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, integer, floating point, string or enum.

- *Boolean*: Values can be written as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (all case-insensitive) or any unambiguous prefix of these.
- *String*: Enclose the value in single-quote. Values are case-insensitive. If multiple values are allowed separate them with commas.
- *Numeric* (integer and floating point): Do not use single-quotes (unless otherwise required) or thousand separators. Typically memory or time related - see comments in that section for detail.

  Zero (special case): If the supplied value is zero the effective value will be parameter specific; though it will be zero unless otherwise noted.
- *Numeric or String with Unit*: Memory & Time. Both of these have an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. An numeric value will use the default, which can be found by referencing `pg_settings.unit`. For convenience, a different unit can also be specified explicitly via a string value. It is case-sensitive and may include a space between the 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).
  - Parameters with default integer units cannot have values with smaller resolution. Any fractional part is silently ignored.

    Example: log_rotation_age; integer minutes and will not accept seconds. Setting it to '10s' will silently round down to zero and disable the feature.
- *"enum"*: These specified in the same way as string parameters, but are restricted to a limited set of values that can be queried from `pg_settings.enumvals`:

  ```
  SELECT name, setting, enumvals FROM pg_settings WHERE enumvals IS NOT NULL;
  ```

  Enum parameter values are case-insensitive.

18.1.2. Parameter Interaction via Configuration File

The primary 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 there 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
```

One parameter is specified per line. The equal sign between name and value is optional. Whitespace is insignificant and blank lines are ignored. Hash marks (`#`) designate the remainder of the line as a comment. Parameter values that are not simple identifiers or numbers must be single-quoted. To embed a single quote in a parameter value, write either two quotes (preferred) or backslash-quote.

Parameters set in this way provide the global default value for the cluster. The setting actually seen by the connecting user or issued statement will be this value unless it is overridden. The next sections describe ways in which the administrator or user can override these defaults.

The configuration file is reread whenever the main server process receives a SIGHUP signal; this is most easily done by running `pg_ctl reload` from the command-line or by calling the SQL function `pg_reload_conf()`. The main server process also propagates this signal to all currently running server processes so that existing sessions also get the new value. Alternatively, you can send the signal to a single server process directly. Some parameters can only be set at server start; any changes to their entries in the configuration file will be ignored until the server is restarted. Invalid parameter settings in the configuration file are likewise ignored (but logged) during SIGHUP processing.

18.1.3. Parameter Interaction via SQL

PostgreSQL provides three SQL commands to establish configuration defaults that override those configured globally. The evaluation of these defaults occurs at the beginning of a new session, upon the user issuing DISCARD, or if the server forces the session to reload it configuration after a SIGHUP

- The ALTER SYSTEM command provides an SQL-accessible means to change the global defaults. Since the server must be running to execute SQL the timing of when the actual value takes effect depends on the variable being changed - but in no case is the current session affected nor will any change take effect before the next configuration reload (SIGHUP) by the server.
- The ALTER DATABASE command allows the database administrator to override global settings on a per-database basis.
- The ALTER ROLE command allows the database administrator to override both global and per-database settings with user-specific values.

Once a client connects to the database PostgreSQL provides two additional SQL commands to interact with session-local system configuration. Both of these

commands have equivalent system administration functions.

- The SHOW command allows inspection of the current value of all parameters. The corresponding function is `current_setting(setting_name text)`.
- The SET command allows modification of the current value some parameters. The corresponding function is `set_config(setting_name, new_value, is_local)`.

Both *SELECT* and *UPDATE* can be issued against the virtual table `pg_settings` to view and affect the session-local configuration. Its definition can be found in Section 48.67.

- SELECT-ing against this relation is the equivalent of issuing SHOW but provides considerably more detail as well as allowing for joining against other relations and specifying filter criteria.
- UPDATE-ing against this relation, specifically the `setting` column is the equivalent of issuing SET though all values must be single-quoted.

  Note that the eqivalent of

  ```
  SET configuration_parameter TO DEFAULT;
  ```

  would be:

  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

18.1.4. Parameter Interaction via Shell

In addition to setting global defaults or attaching overrides at the database or role scope , you may choose to provide them to PostgreSQL via shell facilities. Both the server and libpq client library have defined ways to accept parameter values via the shell.

- On the *server*, command-line options can passed to the `postgres` command directly via the "-c" parameter.

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```

  Settings provided this way override those resolved globally (via postgresql.conf or ALTER SYSTEM) but are otherwise treated as being global for the purpose of database and role overriding.

  Typically, a production system will be adminstered via its `postgresql.conf` file. Use of this mechanism is suggested only for development and testing.

- On the *libpq-client*, command-line options are specified using the `PGOPTIONS` environment variable. Upon connecting to a server the contents of this variable are sent to the server as if they were being executed via a SQL SET at the beginning of the session.

  However, the format for `PGOPTIONS` is similar to that provided when launching `postgres` command. Specifically, the '-c' flag specification is part of the value.

```
env PGOPTIONS="-c geqo=off -c statement_timeout='5 min'" psql
```

Other clients and libraries may provide their own mechanisms, via the shell or otherwise, that allow the user to setup the session configuration without requiring the user to issue SQL commands. Please see their documentation for details.

18.1.5. Configuration File Includes

In addition to parameter settings, the `postgresql.conf` file can contain *include directives*, which specify another file to read and process as if it were inserted into the configuration file at this point. This feature allows a configuration file to be divided into physically separate parts. Include directives simply look like:

```
include 'filename'
```

If the file name is not an absolute path, it is taken as relative to the directory containing the referencing configuration file. Inclusions can be nested.

There is also an `include_if_exists` directive, which acts the same as the `include` directive, except for the behavior when the referenced file does not exist or cannot be read. A regular `include` will consider this an error condition, but `include_if_exists` merely logs a message and continues processing the referencing configuration file.

The `postgresql.conf` file can also contain `include_dir` directives, which specify an entire directory of configuration files to include. It is used similarly:

```
include_dir 'directory'
```

Non-absolute directory names follow the same rules as single file include directives: they are relative to the directory containing the referencing configuration file. Within that directory, only non-directory files whose names end with the suffix `.conf` will be included. File names that start with the `.` character are also excluded, to prevent mistakes as they are hidden on some platforms. Multiple files within an include directory are processed in file name order. The file names are ordered by C locale rules, i.e. numbers before letters, and uppercase letters before lowercase ones.

Include files or directories can be used to logically separate portions of the database configuration, rather than having a single large `postgresql.conf` file. Consider a company that has two database servers, each with a different amount of memory. There are likely elements of the configuration both will share, for things such as logging. But memory-related parameters on the server will vary between the two. And there might be server specific customizations, too. One way to manage this situation is to break the custom configuration changes for your site into three files. You could add this to the end of your `postgresql.conf` file to include them:

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

All systems would have the same `shared.conf`. Each server with a particular amount of memory could share the same `memory.conf`; you might have one for all servers with

8GB of RAM, another for those having 16GB. And finally `server.conf` could have truly server-specific configuration information in it.

Another possibility is to create a configuration file directory and put this information into files there. For example, a `conf.d` directory could be referenced at the end of`postgresql.conf`:

```
include_dir 'conf.d'
```

Then you could name the files in the `conf.d` directory like this:

```
00shared.conf
01memory.conf
02server.conf
```

This shows a clear order in which these files will be loaded. This is important because only the last setting encountered when the server is reading its configuration will be used. Something set in `conf.d/02server.conf` in this example would override a value set in `conf.d/01memory.conf`.

You might instead use this configuration directory approach while naming these files more descriptively:

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

This sort of arrangement gives a unique name for each configuration file variation. This can help eliminate ambiguity when several servers have their configurations all stored in one place, such as in a version control repository. (Storing database configuration files under version control is another good practice to consider).

Prev Server Configuration | Home Up | Next File Locations