

```
SALARY =  88000.0

ID =  4
NAME =  SandeepSharma
ADDRESS =  Haridwar
SALARY =  75000.0

ID =  1
NAME =  SandeepSingh
ADDRESS =  Saharanpur
SALARY =  105000.0

Update Operation done successfully
```

How it works...

The following is an excerpt of the code that was used to update existing records using Python language into `EMPLOYEES` table:

```
cur = conn.cursor()

cur.execute("UPDATE EMPLOYEES set SALARY = 105000.00 where ID=1")
conn.commit()
print "Total number of rows updated :", cur.rowcount
```

If we take a look at the preceding code, we are familiar with the connection object's `cursor()` function and subsequently with the cursor object 's `execute()` function and these have been discussed in the previous recipes. However, for updating records in a table the first change that happens here is that the `UPDATE` statement is used as a part of the `execute()` function to update an underlying record in the table. However, we need to make sure that the changes that we made in the table are visible across the database. Henceforth, we use the connection object's `commit()` function to save the changes that were made by the `UPDATE` statement. Once this is done, the changes that were made by the `UPDATE` statement are visible to anyone who selects the data from the table. We also use the cursor object's `rowcount` read only attribute to find out the number of records that were modified by the last executed statement. The `rowcount` attribute could be used either with the `UPDATE`, `DELETE` or `INSERT` statements.

Deleting records using Python

Here in this recipe we are going to show how to delete records in a table using Python language.

Getting ready

In this recipe, first we are going to show the number of existing records in the table, then we are going to delete some records, see the number of records deleted and then see the available number of records present in the table after deletion.

How to do it...

The following are the steps to delete records in a table:

1. First we are going to check the existing records in the `EMPLOYEES` table.

   ```
   dvdrental=# select * from employees;
    id |      name       | age | address                   | salary
   ----+---------------+-----+---------------------+--------
     1 | SandeepSingh  |  39 | Saharanpur                |  90000
     2 | NeerajKumar   |  42 | Rohtak                    |  90000
     3 | AmitGovil     |  37 | Aligarh                   |  88000
     4 | SandeepSharma |  36 | Haridwar                  |  75000
   ```

2. Next we are going to use the following Python code to delete some records from the `EMPLOYEES` table and save the code in a file called `delete.py`, which we are going to execute from the command line:

   ```
   #!/usr/bin/python

   import psycopg2

   conn = psycopg2.connect(database="dvdrental", user="postgres",
   password="postgres", host="127.0.0.1", port="5432")
   ```

```
print "Opened database successfully"

cur = conn.cursor()

cur.execute("DELETE from EMPLOYEES where ID=2;")
conn.commit()
print "Total number of rows deleted :", cur.rowcount

cur.execute("SELECT id, name, address, salary  from EMPLOYEES")
rows = cur.fetchall()
for row in rows:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "DELETE Operation done successfully";
conn.close()
```

Here, the preceding `DELETE` statement that we are issuing, is used to delete a record from the table where the value of `ID` column is `2`.

3. Next we see the command line output of the preceding code:

```
bash-3.2$ python delete.py
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  SandeepSingh
ADDRESS =  Saharanpur
SALARY =  90000.0

ID =  3
NAME =  AmitGovil
ADDRESS =  Aligarh
SALARY =  88000.0

ID =  4
NAME =  SandeepSharma
ADDRESS =  Haridwar
SALARY =  75000.0

DELETE Operation done successfully
```

How it works...

The following is an excerpt from the code that was used to delete records from the EMPLOYEES table:

```
cur = conn.cursor()

cur.execute("DELETE from EMPLOYEES where ID=2;")
conn.commit()
print "Total number of rows deleted :", cur.rowcount
```

In the preceding code snippet, we use the cursor object's execute function to execute the DELETE statement. We then use the connection object's commit method to commit or save the changes made by the DELETE statement. Finally, we use the cursor object's rowcount attribute to find the number of records that were deleted by the last executed DELETE statement.

12
Data Migration from Other Databases and Upgrading the PostgreSQL Cluster

In this chapter, we will cover the following recipes:

- Using pg_dump to upgrade data
- Using the pg_upgrade utility for a version upgrade
- Replicating data from other databases to PostgreSQL using GoldenGate

Introduction

Often in the career of a database administrator, he/she is required to do major version upgrades of the PostgreSQL server. Over a period of time new terminologies and features get added to PostgreSQL and this results in a major version release. To implement the new features of the new version, the existing PostgreSQL setup needs to be upgraded to the new version. Database upgrades require proper planning, careful execution and planned downtime. PostgreSQL offers two major ways to do a version upgrade.

- With the help of `pg_dump` utility
- With the help of `pg_upgrade` script

Also in this chapter we cover the Oracle GoldenGate tool. **GoldenGate** is a heterogeneous replication software that can be used to replicate data between different databases.

In this chapter we are going to cover heterogeneous replication between Oracle and PostgreSQL.

Using pg_dump to upgrade data

Here, in this recipe, we are going to upgrade PostgreSQL cluster from version 9.2 to 9.3 and we will utilize `pg_dump` utility for this purpose.

Getting ready

The only prerequisites here are that an existing PostgreSQL cluster must be set up and running. The required version here is PostgreSQL version 9.2. These steps are carried out on a 64 bit CentOS machine.

How to do it...

Below are the series of steps that need to be carried out for upgrading PostgreSQL from version 9.2 to 9.3 using `pg_dump`:

1. Backup your database using the `pg_dumpall` command.

   ```
   pg_dumpall >  db.backup
   ```

2. The next step would be to shutdown to the current PostgreSQL server.

   ```
   pg_ctl -D /var/lib/pgsql/9.2/data  stop
   ```

3. The next step would be to rename the old PostgreSQL installation directory.

   ```
   mv /var/lib/pgsql  /var/lib/pgsql.old
   ```

4. The next step would be to install the new version of PostgreSQL which is PostgreSQL version 9.3. Prior to doing that, we will check for existing packages before installing the new ones with the following command and then we will install the new version package:

   ```
      rpm -qa |grep postgresql
   wget http://yum.postgresql.org/9.3/redhat/rhel-6.4-x86_64/pgdg-
   centos93-9.3-1.noarch.rpm


   rpm -Uvh ./pgdg- centos93-9.3-1.noarch.rpm


   yum install postgresql93-server.x86_64 postgresql93-contrib.x86_64
   postgresql93-libs.x86_64 postgresql93.x86_64 postgresql93-devel.
   x86_64
   ```

5. The next step would be to initialize the PostgreSQL version 9.3 server.

   ```
   /usr/pgsql-9.3/bin/initdb -D  /var/lib/pgsql/9.3/data
   ```

6. Once the database cluster is initialized, then the next step is to restore the configuration files from the previous version 9.2 data directory to the current version 9.3 data directory location.

   ```
   cd /var/lib/pgsql.old/9.2/data
   cp  pg_hba.conf postgresql.conf  /var/lib/pgsql/9.3/data
   ```

7. The next step would be to start the PostgreSQL 9.3 database server.

   ```
   pg_ctl -D /var/lib/pgsql/9.3/data  start
   ```

8. Finally, as a last step, restore your data from the backup that was created in step 1.

   ```
   /usr/pgsql-9.3/bin/psql  -d postgres  -f  db.backup
   ```

9. As a next step, we can either remove the old version data directory or else we can continue working alongside both the server versions.

10. If we choose to remove the old version, as mentioned in step 9, we can then remove the respective old version packages, as follows:

    ```
    yum remove postgresql92-server-9.2.3-2PGDG.rhel6.x86_64
    postgresql92-contrib-9.2.3-2PGDG.rhel6.x86_64 postgresql92-libs-
    9.2.3-2PGDG.rhel6.x86_64 postgresql92-9.2.3-2PGDG.rhel6.x86_64
    postgresql92-devel-9.2.3-2PGDG.rhel6.x86_64
    ```

How it works...

Here, initially we take a dump of all the databases in the existing PostgreSQL 9.2 version cluster. We then initiate a clean shutdown of current PostgreSQL server and rename the existing PostgreSQL installation directory to avoid any conflicts with the new version of PostgreSQL, that is version 9.3 that is being installed. Once the respective packages of the new version are installed we then proceed with initializing a database directory for PostgreSQL 9.3 server. To ensure that the desired configuration settings come into effect, we will need to copy the configuration files from the old version's data directory to the new version's data directory and then start the new version PostgreSQL server service using the configuration settings that were defined for the existing environment in the old server. Once the PostgreSQL server version 9.3 has been started we can connect to the databases on this server. Eventually we restore all the tables and databases from the old PostgreSQL server 9.2 to the new PostgreSQL server version 9.3 by using the backup that was made in step 1 in the preceding section.

You may refer to the following links for a more detailed explanation on upgrading a PostgreSQL cluster using `pg_dump`:

`http://www.postgresql.org/docs/9.3/static/upgrading.html`

Using the pg_upgrade utility for a version upgrade

Here, in this recipe, we are going to talk about upgrading a PostgreSQL cluster using `pg_upgrade` utility. We will be covering a PostgreSQL version upgrade from version 9.2 to version 9.3.

Getting ready

The only prerequisites here, are that an existing PostgreSQL cluster must be set up and running. The required version here is PostgreSQL version 9.2. These steps are carried out on a 64 bit CentOS machine.

How to do it...

The following are the steps to upgrade a PostgreSQL cluster from version 9.2 to version 9.3 using the `pg_upgrade` utility:

1. Take a full backup of the data directory using a filesystem dump or use `pg_dumpall` to backup data. Before taking a backup stop the running PostgreSQL server.

```
pg_ctl  -D $PGDATA stop



cd /var/lib/pgsql/9.2/


tar -cvf data.tar data
```

2. The next step would to be install the new version of PostgreSQL.

```
 wget http://yum.postgresql.org/9.3/redhat/rhel-6.4-x86_64/pgdg-
centos93-9.3-1.noarch.rpm
   rpm -ivh ./pgdg- centos93-9.3-1.noarch.rpm
```

3. As the repository is now installed, the next step is to determine which packages need to be installed. For this purpose, check the packages that are installed for the current version and then get the list of packages that are needed to be installed for the new PostgreSQL version 9.3.

```
 rpm -qa | grep postgre | grep 92
postgresql92-server-9.2.3-2PGDG.rhel6.x86_64
postgresql92-contrib-9.2.3-2PGDG.rhel6.x86_64
postgresql92-libs-9.2.3-2PGDG.rhel6.x86_64
postgresql92-9.2.3-2PGDG.rhel6.x86_64
```

```
postgresql92-devel-9.2.3-2PGDG.rhel6.x86_64



 yum list postgres* | grep 93
postgresql93.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-contrib.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-debuginfo.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-devel.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-docs.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-jdbc.x86_64 9.3.1100-1PGDG.rhel6 pgdg93
postgresql93-jdbc-debuginfo.x86_64 9.3.1100-1PGDG.rhel6 pgdg93
postgresql93-libs.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-odbc.x86_64 09.02.0100-1PGDG.rhel6 pgdg93
postgresql93-odbc-debuginfo.x86_64 09.02.0100-1PGDG.rhel6 pgdg93
postgresql93-plperl.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-plpython.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-pltcl.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-server.x86_64 9.3.4-1PGDG.rhel6 pgdg93
postgresql93-test.x86_64 9.3.4-1PGDG.rhel6 pgdg93
```

The packages that will be installed for the new version will match the packages that are currently installed for the old version.

```
 yum install postgresql93-server.x86_64 postgresql93-contrib.
x86_64 postgresql93-libs.x86_64 postgresql93.x86_64 postgresql93-
devel.x86_64
```

4. Now that the new version of PostgreSQL is installed, the next step is to initialize the data directory for the new PostgreSQL version 9.3 database.

```
/etc/init.d/postgresql-9.3 initdb
```

5. Once the data directory has been initialized for the new PostgreSQL version 9.3, the next step is to run the `pg_upgrade` utility.

```
cd  /usr/pgsql-9.3/bin


./pg_upgrade -v -b /usr/pgsql-9.2/bin/ -B /usr/pgsql-9.3/bin/ -d /
var/lib/pgsql/9.2/data/ -D /var/lib/pgsql/9.3/data/
```

Once the upgrade completes , it will then generate two files analyze_new_cluster.sh and delete_old_cluster.sh files respectively. These files are basically used to generate optimizer statistics and delete the old PostgreSQL cluster version's data files.

6. Post the upgrade step we would need to copy the configuration files and the authentication files present in the old version to the new setup, as follows:

```
cd /var/lib/pgsql/9.2/data
cp  -p pg_hba.conf postgresql.conf  /var/lib/pgsql/9.3/data/
```

7. The next step would be to start the PostgreSQL server version 9.3 service.

```
service postgresql-9.3 start
```

8. The next step would be to run the analyze_new_cluster.sh shell script that was generated at the end of step 5. This script is used to collect minimal optimizer statistics in order to get a working and a usable PostgreSQL system.

```
./analyze_new_cluster.sh
```

9. The next step would be to remove the old PostgreSQL directory by running the following script:

```
./delete_old_cluster.sh
```

10. Finally, as a last step, we will remove the old PostgreSQL version 9.2 installed packages.

```
yum remove postgresql92
```

How it works...

After installing the packages for the PostgreSQL server version 9.3, what we are doing here is changing the location of data directory in the startup script, as shown in step 4. After this is done, we initialize the data directory for the new PostgreSQL server. The difference in steps here, and the previous recipe, is that here we are changing the location of the data directory, the log path, and port number of the new PostgreSQL server version, whereas in the earlier recipe we renamed the existing PostgreSQL server version data directory. Once the data directory is initialized, we then we stop the current PostgreSQL server and then launch the pg_upgrade script to upgrade the existing setup to the new version. The pg_upgrade script requires specifying the path of old and new data directories and binaries. Once the upgrade completes it generates two shell scripts analyze_new_cluster.sh and delete_old_cluster.sh to generate statistics and delete the old version PostgreSQL directory. To preserve the existing configuration, we would need the pg_hba.conf and postgresql.conf files from the old version's data directory to the new version's data directory, as shown in step 6 in the preceding section and then we can start the upgraded PostgreSQL server. Once the server has started, we can then proceed to generate statistics via the analyze_new_cluster.sh script and then remove the old version directory via the delete_old_cluster.sh script, as shown in steps 8 and 9 respectively.

You can refer to the following web link for more information regarding the upgrade process:

`http://www.postgresql.org/docs/9.3/static/pgupgrade.html`

`http://no0p.github.io/postgresql/2014/03/29/upgrading-pg-ubuntu.html`

Replicating data from other databases to PostgreSQL using GoldenGate

In this recipe, we are going to cover heterogeneous replication using the Oracle GoldenGate software. We are going to migrate table data from Oracle to PostgreSQL.

Getting ready

Since this recipe talks about replicating data from Oracle to PostgreSQL, it is important to cover Oracle installation. Also, since GoldenGate is the primary tool used, we will also cover the GoldenGate installation for both Oracle and PostgreSQL.

To install the Oracle 11g software on the Linux platform, you may refer to any of the following web links:

- `http://oracle-base.com/articles/11g/oracle-db-11gr2-installation-on-oracle-linux-5.php`
- `http://dbaora.com/install-oracle-11g-release-2-11-2-on-centos-linux-7/`

To install GoldenGate for the Oracle database, refer to the following web link:

`http://docs.oracle.com/cd/E35209_01/doc.1121/e35957.pdf`

Here are the high level installation steps given for the ease of the reader. Please refer to the preceding web link for more detailed information:

- Login to `edelivery.oracle.com`.
- Select **Oracle Fusion Middleware** from the **Select a Product Pack** dropdown menu and select the **Linux x86-64** option from the **Platform** dropdown menu and click on the **Go** button.

- Choose the option whose description says **Oracle GoldenGate on Oracle v11.2.1 Media Pack for Linux x86-64**, click on the **Continue** button and it will open a web link. Then download the file with the name **Oracle GoldenGate V11.2.1.0.3 for Oracle 11g on Linux x86-64**.
- As a next step, extract the downloaded file and change the directory to the new location and then launch the GoldenGate command-line interface using `ggsci` command. Before launching the GoldenGate command-line interface set the GoldenGate installation directory and library path in `PATH` and `LD_LIBRARY_PATH` environment variables respectively.

To install GoldenGate for PostgreSQL, refer to the following web link:

`https://docs.oracle.com/cd/E35209_01/doc.1121/e29642.pdf`

Here are the high level installation steps given for the ease of the reader. Please refer to the preceding web link for more detailed information and instructions:

- Login to `edelivery.oracle.com`
- Select **Oracle Fusion Middleware** from the **Select a Product Pack** dropdown menu and select the **Linux x86-64** option from the **Platform** dropdown menu and click on the **Go** button.
- Choose the option whose description says **Oracle GoldenGate for Non Oracle Database v11.2.1 Media Pack for Linux x86-64**, click on the **Continue** button and you will directed to a web link where you need to click on the **Download** button against the file whose name says **Oracle GoldenGate V11.2.1.0.2 for PostgreSQL on Linux x86-64**.
- Once the file is downloaded, then extract the zip file and change the directory to the newly created location and then launch the GoldenGate command-line interface using the `ggsci` command. Before launching the GoldenGate command-line interface set the GoldenGate installation directory and library path in `PATH` and `LD_LIBRARY_PATH` environment variables respectively.

Here, in this section, we will first cover a brief overview of the procedure used for table data replication from the source database, that is Oracle to target database, that is PostgreSQL.

1. From the source database, that is Oracle here first, we will have to create various subdirectories for GoldenGate and for various database definition files.
2. The next step is to create a parameter file which contains a port number for the manager process for GoldenGate on the source database and then start the manager.
3. Similar to this on the target database, that is PostgreSQL, we will have to create various subdirectories for GoldenGate and for various database definition files.
4. The next step is to create a parameter file which contains a port number for the manager process for GoldenGate on the target database and then start the manager.

5. The next step would be to create two tables with the same structure on both the source, that is Oracle and target database, that is PostgreSQL.
6. Now that we have the tables created on both the source and target database, we will log in to source database from the GoldenGate tool and capture the table definitions for the tables that needs to be replicated.
7. Similar to the preceding step we will log in into the target database using the GoldenGate command-line interface and capture the table definitions for the table which was created in step 5.
8. In the next step, we start the extract process on the source. We first create a parameter file for the extract process, which contains the information about the remote host and consists of a trail file which is used to capture any changes made on the table in the source database and transport these changes to the target database. We then start the extract process and it will capture any changes on the table in the source database, that is Oracle.
9. The next step, is to start the `replicat` process on the target database. For this we set up a `replicat` parameter file. Once the `replicat` process is started, it will read the changes from the trail file which was used in the previous step at the source to capture changes made to the table in the source database. The `replicat` process will read these changes and dump them into the target database, that is PostgreSQL.
10. Now that we have the extract process configured on the source database to capture changes and the `replicat` process set up on the target database to read those changes. We will now begin to add/change some records on the source. With the extract process capturing these changes and recording them in the trail file and the trail file being shipped to the server hosting the target database, the `replicat` process residing on the target reads those changes from the trail file and applies them to the target database.

    We are assuming that a username `nkumar` with password `nkumar` has already been setup on both Oracle and PostgreSQL. We will be using the tables created in `nkumar` schema for replication between Oracle and PostgreSQL.

    For instance, on Oracle, we can create the schema user `nkumar`, as follows, after logging in as the `sys` user:

    ```
    SQL > CREATE USER nkumar identified by nkumar ;
    SQL> GRANT CREATE ANY TABLE to nkumar;
    ```

    For creating `nkumar` user in PostgreSQL, you may refer to *Chapter 1*, *Managing Databases and the PostgreSQL Server* for more details on how to create a user in PostgreSQL and accordingly create this user in PostgreSQL.

How to do it...

The following are the complete sequence of steps required to migrate table data/changes from Oracle to PostgreSQL using GoldenGate:

1. First connect as the superuser `SYS` using the operating system authentication on the machine hosting the Oracle database using the `sqlplus` utility. The utility uses OS authentication by default, so the password is not required to be specified. At the time of Oracle installation, it will usually ask the user to change the password, however if no password is specified you can use the default password `change_on_install`.

   ```
    sqlplus / as sysdba
   ```

2. Once logged in to the Oracle database make the following parameter changes:

   ```
   SQL> alter system set log_archive_dest_1='LOCATION=/home/abcd/
   oracle/oradata/arch';
   ```

3. To make the above mentioned parameter changes come into effect, shutdown and restart the Oracle database.

   ```
   SQL> shutdown immediate

   SQL> startup mount
   ```

4. Configure archiving on the Oracle database to ensure that changes made by transactions are captured and logged in the `archivelog` files.

   ```
   SQL> alter database archivelog;

   SQL> alter database open;
   ```

5. The next step would be to enable minimum supplemental logging.

   ```
   SQL> alter database add supplemental log data;

   SQL> alter database force logging;

   SQL> SELECT force_logging, supplemental_log_data_min FROM
   v$database;


   FOR SUPPLEME

   --- --------

   YES      YES
   ```

6. The next step would be to add the GoldenGate directory path to `PATH` and library path to the `LD_LIBRARY_PATH` environment variables respectively.

   ```
   export PATH=$ORACLE_HOME/bin:$ORACLE_HOME/OPatch:$HOME/ggs:$PATH

   export LD_LIBRARY_PATH=$ORACLE_HOME/lib:$HOME/ggs/lib
   ```

7. The next step would be to launch the GoldenGate command-line interface for Oracle.

   ```
   ./ggsci
   ```

8. The next step would to be create various subdirectories for GoldenGate such as directories for report files, database definition etc.

   ```
   GGSCI> create subdirs


   Creating subdirectories under current directory /home/abcd/oracle/
   ggs


   Parameter files               /home/abcd/oracle/ggs/dirprm:
   already exists
   Report files                  /home/abcd/oracle/ggs/dirrpt:
   created
   Checkpoint files              /home/abcd/oracle/ggs/dirchk:
   created
   Process status files          /home/abcd/oracle/ggs/dirpcs:
   created
   SQL script files              /home/abcd/oracle/ggs/dirsql:
   created
   Database definitions files    /home/abcd/oracle/ggs/dirdef:
   created
   Extract data files            /home/abcd/oracle/ggs/dirdat:
   created
   Temporary files               /home/abcd/oracle/ggs/dirtmp:
   created
   Stdout files                  /home/abcd/oracle/ggs/dirout:
   created
   ```

9. The next step is to create a parameter file for the manager which contains a port number for the manager. Here, we enter port `7809` as the port number.

   ```
   GGSCI > edit param mgr
   GGSCI > view param mgr
   PORT 7809
   ```

10. The next step would be to exit from the manager, start the manager and then verify if it is running.

    ```
    GGSCI > startw mgr
    GGSCI > info all


    Program     Status      Group       Lag at Chkpt  Time Since Chkpt
    ```

```
MANAGER     RUNNING


GGSCI > info mgr
Manager is running (IP port 7809).
```

11. The next step would be to log in to the server hosting the PostgreSQL server and make the GoldenGate configuration steps there. First add the GoldenGate directory to `LD_LIBRARY_PATH` and `PATH` environment variables.

```
export LD_LIBRARY_PATH=/usr/pgsql/lib:/usr/pgsql/ggs/lib
export PATH=/usr/pgsql/bin:/usr/pgsql/ggs:$PATH
```

12. GoldenGate uses an ODBC connection to connect to the `postgres` database. The next step is to create the ODBC file. The ODBC driver is shipped along with the installation on Linux/Unix, you just have to create just the configuration file. If the ODBC driver is not available, you may refer to the following web link to download the respective PostgreSQL driver:

`http://www.uptimemadeeasy.com/linux/install-postgresql-odbc-driver-on-linux/`

```
 view odbc.ini

[ODBC Data Sources]
GG_Postgres=DataDirect 6.1 PostgreSQL Wire Protocol
[ODBC]
IANAAppCodePage=106
InstallDir=/usr/pgsql/ggs
[GG_Postgres]
Driver=/usr/pgsql/ggs/lib/GGpsql25.so
Description=DataDirect 6.1 PostgreSQL Wire Protocol
Database=test
HostName=dbtest
PortNumber=5432
LogonID=nkumar
Password=nkumar
```

13. The next step would be to export the ODBC environment variable, that is `ODBCINI` which should point to the `odbc.ini` file that we have created in the previous step. This variable can be set in the `.profile` file, as well.

```
export ODBCINI=/usr/pgsql/ggs/odbc.ini
```

14. Now that we have the ODBC setup completed, the next step would be to start with the GoldenGate setup for PostgreSQL.

    We will first launch the GoldenGate command-line interpreter for PostgreSQL.

    ```
    ./ggsci
    ```

15. We will now create various subdirectories for the GoldenGate report, definition files, and so on.

    ```
    GGSCI > create subdirs


    Creating subdirectories under current directory /usr/pgsql/ggs


    Parameter files                /usr/pgsql/ggs/dirprm: already
    exists
    Report files                   /usr/pgsql/ggs/dirrpt: created
    Checkpoint files               /usr/pgsql/ggs/dirchk: created
    Process status files           /usr/pgsql/ggs/dirpcs: created
    SQL script files               /usr/pgsql/ggs/dirsql: created
    Database definitions files     /usr/pgsql/ggs/dirdef: created
    Extract data files             /usr/pgsql/ggs/dirdat: created
    Temporary files                /usr/pgsql/ggs/dirtmp: created
    Stdout files                   /usr/pgsql/ggs/dirout: created
    ```

16. The next step would be to create the manager parameter file with port number. Here we enter port number `7809` in the manager parameter file and then start the manager.

    ```
    GGSCI > edit param mgr


    GGSCI > view param mgr


    PORT 7809
    ```

17. Once we created the parameter file we can start the manager and check its status.

```
GGSCI > start mgr

Manager started.

GGSCI > info all

Program     Status      Group       Lag at Chkpt  Time Since Chkpt

MANAGER     RUNNING
GGSCI > info mgr

Manager is running (IP port 7809).
```

18. We will now create a table both in Oracle and PostgreSQL databases and replicate the data between the two. Log in to the Oracle database and create the table.

```
sqlplus nkumar
SQL> create table abcd(col1 number,col2 varchar2(50));

Table created.

SQL> alter table abcd add primary key(col1);

Table altered.
```

19. The next step would be to log in to the PostgreSQL database and create a similar table.

```
 psql -U nkumar -d test -h dbtest

test=> create table "public"."abcd" ( "col1" integer NOT NULL,
"col2" varchar(20),CONSTRAINT "PK_Col111" PRIMARY KEY ("col1"));
```

20. The next step would be log in to Oracle database using the GoldenGate command-line interface, list the tables and capture and check their data types.

```
GGSCI > dblogin userid nkumar, password nkumar
Successfully logged into database.

GGSCI > list tables *
```

```
NKUMAR.ABCD


Found 1 tables matching list criteria.


GGSCI > capture tabledef nkumar.abcd
Table definitions for NKUMAR.ABCD:
COL1                           NUMBER NOT NULL PK
COL2                           VARCHAR (50)
```

21. In the next step we will check our ODBC connection to the PostgreSQL database and use the GoldenGate CLI (command-line interface) for listing the tables and capturing the table definitions.

```
GGSCI > dblogin sourcedb gg_postgres userid nkumar


Password:


2014-11-04 17:56:35  INFO    OGG-03036  Database character set
identified as UTF-8. Locale: en_US.


2014-11-04 17:56:35 INFO    OGG-03037  Session character set
identified as UTF-8.
Successfully logged into database.


GGSCI > list tables *
public.abcd
Found  1 table matching list criteria


GGSCI > capture tabledef "public"."abcd"
Table definitions for public.abcd:
col1       NUMBER (10) NOT NULL PK
col2      VARCHAR (20)
```

22. In the next step we will start the GoldenGate extract process on the Oracle database. First we will create the extract process that captures the changes for the `ABCD` table in the Oracle database and copy these changes directly to the PostgreSQL machine. Every process needs the configuration file, so we will create one for the extract process.

    ```
    GGSCI > edit param epos
    ```

    The parameters created are shown below when viewing the parameter file, as follows:

    ```
    GGSCI > view param epos


    EXTRACT epos
    SETENV (NLS_LANG="AMERICAN_AMERICA.ZHS16GBK")
    SETENV (ORACLE_HOME="/home/abcd/oracle/product/11.2.0/dbhome_1")
    SETENV (ORACLE_SID="orapd")
    USERID nkumar, PASSWORD nkumar
    RMTHOST dbtest, MGRPORT 7809
    RMTTRAIL /usr/pgsql/ggs/dirdat/ep
    TABLE nkumar.abcd;
    ```

23. The extract process is called epos and it connects as user `nkumar` using the password `nkumar` to the Oracle database. Changes made on the Oracle table `abcd` will be extracted and this information will be put in a trail file in the PostgreSQL machine. Now that the parameter file has been created, we can then add the extract process and start it.

    ```
    GGSCI > add extract epos, tranlog, begin now
    EXTRACT added.



    GGSCI > add exttrail /usr/pgsql/ggs/dirdat/ep, extract epos,
    megabytes 5
    EXTTRAIL added.


    GGSCI > start epos


    Sending START request to MANAGER ...
    ```

```
EXTRACT EPOS starting

GGSCI > info all

Program     Status      Group       Lag at Chkpt  Time Since Chkpt

MANAGER     RUNNING
EXTRACT     RUNNING     EPOS        00:00:00      00:00:00

GGSCI > info extract epos
```

24. Since we are replicating the data in the heterogeneous environment, that is data replication is happening from Oracle to PostgreSQL, the process doing the loading in the PostgreSQL would need to provide more details about the data in the extract file. This is done by creating a definition file using `defgen` utility.

    ```
    GGSCI > view param defgen

    DEFSFILE /home/abcd/oracle/ggs/dirdef/ABCD.def
    USERID nkumar, password nkumar
    TABLE NKUMAR.ABCD;
    ```

25. We can now exit from the GoldenGate CLI and call the `defgen` utility on the command line to create the definition file and add the reference to the `defgen` parameter file.

    ```
     ./defgen paramfile ./dirprm/defgen.prm

    Definitions generated for 1 table in /home/abcd/oracle/ggs/dirdef/
    ABCD.def
    ```

26. The next step would be to copy the `defgen` file to the machine where PostgreSQL database is hosted.

    ```
           cd /home/abcd /oracle/ggs/dirdef
     scp dirdef/ABCD.def postgres@dbtest:/usr/pgsql/ggs/dirdef
    ```

27. The next step would be to start the PostgreSQL `replicat` process and we are going to set up the parameter file for this and include the definition file that was copied from the server hosting the Oracle database to the server hosting PostgreSQL.

    ```
    GGSCI > edit param rpos
    ```

    The parameters created when viewing the parameter file are as shown below:

    ```
    GGSCI > view param rpos


    REPLICAT rpos
    SOURCEDEFS /usr/pgsql/ggs/dirdef/ABCD.def
    SETENV ( PGCLIENTENCODING = "UTF8" )
    SETENV (ODBCINI="/usr/pgsql/ggs/odbc.ini" )
    SETENV (NLS_LANG="AMERICAN_AMERICA.AL32UTF8")
    TARGETDB GG_Postgres, USERID nkumar, PASSWORD nkumar
    DISCARDFILE /usr/pgsql/ggs/dirrpt/diskg.dsc, purge
    MAP NKUMAR.ABCD, TARGET public.abcd, COLMAP (COL1=col1,COL2=col2);
    ```

28. In the next step we create the `replicat` process, start it and verify if it is running.

    ```
    GGSCI > add replicat rpos, NODBCHECKPOINT, exttrail /usr/pgsql/
    ggs/dirdat/ep
    REPLICAT added.



    GGSCI > start rpos


    Sending START request to MANAGER ...
    REPLICAT RPOS starting


    GGSCI > info all


    Program     Status      Group       Lag at Chkpt  Time Since Chkpt


    MANAGER     RUNNING
    ```

```
REPLICAT    RUNNING     RPOS        00:00:00      00:00:00


GGSCI > info all

Program     Status      Group       Lag at Chkpt  Time Since Chkpt

MANAGER     RUNNING
REPLICAT    RUNNING     RPOS        00:00:00      00:00:02
GGSCI > view report rpos
```

29. Now that the `extract` and `replicat` processes have been set up on Oracle and PostgreSQL GoldenGate interfaces the next step is to test the configuration. We first begin by logging into the Oracle database and inserting records into the `ABCD` table.

```
 sqlplus nkumar

SQL> insert into abcd values(101,'Neeraj Kumar');

1 row created.

SQL> commit;

Commit complete.

SQL> select * from abcd;

col1 | col2
-----+-------------------
101  | Neeraj Kumar
```

30. Now we will check if the corresponding changes / new records inserted into the `ABCD` table in the Oracle database are visible in the corresponding `ABCD` table in the PostgreSQL database.

```
 psql -U nkumar -d test


test=> select * from abcd;
 col1 |       col2
------+-------------------
  101 |  Neeraj Kumar
(1 row)
```

This setup completes the heterogeneous testing scenario for replicating data /changes from Oracle to PostgreSQL.

How it works...

For the steps mentioned in the preceding section, we are going to discuss the steps 1 to 27 of the preceding section in chunks.

- We will first talk about steps 1 to 6 of the preceding section: Initially we make a superuser connection in Oracle with the `sysdba` privilege and make certain configuration changes. We first enable a destination for holding the archived logs, that is logs that contain information about transactional changes are kept here in the location specified by the `log_archive_dest_1` initialization parameter, as seen in step 2 of the previous section. We then shutdown the database in order to ensure the changes made in step 2 come into effect. Once the database is restarted, we then configure archiving in the database and enable supplemental logging, as seen in step 4 and 5 of the preceding section. In step 6 we configure and include the GoldenGate directory path and library path in `PATH` and `LD_LIBRARY_PATH` environment variables.
- We will now talk about steps 7 and 8 of the preceding section: After GoldenGate is installed on the server hosting the Oracle database, we then launch the GoldenGate CLI and then create various GoldenGate subdirectories for holding parameter files, checkpoint files, database definition files, extract data files, and so on.
- We will now talk about steps 9 and 10 of the preceding section: The GoldenGate manager performs a number of functions like starting the GoldenGate process, trail log file management, and reporting. The manager process needs to be configured both on source and target systems and configuration is carried out with the help of the parameter file, as shown in step 9. We configure the parameter `PORT` to define the port on which the manager is running. Once the parameter file for the manager is setup on the source machine, we then start the manager and verify if it is running. This is shown in step 10 of the preceding section.

- We will now talk about steps 11 to 15 of the preceding section: Once GoldenGate is installed on the machine where the PostgreSQL server is hosted, we then add the GoldenGate directory and library path to `PATH` and `LD_LIBRARY_PATH` environment variables. GoldenGate basically uses an ODBC connection to connect to the PostgreSQL database. For this purpose the we set up an ODBC configuration file called `odbc.ini` which contains connection information to connect to the PostgreSQL server. This is shown in step 12. In the next step, we export the `ODBCINI` environment variable and include the path of the configuration file. Then from step 14 onwards we launch the GoldenGate command-line interface for PostgreSQL and then we create various subdirectories for holding parameter files, database definition files, and so on.

- We will now talk about steps 16 and 17 of the preceding section: Similar to what was performed in steps 9 and 10 on the source system for the manager process in GoldenGate for Oracle, in a similar style we configure the parameter file for the manager process in GoldenGate for the target system, that is PostgreSQL and then start the manager and then verify it, as shown in step 17. The only parameter that has been configured in step 16 is the `PORT` parameter which identified the port on which the manager will listen to.

- We will now talk about steps 18 and 19 in the preceding section: Here we are creating two tables of the same names, having the same structure. One table will be created in the Oracle database and one will be created in PostgreSQL. The tables are created in this manner because the idea of this exercise is that any data changes that happen on the table created in Oracle will be replicated/propagated in PostgreSQL. This is the heterogeneous replication concept.

- Here we will talk about steps 20 and 21 from the preceding section: Basically, in step 20 what we are doing is logging in to the Oracle database using the GoldenGate interface and we capture the table definition for the table that was created in step 18 of the preceding section. Similarly, in step 21, we are checking the ODBC connectivity to the PostgreSQL database from the GoldenGate CLI and once the connection is made we capture the table definitions for the table created in step 19.

- Here we are going to talk about steps 22 and 23 from the preceding section: In step 22 we are creating a parameter file for the `extract` process on the machine hosting the Oracle database since it is used as the source. The extract process happens on the source database. The `extract` process parameter file contains information regarding the Oracle environment, the target remote host, the manager port, the trail file, and the table for which the changes needs to be captured. In step 23, we start the extraction process on the source Oracle database and we add the trail file. The `extract` process will extract any changes made to the Oracle table `ABCD` and will put this information on the trail file which resides on the machine hosting the PostgreSQL server.

- Here we are going to talk about steps 24 and 25 from the preceding section: As the replication is happening in a heterogeneous environment, that is from Oracle to PostgreSQL in this scenario, it is important to get as much detail as possible about the data in the extract file to make things clear for the process loading the data into the PostgreSQL database. For this to happen, we need to create a definition file which will be created on the GoldenGate interface of the Oracle database and will then be shipped to the machine hosting the PostgreSQL server. In step 24, we are basically creating a parameter file for the `defgen` utility. In step 25, we call the `defgen` utility to create the definition file and we also add a reference to the parameter file created in step 24 of the preceding section.
- In step 26 of the preceding section, we copy the definition file created in step 25 from the Oracle machine to the machine hosting the PostgreSQL server.
- Here we are going to talk about steps 27 and 28 of the preceding section. Here we start the `replicat` process. The `replicat` process basically reads the changes from the trail file and distributes them to the PostgreSQL database. In step 27, basically we configure the parameter file for the `replicat` process and in step 28, we start the `replicat` process, add the trail file to the `replicat` process so that it can read changes from the trail file and dump those changes to the PostgreSQL database.
- Here we are going to talk about steps 29 and 30. Basically we are going to test our configuration here. In step 29, we log in to the Oracle database, insert a record in the `ABCD` table and save the changes. Now with GoldenGate extract and `replicat` process running the newly inserted record in the Oracle table should be replicated to the corresponding table in the PostgreSQL database. We confirm this by logging in to the PostgreSQL database and then by selecting the records from the `ABCD` table in step 30. We can see in step 30 that the records inserted in step 29 in the Oracle table are visible in the PostgreSQL database table `ABCD`. This confirms the successful implementation of heterogeneous replication from the Oracle database to the PostgreSQL database.

Index

Thank you for buying

PostgreSQL Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

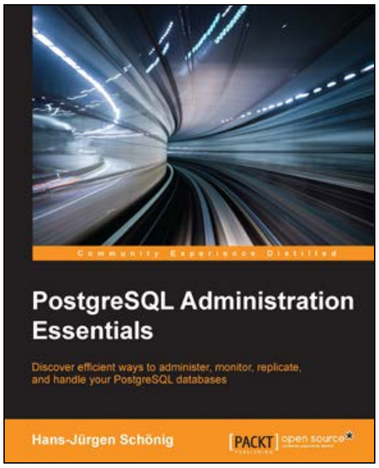
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt open source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's open source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING


Community Experience Distilled
PostgreSQL Administration Essentials
Discover efficient ways to administer, monitor, replicate, and handle your PostgreSQL databases
Hans-Jürgen Schönig
[PACKT] open source

PostgreSQL Administration Essentials

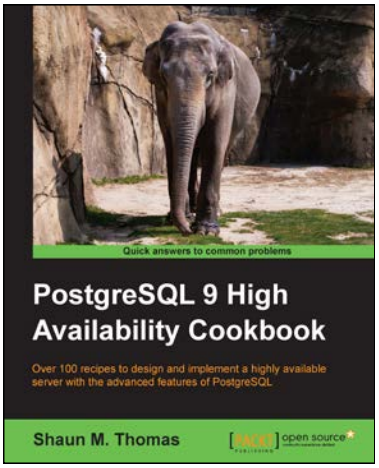
ISBN: 978-1-78398-898-3 Paperback: 142 pages

Discover efficient ways to administer, monitor, replicate, and handle your PostgreSQL databases

1. Learn how to detect bottlenecks and make sure your database systems offer superior performance to your end users.

2. Replicate your databases to achieve full redundancy and create backups quickly and easily.

3. Optimize PostgreSQL configuration parameters and turn your database server into a high-performance machine capable of fulfilling your needs.


Quick answers to common problems
PostgreSQL 9 High Availability Cookbook
Over 100 recipes to design and implement a highly available server with the advanced features of PostgreSQL
Shaun M. Thomas
[PACKT] open source

PostgreSQL 9 High Availability Cookbook

ISBN: 978-1-84951-696-9 Paperback: 398 pages

Over 100 recipes to design and implement a highly available server with the advanced features of PostgreSQL

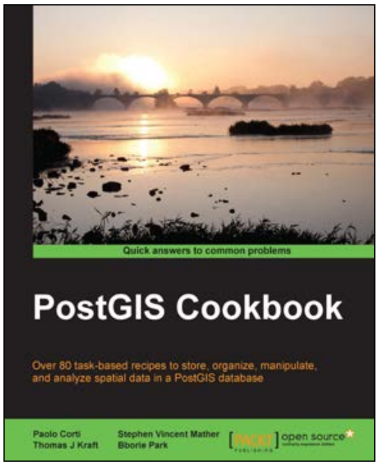
1. Create a PostgreSQL cluster that stays online even when disaster strikes.

2. Avoid costly downtime and data loss that can ruin your business.

3. Perform data replication and monitor your data with hands-on industry-driven recipes and detailed step-by-step explanations.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING


Quick answers to common problems
PostGIS Cookbook
Over 80 task-based recipes to store, organize, manipulate, and analyze spatial data in a PostGIS database
Paolo Corti
Thomas J Kraft
Stephen Vincent Mather
Bborie Park
[PACKT] open source

PostGIS Cookbook

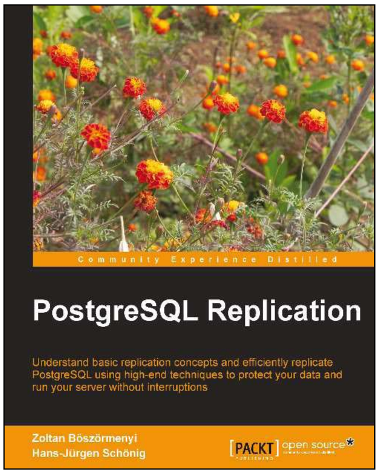
ISBN: 978-1-84951-866-6 Paperback: 484 pages

Over 80 task-based recipes to store, organize, manipulate, and analyze spatial data in a PostGIS database

1. Integrate PostGIS with web frameworks and implement OGC standards such as WMS and WFS using MapServer and GeoServer.

2. Convert 2D and 3D vector data, raster data, and routing data into usable forms.

3. Visualize data from the PostGIS database using a desktop GIS program such as QGIS and OpenJUMP.


Community Experience Distilled
PostgreSQL Replication
Understand basic replication concepts and efficiently replicate PostgreSQL using high-end techniques to protect your data and run your server without interruptions
Zoltan Böszörmenyi
Hans-Jürgen Schönig
[PACKT] open source

PostgreSQL Replication

ISBN: 978-1-84951-672-3 Paperback: 250 pages

Understand basic replication concepts and efficiently replicate PostgreSQL using high-end techniques to protect your data and run your server without interruptions

1. Explains the new replication features introduced in PostgreSQL 9.

2. Contains easy to understand explanations and lots of screenshots that simplify an advanced topic like replication.

3. Teaches PostgreSQL administrators how to maintain consistency between redundant resources and to improve reliability, fault-tolerance, and accessibility.

Please check **www.PacktPub.com** for information on our titles