Community Experience Distilled

PostgreSQL for Data Architects

Discover how to design, develop, and maintain your database application effectively with PostgreSQL

Jayadevan Maymala

[PACKT] PUBLISHING open source* community experience distilled

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 **'Client Tools'**
- A synopsis of the book’s content
- More information on **PostgreSQL for Data Architects**

About the Author

Jayadevan Maymala is a database developer, designer, and architect. He started working with the Oracle database in 1999. Over the years, he has worked with DB2, Sybase, and SQL Server. Of late, he has been working with open source technologies. His database of choice is PostgreSQL. In his career, he has worked in different domains spanning supply chain management, finance, and travel. He has spent an equal amount of time working with databases supporting critical transaction processing systems as well as data warehouses supporting analytical systems.

When he is not working on open source technologies, he spends time reading and updating himself on economic and political issues.

PostgreSQL for Data Architects

PostgreSQL is an incredibly flexible and dependable open source relational database. Harnessing its power will make your applications more reliable and extensible without increasing costs. Using PostgreSQL's advanced features will save you work and increase performance, once you've discovered how to set it up.

PostgreSQL for Data Architects will teach you everything you need to learn in order to get a scalable and optimized PostgreSQL server up and running.

The book starts with basic concepts (such as installing PostgreSQL from source) and covers theoretical aspects (such as concurrency and transaction management). After this, you'll learn how to set up replication, use load balancing to scale horizontally, and troubleshoot errors.

As you continue through this book, you will see the significant impact of configuration parameters on performance, scalability, and transaction management. Finally, you will get acquainted with useful tools available in the PostgreSQL ecosystem used to analyze PostgreSQL logs, set up load balancing, and recovery.

What This Book Covers

Chapter 1, *Installing PostgreSQL*, provides an overview of the process to install PostgreSQL from source. The chapter covers the prerequisites to compile from source, and the process to initialize a cluster in Unix/Linux environment. It also covers the directory structure.

Chapter 2, *Server Architecture*, covers the important processes started when we start a PostgreSQL cluster and how they work along with the memory structures to provide the functionality expected from a database management system.

Chapter 3, *PostgreSQL – Object Hierarchy and Roles*, explains various object types and objects provided by PostgreSQL. Important concepts such as databases, clusters, tablespaces, and schemas are covered in this chapter.

Chapter 4, *Working with Transactions*, covers ACID properties of transactions, isolation levels, and how PostgreSQL provides them. Multiversion concurrency control is another topic dealt with in this chapter.

Chapter 5, *Data Modeling with SQL Power Architect*, talks about how we can model tables and relationships with SQL Power Architect. Some of the aspects that should be considered when we choose a design tool are also covered in this chapter.

Chapter 6, *Client Tools*, covers two clients tools (pgAdmin: a UI tool and psql: a command-line tool). Browsing database objects, generating queries, and generating the execution plan for queries using pgAdmin are covered. Setting up the environment variables for connecting from psql, viewing history of SQL commands executed, and meta-commands are also covered in this chapter.

Chapter 7, *SQL Tuning*, explains query optimization techniques. To set the context, some patterns about database use and theory on how the PostgreSQL optimizer works are covered.

Chapter 8, *Server Tuning*, covers PostgreSQL server settings that have significant impact on query performance. These include memory settings, cost settings, and so on. Two object types: partitions and materialized views are also covered in this chapter.

Chapter 9, *Tools to Move Data in and out of PostgreSQL*, covers common tools/utilities, such as pg_dump, pg_bulkload, and copy used to move data in and out of PostgreSQL.

Chapter 10, Scaling, Replication, and Backup and Recovery, covers methods that are usually used for achievability. A step-by-step method to achieve horizontal scalability using PostgreSQL's streaming replication and pgpool-II is also presented. Point-in-time recovery for PostgreSQL is also covered in this chapter.

Chapter 11, PostgreSQL – Troubleshooting, explains a few of the most common problems developers run into when they start off with PostgreSQL and how to troubleshoot them. Connection issues, privilege issues, and parameter setting issues are also covered.

Chapter 12, PostgreSQL – Extras, covers quite a few topics. Some interesting data types that every data architect should be aware of, a couple of really useful extensions, and a tool to analyze PostgreSQL log files are covered. It also covers a few interesting features available in PostgreSQL 9.4.

6
Client Tools

In the previous chapter, we looked at a tool that is used to design databases. Now, let's cover a few tools that are used with PostgreSQL to manipulate data, create, drop, and alter objects, find out what is happening on the server, and so on. In this chapter, we will cover one GUI and one command-line tool that are used to work with PostgreSQL. We will see how database connections are made, how SQL statements are executed, and how database objects and related metadata can be viewed. We will also look at a couple of advanced use cases (such as generating the plan for queries and changing configuration parameters).

GUI tools and command-line tools

psql is probably the most favored client tool to work with PostgreSQL, and pgAdmin is the popular GUI tool. We will cover a couple of basic features of pgAdmin and also have a look at a few not-so-basic features. To cover the tool exhaustively with screenshots, it will consume many pages and probably not add much value. There are other options such as phpPgAdmin and TOra, (toolkit for Oracle, but supports many databases) which you could explore.

pgAdmin – downloading and installation

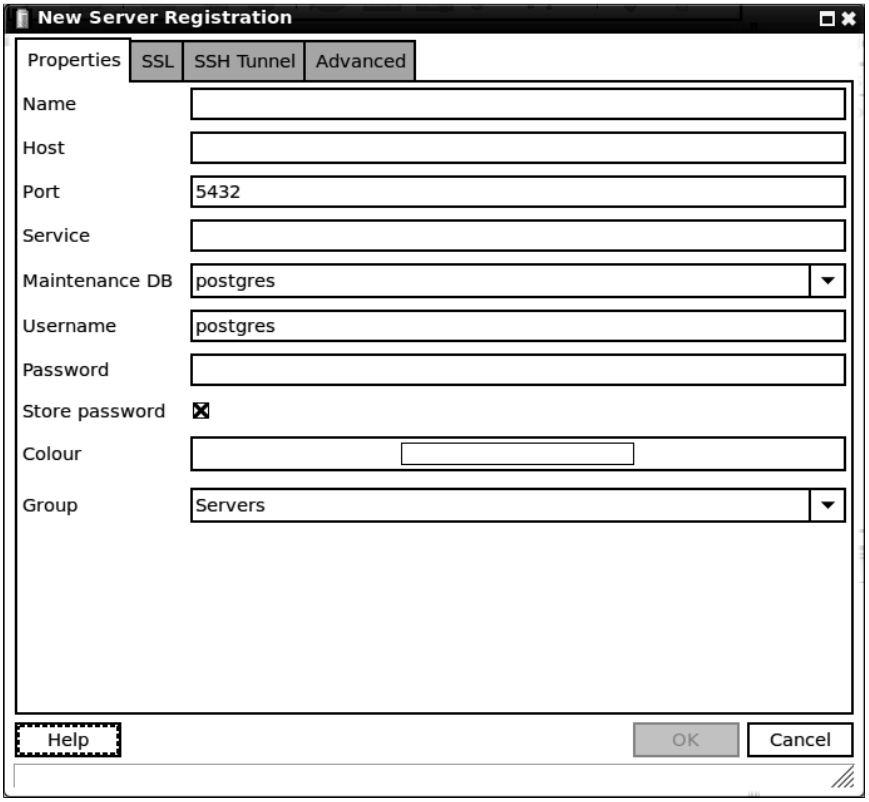
The URL `http://www.pgadmin.org/download/` provides links to various options to get the tool, including downloading and compiling from source. Compiling from source might not be easy, as there are quite a few dependencies and getting all of them to behave is a bit tough, unless you have been installing quite a bit of Linux- or Unix-based software via the compile-from-source option. For Windows, the tool can be installed using the point and click installer. For Linux-based systems, `http://yum.postgresql.org/` and `http://wiki.postgresql.org/wiki/Apt` should provide the setup instructions.

When you use the commands provided in the links to install pgAdmin, pay attention and ensure that you install only the pgAdmin software. If you just copy/paste the commands, you might end up overwriting your PostgreSQL installation.

Adding a server

The first thing we want to do once pgAdmin has been installed and started is set up a connection. For this, we use the **Add Server** option.

Click on **File** and choose **Add Server**. We get a dialog box with various options, as shown in the following screenshot:



Name	
Host	
Port	5432
Service	
Maintenance DB	postgres
Username	postgres
Password	
Store password	☒
Colour	
Group	Servers

Let's look at these options in detail:

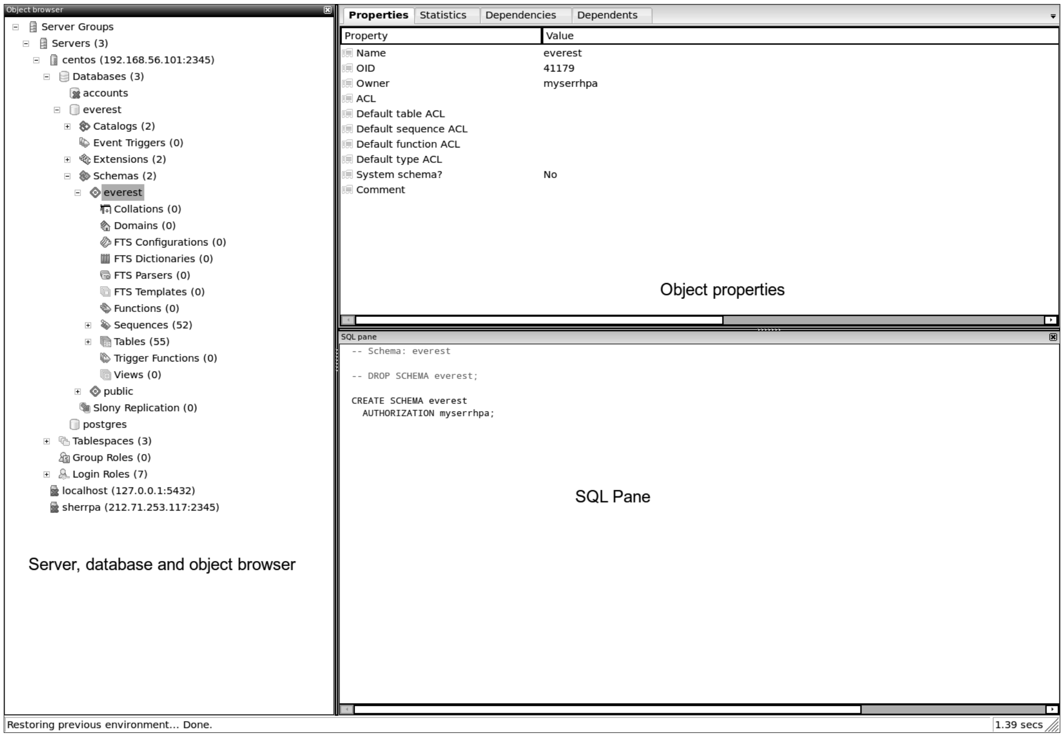
- **Name**: This can be anything that we want to use for reference. It could be `MyProductionServer1`. It's just an identifier to help us figure out which server we are looking at, if we work with many servers.
- **Host**: This is the IP address of the machine where our cluster is running. It can also be the fully qualified hostname.
- **Port**: This is the port where the cluster listens for requests.
- **Maintenance DB**: This is used to specify the initial database that pgAdmin connects to. It's the database where pgAdmin will check for optional but useful modules and extensions (such as `pgAgent` and `adminpack`).
- **Service**: This refers to the name of a service configured in the `pg_service.conf` file. We will not be using it. Detailed documentation about the file is available at `http://www.postgresql.org/docs/current/interactive/libpq-pgservice.html`.
- **Username**: This is the user that you will be connecting as. We could choose to save the password. If we do this, the password will be stored in a password file named `.pgpass` in the `home` directory if we are working in a Unix/Linux environment. In Windows, the password file will be under `%APPDATA%\postgresql\`, and the file will be named `pgpass.conf`. We can use the option under the **File** menu to edit the password file.

This covers the basic options. The other tabs have the not-so-basic settings. For example, we can use the **SSH Tunnel** option if the server does not allow direct connections from the client machine. This is useful when we are working for a client. Only a few machines in the client's network are allowed access from an external network. The machines on which databases are hosted will definitely not be among the machines exposed to external networks.

If you are working with the database cluster on a different machine, check whether you have modified the `postgresql.conf` (`listen_addresses` entry) and `pg_hba.conf` files on the server so that you can connect from a remote machine. The documentation links are `http://www.postgresql.org/docs/current/static/runtime-config-connection.html` and `http://www.postgresql.org/docs/current/static/auth-pg-hba-conf.html`.

The pgAdmin main window

Once we are connected to a server, we can go ahead and see what else is possible with pgAdmin. The main window has three panes, as shown in the following screenshot:



```
Server Groups
  Servers (3)
    centos (192.168.56.101:2345)
      Databases (3)
        accounts
        everest
          Catalogs (2)
          Event Triggers (0)
          Extensions (2)
          Schemas (2)
            everest
              Collations (0)
              Domains (0)
              FTS Configurations (0)
              FTS Dictionaries (0)
              FTS Parsers (0)
              FTS Templates (0)
              Functions (0)
              Sequences (52)
              Tables (55)
              Trigger Functions (0)
              Views (0)
            public
          Slony Replication (0)
        postgres
      Tablespaces (3)
      Group Roles (0)
      Login Roles (7)
    localhost (127.0.0.1:5432)
    sherrpa (212.71.253.117:2345)
```

Property	Value
Name	everest
OID	41179
Owner	myserrhpa
ACL	
Default table ACL	
Default sequence ACL	
Default function ACL	
Default type ACL	
System schema?	No
Comment	

```
-- Schema: everest

-- DROP SCHEMA everest;

CREATE SCHEMA everest
  AUTHORIZATION myserrhpa;
```

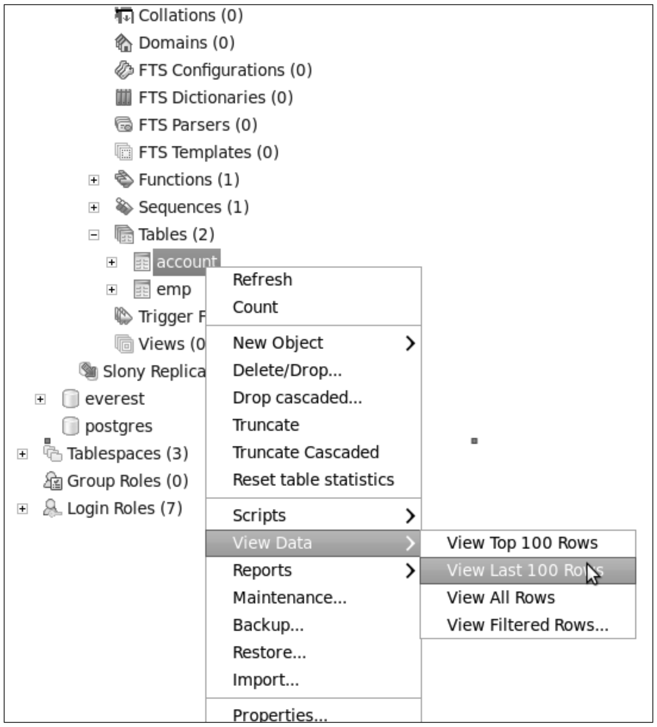
```
Restoring previous environment... Done.                    1.39 secs
```

In the left-hand side pane, we have the **Object Browser** pane that shows all objects in a hierarchical manner. At the top of the hierarchy, we have **Server Groups** and then **Servers**. This is where choosing a good name for the server helps. At the next level and the levels below is the rest of the hierarchy is visible.

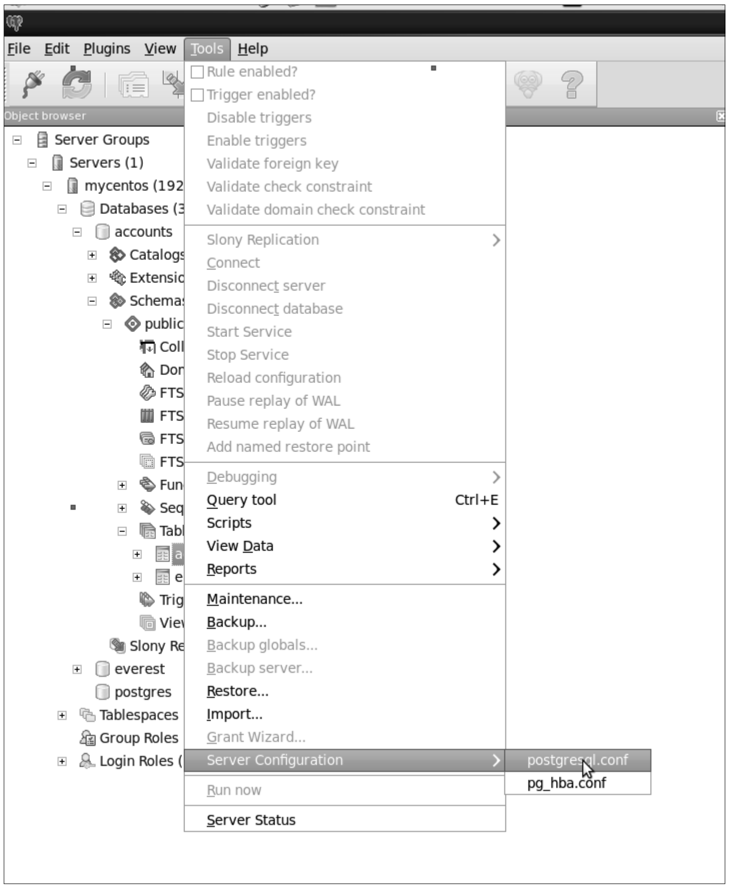
The top right-hand side pane provides quite a lot of very useful information. The first tab provides information such as name of the object, object identifier, estimated rows, and comment. Even more important, in many cases, is the next tab: **Statistics**. This provides information regarding reads, sequential scans, index scans, and other information that will be useful when we are trying to optimize database queries or having another look at the table design. The data shown in the **Statistics** tab as well as the other tabs in the right-hand side pane changes depending on the type of object we select in the object browser.

The bottom right-hand side pane has the SQL for the object selected. Right-clicking on a table in the **Object Browser** pane gives you a few options. One of the options is the data viewer. In the data viewer menu, we can edit existing data, add new records, and delete records. The data viewer menu also lets us apply filters and select specific records.

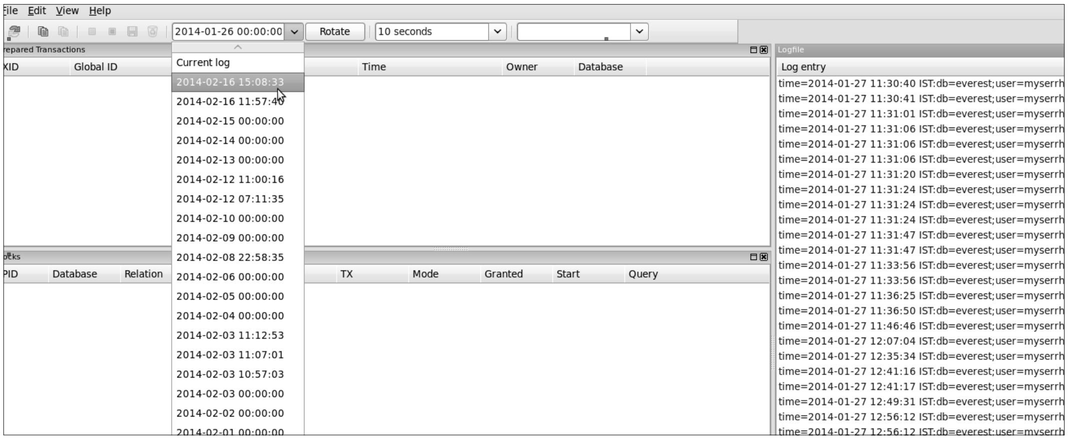
In the case of a table, the right-click options lets us take a backup of the table, import data, restore from a backup, and so on, as shown in the following screenshot:



The **Tools** menu provides many useful options as well. The important ones are listed next. The **Server Configuration** option lets us see the current settings, edit them, and reload the parameter values, as shown in the following screenshot. Some of the settings need a server restart. So, the effect might not be visible immediately. The user must have appropriate permissions to use this feature.



The **Server Status** option lets us see what is happening at the server. It lists the current activity, transactions, locks, and the information getting logged in to the log file. If you are not comfortable with the command-line tool and the PostgreSQL views, which store database activity data, this option can be used to retrieve pretty much the same information that you could get from these views. In addition, we can also retrieve past log files, as shown in the following screenshot:

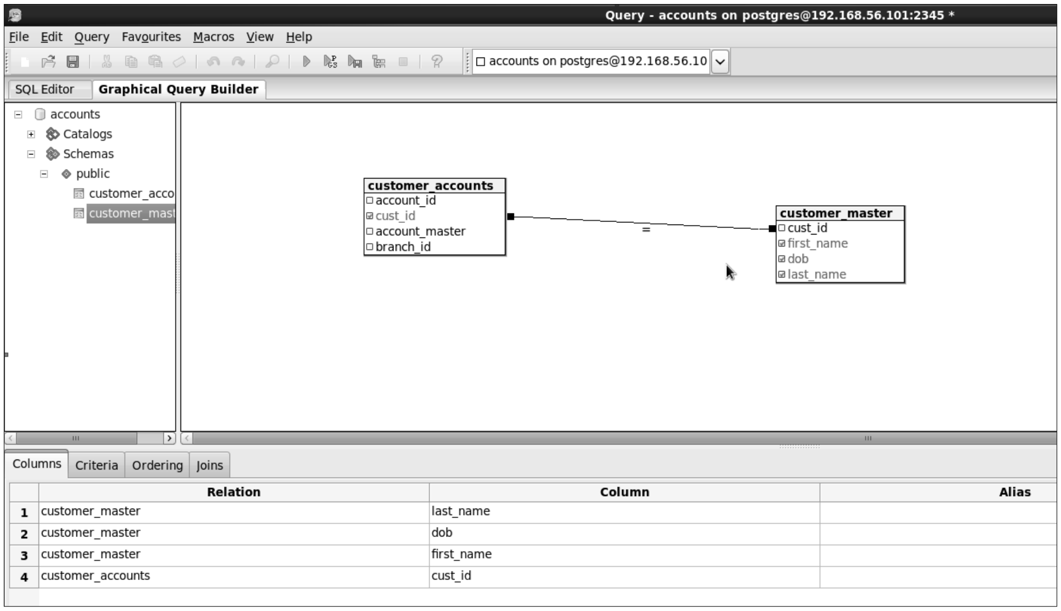


```
time=2014-01-27 11:30:40 IST:db=everest;user=myserrh
time=2014-01-27 11:30:41 IST:db=everest;user=myserrh
time=2014-01-27 11:31:01 IST:db=everest;user=myserrh
time=2014-01-27 11:31:06 IST:db=everest;user=myserrh
time=2014-01-27 11:31:06 IST:db=everest;user=myserrh
time=2014-01-27 11:31:06 IST:db=everest;user=myserrh
time=2014-01-27 11:31:20 IST:db=everest;user=myserrh
time=2014-01-27 11:31:24 IST:db=everest;user=myserrh
time=2014-01-27 11:31:24 IST:db=everest;user=myserrh
time=2014-01-27 11:31:24 IST:db=everest;user=myserrh
time=2014-01-27 11:31:47 IST:db=everest;user=myserrh
time=2014-01-27 11:31:47 IST:db=everest;user=myserrh
time=2014-01-27 11:33:56 IST:db=everest;user=myserrh
time=2014-01-27 11:33:56 IST:db=everest;user=myserrh
time=2014-01-27 11:36:25 IST:db=everest;user=myserrh
time=2014-01-27 11:36:50 IST:db=everest;user=myserrh
time=2014-01-27 11:46:46 IST:db=everest;user=myserrh
time=2014-01-27 12:07:04 IST:db=everest;user=myserrh
time=2014-01-27 12:35:34 IST:db=everest;user=myserrh
time=2014-01-27 12:41:16 IST:db=everest;user=myserrh
time=2014-01-27 12:41:17 IST:db=everest;user=myserrh
time=2014-01-27 12:49:31 IST:db=everest;user=myserrh
time=2014-01-27 12:56:12 IST:db=everest;user=myserrh
time=2014-01-27 12:56:12 IST:db=everest;user=myserrh
```

The Query tool

Once a database with the important tables is in place, we are likely to spend quite a lot of time querying the tables, manipulating the data, writing functions, and so on. For this, the **Query** tool can be used. The **Query** tool is one of the options we get when we choose the **Tools** option. The **Query** tool option gets activated only after we are connected to a database.

In the **Query** tool, writing and executing a SQL statement is a straightforward thing to do. It also has a graphical query builder, which is intuitive for simple queries. As shown in the following screenshot, we can choose the tables and columns we need, join tables by choosing the columns from one table and then drag it to the column to be joined within the other table:

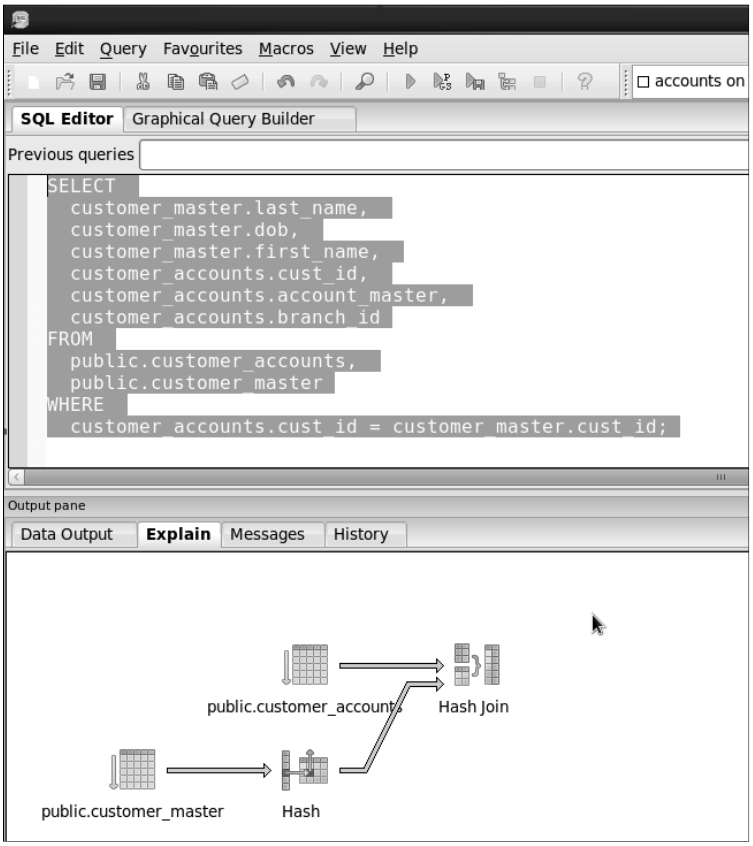


	Relation	Column	Alias
1	customer_master	last_name	
2	customer_master	dob	
3	customer_master	first_name	
4	customer_accounts	cust_id	

The query generated is provided here:

```
SELECT
  customer_master.last_name,
  customer_master.dob,
  customer_master.first_name,
  customer_accounts.cust_id
FROM
  public.customer_accounts,
  public.customer_master
WHERE
  customer_accounts.cust_id = customer_master.cust_id;
```

Another useful feature is the **Explain** pane. We can select a query and use *Shift* + *F7* to generate a graphical query execution plan. The plan for the preceding query is displayed as follows:



```
SELECT
  customer_master.last_name,
  customer_master.dob,
  customer_master.first_name,
  customer_accounts.cust_id,
  customer_accounts.account_master,
  customer_accounts.branch_id
FROM
  public.customer_accounts,
  public.customer_master
WHERE
  customer_accounts.cust_id = customer_master.cust_id;
```

pgAdmin does provide us with many more features. We can click on **File** and then select **Options** to set which object types should be displayed in the **Object Browser** pane. The right-click options in **Database** or **Tables** lets us carry out maintenance activities (such as vacuuming and analyzing). Right-clicking on **Tables** provides a **Script** menu that can be used to generate scripts for the `CREATE`, `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. We can also generate reports from the database.

It's necessary to install the `adminpack` extension for some of the features to work. This can be done using the `CREATE EXTENSION` command.

This covers the features one will use most frequently in pgAdmin. Next, we will cover psql—the command-line utility to work with PostgreSQL clusters.

psql – working from the command line

psql is a very powerful command-line tool and is superior to similar utilities available with other databases. For example, if we are not sure about the name of a table (did I name it `emp`, `employee`, or `employees`?), the only option available in Oracle's SQL* Plus is to query a data dictionary view (such as `user_tables` or `tab`) with a `LIKE` filter. In psql, we can just type `\dt e` and hit *Tab* twice to get the list of tables starting with `e`. In short, psql supports tab completion—a nifty feature that is a great productivity booster. Now that it sounds interesting, let's look at more.

psql – connection options

The `psql --help` command lists all possible options that can be used with psql. We might want to use psql to execute a command, or to connect to a database and remain at the psql prompt to carry out many activities, execute queries, describe tables, create users, alter objects, and so on. Either way, psql needs to know:

- the host on which the PostgreSQL cluster is running
- the port at which the cluster is listening
- the database to which connection is to be made
- the user ID and password (depends on the authentication method)

Each of these options can be provided in a one letter form or in a detailed manner. For example, we could say which database to connect to use `-d mydatabase` or `--dbname=mydatabase`, as shown here:

```
psql --host=localhost --dbname=test
psql (9.3.0)
Type "help" for help.
test=# \q

psql  -h localhost -d test
psql (9.3.0)
Type "help" for help.
test=# \q
```

It makes sense to use the long option initially to get familiar with the possible options. In the case of options (such as port) `--port` is pretty clear, but if we type `-P`, the error appears because `-P` is to specify printing options, not port. We use `-p` to specify port:

```
psql -h localhost -d test -U postgres -P 5432
\pset: unknown option: 5432
psql: could not set printing parameter "5432"
```

While we can definitely execute queries once we are at the psql prompt, it's the rich set of meta-commands that deserve more attention. To say that we can execute SQL is to state the obvious.

The power of \d

Any command that starts with a backslash is referred to as a meta-command. Among the numerous meta-commands, it's likely that you will find `\d` to be most useful. `\d` has a number of options. In its simplest form, without any options, it lists all user-created objects in the database, as shown here (first, create a database `myobjects` and a few objects of different types if you want to follow along, or you could try it on the database you have created):

```
myobjects=# \d
             List of relations
 Schema |    Name    |   Type   |  Owner
--------+------------+----------+----------
 public | mysequence | sequence | postgres
 public | mytable    | table    | postgres
 public | myview     | view     | postgres
(3 rows)
```

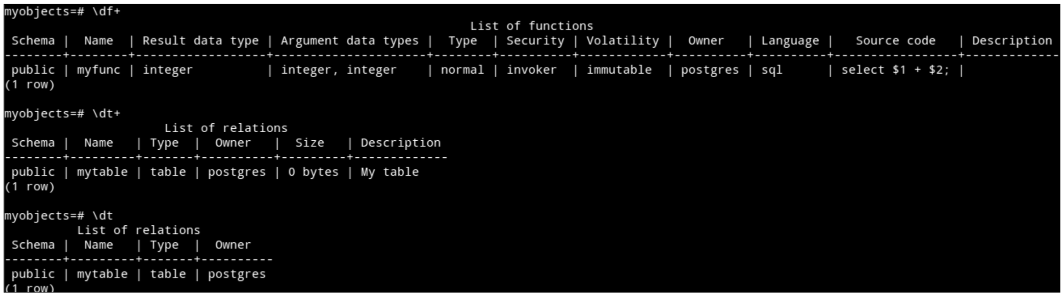
If you want to just see the tales, use `\dt` (`t` for table):

```
myobjects=# \dt
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | mytable | table | postgres
(1 row)
myobjects=# \ds
             List of relations
 Schema |    Name    |   Type   |  Owner
--------+------------+----------+----------
 public | mysequence | sequence | postgres
(1 row)
```

```
myobjects=# \dv
          List of relations
 Schema |  Name  | Type |  Owner
--------+--------+------+----------
 public | myview | view | postgres
(1 row)

myobjects=# \df
                              List of functions
 Schema |  Name  | Result data type | Argument data types |  Type
--------+--------+------------------+---------------------+--------
 public | myfunc | integer          | integer, integer    | normal
(1 row)
```

We can use `s` for sequence, `v` for view, `f` for function, and so on. For schemas, we have to use `n`, which denotes namespace. A `+` sign at the end will give us some more information (such as comments for tables and additional information for functions), as shown in the following screenshot:



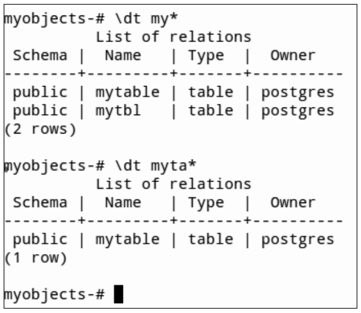
```
myobjects=# \df+
                                                                  List of functions
 Schema |  Name  | Result data type | Argument data types |  Type  | Security | Volatility |  Owner   | Language |   Source code    | Description
--------+--------+------------------+---------------------+--------+----------+------------+----------+----------+------------------+-------------
 public | myfunc | integer          | integer, integer    | normal | invoker  | immutable  | postgres | sql      | select $1 + $2; |
(1 row)

myobjects=# \dt+
                    List of relations
 Schema |  Name   | Type  |  Owner   |  Size   | Description
--------+---------+-------+----------+---------+-------------
 public | mytable | table | postgres | 0 bytes | My table
(1 row)

myobjects=# \dt
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | mytable | table | postgres
(1 row)
```

The `+` sign is very useful when we want to see a view or function definition. `\d` without any letter after it is interpreted as `\dtvsE`. In this, `E` stands for foreign tables. We have already covered the rest.

By the way, we can also use patterns, as shown here:

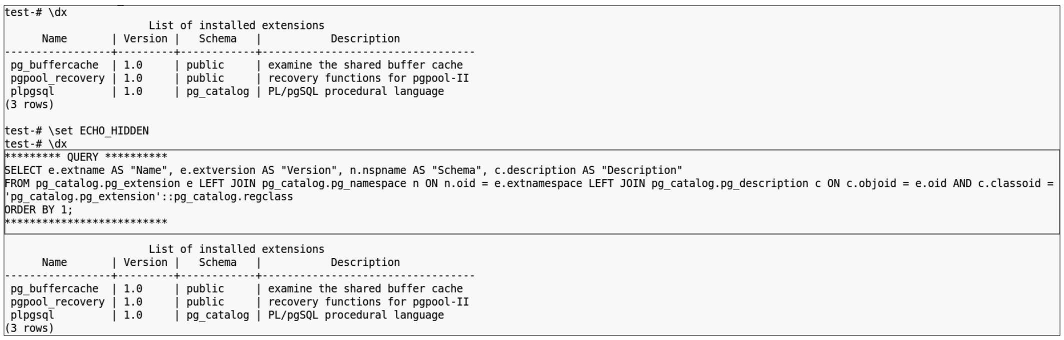


```
myobjects-# \dt my*
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | mytable | table | postgres
 public | mytbl   | table | postgres
(2 rows)

myobjects-# \dt myta*
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | mytable | table | postgres
(1 row)

myobjects-# 
```

We can use psql with the `-E` option. This will display all the SQL statements used by PostgreSQL internally when we execute a `\d` command or other backslash commands. In the following example, the setting is turned `ON` from psql:



```
test-# \dx
                        List of installed extensions
      Name       | Version |   Schema   |          Description
-----------------+---------+------------+---------------------------------
 pg_buffercache  | 1.0     | public     | examine the shared buffer cache
 pgpool_recovery | 1.0     | public     | recovery functions for pgpool-II
 plpgsql         | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)

test-# \set ECHO_HIDDEN
test-# \dx
********* QUERY **********
SELECT e.extname AS "Name", e.extversion AS "Version", n.nspname AS "Schema", c.description AS "Description"
FROM pg_catalog.pg_extension e LEFT JOIN pg_catalog.pg_namespace n ON n.oid = e.extnamespace LEFT JOIN pg_catalog.pg_description c ON c.objoid = e.oid AND c.classoid =
'pg_catalog.pg_extension'::pg_catalog.regclass
ORDER BY 1;
**************************

                        List of installed extensions
      Name       | Version |   Schema   |          Description
-----------------+---------+------------+---------------------------------
 pg_buffercache  | 1.0     | public     | examine the shared buffer cache
 pgpool_recovery | 1.0     | public     | recovery functions for pgpool-II
 plpgsql         | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

After the setting is turned `ON`, we can see the query executed:

psql rocks!

More meta-commands

The `\h` command provides help. So does `\?`. The first one provides help for SQL commands, whereas the second one provides help for psql commands (for example, meta-commands).

In previous chapters, we saw that many host commands can be executed from the psql prompt in the following manner. The example also shows how output can be redirected to a file. Note that in the first command where we redirect output, there is no `!` after \:

```
test=# \o out.txt
test=# show data_directory;
test=# \! cat out.txt
 data_directory
----------------
 /pgdata/9.3
(1 row)

test=# \! ls
out.txt
test=# \o
test=# \! rm out.txt
test=# \! ls
```

The file is gone. Now, we will see how we can execute SQL statements in a file from psql. We use the `\i` option for this:

```
test=# \! echo "SELECT 1; " > a.sql
test=# \i a.sql
 ?column?
----------
        1
(1 row)

test=# \! cat a.sql
SELECT 1;
```

Another important command is to change the setting for timing. How much time the query is taking is something we want to know when we are trying to optimize a query. To enable this, we can use the following code:

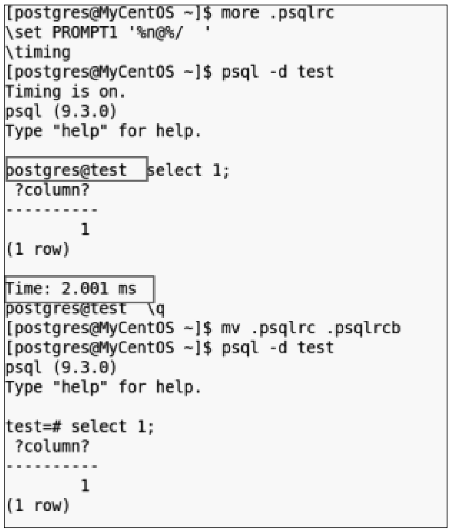
```
postgres=# \timing
Timing is off.
postgres=# \timing
Timing is on.
postgres=# select now();
              now
-------------------------------
 2014-02-19 03:06:53.785452+00
(1 row)

Time: 0.202 ms
```

Timing is a toggle option. It makes sense to have this on by default. In addition to timing, there can be other non-default settings that we want to use. These can be set using the `.psqlrc` file. Let's see how it works. psql will look for the system-wide startup file (`psqlrc`) and the user-specific startup file (`.psqlrc`) and execute commands in these files before accepting user input. If these files are not found, psql will go ahead with the default settings. In Windows, the personal startup file will be `%APPDATA%\postgresql\psqlrc.conf`. We can create these files if they are not present, which is likely. We will create a file with two non-default settings and see how they work. We will change the timing option and psql prompt to non-default values. In the `home` directory, create a file named `.psqlrc`, and make these two entries:

```
\set PROMPT1 '%n@%/  '
\timing
```

The following screenshot illustrates the effect of the settings:



```
[postgres@MyCentOS ~]$ more .psqlrc
\set PROMPT1 '%n@%/  '
\timing
[postgres@MyCentOS ~]$ psql -d test
Timing is on.
psql (9.3.0)
Type "help" for help.

postgres@test  select 1;
 ?column?
----------
        1
(1 row)

Time: 2.001 ms
postgres@test  \q
[postgres@MyCentOS ~]$ mv .psqlrc .psqlrcb
[postgres@MyCentOS ~]$ psql -d test
psql (9.3.0)
Type "help" for help.

test=# select 1;
 ?column?
----------
        1
(1 row)
```

The prompt has changed to include the user (`%n`), the symbol (`@`), and database name (`%/`). The file was renamed to `.psqlrcb` just to see what the prompt looked like if there was no `.psqlrc` file. We can try renaming the `.psqlrcb` file to `.psqlrc` so that it's used by psql. Add an entry in the file, as shown here:

```
\set HISTFILE ~/history-:DBNAME
```

If we connect to a couple of databases and exit, for each database we connected to, there will be a separate history file.

Executing `\conninfo` at the psql prompt gives us information (such as user, database connected to, and port).

By now, we have covered most of the commands that can be used to get information about the environment, switch between databases, read from and write to files, and so on. For an exhaustive listing of the possibilities, we can get help for meta-commands at the psql prompt by typing `\?`. The commands are neatly grouped together into different sets: general, query buffer, input/output, and so on. Also, refer to `http://www.postgresql.org/docs/current/static/app-psql.html` for more options and explanation.

Setting up the environment

We will usually work with a specific set of servers and databases. Typing in the information regarding the database name, user, host, port, and so on, can get tedious. With pgAdmin, we can store the information for different environments. What about psql? This is where environment variables help. The following list covers a few important variables (there are more):

- `PGHOST`: This is the address of the host where the cluster is running
- `PGPORT`: This is the port at which the server is listening
- `PGDATABASE`: This is the database to connect to
- `PGUSER`: This is the database that the user can connect to

Now, we can create a few environment files and source them as required. There can be better options, but this is one. Let's create a file named `.mylocalenv`. Its entries are as follows:

```
$ more .mylocalenv
export PGHOST=localhost
export PGPORT=5432
export PGUSER=postgres
export PGDATABASE=myobjects
```

We can source the file as follows:

```
$ source ./.mylocalenv
```

Ensure that the values have been set as:

```
$ env | grep PG
PGPORT=5432
PGUSER=postgres
PGDATABASE=myobjects
PGHOST=localhost
```

Now, try connecting:

```
$ psql
Password:
psql (9.3.0)
Type "help" for help.

myobjects=# SELECT current_database();
 current_database
------------------
 myobjects
(1 row)
```

One item that we left out is the password. It's not a good idea to set this as an environment variable. For this, we use the `.pgpass` file in our `home` directory. A reference to this file was made in the pgAdmin section.

The `.pgpass` file contains entries in `host:port:database:user:password` form; for example:

```
more ~/.pgpass
localhost:5432:myobjects:postgres:tcc123
```

If we try to connect to the default database (`myobjects`, that is, the database set in the `env` file), it does not ask for a password. However, if we try to connect to another database, the password prompt appears:

```
$ psql
psql (9.3.0)
Type "help" for help.

myobjects=# \q
postgres@jayadevan-Vostro-2520:~$ psql -d postrges
Password:
```

The password is stored as plain text in the `.pgpass` file. This is not the recommended method to store passwords.

History of commands

The `\s` command lists all the commands you have executed so far. The data is stored in `~/.psql_history`, that is, `.psql_history` under the `home` directory of the Linux user under which we are connecting to psql.

Summary

In this chapter, we covered two PostgreSQL tools: one GUI tool and one command-line tool. While the GUI tool might be easier to learn than the other one, it's better to learn the command-line tool because it provides you with a lot of flexibility. For example, you could create SQL files, call them from shell scripts and schedule them in cron. For such automated batch jobs, psql usually proves to be more powerful than a GUI tool. Also, for efficiency, it's better to work from command line. Typing `\d` and hitting *Enter* is faster than moving the mouse around; click on an icon and wait for the data to be displayed. In some cases, such as the execution plan for a query, the visual display might be easier to understand than text. Choosing between a point-and-click option and a command-line option is definitely a matter of personal preference. You can take your pick.

In the next chapter, we will take a look at the PostgreSQL optimizer and see how queries can be optimized with this information.