I. Tutorial

Welcome to the PostgreSQL Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading Part II to gain a more formal knowledge of the SQL language, or Part IV for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read Part III.

Chapter 1. Getting Started

1.1. Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user; no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to Chapter 15 for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you might have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` might also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2. Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called `postgres`.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. To achieve this it starts ("forks") a new process for each connection. From that point on, the client and the new

server process communicate without intervention by the original `postgres` process. Thus, the master server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3. Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. He should have told you what the name of your database is. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named `mydb`, you use the following command:

```
$ createdb mydb
```

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to:

```
createdb: command not found
```

then PostgreSQL was not installed properly. Either it was not installed at all or your shell's search path was not set to include it. Try calling the command with an absolute path instead:

```
$ /usr/local/pgsql/bin/createdb mydb
```

The path at your site might be different. Contact your site administrator or check the installation instructions to correct the situation.

Another response could be this:

```
createdb: could not connect to database postgres: could not connect to server: No such f
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

This means that the server was not started, or it was not started where `createdb` expected it. Again, check the installation instructions or consult the administrator.

Another response could be this:

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see Chapter 20 for help creating accounts. You will need to become the operating system user under which PostgreSQL was installed (usually `postgres`) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the `-U` switch or set the `PGUSER` environment variable to specify your PostgreSQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:

```
createdb: database creation failed: ERROR:  permission denied to create database
```

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 bytes in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type:

```
$ createdb
```

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database `mydb`, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about `createdb` and `dropdb` can be found in createdb and dropdb respectively.

1.4. Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in Part IV.

You probably want to start up `psql` to try the examples in this tutorial. It can be activated for the `mydb` database by typing the command:

```
$ psql mydb
```

If you do not supply the database name then it will default to your user account name. You already discovered this scheme in the previous section using `createdb`.

In `psql`, you will be greeted with the following message:

1. As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. When you connect to a database, you can choose what PostgreSQL user name to connect as; if you don't, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the `-U` option everywhere to select a PostgreSQL user name to connect as.

```
psql (9.5.10)
Type "help" for help.

mydb=>
```

The last line could also be:

```
mydb=#
```

That would mean you are a database superuser, which is most likely the case if you installed the PostgreSQL instance yourself. Being a superuser means that you are not subject to access controls. For the purposes of this tutorial that is not important.

If you encounter problems starting `psql` then go back to the previous section. The diagnostics of `createdb` and `psql` are similar, and if the former worked the latter should work as well.

The last line printed out by `psql` is the prompt, and it indicates that `psql` is listening to you and that you can type SQL queries into a work space maintained by `psql`. Try out these commands:

```
mydb=> SELECT version();
                                  version
 ---------------------------------------------------------------------------
 PostgreSQL 9.5.10 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

The `psql` program has a number of internal commands that are not SQL commands. They begin with the backslash character, "`\`". For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of `psql`, type:

```
mydb=> \q
```

and `psql` will quit and return you to your command shell. (For more internal commands, type `\?` at the `psql` prompt.) The full capabilities of `psql` are documented in psql. In this tutorial we will not use these features explicitly, but you can use them yourself when it is helpful.

Chapter 2. The SQL Language

2.1. Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including *Understanding the New SQL* and *A Guide to the SQL Standard*. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named `mydb`, as described in the previous chapter, and have been able to start psql.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory `src/tutorial/`. (Binary distributions of PostgreSQL might not compile these files.) To use those files, first change to that directory and run make:

```
$ cd ..../src/tutorial
$ make
```

This creates the scripts and compiles the C files containing user-defined functions and types. Then, to start the tutorial, do the following:

```
$ cd ..../tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

The `\i` command reads in commands from the specified file. `psql`'s `-s` option puts you in single step mode which pauses before sending each statement to the server. The commands used in this section are in the file `basics.sql`.

2.2. Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*.

III. Server Administration

This part covers topics that are of interest to a PostgreSQL database administrator. This includes installation of the software, set up and configuration of the server, management of users and databases, and maintenance tasks. Anyone who runs a PostgreSQL server, even for personal use, but especially in production, should be familiar with the topics covered in this part.

The information in this part is arranged approximately in the order in which a new user should read it. But the chapters are self-contained and can be read individually as desired. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should see Part VI.

The first few chapters are written so they can be understood without prerequisite knowledge, so new users who need to set up their own server can begin their exploration with this part. The rest of this part is about tuning and management; that material assumes that the reader is familiar with the general use of the PostgreSQL database system. Readers are encouraged to look at Part I and Part II for additional information.

Chapter 15. Installation from Source Code

This chapter describes the installation of PostgreSQL using the source code distribution. (If you are installing a pre-packaged distribution, such as an RPM or Debian package, ignore this chapter and read the packager's instructions instead.)

15.1. Short Version

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

The long version is the rest of this chapter.

15.2. Requirements

In general, a modern Unix-compatible platform should be able to run PostgreSQL. The platforms that had received specific testing at the time of release are listed in Section 15.6 below. In the `doc` subdirectory of the distribution there are several platform-specific FAQ documents you might wish to consult if you are having trouble.

The following software packages are required for building PostgreSQL:

- GNU make version 3.80 or newer is required; other make programs or older GNU make versions will *not* work. (GNU make is sometimes installed under the name `gmake`.) To test for GNU make enter:

  ```
  make --version
  ```

- You need an ISO/ANSI C compiler (at least C89-compliant). Recent versions of GCC are recommended, but PostgreSQL is known to build using a wide variety of compilers from different vendors.
- tar is required to unpack the source distribution, in addition to either gzip or bzip2.
- The GNU Readline library is used by default. It allows psql (the PostgreSQL command line SQL interpreter) to remember each command you type, and allows you to use arrow keys to recall and edit previous commands. This is very helpful and is strongly recommended. If you don't want to use it then you must specify the `--without-readline` option to `configure`. As an alternative, you can often use the BSD-licensed `libedit` library, originally developed on NetBSD. The `libedit` library is GNU Readline-compatible and is used if `libreadline` is not found, or

Chapter 17. Server Setup and Operation

This chapter discusses how to set up and run the database server and its interactions with the operating system.

17.1. The PostgreSQL User Account

As with any server daemon that is accessible to the outside world, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data that is managed by the server, and should not be shared with other daemons. (For example, using the user `nobody` is a bad idea.) It is not advisable to install executables owned by this user because compromised systems could then modify their own binaries.

To add a Unix user account to your system, look for a command `useradd` or `adduser`. The user name postgres is often used, and is assumed throughout this book, but you can use another name if you like.

17.2. Creating a Database Cluster

Before you can do anything, you must initialize a database storage area on disk. We call this a *database cluster*. (The SQL standard uses the term catalog cluster.) A database cluster is a collection of databases that is managed by a single instance of a running database server. After initialization, a database cluster will contain a database named `postgres`, which is meant as a default database for use by utilities, users and third party applications. The database server itself does not require the `postgres` database to exist, but many external utility programs assume it exists. Another database created within each cluster during initialization is called `template1`. As the name suggests, this will be used as a template for subsequently created databases; it should not be used for actual work. (See Chapter 21 for information about creating new databases within a cluster.)

In file system terms, a database cluster is a single directory under which all data will be stored. We call this the *data directory* or *data area*. It is completely up to you where you choose to store your data. There is no default, although locations such as `/usr/local/pgsql/data` or `/var/lib/pgsql/data` are popular. To initialize a database cluster, use the command initdb, which is installed with PostgreSQL. The desired file system location of your database cluster is indicated by the `-D` option, for example:

```
$ initdb -D /usr/local/pgsql/data
```

Note that you must execute this command while logged into the PostgreSQL user account, which is described in the previous section.

> **Tip:** As an alternative to the `-D` option, you can set the environment variable `PGDATA`.

Alternatively, you can run `initdb` via the pg_ctl program like so:

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

This may be more intuitive if you are using `pg_ctl` for starting and stopping the server (see Section 17.3), so that `pg_ctl` would be the sole command you use for managing the database server instance.

initdb will attempt to create the directory you specify if it does not already exist. Of course, this will fail if initdb does not have permissions to write in the parent directory. It's generally recommendable that the PostgreSQL user own not just the data directory but its parent directory as well, so that this should not be a problem. If the desired parent directory doesn't exist either, you will need to create it first, using root privileges if the grandparent directory isn't writable. So the process might look like this:

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

initdb will refuse to run if the data directory exists and already contains files; this is to prevent accidentally overwriting an existing installation.

Because the data directory contains all the data stored in the database, it is essential that it be secured from unauthorized access. initdb therefore revokes access permissions from everyone but the PostgreSQL user.

However, while the directory contents are secure, the default client authentication setup allows any local user to connect to the database and even become the database superuser. If you do not trust other local users, we recommend you use one of initdb's -W, --pwprompt or --pwfile options to assign a password to the database superuser. Also, specify -A md5 or -A password so that the default trust authentication mode is not used; or modify the generated pg_hba.conf file after running initdb, but *before* you start the server for the first time. (Other reasonable approaches include using peer authentication or file system permissions to restrict connections. See Chapter 19 for more information.)

initdb also initializes the default locale for the database cluster. Normally, it will just take the locale settings in the environment and apply them to the initialized database. It is possible to specify a different locale for the database; more information about that can be found in Section 22.1. The default sort order used within the particular database cluster is set by initdb, and while you can create new databases using different sort order, the order used in the template databases that initdb creates cannot be changed without dropping and recreating them. There is also a performance impact for using locales other than C or POSIX. Therefore, it is important to make this choice correctly the first time.

initdb also sets the default character set encoding for the database cluster. Normally this should be chosen to match the locale setting. For details see Section 22.3.

Non-C and and non-POSIX locales rely on the operating system's collation library for character set ordering. This controls the ordering of keys stored in indexes. For this reason, a cluster cannot switch to an incompatible collation library version, either through snapshot restore, binary streaming replication, a different operating system, or an operating system upgrade.

17.2.1. Use of Secondary File Systems

Many installations create their database clusters on file systems (volumes) other than the machine's "root" volume. If you choose to do this, it is not advisable to try to use the secondary volume's topmost directory (mount point) as the data directory. Best practice is to create a directory within the mount-point directory that is owned by the PostgreSQL user, and then create the data directory within that. This avoids permissions problems, particularly for operations such as pg_upgrade, and it also ensures clean failures if the secondary volume is taken offline.

17.2.2. Use of Network File Systems

Many installations create their database clusters on network file systems. Sometimes this is done via NFS, or by using a Network Attached Storage (NAS) device that uses NFS internally. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives. If the client or server NFS implementation does not provide standard file system semantics, this can cause reliability problems (see http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html). Specifically, delayed (asynchronous) writes to the NFS server can cause data corruption problems. If possible, mount the NFS file system synchronously (without caching) to avoid this hazard. Also, soft-mounting the NFS file system is not recommended.

Storage Area Networks (SAN) typically use communication protocols other than NFS, and may or may not be subject to hazards of this sort. It's advisable to consult the vendor's documentation concerning data consistency guarantees. PostgreSQL cannot be more reliable than the file system it's using.

17.3. Starting the Database Server

Before anyone can access the database, you must start the database server. The database server program is called `postgres`. The `postgres` program must know where to find the data it is supposed to use. This is done with the `-D` option. Thus, the simplest way to start the server is:

```
$ postgres -D /usr/local/pgsql/data
```

which will leave the server running in the foreground. This must be done while logged into the PostgreSQL user account. Without `-D`, the server will try to use the data directory named by the environment variable `PGDATA`. If that variable is not provided either, it will fail.

Normally it is better to start `postgres` in the background. For this, use the usual Unix shell syntax:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

It is important to store the server's stdout and stderr output somewhere, as shown above. It will help for auditing purposes and to diagnose problems. (See Section 23.3 for a more thorough discussion of log file handling.)

The `postgres` program also takes a number of other command-line options. For more information, see the postgres reference page and Chapter 18 below.

This shell syntax can get tedious quickly. Therefore the wrapper program pg_ctl is provided to simplify some tasks. For example:

```
pg_ctl start -l logfile
```

will start the server in the background and put the output into the named log file. The `-D` option has the same meaning here as for `postgres`. `pg_ctl` is also capable of stopping the server.

Normally, you will want to start the database server when the computer boots. Autostart scripts are operating-system-specific. There are a few distributed with PostgreSQL in the `contrib/start-scripts` directory. Installing one will require root privileges.

Different systems have different conventions for starting up daemons at boot time. Many systems have a file `/etc/rc.local` or `/etc/rc.d/rc.local`. Others use `init.d` or `rc.d` directories. Whatever you do, the server must be run by the PostgreSQL user account *and not by root* or any

other user. Therefore you probably should form your commands using `su postgres -c '...'`. For example:

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Here are a few more operating-system-specific suggestions. (In each case be sure to use the proper installation directory and user name where we show generic values.)

- For FreeBSD, look at the file `contrib/start-scripts/freebsd` in the PostgreSQL source distribution.
- On OpenBSD, add the following lines to the file `/etc/rc.local`:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D
      echo -n ' postgresql'
  fi
  ```

- On Linux systems either add

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  to `/etc/rc.d/rc.local` or `/etc/rc.local` or look at the file `contrib/start-scripts/linux` in the PostgreSQL source distribution.
- On NetBSD, use either the FreeBSD or Linux start scripts, depending on preference.
- On Solaris, create a file called `/etc/init.d/postgresql` that contains the following line:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/dat
  ```

  Then, create a symbolic link to it in `/etc/rc3.d` as `S99postgresql`.

While the server is running, its PID is stored in the file `postmaster.pid` in the data directory. This is used to prevent multiple server instances from running in the same data directory and can also be used for shutting down the server.

17.3.1. Server Start-up Failures

There are several common reasons the server might fail to start. Check the server's log file, or start it by hand (without redirecting standard output or standard error) and see what error messages appear. Below we explain some of the most common error messages in more detail.

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds an
FATAL:  could not create TCP/IP listen socket
```

This usually means just what it suggests: you tried to start another server on the same port where one is already running. However, if the kernel error message is not `Address already in use` or some variant of that, there might be a different problem. For example, trying to start a server on a reserved port number might draw something like:

```
$ postgres -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and
FATAL:  could not create TCP/IP listen socket
```

A message like:

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

probably means your kernel's limit on the size of shared memory is smaller than the work area PostgreSQL is trying to create (4011376640 bytes in this example). Or it could mean that you do not have System-V-style shared memory support configured into your kernel at all. As a temporary workaround, you can try starting the server with a smaller-than-normal number of buffers (shared_buffers). You will eventually want to reconfigure your kernel to increase the allowed shared memory size. You might also see this message when trying to start multiple servers on the same machine, if their total space requested exceeds the kernel limit.

An error like:

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

does *not* mean you've run out of disk space. It means your kernel's limit on the number of System V semaphores is smaller than the number PostgreSQL wants to create. As above, you might be able to work around the problem by starting the server with a reduced number of allowed connections (max_connections), but you'll eventually want to increase the kernel limit.

If you get an "illegal system call" error, it is likely that shared memory or semaphores are not supported in your kernel at all. In that case your only option is to reconfigure the kernel to enable these features.

Details about configuring System V IPC facilities are given in Section 17.4.1.

17.3.2. Client Connection Problems

Although the error conditions possible on the client side are quite varied and application-dependent, a few of them might be directly related to how the server was started. Conditions other than those shown below should be documented with the respective client application.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

This is the generic "I couldn't find a server to talk to" failure. It looks like the above when TCP/IP communication is attempted. A common mistake is to forget to configure the server to allow TCP/IP connections.

Alternatively, you'll get this when attempting Unix-domain socket communication to a local server:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

The last line is useful in verifying that the client is trying to connect to the right place. If there is in fact no server running there, the kernel error message will typically be either `Connection refused` or `No such file or directory`, as illustrated. (It is important to realize that `Connection refused` in this context does *not* mean that the server got your connection request

Chapter 18. Server Configuration

There are many configuration parameters that affect the behavior of the database system. In the first section of this chapter we describe how to interact with configuration parameters. The subsequent sections discuss each parameter in detail.

18.1. Setting Parameters

18.1.1. Parameter Names and Values

All parameter names are case-insensitive. Every parameter takes a value of one of five types: boolean, string, integer, floating point, or enumerated (enum). The type determines the syntax for setting the parameter:

- *Boolean:* Values can be written as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (all case-insensitive) or any unambiguous prefix of one of these.
- *String:* In general, enclose the value in single quotes, doubling any single quotes within the value. Quotes can usually be omitted if the value is a simple number or identifier, however.
- *Numeric (integer and floating point):* A decimal point is permitted only for floating-point parameters. Do not use thousands separators. Quotes are not required.
- *Numeric with Unit:* Some numeric parameters have an implicit unit, because they describe quantities of memory or time. The unit might be kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. An unadorned numeric value for one of these settings will use the setting's default unit, which can be learned from `pg_settings.unit`. For convenience, settings can be given with a unit specified explicitly, for example `'120 ms'` for a time value, and they will be converted to whatever the parameter's actual unit is. Note that the value must be written as a string (with quotes) to use this feature. The unit name is case-sensitive, and there can be whitespace between the numeric value and the unit.
  - Valid memory units are `kB` (kilobytes), `MB` (megabytes), `GB` (gigabytes), and `TB` (terabytes). The multiplier for memory units is 1024, not 1000.
  - Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days).
- *Enumerated:* Enumerated-type parameters are written in the same way as string parameters, but are restricted to have one of a limited set of values. The values allowable for such a parameter can be found from `pg_settings.enumvals`. Enum parameter values are case-insensitive.

18.1.2. Parameter Interaction via the Configuration File

The most fundamental way to set these parameters is to edit the file `postgresql.conf`, which is normally kept in the data directory. A default copy is installed when the database cluster directory is initialized. An example of what this file might look like is:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

`listen_addresses` (`string`)

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications. The value takes the form of a comma-separated list of host names and/or numeric IP addresses. The special entry `*` corresponds to all available IP interfaces. The entry `0.0.0.0` allows listening for all IPv4 addresses and `::` allows listening for all IPv6 addresses. If the list is empty, the server does not listen on any IP interface at all, in which case only Unix-domain sockets can be used to connect to it. The default value is localhost, which allows only local TCP/IP "loopback" connections to be made. While client authentication (Chapter 19) allows fine-grained control over who can access the server, `listen_addresses` controls which interfaces accept connection attempts, which can help prevent repeated malicious connection requests on insecure network interfaces. This parameter can only be set at server start.

`port` (`integer`)

The TCP port the server listens on; 5432 by default. Note that the same port number is used for all IP addresses the server listens on. This parameter can only be set at server start.

`max_connections` (`integer`)

Determines the maximum number of concurrent connections to the database server. The default is typically 100 connections, but might be less if your kernel settings will not support it (as determined during initdb). This parameter can only be set at server start.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

`superuser_reserved_connections` (`integer`)

Determines the number of connection "slots" that are reserved for connections by PostgreSQL superusers. At most max_connections connections can ever be active simultaneously. Whenever the number of active concurrent connections is at least `max_connections` minus `superuser_reserved_connections`, new connections will be accepted only for superusers, and no new replication connections will be accepted.

The default value is three connections. The value must be less than the value of `max_connections`. This parameter can only be set at server start.

`unix_socket_directories` (`string`)

Specifies the directory of the Unix-domain socket(s) on which the server is to listen for connections from client applications. Multiple sockets can be created by listing multiple directories separated by commas. Whitespace between entries is ignored; surround a directory name with double quotes if you need to include whitespace or commas in the name. An empty value specifies not listening on any Unix-domain sockets, in which case only TCP/IP sockets can be used to connect to the server. The default value is normally `/tmp`, but that can be changed at build time. This parameter can only be set at server start.

In addition to the socket file itself, which is named `.s.PGSQL.`*`nnnn`* where *`nnnn`* is the server's port number, an ordinary file named `.s.PGSQL.`*`nnnn`*`.lock` will be created in each of the `unix_socket_directories` directories. Neither file should ever be removed manually.

This parameter is irrelevant on Windows, which does not have Unix-domain sockets.

`unix_socket_group` (`string`)

Sets the owning group of the Unix-domain socket(s). (The owning user of the sockets is always the user that starts the server.) In combination with the parameter `unix_socket_permissions` this can be used as an additional access control mechanism for Unix-domain connections. By default this is the empty string, which uses the default group of the server user. This parameter can only be set at server start.

Chapter 19. Client Authentication

When a client application connects to the database server, it specifies which PostgreSQL database user name it wants to connect as, much the same way one logs into a Unix computer as a particular user. Within the SQL environment the active database user name determines access privileges to database objects — see Chapter 20 for more information. Therefore, it is essential to restrict which database users can connect.

> **Note:** As explained in Chapter 20, PostgreSQL actually does privilege management in terms of "roles". In this chapter, we consistently use *database user* to mean "role with the `LOGIN` privilege".

Authentication is the process by which the database server establishes the identity of the client, and by extension determines whether the client application (or the user who runs the client application) is permitted to connect with the database user name that was requested.

PostgreSQL offers a number of different client authentication methods. The method used to authenticate a particular client connection can be selected on the basis of (client) host address, database, and user.

PostgreSQL database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections might have many database users who have no local operating system account, and in such cases there need be no connection between database user names and OS user names.

19.1. The `pg_hba.conf` File

Client authentication is controlled by a configuration file, which traditionally is named `pg_hba.conf` and is stored in the database cluster's data directory. (HBA stands for host-based authentication.) A default `pg_hba.conf` file is installed when the data directory is initialized by `initdb`. It is possible to place the authentication configuration file elsewhere, however; see the hba_file configuration parameter.

The general format of the `pg_hba.conf` file is a set of records, one per line. Blank lines are ignored, as is any text after the `#` comment character. Records cannot be continued across lines. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is double-quoted. Quoting one of the keywords in a database, user, or address field (e.g., `all` or `replication`) makes the word lose its special meaning, and just match a database, user, or host with that name.

Each record specifies a connection type, a client IP address range (if relevant for the connection type), a database name, a user name, and the authentication method to be used for connections matching these parameters. The first record with a matching connection type, client address, requested database, and user name is used to perform authentication. There is no "fall-through" or "backup": if one record is chosen and the authentication fails, subsequent records are not considered. If no record matches, access is denied.

A record can have one of the seven formats

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
```

```
hostssl     database  user  address  auth-method  [auth-options]
hostnossl   database  user  address  auth-method  [auth-options]
host        database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl     database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl   database  user  IP-address  IP-mask  auth-method  [auth-options]
```

The meaning of the fields is as follows:

`local`

This record matches connection attempts using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are disallowed.

`host`

This record matches connection attempts made using TCP/IP. `host` records match either SSL or non-SSL connection attempts.

> **Note:** Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the listen_addresses configuration parameter, since the default behavior is to listen for TCP/IP connections only on the local loopback address `localhost`.

`hostssl`

This record matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption.

To make use of this option the server must be built with SSL support. Furthermore, SSL must be enabled at server start time by setting the ssl configuration parameter (see Section 17.9 for more information).

`hostnossl`

This record type has the opposite behavior of `hostssl`; it only matches connection attempts made over TCP/IP that do not use SSL.

`database`

Specifies which database name(s) this record matches. The value `all` specifies that it matches all databases. The value `sameuser` specifies that the record matches if the requested database has the same name as the requested user. The value `samerole` specifies that the requested user must be a member of the role with the same name as the requested database. (`samegroup` is an obsolete but still accepted spelling of `samerole`.) Superusers are not considered to be members of a role for the purposes of `samerole` unless they are explicitly members of the role, directly or indirectly, and not just by virtue of being a superuser. The value `replication` specifies that the record matches if a replication connection is requested (note that replication connections do not specify any particular database). Otherwise, this is the name of a specific PostgreSQL database. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with `@`.

`user`

Specifies which database user name(s) this record matches. The value `all` specifies that it matches all users. Otherwise, this is either the name of a specific database user, or a group name preceded by +. (Recall that there is no real distinction between users and groups in PostgreSQL; a + mark really means “match any of the roles that are directly or indirectly members of this role”, while a name without a + mark matches only that specific role.) For this purpose, a

superuser is only considered to be a member of a role if they are explicitly a member of the role, directly or indirectly, and not just by virtue of being a superuser. Multiple user names can be supplied by separating them with commas. A separate file containing user names can be specified by preceding the file name with `@`.

address

Specifies the client machine address(es) that this record matches. This field can contain either a host name, an IP address range, or one of the special key words mentioned below.

An IP address range is specified using standard numeric notation for the range's starting address, then a slash (`/`) and a CIDR mask length. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this should be zero in the given IP address. There must not be any white space between the IP address, the `/`, and the CIDR mask length.

Typical examples of an IPv4 address range specified this way are `172.20.143.89/32` for a single host, or `172.20.143.0/24` for a small network, or `10.6.0.0/16` for a larger one. An IPv6 address range might look like `::1/128` for a single host (in this case the IPv6 loopback address) or `fe80::7a31:c1ff:0000:0000/96` for a small network. `0.0.0.0/0` represents all IPv4 addresses, and `::0/0` represents all IPv6 addresses. To specify a single host, use a mask length of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.

An entry given in IPv4 format will match only IPv4 connections, and an entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range. Note that entries in IPv6 format will be rejected if the system's C library does not have support for IPv6 addresses.

You can also write `all` to match any IP address, `samehost` to match any of the server's own IP addresses, or `samenet` to match any address in any subnet that the server is directly connected to.

If a host name is specified (anything that is not an IP address range or a special key word is treated as a host name), that name is compared with the result of a reverse name resolution of the client's IP address (e.g., reverse DNS lookup, if DNS is used). Host name comparisons are case insensitive. If there is a match, then a forward name resolution (e.g., forward DNS lookup) is performed on the host name to check whether any of the addresses it resolves to are equal to the client's IP address. If both directions match, then the entry is considered to match. (The host name that is used in `pg_hba.conf` should be the one that address-to-name resolution of the client's IP address returns, otherwise the line won't be matched. Some host name databases allow associating an IP address with multiple host names, but the operating system will only return one host name when asked to resolve an IP address.)

A host name specification that starts with a dot (`.`) matches a suffix of the actual host name. So `.example.com` would match `foo.example.com` (but not just `example.com`).

When host names are specified in `pg_hba.conf`, you should make sure that name resolution is reasonably fast. It can be of advantage to set up a local name resolution cache such as `nscd`. Also, you may wish to enable the configuration parameter `log_hostname` to see the client's host name instead of the IP address in the log.

This field only applies to `host`, `hostssl`, and `hostnossl` records.

> Users sometimes wonder why host names are handled in this seemingly complicated way, with two name resolutions including a reverse lookup of the client's IP address. This complicates use of the feature in case the client's reverse DNS entry is not set up or yields some undesirable host name. It is done primarily for efficiency: this way, a connection attempt requires at most two resolver lookups, one reverse and one forward. If there is a resolver problem with some address, it becomes only that client's problem. A hypothetical alternative implementation that only did forward lookups would have to resolve every host name mentioned in `pg_hba.conf` during every connection attempt. That could be quite slow if many names are listed. And if there is a resolver problem with one of the host names, it becomes everyone's problem.
>
> Also, a reverse lookup is necessary to implement the suffix matching feature, because the actual client host name needs to be known in order to match it against the pattern.
>
> Note that this behavior is consistent with other popular implementations of host name-based access control, such as the Apache HTTP Server and TCP Wrappers.

IP-address
IP-mask

These two fields can be used as an alternative to the *IP-address/mask-length* notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, `255.0.0.0` represents an IPv4 CIDR mask length of 8, and `255.255.255.255` represents a CIDR mask length of 32.

These fields only apply to `host`, `hostssl`, and `hostnossl` records.

auth-method

Specifies the authentication method to use when a connection matches this record. The possible choices are summarized here; details are in Section 19.3.

`trust`

Allow the connection unconditionally. This method allows anyone that can connect to the PostgreSQL database server to login as any PostgreSQL user they wish, without the need for a password or any other authentication. See Section 19.3.1 for details.

`reject`

Reject the connection unconditionally. This is useful for "filtering out" certain hosts from a group, for example a `reject` line could block a specific host from connecting, while a later line allows the remaining hosts in a specific network to connect.

`md5`

Require the client to supply a double-MD5-hashed password for authentication. See Section 19.3.2 for details.

`password`

Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. See Section 19.3.2 for details.

`gss`

Use GSSAPI to authenticate the user. This is only available for TCP/IP connections. See Section 19.3.3 for details.

`sspi`

Use SSPI to authenticate the user. This is only available on Windows. See Section 19.3.4 for details.

`ident`

Obtain the operating system user name of the client by contacting the ident server on the client and check if it matches the requested database user name. Ident authentication can only be used on TCP/IP connections. When specified for local connections, peer authentication will be used instead. See Section 19.3.5 for details.

`peer`

Obtain the client's operating system user name from the operating system and check if it matches the requested database user name. This is only available for local connections. See Section 19.3.6 for details.

`ldap`

Authenticate using an LDAP server. See Section 19.3.7 for details.

`radius`

Authenticate using a RADIUS server. See Section 19.3.8 for details.

`cert`

Authenticate using SSL client certificates. See Section 19.3.9 for details.

`pam`

Authenticate using the Pluggable Authentication Modules (PAM) service provided by the operating system. See Section 19.3.10 for details.

`auth-options`

After the *`auth-method`* field, there can be field(s) of the form *`name=value`* that specify options for the authentication method. Details about which options are available for which authentication methods appear below.

Files included by `@` constructs are read as lists of names, which can be separated by either whitespace or commas. Comments are introduced by `#`, just as in `pg_hba.conf`, and nested `@` constructs are allowed. Unless the file name following `@` is an absolute path, it is taken to be relative to the directory containing the referencing file.

Since the `pg_hba.conf` records are examined sequentially for each connection attempt, the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, one might wish to use `trust` authentication for local TCP/IP connections but require a password for remote TCP/IP connections. In this case a record specifying `trust` authentication for connections from 127.0.0.1 would appear before a record specifying password authentication for a wider range of allowed client IP addresses.

The `pg_hba.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the postmaster (using `pg_ctl reload` or `kill -HUP`) to make it re-read the file.

> **Tip:** To connect to a particular database, a user must not only pass the `pg_hba.conf` checks, but must have the `CONNECT` privilege for the database. If you wish to restrict which users can connect to which databases, it's usually easier to control this by granting/revoking `CONNECT` privilege than to put the rules in `pg_hba.conf` entries.

Some examples of `pg_hba.conf` entries are shown in Example 19-1. See the next section for details on the different authentication methods.

Example 19-1. Example `pg_hba.conf` Entries

```
# Allow any user on the local system to connect to any database with
# any database user name using Unix-domain sockets (the default for local
# connections).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# The same using local loopback TCP/IP connections.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# The same as the previous line, but using a separate netmask column
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# The same over IPv6.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# The same using a host name (would typically cover both IPv4 and IPv6).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# Allow any user from any host with IP address 192.168.93.x to connect
# to database "postgres" as the same user name that ident reports for
# the connection (typically the operating system user name).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident

# Allow any user from host 192.168.12.10 to connect to database
# "postgres" if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        md5

# Allow any user from hosts in the example.com domain to connect to
```