Part III. Server Administration

This part covers topics that are of interest to a PostgreSQL administrator. This includes installation, configuration of the server, management of users and databases, and maintenance tasks. Anyone running PostgreSQL server, even for personal use, but especially in production, should be familiar with these topics.

The information attempts to be in the order in which a new user should read it. The chapters are self-contained and can be read individually as desired. The information is presented in a narrative form in topical units. Readers looking for a complete description of a command are encouraged to review the 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. 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 review the Part I and Part II parts for additional information.

Table of Contents

Chapter 16. Installation from Binaries

PostgreSQL is available in the form of binary packages for most common operating systems today. When available, this is the recommended way to install PostgreSQL for users of the system. Building from source (see Chapter 17) is only recommended for people developing PostgreSQL or extensions.

For an updated list of platforms providing binary packages, please visit the download section on the PostgreSQL website at https://www.postgresql.org/download/ and follow the instructions for the specific platform.

Chapter 17. 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 see Chapter 16 instead.

17.1. 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 described in Section 17.6 below.

The following software packages are required for building PostgreSQL:

- GNU make version 3.81 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
  ```

- Alternatively, PostgreSQL can be built using Meson[1]. This is currently experimental. If you choose to use Meson, then you don't need GNU make, but the other requirements below still apply.

  The minimum required version of Meson is 0.54.

- You need an ISO/ANSI C compiler (at least C99-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.

- Flex 2.5.35 or later and Bison 2.3 or later are required. Other lex and yacc programs cannot be used.

- Perl 5.14 or later is needed during the build process and to run some test suites. (This requirement is separate from the requirements for building PL/Perl; see below.)

- 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 if `--with-libedit-preferred` is used as an option to `configure`. If you are using a package-based Linux distribution, be aware that you need both the `readline` and `readline-devel` packages, if those are separate in your distribution.

- The zlib compression library is used by default. If you don't want to use it then you must specify the `--without-zlib` option to `configure`. Using this option disables support for compressed archives in pg_dump and pg_restore.

- The ICU library is used by default. If you don't want to use it then you must specify the `--without-icu` option to `configure`. Using this option disables support for ICU collation features (see Section 23.2).

[1] https://mesonbuild.com/

ICU support requires the ICU4C package to be installed. The minimum required version of ICU4C is currently 4.2.

By default, pkg-config will be used to find the required compilation options. This is supported for ICU4C version 4.6 and later. For older versions, or if pkg-config is not available, the variables `ICU_CFLAGS` and `ICU_LIBS` can be specified to `configure`, like in this example:

```
./configure ... ICU_CFLAGS='-I/some/where/include' ICU_LIBS='-L/
some/where/lib -licui18n -licuc -licudata'
```

(If ICU4C is in the default search path for the compiler, then you still need to specify nonempty strings in order to avoid use of pkg-config, for example, `ICU_CFLAGS=' '`.)

The following packages are optional. They are not required in the default configuration, but they are needed when certain build options are enabled, as explained below:

- To build the server programming language PL/Perl you need a full Perl installation, including the `libperl` library and the header files. The minimum required version is Perl 5.14. Since PL/Perl will be a shared library, the `libperl` library must be a shared library also on most platforms. This appears to be the default in recent Perl versions, but it was not in earlier versions, and in any case it is the choice of whomever installed Perl at your site. `configure` will fail if building PL/Perl is selected but it cannot find a shared `libperl`. In that case, you will have to rebuild and install Perl manually to be able to build PL/Perl. During the configuration process for Perl, request a shared library.

  If you intend to make more than incidental use of PL/Perl, you should ensure that the Perl installation was built with the `usemultiplicity` option enabled (`perl -V` will show whether this is the case).

- To build the PL/Python server programming language, you need a Python installation with the header files and the sysconfig module. The minimum required version is Python 3.2.

  Since PL/Python will be a shared library, the `libpython` library must be a shared library also on most platforms. This is not the case in a default Python installation built from source, but a shared library is available in many operating system distributions. `configure` will fail if building PL/Python is selected but it cannot find a shared `libpython`. That might mean that you either have to install additional packages or rebuild (part of) your Python installation to provide this shared library. When building from source, run Python's configure with the `--enable-shared` flag.

- To build the PL/Tcl procedural language, you of course need a Tcl installation. The minimum required version is Tcl 8.4.

- To enable Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English, you need an implementation of the Gettext API. Some operating systems have this built-in (e.g., Linux, NetBSD, Solaris), for other systems you can download an add-on package from https://www.gnu.org/software/gettext/. If you are using the Gettext implementation in the GNU C library, then you will additionally need the GNU Gettext package for some utility programs. For any of the other implementations you will not need it.

- You need OpenSSL, if you want to support encrypted client connections. OpenSSL is also required for random number generation on platforms that do not have `/dev/urandom` (except Windows). The minimum required version is 1.0.2.

- You need MIT Kerberos (for GSSAPI), OpenLDAP, and/or PAM, if you want to support authentication using those services.

- You need LZ4, if you want to support compression of data with that method; see default_toast_compression and wal_compression.

- You need Zstandard, if you want to support compression of data with that method; see wal_compression. The minimum required version is 1.4.0.
- To build the PostgreSQL documentation, there is a separate set of requirements; see Section J.2.

If you need to get a GNU package, you can find it at your local GNU mirror site (see https://www.gnu.org/prep/ftp for a list) or at ftp://ftp.gnu.org/gnu/.

17.2. Getting the Source

The PostgreSQL source code for released versions can be obtained from the download section of our website: https://www.postgresql.org/ftp/source/. Download the `postgresql-`*`version`*`.tar.gz` or `postgresql-`*`version`*`.tar.bz2` file you're interested in, then unpack it:

```
tar xf postgresql-version.tar.bz2
```

This will create a directory `postgresql-`*`version`* under the current directory with the PostgreSQL sources. Change into that directory for the rest of the installation procedure.

Alternatively, you can use the Git version control system; see Section I.1 for more information.

17.3. Building and Installation with Autoconf and Make

17.3.1. Short Version

```
./configure
make
su
make install
adduser postgres
mkdir -p /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/pg_ctl -D /usr/local/pgsql/data -l logfile
 start
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

The long version is the rest of this section.

17.3.2. Installation Procedure

1. **Configuration**

   The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter:

   ```
   ./configure
   ```

   This script will run a number of tests to determine values for various system dependent variables and detect any quirks of your operating system, and finally will create several files in the build tree to record what it found.

You can also run `configure` in a directory outside the source tree, and then build there, if you want to keep the build directory separate from the original source files. This procedure is called a *VPATH* build. Here's how:

```
mkdir build_dir
cd build_dir
/path/to/source/tree/configure [options go here]
make
```

The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under `/usr/local/pgsql` by default.

You can customize the build and installation process by supplying one or more command line options to `configure`. Typically you would customize the install location, or the set of optional features that are built. `configure` has a large number of options, which are described in Section 17.3.3.

Also, `configure` responds to certain environment variables, as described in Section 17.3.4. These provide additional ways to customize the configuration.

2. **Build**

   To start the build, type either of:

   ```
   make
   make all
   ```

   (Remember to use GNU make.) The build will take a few minutes depending on your hardware.

   If you want to build everything that can be built, including the documentation (HTML and man pages), and the additional modules (`contrib`), type instead:

   ```
   make world
   ```

   If you want to build everything that can be built, including the additional modules (`contrib`), but without the documentation, type instead:

   ```
   make world-bin
   ```

   If you want to invoke the build from another makefile rather than manually, you must unset `MAKELEVEL` or set it to zero, for instance like this:

   ```
   build-postgresql:
           $(MAKE) -C postgresql MAKELEVEL=0 all
   ```

   Failure to do that can lead to strange error messages, typically about missing header files.

3. **Regression Tests**

   If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type:

```
make check
```

(This won't work as root; do it as an unprivileged user.) See Chapter 31 for detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

4. **Installing the Files**

> **Note**
>
> If you are upgrading an existing system be sure to read Section 18.6, which has instructions about upgrading a cluster.

To install PostgreSQL enter:

```
make install
```

This will install files into the directories that were specified in Step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you can create the target directories in advance and arrange for appropriate permissions to be granted.

To install the documentation (HTML and man pages), enter:

```
make install-docs
```

If you built the world above, type instead:

```
make install-world
```

This also installs the documentation.

If you built the world without the documentation above, type instead:

```
make install-world-bin
```

You can use `make install-strip` instead of `make install` to strip the executable files and libraries as they are installed. This will save some space. If you built with debugging support, stripping will effectively remove the debugging support, so it should only be done if debugging is no longer needed. `install-strip` tries to do a reasonable job saving space, but it does not have perfect knowledge of how to strip every unneeded byte from an executable file, so if you want to save all the disk space you possibly can, you will have to do manual work.

The standard installation provides all the header files needed for client application development as well as for server-side program development, such as custom functions or data types written in C.

Client-only installation: If you want to install only the client applications and interface libraries, then you can use these commands:

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
```

```
make -C doc install
```

`src/bin` has a few binaries for server-only use, but they are small.

Uninstallation: To undo the installation use the command `make uninstall`. However, this will not remove any created directories.

Cleaning: After the installation you can free disk space by removing the built files from the source tree with the command `make clean`. This will preserve the files made by the `configure` program, so that you can rebuild everything with `make` later on. To reset the source tree to the state in which it was distributed, use `make distclean`. If you are going to build for several platforms within the same source tree you must do this and re-configure for each platform. (Alternatively, use a separate build tree for each platform, so that the source tree remains unmodified.)

If you perform a build and then discover that your `configure` options were wrong, or if you change anything that `configure` investigates (for example, software upgrades), then it's a good idea to do `make distclean` before reconfiguring and rebuilding. Without this, your changes in configuration choices might not propagate everywhere they need to.

17.3.3. `configure` Options

`configure`'s command line options are explained below. This list is not exhaustive (use `./configure --help` to get one that is). The options not covered here are meant for advanced use-cases such as cross-compilation, and are documented in the standard Autoconf documentation.

17.3.3.1. Installation Locations

These options control where `make install` will put the files. The `--prefix` option is sufficient for most cases. If you have special needs, you can customize the installation subdirectories with the other options described in this section. Beware however that changing the relative locations of the different subdirectories may render the installation non-relocatable, meaning you won't be able to move it after installation. (The `man` and `doc` locations are not affected by this restriction.) For relocatable installs, you might want to use the `--disable-rpath` option described later.

`--prefix=`*`PREFIX`*

Install all files under the directory *`PREFIX`* instead of `/usr/local/pgsql`. The actual files will be installed into various subdirectories; no files will ever be installed directly into the *`PREFIX`* directory.

`--exec-prefix=`*`EXEC-PREFIX`*

You can install architecture-dependent files under a different prefix, *`EXEC-PREFIX`*, than what *`PREFIX`* was set to. This can be useful to share architecture-independent files between hosts. If you omit this, then *`EXEC-PREFIX`* is set equal to *`PREFIX`* and both architecture-dependent and independent files will be installed under the same tree, which is probably what you want.

`--bindir=`*`DIRECTORY`*

Specifies the directory for executable programs. The default is *`EXEC-PREFIX`*`/bin`, which normally means `/usr/local/pgsql/bin`.

`--sysconfdir=`*`DIRECTORY`*

Sets the directory for various configuration files, *`PREFIX`*`/etc` by default.

`--libdir=`*`DIRECTORY`*

Sets the location to install libraries and dynamically loadable modules. The default is *`EXEC-PREFIX`*`/lib`.

`--includedir=`*`DIRECTORY`*

Sets the directory for installing C and C++ header files. The default is *`PREFIX`*`/include`.

`--datarootdir=`*`DIRECTORY`*

Sets the root directory for various types of read-only data files. This only sets the default for some of the following options. The default is *`PREFIX`*`/share`.

`--datadir=`*`DIRECTORY`*

Sets the directory for read-only data files used by the installed programs. The default is *`DATAROOTDIR`*. Note that this has nothing to do with where your database files will be placed.

`--localedir=`*`DIRECTORY`*

Sets the directory for installing locale data, in particular message translation catalog files. The default is *`DATAROOTDIR`*`/locale`.

`--mandir=`*`DIRECTORY`*

The man pages that come with PostgreSQL will be installed under this directory, in their respective `man`*`x`* subdirectories. The default is *`DATAROOTDIR`*`/man`.

`--docdir=`*`DIRECTORY`*

Sets the root directory for installing documentation files, except “man” pages. This only sets the default for the following options. The default value for this option is *`DATAROOTDIR`*`/doc/postgresql`.

`--htmldir=`*`DIRECTORY`*

The HTML-formatted documentation for PostgreSQL will be installed under this directory. The default is *`DATAROOTDIR`*.

> **Note**
>
> Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string “`/postgresql`” is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string “`postgres`” or “`pgsql`”. For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. The public C header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into private directories under `includedir`. See the documentation of each interface for information about how to access its header files. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

17.3.3.2. PostgreSQL Features

The options described in this section enable building of various PostgreSQL features that are not built by default. Most of these are non-default only because they require additional software, as described in Section 17.1.

`--enable-nls[=`*`LANGUAGES`*`]`

Enables Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English. *`LANGUAGES`* is an optional space-separated list of codes of the

languages that you want supported, for example `--enable-nls='de fr'`. (The intersection between your list and the set of actually provided translations will be computed automatically.) If you do not specify a list, then all available translations are installed.

To use this option, you will need an implementation of the Gettext API.

`--with-perl`

Build the PL/Perl server-side language.

`--with-python`

Build the PL/Python server-side language.

`--with-tcl`

Build the PL/Tcl server-side language.

`--with-tclconfig=`*`DIRECTORY`*

Tcl installs the file `tclConfig.sh`, which contains configuration information needed to build modules interfacing to Tcl. This file is normally found automatically at a well-known location, but if you want to use a different version of Tcl you can specify the directory in which to look for `tclConfig.sh`.

`--with-llvm`

Build with support for LLVM based JIT compilation (see Chapter 30). This requires the LLVM library to be installed. The minimum required version of LLVM is currently 10.

`llvm-config` will be used to find the required compilation options. `llvm-config` will be searched for in your `PATH`. If that would not yield the desired program, use `LLVM_CONFIG` to specify a path to the correct `llvm-config`. For example

```
./configure ... --with-llvm LLVM_CONFIG='/path/to/llvm/bin/llvm-config'
```

LLVM support requires a compatible `clang` compiler (specified, if necessary, using the `CLANG` environment variable), and a working C++ compiler (specified, if necessary, using the `CXX` environment variable).

`--with-lz4`

Build with LZ4 compression support.

`--with-zstd`

Build with Zstandard compression support.

`--with-ssl=`*`LIBRARY`*

Build with support for SSL (encrypted) connections. The only *`LIBRARY`* supported is `openssl`. This requires the OpenSSL package to be installed. `configure` will check for the required header files and libraries to make sure that your OpenSSL installation is sufficient before proceeding.

`--with-openssl`

Obsolete equivalent of `--with-ssl=openssl`.

`--with-gssapi`

Build with support for GSSAPI authentication. MIT Kerberos is required to be installed for GSSAPI. On many systems, the GSSAPI system (a part of the MIT Kerberos installation) is not

installed in a location that is searched by default (e.g., `/usr/include`, `/usr/lib`), so you must use the options `--with-includes` and `--with-libraries` in addition to this option. `configure` will check for the required header files and libraries to make sure that your GSSAPI installation is sufficient before proceeding.

`--with-ldap`

Build with LDAP support for authentication and connection parameter lookup (see Section 32.18 and Section 20.10 for more information). On Unix, this requires the OpenLDAP package to be installed. On Windows, the default WinLDAP library is used. `configure` will check for the required header files and libraries to make sure that your OpenLDAP installation is sufficient before proceeding.

`--with-pam`

Build with PAM (Pluggable Authentication Modules) support.

`--with-bsd-auth`

Build with BSD Authentication support. (The BSD Authentication framework is currently only available on OpenBSD.)

`--with-systemd`

Build with support for systemd service notifications. This improves integration if the server is started under systemd but has no impact otherwise; see Section 18.3 for more information. libsystemd and the associated header files need to be installed to use this option.

`--with-bonjour`

Build with support for Bonjour automatic service discovery. This requires Bonjour support in your operating system. Recommended on macOS.

`--with-uuid=`*`LIBRARY`*

Build the uuid-ossp module (which provides functions to generate UUIDs), using the specified UUID library. *`LIBRARY`* must be one of:

- `bsd` to use the UUID functions found in FreeBSD and some other BSD-derived systems
- `e2fs` to use the UUID library created by the `e2fsprogs` project; this library is present in most Linux systems and in macOS, and can be obtained for other platforms as well
- `ossp` to use the OSSP UUID library[2]

`--with-ossp-uuid`

Obsolete equivalent of `--with-uuid=ossp`.

`--with-libxml`

Build with libxml2, enabling SQL/XML support. Libxml2 version 2.6.23 or later is required for this feature.

To detect the required compiler and linker options, PostgreSQL will query `pkg-config`, if that is installed and knows about libxml2. Otherwise the program `xml2-config`, which is installed by libxml2, will be used if it is found. Use of `pkg-config` is preferred, because it can deal with multi-architecture installations better.

To use a libxml2 installation that is in an unusual location, you can set `pkg-config`-related environment variables (see its documentation), or set the environment variable `XML2_CONFIG`

[2] http://www.ossp.org/pkg/lib/uuid/

to point to the `xml2-config` program belonging to the libxml2 installation, or set the variables `XML2_CFLAGS` and `XML2_LIBS`. (If `pkg-config` is installed, then to override its idea of where libxml2 is you must either set `XML2_CONFIG` or set both `XML2_CFLAGS` and `XML2_LIBS` to nonempty strings.)

`--with-libxslt`

Build with libxslt, enabling the xml2 module to perform XSL transformations of XML. `--with-libxml` must be specified as well.

`--with-selinux`

Build with SElinux support, enabling the sepgsql extension.

17.3.3.3. Anti-Features

The options described in this section allow disabling certain PostgreSQL features that are built by default, but which might need to be turned off if the required software or system features are not available. Using these options is not recommended unless really necessary.

`--without-icu`

Build without support for the ICU library, disabling the use of ICU collation features (see Section 23.2).

`--without-readline`

Prevents use of the Readline library (and libedit as well). This option disables command-line editing and history in psql.

`--with-libedit-preferred`

Favors the use of the BSD-licensed libedit library rather than GPL-licensed Readline. This option is significant only if you have both libraries installed; the default in that case is to use Readline.

`--without-zlib`

Prevents use of the Zlib library. This disables support for compressed archives in pg_dump and pg_restore.

`--disable-spinlocks`

Allow the build to succeed even if PostgreSQL has no CPU spinlock support for the platform. The lack of spinlock support will result in very poor performance; therefore, this option should only be used if the build aborts and informs you that the platform lacks spinlock support. If this option is required to build PostgreSQL on your platform, please report the problem to the PostgreSQL developers.

`--disable-atomics`

Disable use of CPU atomic operations. This option does nothing on platforms that lack such operations. On platforms that do have them, this will result in poor performance. This option is only useful for debugging or making performance comparisons.

17.3.3.4. Build Process Details

`--with-includes=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `--with-libraries` option.

Example: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`DIRECTORIES`*

`DIRECTORIES` is a colon-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `--with-includes` option) if you have packages installed in non-standard locations.

Example: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--with-system-tzdata=`*`DIRECTORY`*

PostgreSQL includes its own time zone database, which it requires for date and time operations. This time zone database is in fact compatible with the IANA time zone database provided by many operating systems such as FreeBSD, Linux, and Solaris, so it would be redundant to install it again. When this option is used, the system-supplied time zone database in *`DIRECTORY`* is used instead of the one included in the PostgreSQL source distribution. *`DIRECTORY`* must be specified as an absolute path. `/usr/share/zoneinfo` is a likely directory on some operating systems. Note that the installation routine will not detect mismatching or erroneous time zone data. If you use this option, you are advised to run the regression tests to verify that the time zone data you have pointed to works correctly with PostgreSQL.

This option is mainly aimed at binary package distributors who know their target operating system well. The main advantage of using this option is that the PostgreSQL package won't need to be upgraded whenever any of the many local daylight-saving time rules change. Another advantage is that PostgreSQL can be cross-compiled more straightforwardly if the time zone database files do not need to be built during the installation.

`--with-extra-version=`*`STRING`*

Append *`STRING`* to the PostgreSQL version number. You can use this, for example, to mark binaries built from unreleased Git snapshots or containing custom patches with an extra version string, such as a `git describe` identifier or a distribution package release number.

`--disable-rpath`

Do not mark PostgreSQL's executables to indicate that they should search for shared libraries in the installation's library directory (see `--libdir`). On most platforms, this marking uses an absolute path to the library directory, so that it will be unhelpful if you relocate the installation later. However, you will then need to provide some other way for the executables to find the shared libraries. Typically this requires configuring the operating system's dynamic linker to search the library directory; see Section 17.5.1 for more detail.

17.3.3.5. Miscellaneous

It's fairly common, particularly for test builds, to adjust the default port number with `--with-pg-port`. The other options in this section are recommended only for advanced users.

`--with-pgport=`*`NUMBER`*

Set *`NUMBER`* as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.

`--with-krb-srvnam=`*`NAME`*

The default name of the Kerberos service principal used by GSSAPI. `postgres` is the default. There's usually no reason to change this unless you are building for a Windows environment, in which case it must be set to upper case `POSTGRES`.

`--with-segsize=`*`SEGSIZE`*

Set the *segment size*, in gigabytes. Large tables are divided into multiple operating-system files, each of size equal to the segment size. This avoids problems with file size limits that exist on many platforms. The default segment size, 1 gigabyte, is safe on all supported platforms. If your operating system has "largefile" support (which most do, nowadays), you can use a larger segment size. This can be helpful to reduce the number of file descriptors consumed when working with very large tables. But be careful not to select a value larger than is supported by your platform and the file systems you intend to use. Other tools you might wish to use, such as tar, could also set limits on the usable file size. It is recommended, though not absolutely required, that this value be a power of 2. Note that changing this value breaks on-disk database compatibility, meaning you cannot use `pg_upgrade` to upgrade to a build with a different segment size.

`--with-blocksize=`*`BLOCKSIZE`*

Set the *block size*, in kilobytes. This is the unit of storage and I/O within tables. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 32 (kilobytes). Note that changing this value breaks on-disk database compatibility, meaning you cannot use `pg_upgrade` to upgrade to a build with a different block size.

`--with-wal-blocksize=`*`BLOCKSIZE`*

Set the *WAL block size*, in kilobytes. This is the unit of storage and I/O within the WAL log. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 64 (kilobytes). Note that changing this value breaks on-disk database compatibility, meaning you cannot use `pg_upgrade` to upgrade to a build with a different WAL block size.

17.3.3.6. Developer Options

Most of the options in this section are only of interest for developing or debugging PostgreSQL. They are not recommended for production builds, except for `--enable-debug`, which can be useful to enable detailed bug reports in the unlucky event that you encounter a bug. On platforms supporting DTrace, `--enable-dtrace` may also be reasonable to use in production.

When building an installation that will be used to develop code inside the server, it is recommended to use at least the options `--enable-debug` and `--enable-cassert`.

`--enable-debug`

Compiles all programs and libraries with debugging symbols. This means that you can run the programs in a debugger to analyze problems. This enlarges the size of the installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that might arise. Currently, this option is recommended for production installations only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

`--enable-cassert`

Enables *assertion* checks in the server, which test for many "cannot happen" conditions. This is invaluable for code development purposes, but the tests can slow down the server significantly. Also, having the tests turned on won't necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. This option is not recommended for production use, but you should have it on for development work or when running a beta version.

`--enable-tap-tests`

Enable tests using the Perl TAP tools. This requires a Perl installation and the Perl module `IPC::Run`. See Section 31.4 for more information.

`--enable-depend`

Enables automatic dependency tracking. With this option, the makefiles are set up so that all affected object files will be rebuilt when any header file is changed. This is useful if you are doing development work, but is just wasted overhead if you intend only to compile once and install. At present, this option only works with GCC.

`--enable-coverage`

If using GCC, all programs and libraries are compiled with code coverage testing instrumentation. When run, they generate files in the build directory with code coverage metrics. See Section 31.5 for more information. This option is for use only with GCC and when doing development work.

`--enable-profiling`

If using GCC, all programs and libraries are compiled so they can be profiled. On backend exit, a subdirectory will be created that contains the `gmon.out` file containing profile data. This option is for use only with GCC and when doing development work.

`--enable-dtrace`

Compiles PostgreSQL with support for the dynamic tracing tool DTrace. See Section 27.5 for more information.

To point to the `dtrace` program, the environment variable `DTRACE` can be set. This will often be necessary because `dtrace` is typically installed under `/usr/sbin`, which might not be in your `PATH`.

Extra command-line options for the `dtrace` program can be specified in the environment variable `DTRACEFLAGS`. On Solaris, to include DTrace support in a 64-bit binary, you must specify `DTRACEFLAGS="-64"`. For example, using the GCC compiler:

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

Using Sun's compiler:

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --
enable-dtrace DTRACEFLAGS='-64' ...
```

`--enable-injection-points`

Compiles PostgreSQL with support for injection points in the server. Injection points allow to run user-defined code from within the server in pre-defined code paths. This helps in testing and in the investigation of concurrency scenarios in a controlled fashion. This option is disabled by default. See Section 36.10.13 for more details. This option is intended to be used only by developers for testing.

`--with-segsize-blocks=SEGSIZE_BLOCKS`

Specify the relation segment size in blocks. If both `--with-segsize` and this option are specified, this option wins. This option is only for developers, to test segment related code.

17.3.4. `configure` Environment Variables

In addition to the ordinary command-line options described above, `configure` responds to a number of environment variables. You can specify environment variables on the `configure` command line, for example:

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

In this usage an environment variable is little different from a command-line option. You can also set such variables beforehand:

```
export CC=/opt/bin/gcc
export CFLAGS='-O2 -pipe'
./configure
```

This usage can be convenient because many programs' configuration scripts respond to these variables in similar ways.

The most commonly used of these environment variables are `CC` and `CFLAGS`. If you prefer a C compiler different from the one `configure` picks, you can set the variable `CC` to the program of your choice. By default, `configure` will pick `gcc` if available, else the platform's default (usually `cc`). Similarly, you can override the default compiler flags if needed with the `CFLAGS` variable.

Here is a list of the significant variables that can be set in this manner:

`BISON`

Bison program

`CC`

C compiler

`CFLAGS`

options to pass to the C compiler

`CLANG`

path to `clang` program used to process source code for inlining when compiling with `--with-llvm`

`CPP`

C preprocessor

`CPPFLAGS`

options to pass to the C preprocessor

`CXX`

C++ compiler

`CXXFLAGS`

options to pass to the C++ compiler

`DTRACE`

location of the `dtrace` program

`DTRACEFLAGS`

options to pass to the `dtrace` program

`FLEX`

Flex program

`LDFLAGS`

options to use when linking either executables or shared libraries

`LDFLAGS_EX`

additional options for linking executables only

`LDFLAGS_SL`

additional options for linking shared libraries only

`LLVM_CONFIG`

`llvm-config` program used to locate the LLVM installation

`MSGFMT`

`msgfmt` program for native language support

`PERL`

Perl interpreter program. This will be used to determine the dependencies for building PL/Perl. The default is `perl`.

`PYTHON`

Python interpreter program. This will be used to determine the dependencies for building PL/Python. If this is not set, the following are probed in this order: `python3 python`.

`TCLSH`

Tcl interpreter program. This will be used to determine the dependencies for building PL/Tcl. If this is not set, the following are probed in this order: `tclsh tcl tclsh8.6 tclsh86 tclsh8.5 tclsh85 tclsh8.4 tclsh84`.

`XML2_CONFIG`

`xml2-config` program used to locate the libxml2 installation

Sometimes it is useful to add compiler flags after-the-fact to the set that were chosen by `configure`. An important example is that gcc's `-Werror` option cannot be included in the `CFLAGS` passed to `configure`, because it will break many of `configure`'s built-in tests. To add such flags, include them in the `COPT` environment variable while running `make`. The contents of `COPT` are added to both the `CFLAGS` and `LDFLAGS` options set up by `configure`. For example, you could do

```
make COPT='-Werror'
```

or

```
export COPT='-Werror'
make
```

> **Note**
>
> If using GCC, it is best to build with an optimization level of at least `-O1`, because using no optimization (`-O0`) disables some important compiler warnings (such as the use of uninitialized variables). However, non-zero optimization levels can complicate debugging because stepping through compiled code will usually not match up one-to-one with source code lines. If you get confused while trying to debug optimized code, recompile the specific files of in-

terest with `-O0`. An easy way to do this is by passing an option to make: `make PROFILE=-O0 file.o`.

The `COPT` and `PROFILE` environment variables are actually handled identically by the PostgreSQL makefiles. Which to use is a matter of preference, but a common habit among developers is to use `PROFILE` for one-time flag adjustments, while `COPT` might be kept set all the time.

17.4. Building and Installation with Meson

17.4.1. Short Version

```
meson setup build --prefix=/usr/local/pgsql
cd build
ninja
su
ninja install
adduser postgres
mkdir -p /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/pg_ctl -D /usr/local/pgsql/data -l logfile
 start
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

The long version is the rest of this section.

17.4.2. Installation Procedure

1. **Configuration**

   The first step of the installation procedure is to configure the build tree for your system and choose the options you would like. To create and configure the build directory, you can start with the `meson setup` command.

   `meson setup build`

   The setup command takes a `builddir` and a `srcdir` argument. If no `srcdir` is given, Meson will deduce the `srcdir` based on the current directory and the location of `meson.build`. The `builddir` is mandatory.

   Running `meson setup` loads the build configuration file and sets up the build directory. Additionally, you can also pass several build options to Meson. Some commonly used options are mentioned in the subsequent sections. For example:

   ```
   # configure with a different installation prefix
   meson setup build --prefix=/home/user/pg-install

   # configure to generate a debug build
   meson setup build --buildtype=debug

   # configure to build with OpenSSL support
   ```

```
meson setup build -Dssl=openssl
```

Setting up the build directory is a one-time step. To reconfigure before a new build, you can simply use the `meson configure` command

```
meson configure -Dcassert=true
```

`meson configure`'s commonly used command-line options are explained in Section 17.4.3.

2. **Build**

   By default, Meson uses the Ninja[3] build tool. To build PostgreSQL from source using Meson, you can simply use the `ninja` command in the build directory.

   ```
   ninja
   ```

   Ninja will automatically detect the number of CPUs in your computer and parallelize itself accordingly. You can override the number of parallel processes used with the command line argument `-j`.

   It should be noted that after the initial configure step, `ninja` is the only command you ever need to type to compile. No matter how you alter your source tree (short of moving it to a completely new location), Meson will detect the changes and regenerate itself accordingly. This is especially handy if you have multiple build directories. Often one of them is used for development (the "debug" build) and others only every now and then (such as a "static analysis" build). Any configuration can be built just by cd'ing to the corresponding directory and running Ninja.

   If you'd like to build with a backend other than ninja, you can use configure with the `--backend` option to select the one you want to use and then build using `meson compile`. To learn more about these backends and other arguments you can provide to ninja, you can refer to the Meson documentation[4].

3. **Regression Tests**

   If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type:

   ```
   meson test
   ```

   (This won't work as root; do it as an unprivileged user.) See Chapter 31 for detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

   To run pg_regress and pg_isolation_regress tests against a running postgres instance, specify **`--setup running`** as an argument to **`meson test`**.

4. **Installing the Files**

> **Note**
>
> If you are upgrading an existing system be sure to read Section 18.6, which has instructions about upgrading a cluster.

[3] https://ninja-build.org/
[4] https://mesonbuild.com/Running-Meson.html#building-from-the-source

Once PostgreSQL is built, you can install it by simply running the `ninja install` command.

```
ninja install
```

This will install files into the directories that were specified in Step 1. Make sure that you have appropriate permissions to write into that area. You might need to do this step as root. Alternatively, you can create the target directories in advance and arrange for appropriate permissions to be granted. The standard installation provides all the header files needed for client application development as well as for server-side program development, such as custom functions or data types written in C.

`ninja install` should work for most cases, but if you'd like to use more options (such as `--quiet` to suppress extra output), you could also use `meson install` instead. You can learn more about meson install[5] and its options in the Meson documentation.

Uninstallation: To undo the installation, you can use the `ninja uninstall` command.

Cleaning: After the installation, you can free disk space by removing the built files from the source tree with the `ninja clean` command.

17.4.3. `meson setup` Options

`meson setup`'s command-line options are explained below. This list is not exhaustive (use `meson configure --help` to get one that is). The options not covered here are meant for advanced use-cases, and are documented in the standard Meson documentation[6]. These arguments can be used with `meson setup` as well.

17.4.3.1. Installation Locations

These options control where `ninja install` (or `meson install`) will put the files. The `--prefix` option (example Section 17.4.1) is sufficient for most cases. If you have special needs, you can customize the installation subdirectories with the other options described in this section. Beware however that changing the relative locations of the different subdirectories may render the installation non-relocatable, meaning you won't be able to move it after installation. (The `man` and `doc` locations are not affected by this restriction.) For relocatable installs, you might want to use the `-Dpath=false` option described later.

`--prefix=`*`PREFIX`*

Install all files under the directory *`PREFIX`* instead of `/usr/local/pgsql` (on Unix based systems) or *`current drive letter`*`:/usr/local/pgsql` (on Windows). The actual files will be installed into various subdirectories; no files will ever be installed directly into the *`PREFIX`* directory.

`--bindir=`*`DIRECTORY`*

Specifies the directory for executable programs. The default is *`PREFIX`*`/bin`.

`--sysconfdir=`*`DIRECTORY`*

Sets the directory for various configuration files, *`PREFIX`*`/etc` by default.

`--libdir=`*`DIRECTORY`*

Sets the location to install libraries and dynamically loadable modules. The default is *`PREFIX`*`/lib`.

[5] https://mesonbuild.com/Commands.html#install
[6] https://mesonbuild.com/Commands.html#configure

`--includedir=`*`DIRECTORY`*

Sets the directory for installing C and C++ header files. The default is *`PREFIX`*`/include`.

`--datadir=`*`DIRECTORY`*

Sets the directory for read-only data files used by the installed programs. The default is *`PREFIX`*`/share`. Note that this has nothing to do with where your database files will be placed.

`--localedir=`*`DIRECTORY`*

Sets the directory for installing locale data, in particular message translation catalog files. The default is *`DATADIR`*`/locale`.

`--mandir=`*`DIRECTORY`*

The man pages that come with PostgreSQL will be installed under this directory, in their respective `man`*`x`* subdirectories. The default is *`DATADIR`*`/man`.

> **Note**
>
> Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string "`/postgresql`" is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string "`postgres`" or "`pgsql`". For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. The public C header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into private directories under `includedir`. See the documentation of each interface for information about how to access its header files. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

17.4.3.2. PostgreSQL Features

The options described in this section enable building of various optional PostgreSQL features. Most of these require additional software, as described in Section 17.1, and will be automatically enabled if the required software is found. You can change this behavior by manually setting these features to `enabled` to require them or `disabled` to not build with them.

To specify PostgreSQL-specific options, the name of the option must be prefixed by `-D`.

`-Dnls={ auto | enabled | disabled }`

Enables or disables Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English. Defaults to auto and will be enabled automatically if an implementation of the Gettext API is found.

`-Dplperl={ auto | enabled | disabled }`

Build the PL/Perl server-side language. Defaults to auto.

`-Dplpython={ auto | enabled | disabled }`

Build the PL/Python server-side language. Defaults to auto.

`-Dpltcl={ auto | enabled | disabled }`

Build the PL/Tcl server-side language. Defaults to auto.

`-Dtcl_version=`*`TCL_VERSION`*

Specifies the Tcl version to use when building PL/Tcl.

`-Dicu={ auto | enabled | disabled }`

Build with support for the ICU library, enabling use of ICU collation features (see Section 23.2). Defaults to auto and requires the ICU4C package to be installed. The minimum required version of ICU4C is currently 4.2.

`-Dllvm={ auto | enabled | disabled }`

Build with support for LLVM based JIT compilation (see Chapter 30). This requires the LLVM library to be installed. The minimum required version of LLVM is currently 10. Disabled by default.

`llvm-config` will be used to find the required compilation options. `llvm-config`, and then `llvm-config-$version` for all supported versions, will be searched for in your `PATH`. If that would not yield the desired program, use `LLVM_CONFIG` to specify a path to the correct `llvm-config`.

`-Dlz4={ auto | enabled | disabled }`

Build with LZ4 compression support. Defaults to auto.

`-Dzstd={ auto | enabled | disabled }`

Build with Zstandard compression support. Defaults to auto.

`-Dssl={ auto | `*`LIBRARY`*` }`

Build with support for SSL (encrypted) connections. The only *`LIBRARY`* supported is `openssl`. This requires the OpenSSL package to be installed. Building with this will check for the required header files and libraries to make sure that your OpenSSL installation is sufficient before proceeding. The default for this option is auto.

`-Dgssapi={ auto | enabled | disabled }`

Build with support for GSSAPI authentication. MIT Kerberos is required to be installed for GSSAPI. On many systems, the GSSAPI system (a part of the MIT Kerberos installation) is not installed in a location that is searched by default (e.g., `/usr/include`, `/usr/lib`). In those cases, PostgreSQL will query `pkg-config` to detect the required compiler and linker options. Defaults to auto. `meson configure` will check for the required header files and libraries to make sure that your GSSAPI installation is sufficient before proceeding.

`-Dldap={ auto | enabled | disabled }`

Build with LDAP support for authentication and connection parameter lookup (see Section 32.18 and Section 20.10 for more information). On Unix, this requires the OpenLDAP package to be installed. On Windows, the default WinLDAP library is used. Defaults to auto. `meson configure` will check for the required header files and libraries to make sure that your OpenLDAP installation is sufficient before proceeding.

`-Dpam={ auto | enabled | disabled }`

Build with PAM (Pluggable Authentication Modules) support. Defaults to auto.

`-Dbsd_auth={ auto | enabled | disabled }`

Build with BSD Authentication support. (The BSD Authentication framework is currently only available on OpenBSD.) Defaults to auto.

`-Dsystemd={ auto | enabled | disabled }`

Build with support for systemd service notifications. This improves integration if the server is started under systemd but has no impact otherwise; see Section 18.3 for more information. Defaults to auto. libsystemd and the associated header files need to be installed to use this option.

`-Dbonjour={ auto | enabled | disabled }`

Build with support for Bonjour automatic service discovery. Defaults to auto and requires Bonjour support in your operating system. Recommended on macOS.

`-Duuid=`*`LIBRARY`*

Build the uuid-ossp module (which provides functions to generate UUIDs), using the specified UUID library. *`LIBRARY`* must be one of:

- `none` to not build the uuid module. This is the default.
- `bsd` to use the UUID functions found in FreeBSD, and some other BSD-derived systems
- `e2fs` to use the UUID library created by the `e2fsprogs` project; this library is present in most Linux systems and in macOS, and can be obtained for other platforms as well
- `ossp` to use the OSSP UUID library[7]

`-Dlibxml={ auto | enabled | disabled }`

Build with libxml2, enabling SQL/XML support. Defaults to auto. Libxml2 version 2.6.23 or later is required for this feature.

To use a libxml2 installation that is in an unusual location, you can set `pkg-config`-related environment variables (see its documentation).

`-Dlibxslt={ auto | enabled | disabled }`

Build with libxslt, enabling the xml2 module to perform XSL transformations of XML. `-Dlibxml` must be specified as well. Defaults to auto.

`-Dselinux={ auto | enabled | disabled }`

Build with SElinux support, enabling the sepgsql extension. Defaults to auto.

17.4.3.3. Anti-Features

`-Dreadline={ auto | enabled | disabled }`

Allows use of the Readline library (and libedit as well). This option defaults to auto and enables command-line editing and history in psql and is strongly recommended.

`-Dlibedit_preferred={ true | false }`

Setting this to true favors the use of the BSD-licensed libedit library rather than GPL-licensed Readline. This option is significant only if you have both libraries installed; the default is false, that is to use Readline.

`-Dzlib={ auto | enabled | disabled }`

Enables use of the Zlib library. It defaults to auto and enables support for compressed archives in pg_dump, pg_restore and pg_basebackup and is recommended.

[7] http://www.ossp.org/pkg/lib/uuid/

`-Dspinlocks={ true | false }`

This option is set to true by default; setting it to false will allow the build to succeed even if PostgreSQL has no CPU spinlock support for the platform. The lack of spinlock support will result in very poor performance; therefore, this option should only be changed if the build aborts and informs you that the platform lacks spinlock support. If setting this option to false is required to build PostgreSQL on your platform, please report the problem to the PostgreSQL developers.

`-Datomics={ true | false }`

This option is set to true by default; setting it to false will disable use of CPU atomic operations. The option does nothing on platforms that lack such operations. On platforms that do have them, disabling atomics will result in poor performance. Changing this option is only useful for debugging or making performance comparisons.

17.4.3.4. Build Process Details

`--auto_features={ auto | enabled | disabled }`

Setting this option allows you to override the value of all "auto" features (features that are enabled automatically if the required software is found). This can be useful when you want to disable or enable all the "optional" features at once without having to set each of them manually. The default value for this parameter is auto.

`--backend=`*`BACKEND`*

The default backend Meson uses is ninja and that should suffice for most use cases. However, if you'd like to fully integrate with Visual Studio, you can set the *`BACKEND`* to `vs`.

`-Dc_args=`*`OPTIONS`*

This option can be used to pass extra options to the C compiler.

`-Dc_link_args=`*`OPTIONS`*

This option can be used to pass extra options to the C linker.

`-Dextra_include_dirs=`*`DIRECTORIES`*

`DIRECTORIES` is a comma-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `-Dextra_lib_dirs` option.

Example: `-Dextra_include_dirs=/opt/gnu/include,/usr/sup/include`.

`-Dextra_lib_dirs=`*`DIRECTORIES`*

`DIRECTORIES` is a comma-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `-Dextra_include_dirs` option) if you have packages installed in non-standard locations.

Example: `-Dextra_lib_dirs=/opt/gnu/lib,/usr/sup/lib`.

`-Dsystem_tzdata=`*`DIRECTORY`*

PostgreSQL includes its own time zone database, which it requires for date and time operations. This time zone database is in fact compatible with the IANA time zone database provided by many operating systems such as FreeBSD, Linux, and Solaris, so it would be redundant to install it again. When this option is used, the system-supplied time zone database in *`DIRECTORY`* is used instead of the one included in the PostgreSQL source distribution. *`DIRECTORY`* must be specified as an absolute path. `/usr/share/zoneinfo` is a likely directory on some operating systems.

Note that the installation routine will not detect mismatching or erroneous time zone data. If you use this option, you are advised to run the regression tests to verify that the time zone data you have pointed to works correctly with PostgreSQL.

This option is mainly aimed at binary package distributors who know their target operating system well. The main advantage of using this option is that the PostgreSQL package won't need to be upgraded whenever any of the many local daylight-saving time rules change. Another advantage is that PostgreSQL can be cross-compiled more straightforwardly if the time zone database files do not need to be built during the installation.

`-Dextra_version=`*`STRING`*

Append *`STRING`* to the PostgreSQL version number. You can use this, for example, to mark binaries built from unreleased Git snapshots or containing custom patches with an extra version string, such as a `git describe` identifier or a distribution package release number.

`-Drpath={ true | false }`

This option is set to true by default. If set to false, do not mark PostgreSQL's executables to indicate that they should search for shared libraries in the installation's library directory (see `--libdir`). On most platforms, this marking uses an absolute path to the library directory, so that it will be unhelpful if you relocate the installation later. However, you will then need to provide some other way for the executables to find the shared libraries. Typically this requires configuring the operating system's dynamic linker to search the library directory; see Section 17.5.1 for more detail.

`-D`*`BINARY_NAME=PATH`*

If a program required to build PostgreSQL (with or without optional flags) is stored at a non-standard path, you can specify it manually to `meson configure`. The complete list of programs for which this is supported can be found by running `meson configure`. Example:

```
meson configure -DBISON=PATH_TO_BISON
```

17.4.3.5. Documentation

See Section J.2 for the tools needed for building the documentation.

`-Ddocs={ auto | enabled | disabled }`

Enables building the documentation in HTML and man format. It defaults to auto.

`-Ddocs_pdf={ auto | enabled | disabled }`

Enables building the documentation in PDF format. It defaults to auto.

`-Ddocs_html_style={ simple | website }`

Controls which CSS stylesheet is used. The default is `simple`. If set to `website`, the HTML documentation will reference the stylesheet for postgresql.org[8].

17.4.3.6. Miscellaneous

`-Dpgport=`*`NUMBER`*

Set *`NUMBER`* as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.

[8] https://www.postgresql.org/docs/current/

`-Dkrb_srvnam=`*`NAME`*

The default name of the Kerberos service principal used by GSSAPI. `postgres` is the default. There's usually no reason to change this unless you are building for a Windows environment, in which case it must be set to upper case `POSTGRES`.

`-Dsegsize=`*`SEGSIZE`*

Set the *segment size*, in gigabytes. Large tables are divided into multiple operating-system files, each of size equal to the segment size. This avoids problems with file size limits that exist on many platforms. The default segment size, 1 gigabyte, is safe on all supported platforms. If your operating system has “largefile” support (which most do, nowadays), you can use a larger segment size. This can be helpful to reduce the number of file descriptors consumed when working with very large tables. But be careful not to select a value larger than is supported by your platform and the file systems you intend to use. Other tools you might wish to use, such as tar, could also set limits on the usable file size. It is recommended, though not absolutely required, that this value be a power of 2.

`-Dblocksize=`*`BLOCKSIZE`*

Set the *block size*, in kilobytes. This is the unit of storage and I/O within tables. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 32 (kilobytes).

`-Dwal_blocksize=`*`BLOCKSIZE`*

Set the *WAL block size*, in kilobytes. This is the unit of storage and I/O within the WAL log. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 64 (kilobytes).

17.4.3.7. Developer Options

Most of the options in this section are only of interest for developing or debugging PostgreSQL. They are not recommended for production builds, except for `--debug`, which can be useful to enable detailed bug reports in the unlucky event that you encounter a bug. On platforms supporting DTrace, `-Ddtrace` may also be reasonable to use in production.

When building an installation that will be used to develop code inside the server, it is recommended to use at least the `--buildtype=debug` and `-Dcassert` options.

`--buildtype=`*`BUILDTYPE`*

This option can be used to specify the buildtype to use; defaults to `debugoptimized`. If you'd like finer control on the debug symbols and optimization levels than what this option provides, you can refer to the `--debug` and `--optimization` flags.

The following build types are generally used: `plain`, `debug`, `debugoptimized` and `release`. More information about them can be found in the Meson documentation[9].

`--debug`

Compiles all programs and libraries with debugging symbols. This means that you can run the programs in a debugger to analyze problems. This enlarges the size of the installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that might arise. Currently, this option is recommended for production installations only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

[9] https://mesonbuild.com/Running-Meson.html#configuring-the-build-directory

`--optimization=`*`LEVEL`*

Specify the optimization level. `LEVEL` can be set to any of {0,g,1,2,3,s}.

`--werror`

Setting this option asks the compiler to treat warnings as errors. This can be useful for code development.

`-Dcassert={ true | false }`

Enables *assertion* checks in the server, which test for many “cannot happen” conditions. This is invaluable for code development purposes, but the tests slow down the server significantly. Also, having the tests turned on won't necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. This option is not recommended for production use, but you should have it on for development work or when running a beta version.

`-Dtap_tests={ auto | enabled | disabled }`

Enable tests using the Perl TAP tools. Defaults to auto and requires a Perl installation and the Perl module `IPC::Run`. See Section 31.4 for more information.

`-DPG_TEST_EXTRA=`*`TEST_SUITES`*

Enable test suites which require special software to run. This option accepts arguments via a whitespace-separated list. See Section 31.1.3 for details.

`-Db_coverage={ true | false }`

If using GCC, all programs and libraries are compiled with code coverage testing instrumentation. When run, they generate files in the build directory with code coverage metrics. See Section 31.5 for more information. This option is for use only with GCC and when doing development work.

`-Ddtrace={ auto | enabled | disabled }`

Enabling this compiles PostgreSQL with support for the dynamic tracing tool DTrace. See Section 27.5 for more information.

To point to the `dtrace` program, the `DTRACE` option can be set. This will often be necessary because `dtrace` is typically installed under `/usr/sbin`, which might not be in your `PATH`.

`-Dinjection_points={ true | false }`

Compiles PostgreSQL with support for injection points in the server. Injection points allow to run user-defined code from within the server in pre-defined code paths. This helps in testing and in the investigation of concurrency scenarios in a controlled fashion. This option is disabled by default. See Section 36.10.13 for more details. This option is intended to be used only by developers for testing.

`-Dsegsize_blocks=SEGSIZE_BLOCKS`

Specify the relation segment size in blocks. If both `-Dsegsize` and this option are specified, this option wins. This option is only for developers, to test segment related code.

17.4.4. `meson` Build Targets

Individual build targets can be built using `ninja` *`target`*. When no target is specified, everything except documentation is built. Individual build products can be built using the path/filename as *`target`*.

17.4.4.1. Code Targets

`all`

Build everything other than documentation

`backend`

Build backend and related modules

`bin`

Build frontend binaries

`contrib`

Build contrib modules

`pl`

Build procedural languages

17.4.4.2. Developer Targets

`reformat-dat-files`

Rewrite catalog data files into standard format

`expand-dat-files`

Expand all data files to include defaults

`update-unicode`

Update unicode data to new version

17.4.4.3. Documentation Targets

`html`

Build documentation in multi-page HTML format

`man`

Build documentation in man page format

`docs`

Build documentation in multi-page HTML and man page format

`doc/src/sgml/postgres-A4.pdf`

Build documentation in PDF format, with A4 pages

`doc/src/sgml/postgres-US.pdf`

Build documentation in PDF format, with US letter pages

`doc/src/sgml/postgres.html`

Build documentation in single-page HTML format

`alldocs`

Build documentation in all supported formats

17.4.4.4. Installation Targets

`install`

Install postgres, excluding documentation

`install-docs`

Install documentation in multi-page HTML and man page formats

`install-html`

Install documentation in multi-page HTML format

`install-man`

Install documentation in man page format

`install-quiet`

Like "install", but installed files are not displayed

`install-world`

Install postgres, including multi-page HTML and man page documentation

`uninstall`

Remove installed files

17.4.4.5. Other Targets

`clean`

Remove all build products

`test`

Run all enabled tests (including contrib)

`world`

Build everything, including documentation

`help`

List important targets

17.5. Post-Installation Setup

17.5.1. Shared Libraries

On some systems with shared libraries you need to tell the system how to find the newly installed shared libraries. The systems on which this is *not* necessary include FreeBSD, Linux, NetBSD, OpenBSD, and Solaris.

The method to set the shared library search path varies between platforms, but the most widely-used method is to set the environment variable `LD_LIBRARY_PATH` like so: In Bourne shells (`sh`, `ksh`, `bash`, `zsh`):

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

or in `csh` or `tcsh`:

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Replace `/usr/local/pgsql/lib` with whatever you set `--libdir` to in Step 1. You should put these commands into a shell start-up file such as `/etc/profile` or `~/.bash_profile`. Some good information about the caveats associated with this method can be found at http://xahlee.info/UnixResource_dir/_/ldpath.html.

On some systems it might be preferable to set the environment variable `LD_RUN_PATH` *before* building.

On Cygwin, put the library directory in the `PATH` or move the `.dll` files into the `bin` directory.

If in doubt, refer to the manual pages of your system (perhaps `ld.so` or `rld`). If you later get a message like:

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or
 directory
```

then this step was necessary. Simply take care of it then.

If you are on Linux and you have root access, you can run:

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(or equivalent directory) after installation to enable the run-time linker to find the shared libraries faster. Refer to the manual page of `ldconfig` for more information. On FreeBSD, NetBSD, and OpenBSD the command is:

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

instead. Other systems are not known to have an equivalent command.

17.5.2. Environment Variables

If you installed into `/usr/local/pgsql` or some other location that is not searched for programs by default, you should add `/usr/local/pgsql/bin` (or whatever you set `--bindir` to in Step 1) into your `PATH`. Strictly speaking, this is not necessary, but it will make the use of PostgreSQL much more convenient.

To do this, add the following to your shell start-up file, such as `~/.bash_profile` (or `/etc/profile`, if you want it to affect all users):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

If you are using `csh` or `tcsh`, then use this command:

```
set path = ( /usr/local/pgsql/bin $path )
```

To enable your system to find the man documentation, you need to add lines like the following to a shell start-up file unless you installed into a location that is searched by default:

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

The environment variables `PGHOST` and `PGPORT` specify to client applications the host and port of the database server, overriding the compiled-in defaults. If you are going to run client applications remotely then it is convenient if every user that plans to use the database sets `PGHOST`. This is not required, however; the settings can be communicated via command line options to most client programs.

17.6. Supported Platforms

A platform (that is, a CPU architecture and operating system combination) is considered supported by the PostgreSQL development community if the code contains provisions to work on that platform and it has recently been verified to build and pass its regression tests on that platform. Currently, most testing of platform compatibility is done automatically by test machines in the PostgreSQL Build Farm[10]. If you are interested in using PostgreSQL on a platform that is not represented in the build farm, but on which the code works or can be made to work, you are strongly encouraged to set up a build farm member machine so that continued compatibility can be assured.

In general, PostgreSQL can be expected to work on these CPU architectures: x86, PowerPC, S/390, SPARC, ARM, MIPS, RISC-V, and PA-RISC, including big-endian, little-endian, 32-bit, and 64-bit variants where applicable. It is often possible to build on an unsupported CPU type by configuring with `--disable-spinlocks`, but performance will be poor.

PostgreSQL can be expected to work on current versions of these operating systems: Linux, Windows, FreeBSD, OpenBSD, NetBSD, DragonFlyBSD, macOS, Solaris, and illumos. Other Unix-like systems may also work but are not currently being tested. In most cases, all CPU architectures supported by a given operating system will work. Look in Section 17.7 below to see if there is information specific to your operating system, particularly if using an older system.

If you have installation problems on a platform that is known to be supported according to recent build farm results, please report it to `<pgsql-bugs@lists.postgresql.org>`. If you are interested in porting PostgreSQL to a new platform, `<pgsql-hackers@lists.postgresql.org>` is the appropriate place to discuss that.

Historical versions of PostgreSQL or POSTGRES also ran on CPU architectures including Alpha, Itanium, M32R, M68K, M88K, NS32K, SuperH, and VAX, and operating systems including 4.3BSD, AIX, BEOS, BSD/OS, DG/UX, Dynix, HP-UX, IRIX, NeXTSTEP, QNX, SCO, SINIX, Sprite, SunOS, Tru64 UNIX, and ULTRIX.

17.7. Platform-Specific Notes

This section documents additional platform-specific issues regarding the installation and setup of PostgreSQL. Be sure to read the installation instructions, and in particular Section 17.1 as well. Also, check Chapter 31 regarding the interpretation of regression test results.

Platforms that are not covered here have no known platform-specific installation issues.

17.7.1. Cygwin

PostgreSQL can be built using Cygwin, a Linux-like environment for Windows, but that method is inferior to the native Windows build and running a server under Cygwin is no longer recommended.

[10] https://buildfarm.postgresql.org/

When building from source, proceed according to the Unix-style installation procedure (i.e., `./configure; make`; etc.), noting the following Cygwin-specific differences:

- Set your path to use the Cygwin bin directory before the Windows utilities. This will help prevent problems with compilation.
- The `adduser` command is not supported; use the appropriate user management application on Windows. Otherwise, skip this step.
- The `su` command is not supported; use ssh to simulate su on Windows. Otherwise, skip this step.
- OpenSSL is not supported.
- Start `cygserver` for shared memory support. To do this, enter the command `/usr/sbin/cygserver &`. This program needs to be running anytime you start the PostgreSQL server or initialize a database cluster (`initdb`). The default `cygserver` configuration may need to be changed (e.g., increase `SEMMNS`) to prevent PostgreSQL from failing due to a lack of system resources.
- Building might fail on some systems where a locale other than C is in use. To fix this, set the locale to C by doing `export LANG=C.utf8` before building, and then setting it back to the previous setting after you have installed PostgreSQL.
- The parallel regression tests (`make check`) can generate spurious regression test failures due to overflowing the `listen()` backlog queue which causes connection refused errors or hangs. You can limit the number of connections using the make variable `MAX_CONNECTIONS` thus:

  ```
  make MAX_CONNECTIONS=5 check
  ```

  (On some systems you can have up to about 10 simultaneous connections.)

It is possible to install `cygserver` and the PostgreSQL server as Windows NT services. For information on how to do this, please refer to the `README` document included with the PostgreSQL binary package on Cygwin. It is installed in the directory `/usr/share/doc/Cygwin`.

17.7.2. macOS

To build PostgreSQL from source on macOS, you will need to install Apple's command line developer tools, which can be done by issuing

```
xcode-select --install
```

(note that this will pop up a GUI dialog window for confirmation). You may or may not wish to also install Xcode.

On recent macOS releases, it's necessary to embed the “sysroot” path in the include switches used to find some system header files. This results in the outputs of the configure script varying depending on which SDK version was used during configure. That shouldn't pose any problem in simple scenarios, but if you are trying to do something like building an extension on a different machine than the server code was built on, you may need to force use of a different sysroot path. To do that, set `PG_SYSROOT`, for example

```
make PG_SYSROOT=/desired/path all
```

To find out the appropriate path on your machine, run

```
xcrun --show-sdk-path
```

Note that building an extension using a different sysroot version than was used to build the core server is not really recommended; in the worst case it could result in hard-to-debug ABI inconsistencies.

You can also select a non-default sysroot path when configuring, by specifying `PG_SYSROOT` to configure:

```
./configure ... PG_SYSROOT=/desired/path
```

This would primarily be useful to cross-compile for some other macOS version. There is no guarantee that the resulting executables will run on the current host.

To suppress the `-isysroot` options altogether, use

```
./configure ... PG_SYSROOT=none
```

(any nonexistent pathname will work). This might be useful if you wish to build with a non-Apple compiler, but beware that that case is not tested or supported by the PostgreSQL developers.

macOS's "System Integrity Protection" (SIP) feature breaks `make check`, because it prevents passing the needed setting of `DYLD_LIBRARY_PATH` down to the executables being tested. You can work around that by doing `make install` before `make check`. Most PostgreSQL developers just turn off SIP, though.

17.7.3. MinGW

PostgreSQL for Windows can be built using MinGW, a Unix-like build environment for Microsoft operating systems. The MinGW build procedure uses the normal build system described in this chapter.

MinGW, the Unix-like build tools, and MSYS, a collection of Unix tools required to run shell scripts like `configure`, can be downloaded from http://www.mingw.org/. Neither is required to run the resulting binaries; they are needed only for creating the binaries.

To build 64 bit binaries using MinGW, install the 64 bit tool set from https://mingw-w64.org/, put its bin directory in the `PATH`, and run `configure` with the `--host=x86_64-w64-mingw32` option.

After you have everything installed, it is suggested that you run psql under `CMD.EXE`, as the MSYS console has buffering issues.

17.7.3.1. Collecting Crash Dumps

If PostgreSQL on Windows crashes, it has the ability to generate minidumps that can be used to track down the cause for the crash, similar to core dumps on Unix. These dumps can be read using the Windows Debugger Tools or using Visual Studio. To enable the generation of dumps on Windows, create a subdirectory named `crashdumps` inside the cluster data directory. The dumps will then be written into this directory with a unique name based on the identifier of the crashing process and the current time of the crash.

17.7.4. Solaris

PostgreSQL is well-supported on Solaris. The more up to date your operating system, the fewer issues you will experience.

17.7.4.1. Required Tools

You can build with either GCC or Sun's compiler suite. For better code optimization, Sun's compiler is strongly recommended on the SPARC architecture. If you are using Sun's compiler, be careful not to select `/usr/ucb/cc`; use `/opt/SUNWspro/bin/cc`.

You can download Sun Studio from https://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/. Many GNU tools are integrated into Solaris 10, or they are present on the Solaris companion CD. If you need packages for older versions of Solaris, you can find these tools at http://www.sunfreeware.com. If you prefer sources, look at https://www.gnu.org/prep/ftp.

17.7.4.2. configure Complains About a Failed Test Program

If `configure` complains about a failed test program, this is probably a case of the run-time linker being unable to find some library, probably libz, libreadline or some other non-standard library such as libssl. To point it to the right location, set the `LDFLAGS` environment variable on the `configure` command line, e.g.,

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

See the ld man page for more information.

17.7.4.3. Compiling for Optimal Performance

On the SPARC architecture, Sun Studio is strongly recommended for compilation. Try using the `-xO5` optimization flag to generate significantly faster binaries. Do not use any flags that modify behavior of floating-point operations and `errno` processing (e.g., `-fast`).

If you do not have a reason to use 64-bit binaries on SPARC, prefer the 32-bit version. The 64-bit operations are slower and 64-bit binaries are slower than the 32-bit variants. On the other hand, 32-bit code on the AMD64 CPU family is not native, so 32-bit code is significantly slower on that CPU family.

17.7.4.4. Using DTrace for Tracing PostgreSQL

Yes, using DTrace is possible. See Section 27.5 for further information.

If you see the linking of the `postgres` executable abort with an error message like:

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

your DTrace installation is too old to handle probes in static functions. You need Solaris 10u4 or newer to use DTrace.

17.7.5. Visual Studio

It is recommended that most users download the binary distribution for Windows, available as a graphical installer package from the PostgreSQL website at https://www.postgresql.org/download/. Building from source is only intended for people developing PostgreSQL or extensions.

PostgreSQL for Windows with Visual Studio can be built using Meson, as described in Section 17.4. The native Windows port requires a 32 or 64-bit version of Windows 10 or later.

Native builds of psql don't support command line editing. The Cygwin build does support command line editing, so it should be used where psql is needed for interactive use on Windows.

PostgreSQL can be built using the Visual C++ compiler suite from Microsoft. These compilers can be either from Visual Studio, Visual Studio Express or some versions of the Microsoft Windows

SDK. If you do not already have a Visual Studio environment set up, the easiest ways are to use the compilers from Visual Studio 2022 or those in the Windows SDK 10, which are both free downloads from Microsoft.

Both 32-bit and 64-bit builds are possible with the Microsoft Compiler suite. 32-bit PostgreSQL builds are possible with Visual Studio 2015 to Visual Studio 2022, as well as standalone Windows SDK releases 10 and above. 64-bit PostgreSQL builds are supported with Microsoft Windows SDK version 10 and above or Visual Studio 2015 and above.

If your build environment doesn't ship with a supported version of the Microsoft Windows SDK it is recommended that you upgrade to the latest version (currently version 10), available for download from https://www.microsoft.com/download.

You must always include the Windows Headers and Libraries part of the SDK. If you install a Windows SDK including the Visual C++ Compilers, you don't need Visual Studio to build. Note that as of Version 8.0a the Windows SDK no longer ships with a complete command-line build environment.

17.7.5.1. Requirements

The following additional products are required to build PostgreSQL on Windows.

Strawberry Perl

Strawberry Perl is required to run the build generation scripts. MinGW or Cygwin Perl will not work. It must also be present in the PATH. Binaries can be downloaded from https://strawberryperl.com.

Bison and Flex

Bison and Flex are required. Only Bison versions 2.3 and later will work. Flex must be version 2.5.35 or later.

Both Bison and Flex are included in the msys tool suite, available from http://www.mingw.org/wiki/MSYS as part of the MinGW compiler suite.

You will need to add the directory containing `flex.exe` and `bison.exe` to the PATH environment variable. In the case of MinGW, the directory is the `\msys\1.0\bin` subdirectory of your MinGW installation directory.

> **Note**
>
> The Bison distribution from GnuWin32 appears to have a bug that causes Bison to malfunction when installed in a directory with spaces in the name, such as the default location on English installations `C:\Program Files\GnuWin32`. Consider installing into `C:\GnuWin32` or use the NTFS short name path to GnuWin32 in your PATH environment setting (e.g., `C:\PROGRA~1\GnuWin32`).

The following additional products are not required to get started, but are required to build the complete package.

Magicsplat Tcl

Required for building PL/Tcl. Binaries can be downloaded from https://www.magicsplat.com/tcl-installer/index.html.

Diff

Diff is required to run the regression tests, and can be downloaded from http://gnuwin32.sourceforge.net.

Gettext

Gettext is required to build with NLS support, and can be downloaded from http://gnuwin32.sourceforge.net. Note that binaries, dependencies and developer files are all needed.

MIT Kerberos

Required for GSSAPI authentication support. MIT Kerberos can be downloaded from https://web.mit.edu/Kerberos/dist/index.html.

libxml2 and libxslt

Required for XML support. Binaries can be downloaded from https://zlatkovic.com/pub/libxml or source from http://xmlsoft.org. Note that libxml2 requires iconv, which is available from the same download location.

LZ4

Required for supporting LZ4 compression. Binaries and source can be downloaded from https://github.com/lz4/lz4/releases.

Zstandard

Required for supporting Zstandard compression. Binaries and source can be downloaded from https://github.com/facebook/zstd/releases.

OpenSSL

Required for SSL support. Binaries can be downloaded from https://slproweb.com/products/Win32OpenSSL.html or source from https://www.openssl.org.

ossp-uuid

Required for UUID-OSSP support (contrib only). Source can be downloaded from http://www.ossp.org/pkg/lib/uuid/.

Python

Required for building PL/Python. Binaries can be downloaded from https://www.python.org.

zlib

Required for compression support in pg_dump and pg_restore. Binaries can be downloaded from https://www.zlib.net.

17.7.5.2. Special Considerations for 64-Bit Windows

PostgreSQL will only build for the x64 architecture on 64-bit Windows.

Mixing 32- and 64-bit versions in the same build tree is not supported. The build system will automatically detect if it's running in a 32- or 64-bit environment, and build PostgreSQL accordingly. For this reason, it is important to start the correct command prompt before building.

To use a server-side third party library such as Python or OpenSSL, this library *must* also be 64-bit. There is no support for loading a 32-bit library in a 64-bit server. Several of the third party libraries that PostgreSQL supports may only be available in 32-bit versions, in which case they cannot be used with 64-bit PostgreSQL.

17.7.5.3. Collecting Crash Dumps

If PostgreSQL on Windows crashes, it has the ability to generate minidumps that can be used to track down the cause for the crash, similar to core dumps on Unix. These dumps can be read using the

Windows Debugger Tools or using Visual Studio. To enable the generation of dumps on Windows, create a subdirectory named `crashdumps` inside the cluster data directory. The dumps will then be written into this directory with a unique name based on the identifier of the crashing process and the current time of the crash.

Chapter 18. Server Setup and Operation

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

The directions in this chapter assume that you are working with plain PostgreSQL without any additional infrastructure, for example a copy that you built from source according to the directions in the preceding chapters. If you are working with a pre-packaged or vendor-supplied version of PostgreSQL, it is likely that the packager has made special provisions for installing and starting the database server according to your system's conventions. Consult the package-level documentation for details.

18.1. 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.) In particular, it is advisable that this user account not own the PostgreSQL executable files, to ensure that a compromised server process could not modify those executables.

Pre-packaged versions of PostgreSQL will typically create a suitable user account automatically during package installation.

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.

18.2. 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. There are two more databases created within each cluster during initialization, named `template1` and `template0`. As the names suggest, these will be used as templates for subsequently-created databases; they should not be used for actual work. (See Chapter 22 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. The data directory must be initialized before being used, using the program initdb which is installed with PostgreSQL.

If you are using a pre-packaged version of PostgreSQL, it may well have a specific convention for where to place the data directory, and it may also provide a script for creating the data directory. In that case you should use that script in preference to running `initdb` directly. Consult the package-level documentation for details.

To initialize a database cluster manually, run `initdb` and specify the desired file system location of the database cluster with 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 18.3), 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, and optionally, group. Group access, when enabled, is read-only. This allows an unprivileged user in the same group as the cluster owner to take a backup of the cluster data or perform other operations that only require read access.

Note that enabling or disabling group access on an existing cluster requires the cluster to be shut down and the appropriate mode to be set on all directories and files before restarting PostgreSQL. Otherwise, a mix of modes might exist in the data directory. For clusters that allow access only by the owner, the appropriate modes are `0700` for directories and `0600` for files. For clusters that also allow reads by the group, the appropriate modes are `0750` for directories and `0640` for files.

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 scram-sha-256` 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 20 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 23.1. 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 23.3.

Non-`C` 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.

18.2.1. 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.

18.2.2. File Systems

Generally, any file system with POSIX semantics can be used for PostgreSQL. Users prefer different file systems for a variety of reasons, including vendor support, performance, and familiarity. Experience suggests that, all other things being equal, one should not expect major performance or behavior changes merely from switching file systems or making minor file system configuration changes.

18.2.2.1. NFS

It is possible to use an NFS file system for storing the PostgreSQL data directory. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives. PostgreSQL does not use any functionality that is known to have nonstandard behavior on NFS, such as file locking.

The only firm requirement for using NFS with PostgreSQL is that the file system is mounted using the `hard` option. With the `hard` option, processes can "hang" indefinitely if there are network problems, so this configuration will require a careful monitoring setup. The `soft` option will interrupt system calls in case of network problems, but PostgreSQL will not repeat system calls interrupted in this way, so any such interruption will result in an I/O error being reported.

It is not necessary to use the `sync` mount option. The behavior of the `async` option is sufficient, since PostgreSQL issues `fsync` calls at appropriate times to flush the write caches. (This is analogous to how it works on a local file system.) However, it is strongly recommended to use the `sync` export option on the NFS *server* on systems where it exists (mainly Linux). Otherwise, an `fsync` or equivalent on the NFS client is not actually guaranteed to reach permanent storage on the server, which could cause corruption similar to running with the parameter fsync off. The defaults of these mount and export options differ between vendors and versions, so it is recommended to check and perhaps specify them explicitly in any case to avoid any ambiguity.

In some cases, an external storage product can be accessed either via NFS or a lower-level protocol such as iSCSI. In the latter case, the storage appears as a block device and any available file system can be created on it. That approach might relieve the DBA from having to deal with some of the idiosyncrasies of NFS, but of course the complexity of managing remote storage then happens at other levels.

18.3. Starting the Database Server

Before anyone can access the database, you must start the database server. The database server program is called `postgres`.

If you are using a pre-packaged version of PostgreSQL, it almost certainly includes provisions for running the server as a background task according to the conventions of your operating system. Using

the package's infrastructure to start the server will be much less work than figuring out how to do this yourself. Consult the package-level documentation for details.

The bare-bones way to start the server manually is just to invoke `postgres` directly, specifying the location of the data directory with the `-D` option, for example:

```
$ 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 24.3 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 19 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 example scripts 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 /usr/local/pgsql/data'
    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.

  When using systemd, you can use the following service unit file (e.g., at `/etc/systemd/system/postgresql.service`):

  ```
  [Unit]
  Description=PostgreSQL database server
  Documentation=man:postgres(1)
  After=network-online.target
  Wants=network-online.target

  [Service]
  Type=notify
  User=postgres
  ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  ExecReload=/bin/kill -HUP $MAINPID
  KillMode=mixed
  KillSignal=SIGINT
  TimeoutSec=infinity

  [Install]
  WantedBy=multi-user.target
  ```

  Using `Type=notify` requires that the server binary was built with `configure --with-systemd`.

  Consider carefully the timeout setting. systemd has a default timeout of 90 seconds as of this writing and will kill a process that does not report readiness within that time. But a PostgreSQL server that might have to perform crash recovery at startup could take much longer to become ready. The suggested value of `infinity` disables the timeout logic.

- 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/data"
  ```

  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.

18.3.1. 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 address "127.0.0.1": Address already in
 use
HINT:  Is another postmaster already running on port 5432? If not,
 wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

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 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not,
 wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

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). This is only likely to happen if you have set `shared_memory_type` to `sysv`. In that case, you can try starting the server with a smaller-than-normal number of buffers (shared_buffers), or 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.

Details about configuring System V IPC facilities are given in Section 18.4.1.

18.3.2. 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: error: connection to server at
 "server.joe.com" (123.123.123.123), port 5432 failed: Connection
 refused
        Is the server running on that host and accepting TCP/IP
 connections?
```

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 might get this when attempting Unix-domain socket communication to a local server:

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432"
 failed: No such file or directory
        Is the server running locally and accepting connections on
 that socket?
```

If the server is indeed running, check that the client's idea of the socket path (here `/tmp`) agrees with the server's unix_socket_directories setting.

A connection failure message always shows the server address or socket path name, which is useful in verifying that the client is trying to connect to the right place. If there is in fact no server listening 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 and rejected it. That case will produce a different message, as shown in Section 20.15.) Other error messages such as `Connection timed out` might indicate more fundamental problems, like lack of network connectivity, or a firewall blocking the connection.

18.4. Managing Kernel Resources

PostgreSQL can sometimes exhaust various operating system resource limits, especially when multiple copies of the server are running on the same system, or in very large installations. This section explains the kernel resources used by PostgreSQL and the steps you can take to resolve problems related to kernel resource consumption.

18.4.1. Shared Memory and Semaphores

PostgreSQL requires the operating system to provide inter-process communication (IPC) features, specifically shared memory and semaphores. Unix-derived systems typically provide “System V” IPC, “POSIX” IPC, or both. Windows has its own implementation of these features and is not discussed here.

By default, PostgreSQL allocates a very small amount of System V shared memory, as well as a much larger amount of anonymous `mmap` shared memory. Alternatively, a single large System V shared memory region can be used (see shared_memory_type). In addition a significant number of semaphores, which can be either System V or POSIX style, are created at server startup. Currently, POSIX semaphores are used on Linux and FreeBSD systems while other platforms use System V semaphores.

System V IPC features are typically constrained by system-wide allocation limits. When PostgreSQL exceeds one of these limits, the server will refuse to start and should leave an instructive error message describing the problem and what to do about it. (See also Section 18.3.1.) The relevant kernel parameters are named consistently across different systems; Table 18.1 gives an overview. The methods to set them, however, vary. Suggestions for some platforms are given below.

Table 18.1. System V IPC Parameters

Name	Description	Values needed to run one PostgreSQL instance
SHMMAX	Maximum size of shared memory segment (bytes)	at least 1kB, but the default is usually much higher

Name	Description	Values needed to run one PostgreSQL instance
SHMMIN	Minimum size of shared memory segment (bytes)	1
SHMALL	Total amount of shared memory available (bytes or pages)	same as SHMMAX if bytes, or ceil(SHMMAX/PAGE_SIZE) if pages, plus room for other applications
SHMSEG	Maximum number of shared memory segments per process	only 1 segment is needed, but the default is much higher
SHMMNI	Maximum number of shared memory segments system-wide	like SHMSEG plus room for other applications
SEMMNI	Maximum number of semaphore identifiers (i.e., sets)	at least ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19) plus room for other applications
SEMMNS	Maximum number of semaphores system-wide	ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19) * 20 plus room for other applications
SEMMSL	Maximum number of semaphores per set	at least 20
SEMMAP	Number of entries in semaphore map	see text
SEMVMX	Maximum value of semaphore	at least 1000 (The default is often 32767; do not change unless necessary)

PostgreSQL requires a few bytes of System V shared memory (typically 48 bytes, on 64-bit platforms) for each copy of the server. On most modern operating systems, this amount can easily be allocated. However, if you are running many copies of the server or you explicitly configure the server to use large amounts of System V shared memory (see shared_memory_type and dynamic_shared_memory_type), it may be necessary to increase SHMALL, which is the total amount of System V shared memory system-wide. Note that SHMALL is measured in pages rather than bytes on many systems.

Less likely to cause problems is the minimum size for shared memory segments (SHMMIN), which should be at most approximately 32 bytes for PostgreSQL (it is usually just 1). The maximum number of segments system-wide (SHMMNI) or per-process (SHMSEG) are unlikely to cause a problem unless your system has them set to zero.

When using System V semaphores, PostgreSQL uses one semaphore per allowed connection (max_connections), allowed autovacuum worker process (autovacuum_max_workers), allowed WAL sender process (max_wal_senders), and allowed background process (max_worker_processes), in sets of 19. Each such set will also contain a 20th semaphore which contains a “magic number”, to detect collision with semaphore sets used by other applications. The maximum number of semaphores in the system is set by SEMMNS, which consequently must be at least as high as max_connections plus autovacuum_max_workers plus max_wal_senders, plus max_worker_processes, plus one extra for each 19 allowed connections plus workers (see the formula in Table 18.1). The parameter SEMMNI determines the limit on the number of semaphore sets that can exist on the system at one time. Hence this parameter must be at least ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19). Lowering the number of allowed connections is a temporary workaround for failures, which are usually confusingly worded “No space left on device”, from the function semget.

In some cases it might also be necessary to increase SEMMAP to be at least on the order of SEMMNS. If the system has this parameter (many do not), it defines the size of the semaphore resource map, in which each contiguous block of available semaphores needs an entry. When a semaphore set is freed it

is either added to an existing entry that is adjacent to the freed block or it is registered under a new map entry. If the map is full, the freed semaphores get lost (until reboot). Fragmentation of the semaphore space could over time lead to fewer available semaphores than there should be.

Various other settings related to “semaphore undo”, such as SEMMNU and SEMUME, do not affect PostgreSQL.

When using POSIX semaphores, the number of semaphores needed is the same as for System V, that is one semaphore per allowed connection (max_connections), allowed autovacuum worker process (autovacuum_max_workers), allowed WAL sender process (max_wal_senders), and allowed background process (max_worker_processes). On the platforms where this option is preferred, there is no specific kernel limit on the number of POSIX semaphores.

FreeBSD

The default shared memory settings are usually good enough, unless you have set shared_memory_type to sysv. System V semaphores are not used on this platform.

The default IPC settings can be changed using the sysctl or loader interfaces. The following parameters can be set using sysctl:

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

To make these settings persist over reboots, modify /etc/sysctl.conf.

If you have set shared_memory_type to sysv, you might also want to configure your kernel to lock System V shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

If running in a FreeBSD jail, you should set its sysvshm parameter to new, so that it has its own separate System V shared memory namespace. (Before FreeBSD 11.0, it was necessary to enable shared access to the host's IPC namespace from jails, and take measures to avoid collisions.)

NetBSD

The default shared memory settings are usually good enough, unless you have set shared_memory_type to sysv. You will usually want to increase kern.ipc.semmni and kern.ipc.semmns, as NetBSD's default settings for these are uncomfortably small.

IPC parameters can be adjusted using sysctl, for example:

```
# sysctl -w kern.ipc.semmni=100
```

To make these settings persist over reboots, modify /etc/sysctl.conf.

If you have set shared_memory_type to sysv, you might also want to configure your kernel to lock System V shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

OpenBSD

The default shared memory settings are usually good enough, unless you have set shared_memory_type to sysv. You will usually want to increase kern.seminfo.semmni and kern.seminfo.semmns, as OpenBSD's default settings for these are uncomfortably small.

IPC parameters can be adjusted using sysctl, for example:

```
# sysctl kern.seminfo.semmni=100
```

To make these settings persist over reboots, modify `/etc/sysctl.conf`.

Linux

The default shared memory settings are usually good enough, unless you have set `shared_memory_type` to `sysv`, and even then only on older kernel versions that shipped with low defaults. System V semaphores are not used on this platform.

The shared memory size settings can be changed via the `sysctl` interface. For example, to allow 16 GB:

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

To make these settings persist over reboots, see `/etc/sysctl.conf`.

macOS

The default shared memory and semaphore settings are usually good enough, unless you have set `shared_memory_type` to `sysv`.

The recommended method for configuring shared memory in macOS is to create a file named `/etc/sysctl.conf`, containing variable assignments such as:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Note that in some macOS versions, *all five* shared-memory parameters must be set in `/etc/sysctl.conf`, else the values will be ignored.

`SHMMAX` can only be set to a multiple of 4096.

`SHMALL` is measured in 4 kB pages on this platform.

It is possible to change all but `SHMMNI` on the fly, using sysctl. But it's still best to set up your preferred values via `/etc/sysctl.conf`, so that the values will be kept across reboots.

Solaris
illumos

The default shared memory and semaphore settings are usually good enough for most PostgreSQL applications. Solaris defaults to a `SHMMAX` of one-quarter of system RAM. To further adjust this setting, use a project setting associated with the `postgres` user. For example, run the following as `root`:

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-
memory=(privileged,8GB,deny)" -U postgres -G postgres
 user.postgres
```

This command adds the `user.postgres` project and sets the shared memory maximum for the `postgres` user to 8GB, and takes effect the next time that user logs in, or when you restart PostgreSQL (not reload). The above assumes that PostgreSQL is run by the `postgres` user in the `postgres` group. No server reboot is required.

Other recommended kernel setting changes for database servers which will have a large number of connections are:

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

Additionally, if you are running PostgreSQL inside a zone, you may need to raise the zone resource usage limits as well. See "Chapter2: Projects and Tasks" in the *System Administrator's Guide* for more information on `projects` and `prctl`.

18.4.2. systemd RemoveIPC

If systemd is in use, some care must be taken that IPC resources (including shared memory) are not prematurely removed by the operating system. This is especially of concern when installing PostgreSQL from source. Users of distribution packages of PostgreSQL are less likely to be affected, as the `postgres` user is then normally created as a system user.

The setting `RemoveIPC` in `logind.conf` controls whether IPC objects are removed when a user fully logs out. System users are exempt. This setting defaults to on in stock systemd, but some operating system distributions default it to off.

A typical observed effect when this setting is on is that shared memory objects used for parallel query execution are removed at apparently random times, leading to errors and warnings while attempting to open and remove them, like

```
WARNING:  could not remove shared memory segment "/
PostgreSQL.1450751626": No such file or directory
```

Different types of IPC objects (shared memory vs. semaphores, System V vs. POSIX) are treated slightly differently by systemd, so one might observe that some IPC resources are not removed in the same way as others. But it is not advisable to rely on these subtle differences.

A “user logging out” might happen as part of a maintenance job or manually when an administrator logs in as the `postgres` user or something similar, so it is hard to prevent in general.

What is a “system user” is determined at systemd compile time from the `SYS_UID_MAX` setting in `/etc/login.defs`.

Packaging and deployment scripts should be careful to create the `postgres` user as a system user by using `useradd -r`, `adduser --system`, or equivalent.

Alternatively, if the user account was created incorrectly or cannot be changed, it is recommended to set

```
RemoveIPC=no
```

in `/etc/systemd/logind.conf` or another appropriate configuration file.

> **Caution**
>
> At least one of these two things has to be ensured, or the PostgreSQL server will be very unreliable.

18.4.3. Resource Limits

Unix-like operating systems enforce various kinds of resource limits that might interfere with the operation of your PostgreSQL server. Of particular importance are limits on the number of processes

per user, the number of open files per process, and the amount of memory available to each process. Each of these have a “hard” and a “soft” limit. The soft limit is what actually counts but it can be changed by the user up to the hard limit. The hard limit can only be changed by the root user. The system call `setrlimit` is responsible for setting these parameters. The shell's built-in command `ulimit` (Bourne shells) or `limit` (csh) is used to control the resource limits from the command line. On BSD-derived systems the file `/etc/login.conf` controls the various resource limits set during login. See the operating system documentation for details. The relevant parameters are `maxproc`, `openfiles`, and `datasize`. For example:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` is the soft limit. Append `-max` to set the hard limit.)

Kernels can also have system-wide limits on some resources.

- On Linux the kernel parameter `fs.file-max` determines the maximum number of open files that the kernel will support. It can be changed with `sysctl -w fs.file-max=`*`N`*. To make the setting persist across reboots, add an assignment in `/etc/sysctl.conf`. The maximum limit of files per process is fixed at the time the kernel is compiled; see `/usr/src/linux/Documentation/proc.txt` for more information.

The PostgreSQL server uses one process per connection so you should provide for at least as many processes as allowed connections, in addition to what you need for the rest of your system. This is usually not a problem but if you run several servers on one machine things might get tight.

The factory default limit on open files is often set to “socially friendly” values that allow many users to coexist on a machine without using an inappropriate fraction of the system resources. If you run many servers on a machine this is perhaps what you want, but on dedicated servers you might want to raise this limit.

On the other side of the coin, some systems allow individual processes to open large numbers of files; if more than a few processes do so then the system-wide limit can easily be exceeded. If you find this happening, and you do not want to alter the system-wide limit, you can set PostgreSQL's max_files_per_process configuration parameter to limit the consumption of open files.

Another kernel limit that may be of concern when supporting large numbers of client connections is the maximum socket connection queue length. If more than that many connection requests arrive within a very short period, some may get rejected before the PostgreSQL server can service the requests, with those clients receiving unhelpful connection failure errors such as “Resource temporarily unavailable” or “Connection refused”. The default queue length limit is 128 on many platforms. To raise it, adjust the appropriate kernel parameter via sysctl, then restart the PostgreSQL server. The parameter is variously named `net.core.somaxconn` on Linux, `kern.ipc.soacceptqueue` on newer FreeBSD, and `kern.ipc.somaxconn` on macOS and other BSD variants.

18.4.4. Linux Memory Overcommit

The default virtual memory behavior on Linux is not optimal for PostgreSQL. Because of the way that the kernel implements memory overcommit, the kernel might terminate the PostgreSQL postmaster (the supervisor server process) if the memory demands of either PostgreSQL or another process cause the system to run out of virtual memory.

If this happens, you will see a kernel message that looks like this (consult your system documentation and configuration on where to look for such a message):

```
Out of Memory: Killed process 12345 (postgres).
```

This indicates that the `postgres` process has been terminated due to memory pressure. Although existing database connections will continue to function normally, no new connections will be accepted. To recover, PostgreSQL will need to be restarted.

One way to avoid this problem is to run PostgreSQL on a machine where you can be sure that other processes will not run the machine out of memory. If memory is tight, increasing the swap space of the operating system can help avoid the problem, because the out-of-memory (OOM) killer is invoked only when physical memory and swap space are exhausted.

If PostgreSQL itself is the cause of the system running out of memory, you can avoid the problem by changing your configuration. In some cases, it may help to lower memory-related configuration parameters, particularly `shared_buffers`, `work_mem`, and `hash_mem_multiplier`. In other cases, the problem may be caused by allowing too many connections to the database server itself. In many cases, it may be better to reduce `max_connections` and instead make use of external connection-pooling software.

It is possible to modify the kernel's behavior so that it will not "overcommit" memory. Although this setting will not prevent the OOM killer[1] from being invoked altogether, it will lower the chances significantly and will therefore lead to more robust system behavior. This is done by selecting strict overcommit mode via `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

or placing an equivalent entry in `/etc/sysctl.conf`. You might also wish to modify the related setting `vm.overcommit_ratio`. For details see the kernel documentation file https://www.kernel.org/doc/Documentation/vm/overcommit-accounting.

Another approach, which can be used with or without altering `vm.overcommit_memory`, is to set the process-specific *OOM score adjustment* value for the postmaster process to `-1000`, thereby guaranteeing it will not be targeted by the OOM killer. The simplest way to do this is to execute

```
echo -1000 > /proc/self/oom_score_adj
```

in the PostgreSQL startup script just before invoking `postgres`. Note that this action must be done as root, or it will have no effect; so a root-owned startup script is the easiest place to do it. If you do this, you should also set these environment variables in the startup script before invoking `postgres`:

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

These settings will cause postmaster child processes to run with the normal OOM score adjustment of zero, so that the OOM killer can still target them at need. You could use some other value for `PG_OOM_ADJUST_VALUE` if you want the child processes to run with some other OOM score adjustment. (`PG_OOM_ADJUST_VALUE` can also be omitted, in which case it defaults to zero.) If you do not set `PG_OOM_ADJUST_FILE`, the child processes will run with the same OOM score adjustment as the postmaster, which is unwise since the whole point is to ensure that the postmaster has a preferential setting.

18.4.5. Linux Huge Pages

Using huge pages reduces overhead when using large contiguous chunks of memory, as PostgreSQL does, particularly when using large values of shared_buffers. To use this feature in PostgreSQL you

[1] https://lwn.net/Articles/104179/

need a kernel with `CONFIG_HUGETLBFS=y` and `CONFIG_HUGETLB_PAGE=y`. You will also have to configure the operating system to provide enough huge pages of the desired size. The runtime-computed parameter shared_memory_size_in_huge_pages reports the number of huge pages required. This parameter can be viewed before starting the server with a `postgres` command like:

```
$ postgres -D $PGDATA -C shared_memory_size_in_huge_pages
3170
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
$ ls /sys/kernel/mm/hugepages
hugepages-1048576kB  hugepages-2048kB
```

In this example the default is 2MB, but you can also explicitly request either 2MB or 1GB with huge_page_size to adapt the number of pages calculated by `shared_memory_size_in_huge_pages`. While we need at least `3170` huge pages in this example, a larger setting would be appropriate if other programs on the machine also need huge pages. We can set this with:

```
# sysctl -w vm.nr_hugepages=3170
```

Don't forget to add this setting to `/etc/sysctl.conf` so that it is reapplied after reboots. For non-default huge page sizes, we can instead use:

```
# echo 3170 > /sys/kernel/mm/hugepages/hugepages-2048kB/
nr_hugepages
```

It is also possible to provide these settings at boot time using kernel parameters such as `hugepagesz=2M hugepages=3170`.

Sometimes the kernel is not able to allocate the desired number of huge pages immediately due to fragmentation, so it might be necessary to repeat the command or to reboot. (Immediately after a reboot, most of the machine's memory should be available to convert into huge pages.) To verify the huge page allocation situation for a given size, use:

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

It may also be necessary to give the database server's operating system user permission to use huge pages by setting `vm.hugetlb_shm_group` via sysctl, and/or give permission to lock memory with `ulimit -l`.

The default behavior for huge pages in PostgreSQL is to use them when possible, with the system's default huge page size, and to fall back to normal pages on failure. To enforce the use of huge pages, you can set huge_pages to `on` in `postgresql.conf`. Note that with this setting PostgreSQL will fail to start if not enough huge pages are available.

For a detailed description of the Linux huge pages feature have a look at https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

18.5. Shutting Down the Server

There are several ways to shut down the database server. Under the hood, they all reduce to sending a signal to the supervisor `postgres` process.

If you are using a pre-packaged version of PostgreSQL, and you used its provisions for starting the server, then you should also use its provisions for stopping the server. Consult the package-level documentation for details.

When managing the server directly, you can control the type of shutdown by sending different signals to the `postgres` process:

SIGTERM

This is the *Smart Shutdown* mode. After receiving SIGTERM, the server disallows new connections, but lets existing sessions end their work normally. It shuts down only after all of the sessions terminate. If the server is in recovery when a smart shutdown is requested, recovery and streaming replication will be stopped only after all regular sessions have terminated.

SIGINT

This is the *Fast Shutdown* mode. The server disallows new connections and sends all existing server processes SIGTERM, which will cause them to abort their current transactions and exit promptly. It then waits for all server processes to exit and finally shuts down.

SIGQUIT

This is the *Immediate Shutdown* mode. The server will send SIGQUIT to all child processes and wait for them to terminate. If any do not terminate within 5 seconds, they will be sent SIGKILL. The supervisor server process exits as soon as all child processes have exited, without doing normal database shutdown processing. This will lead to recovery (by replaying the WAL log) upon next start-up. This is recommended only in emergencies.

The pg_ctl program provides a convenient interface for sending these signals to shut down the server. Alternatively, you can send the signal directly using `kill` on non-Windows systems. The PID of the `postgres` process can be found using the `ps` program, or from the file `postmaster.pid` in the data directory. For example, to do a fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

Important

It is best not to use SIGKILL to shut down the server. Doing so will prevent the server from releasing shared memory and semaphores. Furthermore, SIGKILL kills the `postgres` process without letting it relay the signal to its subprocesses, so it might be necessary to kill the individual subprocesses by hand as well.

To terminate an individual session while allowing other sessions to continue, use `pg_terminate_backend()` (see Table 9.94) or send a SIGTERM signal to the child process associated with the session.

18.6. Upgrading a PostgreSQL Cluster

This section discusses how to upgrade your database data from one PostgreSQL release to a newer one.

Current PostgreSQL version numbers consist of a major and a minor version number. For example, in the version number 10.1, the 10 is the major version number and the 1 is the minor version number, meaning this would be the first minor release of the major release 10. For releases before PostgreSQL version 10.0, version numbers consist of three numbers, for example, 9.5.3. In those cases, the major version consists of the first two digit groups of the version number, e.g., 9.5, and the minor version is the third number, e.g., 3, meaning this would be the third minor release of the major release 9.5.

Minor releases never change the internal storage format and are always compatible with earlier and later minor releases of the same major version number. For example, version 10.1 is compatible with version 10.0 and version 10.6. Similarly, for example, 9.5.3 is compatible with 9.5.0, 9.5.1, and 9.5.6.

To update between compatible versions, you simply replace the executables while the server is down and restart the server. The data directory remains unchanged — minor upgrades are that simple.

For *major* releases of PostgreSQL, the internal data storage format is subject to change, thus complicating upgrades. The traditional method for moving data to a new major version is to dump and restore the database, though this can be slow. A faster method is pg_upgrade. Replication methods are also available, as discussed below. (If you are using a pre-packaged version of PostgreSQL, it may provide scripts to assist with major version upgrades. Consult the package-level documentation for details.)

New major versions also typically introduce some user-visible incompatibilities, so application programming changes might be required. All user-visible changes are listed in the release notes (Appendix E); pay particular attention to the section labeled "Migration". Though you can upgrade from one major version to another without upgrading to intervening versions, you should read the major release notes of all intervening versions.

Cautious users will want to test their client applications on the new version before switching over fully; therefore, it's often a good idea to set up concurrent installations of old and new versions. When testing a PostgreSQL major upgrade, consider the following categories of possible changes:

Administration

The capabilities available for administrators to monitor and control the server often change and improve in each major release.

SQL

Typically this includes new SQL command capabilities and not changes in behavior, unless specifically mentioned in the release notes.

Library API

Typically libraries like libpq only add new functionality, again unless mentioned in the release notes.

System Catalogs

System catalog changes usually only affect database management tools.

Server C-language API

This involves changes in the backend function API, which is written in the C programming language. Such changes affect code that references backend functions deep inside the server.

18.6.1. Upgrading Data via pg_dumpall

One upgrade method is to dump data from one major version of PostgreSQL and restore it in another — to do this, you must use a *logical* backup tool like pg_dumpall; file system level backup methods will not work. (There are checks in place that prevent you from using a data directory with an incompatible version of PostgreSQL, so no great harm can be done by trying to start the wrong server version on a data directory.)

It is recommended that you use the pg_dump and pg_dumpall programs from the *newer* version of PostgreSQL, to take advantage of enhancements that might have been made in these programs. Current releases of the dump programs can read data from any server version back to 9.2.

These instructions assume that your existing installation is under the `/usr/local/pgsql` directory, and that the data area is in `/usr/local/pgsql/data`. Substitute your paths appropriately.

1. If making a backup, make sure that your database is not being updated. This does not affect the integrity of the backup, but the changed data would of course not be included. If necessary, edit the permissions in the file `/usr/local/pgsql/data/pg_hba.conf` (or equivalent) to

disallow access from everyone except you. See Chapter 20 for additional information on access control.

To back up your database installation, type:

```
pg_dumpall > outputfile
```

To make the backup, you can use the pg_dumpall command from the version you are currently running; see Section 25.1.2 for more details. For best results, however, try to use the pg_dumpall command from PostgreSQL 17.4, since this version contains bug fixes and improvements over older versions. While this advice might seem idiosyncratic since you haven't installed the new version yet, it is advisable to follow it if you plan to install the new version in parallel with the old version. In that case you can complete the installation normally and transfer the data later. This will also decrease the downtime.

2. Shut down the old server:

```
pg_ctl stop
```

On systems that have PostgreSQL started at boot time, there is probably a start-up file that will accomplish the same thing. For example, on a Red Hat Linux system one might find that this works:

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

See Chapter 18 for details about starting and stopping the server.

3. If restoring from backup, rename or delete the old installation directory if it is not version-specific. It is a good idea to rename the directory, rather than delete it, in case you have trouble and need to revert to it. Keep in mind the directory might consume significant disk space. To rename the directory, use a command like this:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

(Be sure to move the directory as a single unit so relative paths remain unchanged.)

4. Install the new version of PostgreSQL as outlined in Chapter 17.

5. Create a new database cluster if needed. Remember that you must execute these commands while logged in to the special database user account (which you already have if you are upgrading).

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

6. Restore your previous `pg_hba.conf` and any `postgresql.conf` modifications.

7. Start the database server, again using the special database user account:

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

8. Finally, restore your data from backup with:

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

using the *new* psql.

The least downtime can be achieved by installing the new server in a different directory and running both the old and the new servers in parallel, on different ports. Then you can use something like:

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

to transfer your data.

18.6.2. Upgrading Data via pg_upgrade

The pg_upgrade module allows an installation to be migrated in-place from one major PostgreSQL version to another. Upgrades can be performed in minutes, particularly with `--link` mode. It requires steps similar to pg_dumpall above, e.g., starting/stopping the server, running initdb. The pg_upgrade documentation outlines the necessary steps.

18.6.3. Upgrading Data via Replication

It is also possible to use logical replication methods to create a standby server with the updated version of PostgreSQL. This is possible because logical replication supports replication between different major versions of PostgreSQL. The standby can be on the same computer or a different computer. Once it has synced up with the primary server (running the older version of PostgreSQL), you can switch primaries and make the standby the primary and shut down the older database instance. Such a switch-over results in only several seconds of downtime for an upgrade.

This method of upgrading can be performed using the built-in logical replication facilities as well as using external logical replication systems such as pglogical, Slony, Londiste, and Bucardo.

18.7. Preventing Server Spoofing

While the server is running, it is not possible for a malicious user to take the place of the normal database server. However, when the server is down, it is possible for a local user to spoof the normal server by starting their own server. The spoof server could read passwords and queries sent by clients, but could not return any data because the PGDATA directory would still be secure because of directory permissions. Spoofing is possible because any user can start a database server; a client cannot identify an invalid server unless it is specially configured.

One way to prevent spoofing of `local` connections is to use a Unix domain socket directory (unix_socket_directories) that has write permission only for a trusted local user. This prevents a malicious user from creating their own socket file in that directory. If you are concerned that some applications might still reference `/tmp` for the socket file and hence be vulnerable to spoofing, during operating system startup create a symbolic link `/tmp/.s.PGSQL.5432` that points to the relocated socket file. You also might need to modify your `/tmp` cleanup script to prevent removal of the symbolic link.

Another option for `local` connections is for clients to use `requirepeer` to specify the required owner of the server process connected to the socket.

To prevent spoofing on TCP connections, either use SSL certificates and make sure that clients check the server's certificate, or use GSSAPI encryption (or both, if they're on separate connections).

To prevent spoofing with SSL, the server must be configured to accept only `hostssl` connections (Section 20.1) and have SSL key and certificate files (Section 18.9). The TCP client must connect using `sslmode=verify-ca` or `verify-full` and have the appropriate root certificate file installed (Section 32.19.1). Alternatively the system CA pool can be used using `sslrootcert=system`; in this case, `sslmode=verify-full` is forced for safety, since it is generally trivial to obtain certificates which are signed by a public CA.

To prevent server spoofing from occurring when using scram-sha-256 password authentication over a network, you should ensure that you connect to the server using SSL and with one of the anti-spoofing

methods described in the previous paragraph. Additionally, the SCRAM implementation in libpq cannot protect the entire authentication exchange, but using the `channel_binding=require` connection parameter provides a mitigation against server spoofing. An attacker that uses a rogue server to intercept a SCRAM exchange can use offline analysis to potentially determine the hashed password from the client.

To prevent spoofing with GSSAPI, the server must be configured to accept only `hostgssenc` connections (Section 20.1) and use `gss` authentication with them. The TCP client must connect using `gssencmode=require`.

18.8. Encryption Options

PostgreSQL offers encryption at several levels, and provides flexibility in protecting data from disclosure due to database server theft, unscrupulous administrators, and insecure networks. Encryption might also be required to secure sensitive data such as medical records or financial transactions.

Password Encryption

Database user passwords are stored as hashes (determined by the setting password_encryption), so the administrator cannot determine the actual password assigned to the user. If SCRAM or MD5 encryption is used for client authentication, the unencrypted password is never even temporarily present on the server because the client encrypts it before being sent across the network. SCRAM is preferred, because it is an Internet standard and is more secure than the PostgreSQL-specific MD5 authentication protocol.

Encryption For Specific Columns

The pgcrypto module allows certain fields to be stored encrypted. This is useful if only some of the data is sensitive. The client supplies the decryption key and the data is decrypted on the server and then sent to the client.

The decrypted data and the decryption key are present on the server for a brief time while it is being decrypted and communicated between the client and server. This presents a brief moment where the data and keys can be intercepted by someone with complete access to the database server, such as the system administrator.

Data Partition Encryption

Storage encryption can be performed at the file system level or the block level. Linux file system encryption options include eCryptfs and EncFS, while FreeBSD uses PEFS. Block level or full disk encryption options include dm-crypt + LUKS on Linux and GEOM modules geli and gbde on FreeBSD. Many other operating systems support this functionality, including Windows.

This mechanism prevents unencrypted data from being read from the drives if the drives or the entire computer is stolen. This does not protect against attacks while the file system is mounted, because when mounted, the operating system provides an unencrypted view of the data. However, to mount the file system, you need some way for the encryption key to be passed to the operating system, and sometimes the key is stored somewhere on the host that mounts the disk.

Encrypting Data Across A Network

SSL connections encrypt all data sent across the network: the password, the queries, and the data returned. The `pg_hba.conf` file allows administrators to specify which hosts can use non-encrypted connections (`host`) and which require SSL-encrypted connections (`hostssl`). Also, clients can specify that they connect to servers only via SSL.

GSSAPI-encrypted connections encrypt all data sent across the network, including queries and data returned. (No password is sent across the network.) The `pg_hba.conf` file allows administrators to specify which hosts can use non-encrypted connections (`host`) and which require

GSSAPI-encrypted connections (`hostgssenc`). Also, clients can specify that they connect to servers only on GSSAPI-encrypted connections (`gssencmode=require`).

Stunnel or SSH can also be used to encrypt transmissions.

SSL Host Authentication

It is possible for both the client and server to provide SSL certificates to each other. It takes some extra configuration on each side, but this provides stronger verification of identity than the mere use of passwords. It prevents a computer from pretending to be the server just long enough to read the password sent by the client. It also helps prevent “man in the middle” attacks where a computer between the client and server pretends to be the server and reads and passes all data between the client and server.

Client-Side Encryption

If the system administrator for the server's machine cannot be trusted, it is necessary for the client to encrypt the data; this way, unencrypted data never appears on the database server. Data is encrypted on the client before being sent to the server, and database results have to be decrypted on the client before being used.

18.9. Secure TCP/IP Connections with SSL

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. This requires that OpenSSL is installed on both client and server systems and that support in PostgreSQL is enabled at build time (see Chapter 17).

The terms SSL and TLS are often used interchangeably to mean a secure encrypted connection using a TLS protocol. SSL protocols are the precursors to TLS protocols, and the term SSL is still used for encrypted connections even though SSL protocols are no longer supported. SSL is used interchangeably with TLS in PostgreSQL.

18.9.1. Basic Setup

With SSL support compiled in, the PostgreSQL server can be started with support for encrypted connections using TLS protocols enabled by setting the parameter ssl to `on` in `postgresql.conf`. The server will listen for both normal and SSL connections on the same TCP port, and will negotiate with any connecting client on whether to use SSL. By default, this is at the client's option; see Section 20.1 about how to set up the server to require use of SSL for some or all connections.

To start in SSL mode, files containing the server certificate and private key must exist. By default, these files are expected to be named `server.crt` and `server.key`, respectively, in the server's data directory, but other names and locations can be specified using the configuration parameters ssl_cert_file and ssl_key_file.

On Unix systems, the permissions on `server.key` must disallow any access to world or group; achieve this by the command `chmod 0600 server.key`. Alternatively, the file can be owned by root and have group read access (that is, `0640` permissions). That setup is intended for installations where certificate and key files are managed by the operating system. The user under which the PostgreSQL server runs should then be made a member of the group that has access to those certificate and key files.

If the data directory allows group read access then certificate files may need to be located outside of the data directory in order to conform to the security requirements outlined above. Generally, group access is enabled to allow an unprivileged user to backup the database, and in that case the backup software will not be able to read the certificate files and will likely error.

If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered. Using a passphrase by default disables the ability to change the server's

SSL configuration without a server restart, but see ssl_passphrase_command_supports_reload. Furthermore, passphrase-protected private keys cannot be used at all on Windows.

The first certificate in `server.crt` must be the server's certificate because it must match the server's private key. The certificates of "intermediate" certificate authorities can also be appended to the file. Doing this avoids the necessity of storing intermediate certificates on clients, assuming the root and intermediate certificates were created with `v3_ca` extensions. (This sets the certificate's basic constraint of `CA` to `true`.) This allows easier expiration of intermediate certificates.

It is not necessary to add the root certificate to `server.crt`. Instead, clients must have the root certificate of the server's certificate chain.

18.9.2. OpenSSL Configuration

PostgreSQL reads the system-wide OpenSSL configuration file. By default, this file is named `openssl.cnf` and is located in the directory reported by `openssl version -d`. This default can be overridden by setting environment variable `OPENSSL_CONF` to the name of the desired configuration file.

OpenSSL supports a wide range of ciphers and authentication algorithms, of varying strength. While a list of ciphers can be specified in the OpenSSL configuration file, you can specify ciphers specifically for use by the database server by modifying ssl_ciphers in `postgresql.conf`.

> **Note**
>
> It is possible to have authentication without encryption overhead by using `NULL-SHA` or `NULL-MD5` ciphers. However, a man-in-the-middle could read and pass communications between client and server. Also, encryption overhead is minimal compared to the overhead of authentication. For these reasons NULL ciphers are not recommended.

18.9.3. Using Client Certificates

To require the client to supply a trusted certificate, place certificates of the root certificate authorities (CAs) you trust in a file in the data directory, set the parameter ssl_ca_file in `postgresql.conf` to the new file name, and add the authentication option `clientcert=verify-ca` or `clientcert=verify-full` to the appropriate `hostssl` line(s) in `pg_hba.conf`. A certificate will then be requested from the client during SSL connection startup. (See Section 32.19 for a description of how to set up certificates on the client.)

For a `hostssl` entry with `clientcert=verify-ca`, the server will verify that the client's certificate is signed by one of the trusted certificate authorities. If `clientcert=verify-full` is specified, the server will not only verify the certificate chain, but it will also check whether the username or its mapping matches the `cn` (Common Name) of the provided certificate. Note that certificate chain validation is always ensured when the `cert` authentication method is used (see Section 20.12).

Intermediate certificates that chain up to existing root certificates can also appear in the ssl_ca_file file if you wish to avoid storing them on clients (assuming the root and intermediate certificates were created with `v3_ca` extensions). Certificate Revocation List (CRL) entries are also checked if the parameter ssl_crl_file or ssl_crl_dir is set.

The `clientcert` authentication option is available for all authentication methods, but only in `pg_hba.conf` lines specified as `hostssl`. When `clientcert` is not specified, the server verifies the client certificate against its CA file only if a client certificate is presented and the CA is configured.

There are two approaches to enforce that users provide a certificate during login.

The first approach makes use of the `cert` authentication method for `hostssl` entries in `pg_hba.conf`, such that the certificate itself is used for authentication while also providing ssl connection security. See Section 20.12 for details. (It is not necessary to specify any `clientcert` options explicitly when using the `cert` authentication method.) In this case, the `cn` (Common Name) provided in the certificate is checked against the user name or an applicable mapping.

The second approach combines any authentication method for `hostssl` entries with the verification of client certificates by setting the `clientcert` authentication option to `verify-ca` or `verify-full`. The former option only enforces that the certificate is valid, while the latter also ensures that the `cn` (Common Name) in the certificate matches the user name or an applicable mapping.

18.9.4. SSL Server File Usage

Table 18.2 summarizes the files that are relevant to the SSL setup on the server. (The shown file names are default names. The locally configured names could be different.)

Table 18.2. SSL Server File Usage

File	Contents	Effect
ssl_cert_file (`$PGDATA/server.crt`)	server certificate	sent to client to indicate server's identity
ssl_key_file (`$PGDATA/server.key`)	server private key	proves server certificate was sent by the owner; does not indicate certificate owner is trustworthy
ssl_ca_file	trusted certificate authorities	checks that client certificate is signed by a trusted certificate authority
ssl_crl_file	certificates revoked by certificate authorities	client certificate must not be on this list

The server reads these files at server start and whenever the server configuration is reloaded. On Windows systems, they are also re-read whenever a new backend process is spawned for a new client connection.

If an error in these files is detected at server start, the server will refuse to start. But if an error is detected during a configuration reload, the files are ignored and the old SSL configuration continues to be used. On Windows systems, if an error in these files is detected at backend start, that backend will be unable to establish an SSL connection. In all these cases, the error condition is reported in the server log.

18.9.5. Creating Certificates

To create a simple self-signed certificate for the server, valid for 365 days, use the following OpenSSL command, replacing *`dbhost.yourdomain.com`* with the server's host name:

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Then do:

```
chmod og-rwx server.key
```

because the server will reject the file if its permissions are more liberal than this. For more details on how to create your server private key and certificate, refer to the OpenSSL documentation.

While a self-signed certificate can be used for testing, a certificate signed by a certificate authority (CA) (usually an enterprise-wide root CA) should be used in production.

To create a server certificate whose identity can be validated by clients, first create a certificate signing request (CSR) and a public/private key file:

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Then, sign the request with the key to create a root certificate authority (using the default OpenSSL configuration file location on Linux):

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Finally, create a server certificate signed by the new root certificate authority:

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` and `server.key` should be stored on the server, and `root.crt` should be stored on the client so the client can verify that the server's leaf certificate was signed by its trusted root certificate. `root.key` should be stored offline for use in creating future certificates.

It is also possible to create a chain of trust that includes intermediate certificates:

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
```

```
-CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
-out server.crt
```

`server.crt` and `intermediate.crt` should be concatenated into a certificate file bundle and stored on the server. `server.key` should also be stored on the server. `root.crt` should be stored on the client so the client can verify that the server's leaf certificate was signed by a chain of certificates linked to its trusted root certificate. `root.key` and `intermediate.key` should be stored offline for use in creating future certificates.

18.10. Secure TCP/IP Connections with GSSAPI Encryption

PostgreSQL also has native support for using GSSAPI to encrypt client/server communications for increased security. Support requires that a GSSAPI implementation (such as MIT Kerberos) is installed on both client and server systems, and that support in PostgreSQL is enabled at build time (see Chapter 17).

18.10.1. Basic Setup

The PostgreSQL server will listen for both normal and GSSAPI-encrypted connections on the same TCP port, and will negotiate with any connecting client whether to use GSSAPI for encryption (and for authentication). By default, this decision is up to the client (which means it can be downgraded by an attacker); see Section 20.1 about setting up the server to require the use of GSSAPI for some or all connections.

When using GSSAPI for encryption, it is common to use GSSAPI for authentication as well, since the underlying mechanism will determine both client and server identities (according to the GSSAPI implementation) in any case. But this is not required; another PostgreSQL authentication method can be chosen to perform additional verification.

Other than configuration of the negotiation behavior, GSSAPI encryption requires no setup beyond that which is necessary for GSSAPI authentication. (For more information on configuring that, see Section 20.6.)

18.11. Secure TCP/IP Connections with SSH Tunnels

It is possible to use SSH to encrypt the network connection between clients and a PostgreSQL server. Done properly, this provides an adequately secure network connection, even for non-SSL-capable clients.

First make sure that an SSH server is running properly on the same machine as the PostgreSQL server and that you can log in using `ssh` as some user; you then can establish a secure tunnel to the remote server. A secure tunnel listens on a local port and forwards all traffic to a port on the remote machine. Traffic sent to the remote port can arrive on its `localhost` address, or different bind address if desired; it does not appear as coming from your local machine. This command creates a secure tunnel from the client machine to the remote machine `foo.com`:

```
ssh -L 63333:localhost:5432 joe@foo.com
```

The first number in the `-L` argument, 63333, is the local port number of the tunnel; it can be any unused port. (IANA reserves ports 49152 through 65535 for private use.) The name or IP address after this is the remote bind address you are connecting to, i.e., `localhost`, which is the default. The second number, 5432, is the remote end of the tunnel, e.g., the port number your database server

is using. In order to connect to the database server using this tunnel, you connect to port 63333 on the local machine:

```
psql -h localhost -p 63333 postgres
```

To the database server it will then look as though you are user `joe` on host `foo.com` connecting to the `localhost` bind address, and it will use whatever authentication procedure was configured for connections by that user to that bind address. Note that the server will not think the connection is SSL-encrypted, since in fact it is not encrypted between the SSH server and the PostgreSQL server. This should not pose any extra security risk because they are on the same machine.

In order for the tunnel setup to succeed you must be allowed to connect via `ssh` as `joe@foo.com`, just as if you had attempted to use `ssh` to create a terminal session.

You could also have set up port forwarding as

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

but then the database server will see the connection as coming in on its `foo.com` bind address, which is not opened by the default setting `listen_addresses = 'localhost'`. This is usually not what you want.

If you have to "hop" to the database server via some login host, one possible setup could look like this:

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Note that this way the connection from `shell.foo.com` to `db.foo.com` will not be encrypted by the SSH tunnel. SSH offers quite a few configuration possibilities when the network is restricted in various ways. Please refer to the SSH documentation for details.

> **Tip**
>
> Several other applications exist that can provide secure tunnels using a procedure similar in concept to the one just described.

18.12. Registering Event Log on Windows

To register a Windows event log library with the operating system, issue this command:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

This creates registry entries used by the event viewer, under the default event source named `PostgreSQL`.

To specify a different event source name (see event_source), use the `/n` and `/i` options:

```
regsvr32 /n /i:event_source_name pgsql_library_directory/
pgevent.dll
```

To unregister the event log library from the operating system, issue this command:

```
regsvr32 /u [/i:event_source_name] pgsql_library_directory/
pgevent.dll
```

> **Note**
>
> To enable event logging in the database server, modify log_destination to include `eventlog` in `postgresql.conf`.

Chapter 19. 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.

19.1. Setting Parameters

19.1.1. 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. (Values that match an SQL keyword require quoting in some contexts.)
- *Numeric (integer and floating point):* Numeric parameters can be specified in the customary integer and floating-point formats; fractional values are rounded to the nearest integer if the parameter is of integer type. Integer parameters additionally accept hexadecimal input (beginning with `0x`) and octal input (beginning with `0`), but these formats cannot have a fraction. Do not use thousands separators. Quotes are not required, except for hexadecimal input.
- *Numeric with Unit:* Some numeric parameters have an implicit unit, because they describe quantities of memory or time. The unit might be bytes, 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 `B` (bytes), `kB` (kilobytes), `MB` (megabytes), `GB` (gigabytes), and `TB` (terabytes). The multiplier for memory units is 1024, not 1000.
  - Valid time units are `us` (microseconds), `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days).

  If a fractional value is specified with a unit, it will be rounded to a multiple of the next smaller unit if there is one. For example, `30.1 GB` will be converted to `30822 MB` not `32319628902 B`. If the parameter is of integer type, a final rounding to integer occurs after any unit conversion.
- *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.

19.1.2. 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
```

One parameter is specified per line. The equal sign between name and value is optional. Whitespace is insignificant (except within a quoted parameter value) 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. If the file contains multiple entries for the same parameter, all but the last one are ignored.

Parameters set in this way provide default values for the cluster. The settings seen by active sessions will be these values unless they are overridden. The following 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 signal is most easily sent 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 adopt the new values (this will happen after they complete any currently-executing client command). 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.

In addition to `postgresql.conf`, a PostgreSQL data directory contains a file `postgresql.auto.conf`, which has the same format as `postgresql.conf` but is intended to be edited automatically, not manually. This file holds settings provided through the `ALTER SYSTEM` command. This file is read whenever `postgresql.conf` is, and its settings take effect in the same way. Settings in `postgresql.auto.conf` override those in `postgresql.conf`.

External tools may also modify `postgresql.auto.conf`. It is not recommended to do this while the server is running unless allow_alter_system is set to `off`, since a concurrent `ALTER SYSTEM` command could overwrite such changes. Such tools might simply append new settings to the end, or they might choose to remove duplicate settings and/or comments (as `ALTER SYSTEM` will).

The system view `pg_file_settings` can be helpful for pre-testing changes to the configuration files, or for diagnosing problems if a SIGHUP signal did not have the desired effects.

19.1.3. Parameter Interaction via SQL

PostgreSQL provides three SQL commands to establish configuration defaults. The already-mentioned `ALTER SYSTEM` command provides an SQL-accessible means of changing global defaults; it is functionally equivalent to editing `postgresql.conf`. In addition, there are two commands that allow setting of defaults on a per-database or per-role basis:

- The `ALTER DATABASE` command allows global settings to be overridden on a per-database basis.
- The `ALTER ROLE` command allows both global and per-database settings to be overridden with user-specific values.

Values set with `ALTER DATABASE` and `ALTER ROLE` are applied only when starting a fresh database session. They override values obtained from the configuration files or server command line, and constitute defaults for the rest of the session. Note that some settings cannot be changed after server start, and so cannot be set with these commands (or the ones listed below).

Once a client is connected to the database, PostgreSQL provides two additional SQL commands (and equivalent functions) to interact with session-local configuration settings:

- The `SHOW` command allows inspection of the current value of any parameter. The corresponding SQL function is `current_setting(setting_name text)` (see Section 9.28.1).

- The SET command allows modification of the current value of those parameters that can be set locally to a session; it has no effect on other sessions. Many parameters can be set this way by any user, but some can only be set by superusers and users who have been granted SET privilege on that parameter. The corresponding SQL function is set_config(setting_name, new_value, is_local) (see Section 9.28.1).

In addition, the system view pg_settings can be used to view and change session-local values:

- Querying this view is similar to using SHOW ALL but provides more detail. It is also more flexible, since it's possible to specify filter conditions or join against other relations.

- Using UPDATE on this view, specifically updating the setting column, is the equivalent of issuing SET commands. For example, the equivalent of

```
SET configuration_parameter TO DEFAULT;
```

  is:

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

19.1.4. Parameter Interaction via the Shell

In addition to setting global defaults or attaching overrides at the database or role level, you can pass settings to PostgreSQL via shell facilities. Both the server and libpq client library accept parameter values via the shell.

- During server startup, parameter settings can be passed to the postgres command via the -c name=value command-line parameter, or its equivalent --name=value variation. For example,

```
postgres -c log_connections=yes --log-destination='syslog'
```

  Settings provided in this way override those set via postgresql.conf or ALTER SYSTEM, so they cannot be changed globally without restarting the server.

- When starting a client session via libpq, parameter settings can be specified using the PGOPTIONS environment variable. Settings established in this way constitute defaults for the life of the session, but do not affect other sessions. For historical reasons, the format of PGOPTIONS is similar to that used when launching the postgres command; specifically, the -c, or prepended --, before the name must be specified. For example,

```
env PGOPTIONS="-c geqo=off --statement-timeout=5min" psql
```

  Other clients and libraries might provide their own mechanisms, via the shell or otherwise, that allow the user to alter session settings without direct use of SQL commands.

19.1.5. Managing Configuration File Contents

PostgreSQL provides several features for breaking down complex postgresql.conf files into sub-files. These features are especially useful when managing multiple servers with related, but not identical, configurations.

In addition to individual 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 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. These look like

```
include_dir 'directory'
```

Non-absolute directory names are taken as relative to the directory containing the referencing configuration file. Within the specified directory, only non-directory files whose names end with the suffix `.conf` will be included. File names that start with the `.` character are also ignored, to prevent mistakes since such files are hidden on some platforms. Multiple files within an include directory are processed in file name order (according to 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 naming convention establishes a clear order in which these files will be loaded. This is important because only the last setting encountered for a particular parameter while the server is reading configuration files will be used. In this example, something set in `conf.d/02server.conf` would override a value set in `conf.d/01memory.conf`.

You might instead use this approach to naming the files 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.)

19.2. File Locations

In addition to the `postgresql.conf` file already mentioned, PostgreSQL uses two other manually-edited configuration files, which control client authentication (their use is discussed in Chapter 20). By default, all three configuration files are stored in the database cluster's data directory. The parameters described in this section allow the configuration files to be placed elsewhere. (Doing so can ease administration. In particular it is often easier to ensure that the configuration files are properly backed-up when they are kept separate.)

`data_directory` (`string`)

Specifies the directory to use for data storage. This parameter can only be set at server start.

`config_file` (`string`)

Specifies the main server configuration file (customarily called `postgresql.conf`). This parameter can only be set on the `postgres` command line.

`hba_file` (`string`)

Specifies the configuration file for host-based authentication (customarily called `pg_hba.conf`). This parameter can only be set at server start.

`ident_file` (`string`)

Specifies the configuration file for user name mapping (customarily called `pg_ident.conf`). This parameter can only be set at server start. See also Section 20.2.

`external_pid_file` (`string`)

Specifies the name of an additional process-ID (PID) file that the server should create for use by server administration programs. This parameter can only be set at server start.

In a default installation, none of the above parameters are set explicitly. Instead, the data directory is specified by the `-D` command-line option or the `PGDATA` environment variable, and the configuration files are all found within the data directory.

If you wish to keep the configuration files elsewhere than the data directory, the `postgres -D` command-line option or `PGDATA` environment variable must point to the directory containing the configuration files, and the `data_directory` parameter must be set in `postgresql.conf` (or on the command line) to show where the data directory is actually located. Notice that `data_directory` overrides `-D` and `PGDATA` for the location of the data directory, but not for the location of the configuration files.

If you wish, you can specify the configuration file names and locations individually using the parameters `config_file`, `hba_file` and/or `ident_file`. `config_file` can only be specified on the `postgres` command line, but the others can be set within the main configuration file. If all three parameters plus `data_directory` are explicitly set, then it is not necessary to specify `-D` or `PGDATA`.

When setting any of these parameters, a relative path will be interpreted with respect to the directory in which `postgres` is started.

19.3. Connections and Authentication

19.3.1. Connection Settings

`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. If the list is not empty, the server will start if it can listen on at least one TCP/IP address. A warning will be emitted for any TCP/IP address which cannot be opened. The default value is localhost, which allows only local TCP/IP "loopback" connections to be made.

While client authentication (Chapter 20) 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.

PostgreSQL sizes certain resources based directly on the value of `max_connections`. Increasing its value leads to higher allocation of those resources, including shared memory.

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

`reserved_connections` (`integer`)

Determines the number of connection "slots" that are reserved for connections by roles with privileges of the `pg_use_reserved_connections` role. Whenever the number of free connection slots is greater than superuser_reserved_connections but less than or equal to the sum of `superuser_reserved_connections` and `reserved_connections`, new connections will be accepted only for superusers and roles with privileges of `pg_use_reserved_connections`. If `superuser_reserved_connections` or fewer connection slots are available, new connections will be accepted only for superusers.

The default value is zero connections. The value must be less than `max_connections` minus `superuser_reserved_connections`. This parameter can only be set at server start.

`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 `supe-`

`ruser_reserved_connections`, new connections will be accepted only for superusers. The connection slots reserved by this parameter are intended as final reserve for emergency use after the slots reserved by reserved_connections have been exhausted.

The default value is three connections. The value must be less than `max_connections` minus `reserved_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.

A value that starts with `@` specifies that a Unix-domain socket in the abstract namespace should be created (currently supported on Linux only). In that case, this value does not specify a "directory" but a prefix from which the actual socket name is computed in the same manner as for the file-system namespace. While the abstract socket name prefix can be chosen freely, since it is not a file-system location, the convention is to nonetheless use file-system-like values such as `@/tmp`.

The default value is normally `/tmp`, but that can be changed at build time. On Windows, the default is empty, which means no Unix-domain socket is created by default. 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. For sockets in the abstract namespace, no lock file is created.

`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.

This parameter is not supported on Windows. Any setting will be ignored. Also, sockets in the abstract namespace have no file owner, so this setting is also ignored in that case.

`unix_socket_permissions` (`integer`)

Sets the access permissions of the Unix-domain socket(s). Unix-domain sockets use the usual Unix file system permission set. The parameter value is expected to be a numeric mode specified in the format accepted by the `chmod` and `umask` system calls. (To use the customary octal format the number must start with a `0` (zero).)

The default permissions are `0777`, meaning anyone can connect. Reasonable alternatives are `0770` (only user and group, see also `unix_socket_group`) and `0700` (only user). (Note that for a Unix-domain socket, only write permission matters, so there is no point in setting or revoking read or execute permissions.)

This access control mechanism is independent of the one described in Chapter 20.

This parameter can only be set at server start.

This parameter is irrelevant on systems, notably Solaris as of Solaris 10, that ignore socket permissions entirely. There, one can achieve a similar effect by pointing `unix_socket_directories` to a directory having search permission limited to the desired audience.

Sockets in the abstract namespace have no file permissions, so this setting is also ignored in that case.

`bonjour` (`boolean`)

Enables advertising the server's existence via Bonjour. The default is off. This parameter can only be set at server start.

`bonjour_name` (`string`)

Specifies the Bonjour service name. The computer name is used if this parameter is set to the empty string `''` (which is the default). This parameter is ignored if the server was not compiled with Bonjour support. This parameter can only be set at server start.

19.3.2. TCP Settings

`tcp_keepalives_idle` (`integer`)

Specifies the amount of time with no network activity after which the operating system should send a TCP keepalive message to the client. If this value is specified without units, it is taken as seconds. A value of 0 (the default) selects the operating system's default. On Windows, setting a value of 0 will set this parameter to 2 hours, since Windows does not provide a way to read the system default value. This parameter is supported only on systems that support `TCP_KEEPIDLE` or an equivalent socket option, and on Windows; on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

`tcp_keepalives_interval` (`integer`)

Specifies the amount of time after which a TCP keepalive message that has not been acknowledged by the client should be retransmitted. If this value is specified without units, it is taken as seconds. A value of 0 (the default) selects the operating system's default. On Windows, setting a value of 0 will set this parameter to 1 second, since Windows does not provide a way to read the system default value. This parameter is supported only on systems that support `TCP_KEEPINTVL` or an equivalent socket option, and on Windows; on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

`tcp_keepalives_count` (`integer`)

Specifies the number of TCP keepalive messages that can be lost before the server's connection to the client is considered dead. A value of 0 (the default) selects the operating system's default. This parameter is supported only on systems that support `TCP_KEEPCNT` or an equivalent socket option (which does not include Windows); on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

`tcp_user_timeout` (`integer`)

Specifies the amount of time that transmitted data may remain unacknowledged before the TCP connection is forcibly closed. If this value is specified without units, it is taken as milliseconds. A value of 0 (the default) selects the operating system's default. This parameter is supported only on systems that support `TCP_USER_TIMEOUT` (which does not include Windows); on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

`client_connection_check_interval` (`integer`)

Sets the time interval between optional checks that the client is still connected, while running queries. The check is performed by polling the socket, and allows long running queries to be aborted sooner if the kernel reports that the connection is closed.

This option relies on kernel events exposed by Linux, macOS, illumos and the BSD family of operating systems, and is not currently available on other systems.

If the value is specified without units, it is taken as milliseconds. The default value is `0`, which disables connection checks. Without connection checks, the server will detect the loss of the connection only at the next interaction with the socket, when it waits for, receives or sends data.

For the kernel itself to detect lost TCP connections reliably and within a known time-frame in all scenarios including network failure, it may also be necessary to adjust the TCP keepalive settings of the operating system, or the tcp_keepalives_idle, tcp_keepalives_interval and tcp_keepalives_count settings of PostgreSQL.

19.3.3. Authentication

`authentication_timeout` (`integer`)

Maximum amount of time allowed to complete client authentication. If a would-be client has not completed the authentication protocol in this much time, the server closes the connection. This prevents hung clients from occupying a connection indefinitely. If this value is specified without units, it is taken as seconds. The default is one minute (`1m`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`password_encryption` (`enum`)

When a password is specified in CREATE ROLE or ALTER ROLE, this parameter determines the algorithm to use to encrypt the password. Possible values are `scram-sha-256`, which will encrypt the password with SCRAM-SHA-256, and `md5`, which stores the password as an MD5 hash. The default is `scram-sha-256`.

Note that older clients might lack support for the SCRAM authentication mechanism, and hence not work with passwords encrypted with SCRAM-SHA-256. See Section 20.5 for more details.

`scram_iterations` (`integer`)

The number of computational iterations to be performed when encrypting a password using SCRAM-SHA-256. The default is `4096`. A higher number of iterations provides additional protection against brute-force attacks on stored passwords, but makes authentication slower. Changing the value has no effect on existing passwords encrypted with SCRAM-SHA-256 as the iteration count is fixed at the time of encryption. In order to make use of a changed value, a new password must be set.

`krb_server_keyfile` (`string`)

Sets the location of the server's Kerberos key file. The default is `FILE:/usr/local/pgsql/etc/krb5.keytab` (where the directory part is whatever was specified as `sysconfdir` at build time; use `pg_config --sysconfdir` to determine that). If this parameter is set to an empty string, it is ignored and a system-dependent default is used. This parameter can only be set in the `postgresql.conf` file or on the server command line. See Section 20.6 for more information.

`krb_caseins_users` (`boolean`)

Sets whether GSSAPI user names should be treated case-insensitively. The default is `off` (case sensitive). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`gss_accept_delegation` (`boolean`)

Sets whether GSSAPI delegation should be accepted from the client. The default is `off` meaning credentials from the client will *not* be accepted. Changing this to `on` will make the server accept credentials delegated to it from the client. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.3.4. SSL

See Section 18.9 for more information about setting up SSL. The configuration parameters for controlling transfer encryption using TLS protocols are named `ssl` for historic reasons, even though support for the SSL protocol has been deprecated. SSL is in this context used interchangeably with TLS.

`ssl` (`boolean`)

Enables SSL connections. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `off`.

`ssl_ca_file` (`string`)

Specifies the name of the file containing the SSL server certificate authority (CA). Relative paths are relative to the data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is empty, meaning no CA file is loaded, and client certificate verification is not performed.

`ssl_cert_file` (`string`)

Specifies the name of the file containing the SSL server certificate. Relative paths are relative to the data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `server.crt`.

`ssl_crl_file` (`string`)

Specifies the name of the file containing the SSL client certificate revocation list (CRL). Relative paths are relative to the data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is empty, meaning no CRL file is loaded (unless ssl_crl_dir is set).

`ssl_crl_dir` (`string`)

Specifies the name of the directory containing the SSL client certificate revocation list (CRL). Relative paths are relative to the data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is empty, meaning no CRLs are used (unless ssl_crl_file is set).

The directory needs to be prepared with the OpenSSL command `openssl rehash` or `c_rehash`. See its documentation for details.

When using this setting, CRLs in the specified directory are loaded on-demand at connection time. New CRLs can be added to the directory and will be used immediately. This is unlike ssl_crl_file, which causes the CRL in the file to be loaded at server start time or when the configuration is reloaded. Both settings can be used together.

`ssl_key_file` (`string`)

Specifies the name of the file containing the SSL server private key. Relative paths are relative to the data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `server.key`.

`ssl_ciphers` (`string`)

Specifies a list of SSL cipher suites that are allowed to be used by SSL connections. See the ciphers manual page in the OpenSSL package for the syntax of this setting and a list of supported values. Only connections using TLS version 1.2 and lower are affected. There is currently no setting that controls the cipher choices used by TLS version 1.3 connections. The default value is `HIGH:MEDIUM:+3DES:!aNULL`. The default is usually a reasonable choice unless you have specific security requirements.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

Explanation of the default value:

`HIGH`

Cipher suites that use ciphers from `HIGH` group (e.g., AES, Camellia, 3DES)

`MEDIUM`

Cipher suites that use ciphers from `MEDIUM` group (e.g., RC4, SEED)

`+3DES`

The OpenSSL default order for `HIGH` is problematic because it orders 3DES higher than AES128. This is wrong because 3DES offers less security than AES128, and it is also much slower. `+3DES` reorders it after all other `HIGH` and `MEDIUM` ciphers.

`!aNULL`

Disables anonymous cipher suites that do no authentication. Such cipher suites are vulnerable to MITM attacks and therefore should not be used.

Available cipher suite details will vary across OpenSSL versions. Use the command `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'` to see actual details for the currently installed OpenSSL version. Note that this list is filtered at run time based on the server key type.

`ssl_prefer_server_ciphers` (`boolean`)

Specifies whether to use the server's SSL cipher preferences, rather than the client's. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `on`.

PostgreSQL versions before 9.4 do not have this setting and always use the client's preferences. This setting is mainly for backward compatibility with those versions. Using the server's preferences is usually better because it is more likely that the server is appropriately configured.

`ssl_ecdh_curve` (`string`)

Specifies the name of the curve to use in ECDH key exchange. It needs to be supported by all clients that connect. It does not need to be the same curve used by the server's Elliptic Curve key. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `prime256v1`.

OpenSSL names for the most common curves are: `prime256v1` (NIST P-256), `secp384r1` (NIST P-384), `secp521r1` (NIST P-521). The full list of available curves can be shown with the command `openssl ecparam -list_curves`. Not all of them are usable in TLS though.

`ssl_min_protocol_version` (`enum`)

Sets the minimum SSL/TLS protocol version to use. Valid values are currently: `TLSv1`, `TLSv1.1`, `TLSv1.2`, `TLSv1.3`. Older versions of the OpenSSL library do not support all values; an error will be raised if an unsupported setting is chosen. Protocol versions before TLS 1.0, namely SSL version 2 and 3, are always disabled.

The default is `TLSv1.2`, which satisfies industry best practices as of this writing.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`ssl_max_protocol_version` (`enum`)

Sets the maximum SSL/TLS protocol version to use. Valid values are as for ssl_min_protocol_version, with addition of an empty string, which allows any protocol version. The default is

to allow any version. Setting the maximum protocol version is mainly useful for testing or if some component has issues working with a newer protocol.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`ssl_dh_params_file` (`string`)

Specifies the name of the file containing Diffie-Hellman parameters used for so-called ephemeral DH family of SSL ciphers. The default is empty, in which case compiled-in default DH parameters used. Using custom DH parameters reduces the exposure if an attacker manages to crack the well-known compiled-in DH parameters. You can create your own DH parameters file with the command `openssl dhparam -out dhparams.pem 2048`.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`ssl_passphrase_command` (`string`)

Sets an external command to be invoked when a passphrase for decrypting an SSL file such as a private key needs to be obtained. By default, this parameter is empty, which means the built-in prompting mechanism is used.

The command must print the passphrase to the standard output and exit with code 0. In the parameter value, `%p` is replaced by a prompt string. (Write `%%` for a literal `%`.) Note that the prompt string will probably contain whitespace, so be sure to quote adequately. A single newline is stripped from the end of the output if present.

The command does not actually have to prompt the user for a passphrase. It can read it from a file, obtain it from a keychain facility, or similar. It is up to the user to make sure the chosen mechanism is adequately secure.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`ssl_passphrase_command_supports_reload` (`boolean`)

This parameter determines whether the passphrase command set by `ssl_passphrase_command` will also be called during a configuration reload if a key file needs a passphrase. If this parameter is off (the default), then `ssl_passphrase_command` will be ignored during a reload and the SSL configuration will not be reloaded if a passphrase is needed. That setting is appropriate for a command that requires a TTY for prompting, which might not be available when the server is running. Setting this parameter to on might be appropriate if the passphrase is obtained from a file, for example.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.4. Resource Consumption

19.4.1. Memory

`shared_buffers` (`integer`)

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 128 megabytes (`128MB`), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes. However, settings significantly higher than the minimum are usually needed for good performance. If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. (Non-default values of `BLCKSZ` change the minimum value.) This parameter can only be set at server start.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads where even larger settings for `shared_buffers` are effective, but because PostgreSQL al-

so relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount. Larger settings for `shared_buffers` usually require a corresponding increase in `max_wal_size`, in order to spread out the process of writing large quantities of new or changed data over a longer period of time.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system.

`huge_pages` (`enum`)

Controls whether huge pages are requested for the main shared memory area. Valid values are `try` (the default), `on`, and `off`. With `huge_pages` set to `try`, the server will try to request huge pages, but fall back to the default if that fails. With `on`, failure to request huge pages will prevent the server from starting up. With `off`, huge pages will not be requested. The actual state of huge pages is indicated by the server variable huge_pages_status.

At present, this setting is supported only on Linux and Windows. The setting is ignored on other systems when set to `try`. On Linux, it is only supported when `shared_memory_type` is set to `mmap` (the default).

The use of huge pages results in smaller page tables and less CPU time spent on memory management, increasing performance. For more details about using huge pages on Linux, see Section 18.4.5.

Huge pages are known as large pages on Windows. To use them, you need to assign the user right “Lock pages in memory” to the Windows user account that runs PostgreSQL. You can use Windows Group Policy tool (gpedit.msc) to assign the user right “Lock pages in memory”. To start the database server on the command prompt as a standalone process, not as a Windows service, the command prompt must be run as an administrator or User Access Control (UAC) must be disabled. When the UAC is enabled, the normal command prompt revokes the user right “Lock pages in memory” when started.

Note that this setting only affects the main shared memory area. Operating systems such as Linux, FreeBSD, and Illumos can also use huge pages (also known as “super” pages or “large” pages) automatically for normal memory allocation, without an explicit request from PostgreSQL. On Linux, this is called “transparent huge pages” (THP). That feature has been known to cause performance degradation with PostgreSQL for some users on some Linux versions, so its use is currently discouraged (unlike explicit use of `huge_pages`).

`huge_page_size` (`integer`)

Controls the size of huge pages, when they are enabled with huge_pages. The default is zero (`0`). When set to `0`, the default huge page size on the system will be used. This parameter can only be set at server start.

Some commonly available page sizes on modern 64 bit server architectures include: `2MB` and `1GB` (Intel and AMD), `16MB` and `16GB` (IBM POWER), and `64kB`, `2MB`, `32MB` and `1GB` (ARM). For more information about usage and support, see Section 18.4.5.

Non-default settings are currently supported only on Linux.

`temp_buffers` (`integer`)

Sets the maximum amount of memory used for temporary buffers within each database session. These are session-local buffers used only for access to temporary tables. If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default is eight megabytes (`8MB`). (If `BLCKSZ` is not 8kB, the default value scales proportionally to it.) This setting can be changed within individual sessions, but only before the first use of temporary tables within the session; subsequent attempts to change the value will have no effect on that session.

A session will allocate temporary buffers as needed up to the limit given by `temp_buffers`. The cost of setting a large value in sessions that do not actually need many temporary buffers is only a buffer descriptor, or about 64 bytes, per increment in `temp_buffers`. However if a buffer is actually used an additional 8192 bytes will be consumed for it (or in general, `BLCKSZ` bytes).

`max_prepared_transactions` (`integer`)

Sets the maximum number of transactions that can be in the "prepared" state simultaneously (see PREPARE TRANSACTION). Setting this parameter to zero (which is the default) disables the prepared-transaction feature. This parameter can only be set at server start.

If you are not planning to use prepared transactions, this parameter should be set to zero to prevent accidental creation of prepared transactions. If you are using prepared transactions, you will probably want `max_prepared_transactions` to be at least as large as max_connections, so that every session can have a prepared transaction pending.

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

`work_mem` (`integer`)

Sets the base maximum amount of memory to be used by a query operation (such as a sort or hash table) before writing to temporary disk files. If this value is specified without units, it is taken as kilobytes. The default value is four megabytes (`4MB`). Note that a complex query might perform several sort and hash operations at the same time, with each operation generally being allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of `work_mem`; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins. Hash tables are used in hash joins, hash-based aggregation, memoize nodes and hash-based processing of `IN` subqueries.

Hash-based operations are generally more sensitive to memory availability than equivalent sort-based operations. The memory limit for a hash table is computed by multiplying `work_mem` by `hash_mem_multiplier`. This makes it possible for hash-based operations to use an amount of memory that exceeds the usual `work_mem` base amount.

`hash_mem_multiplier` (`floating point`)

Used to compute the maximum amount of memory that hash-based operations can use. The final limit is determined by multiplying `work_mem` by `hash_mem_multiplier`. The default value is 2.0, which makes hash-based operations use twice the usual `work_mem` base amount.

Consider increasing `hash_mem_multiplier` in environments where spilling by query operations is a regular occurrence, especially when simply increasing `work_mem` results in memory pressure (memory pressure typically takes the form of intermittent out of memory errors). The default setting of 2.0 is often effective with mixed workloads. Higher settings in the range of 2.0 - 8.0 or more may be effective in environments where `work_mem` has already been increased to 40MB or more.

`maintenance_work_mem` (`integer`)

Specifies the maximum amount of memory to be used by maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. If this value is specified without units, it is taken as kilobytes. It defaults to 64 megabytes (`64MB`). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to autovacuum_max_workers times this memory may be allocated, so be careful not to set the default value too high. It may be useful to control for this by separately setting autovacuum_work_mem.

`autovacuum_work_mem` (`integer`)

Specifies the maximum amount of memory to be used by each autovacuum worker process. If this value is specified without units, it is taken as kilobytes. It defaults to -1, indicating that the value of maintenance_work_mem should be used instead. The setting has no effect on the behavior of `VACUUM` when run in other contexts. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`vacuum_buffer_usage_limit` (`integer`)

Specifies the size of the *Buffer Access Strategy* used by the `VACUUM` and `ANALYZE` commands. A setting of `0` will allow the operation to use any number of `shared_buffers`. Otherwise valid sizes range from `128 kB` to `16 GB`. If the specified size would exceed 1/8 the size of `shared_buffers`, the size is silently capped to that value. The default value is `2MB`. If this value is specified without units, it is taken as kilobytes. This parameter can be set at any time. It can be overridden for VACUUM and ANALYZE when passing the `BUFFER_USAGE_LIMIT` option. Higher settings can allow `VACUUM` and `ANALYZE` to run more quickly, but having too large a setting may cause too many other useful pages to be evicted from shared buffers.

`logical_decoding_work_mem` (`integer`)

Specifies the maximum amount of memory to be used by logical decoding, before some of the decoded changes are written to local disk. This limits the amount of memory used by logical streaming replication connections. It defaults to 64 megabytes (`64MB`). Since each replication connection only uses a single buffer of this size, and an installation normally doesn't have many such connections concurrently (as limited by `max_wal_senders`), it's safe to set this value significantly higher than `work_mem`, reducing the amount of decoded changes written to disk.

`commit_timestamp_buffers` (`integer`)

Specifies the amount of memory to use to cache the contents of `pg_commit_ts` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `0`, which requests `shared_buffers`/512 up to 1024 blocks, but not fewer than 16 blocks. This parameter can only be set at server start.

`multixact_member_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_multixact/members` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `32`. This parameter can only be set at server start.

`multixact_offset_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_multixact/offsets` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `16`. This parameter can only be set at server start.

`notify_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_notify` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `16`. This parameter can only be set at server start.

`serializable_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_serial` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `32`. This parameter can only be set at server start.

`subtransaction_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_subtrans` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `0`, which requests `shared_buffers`/512 up to 1024 blocks, but not fewer than 16 blocks. This parameter can only be set at server start.

`transaction_buffers` (`integer`)

Specifies the amount of shared memory to use to cache the contents of `pg_xact` (see Table 65.1). If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The default value is `0`, which requests `shared_buffers`/512 up to 1024 blocks, but not fewer than 16 blocks. This parameter can only be set at server start.

`max_stack_depth` (`integer`)

Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by `ulimit -s` or local equivalent), less a safety margin of a megabyte or so. The safety margin is needed because the stack depth is not checked in every routine in the server, but only in key potentially-recursive routines. If this value is specified without units, it is taken as kilobytes. The default setting is two megabytes (`2MB`), which is conservatively small and unlikely to risk crashes. However, it might be too small to allow execution of complex functions. Only superusers and users with the appropriate `SET` privilege can change this setting.

Setting `max_stack_depth` higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process. On platforms where PostgreSQL can determine the kernel limit, the server will not allow this variable to be set to an unsafe value. However, not all platforms provide the information, so caution is recommended in selecting a value.

`shared_memory_type` (`enum`)

Specifies the shared memory implementation that the server should use for the main shared memory region that holds PostgreSQL's shared buffers and other shared data. Possible values are `mmap` (for anonymous shared memory allocated using `mmap`), `sysv` (for System V shared memory allocated via `shmget`) and `windows` (for Windows shared memory). Not all values are supported on all platforms; the first supported option is the default for that platform. The use of the `sysv` option, which is not the default on any platform, is generally discouraged because it typically requires non-default kernel settings to allow for large allocations (see Section 18.4.1).

`dynamic_shared_memory_type` (`enum`)

Specifies the dynamic shared memory implementation that the server should use. Possible values are `posix` (for POSIX shared memory allocated using `shm_open`), `sysv` (for System V shared memory allocated via `shmget`), `windows` (for Windows shared memory), and `mmap` (to simulate shared memory using memory-mapped files stored in the data directory). Not all values are supported on all platforms; the first supported option is usually the default for that platform. The use of the `mmap` option, which is not the default on any platform, is generally discouraged because the operating system may write modified pages back to disk repeatedly, increasing system I/O load; however, it may be useful for debugging, when the `pg_dynshmem` directory is stored on a RAM disk, or when other shared memory facilities are not available.

`min_dynamic_shared_memory` (`integer`)

Specifies the amount of memory that should be allocated at server startup for use by parallel queries. When this memory region is insufficient or exhausted by concurrent queries, new parallel queries try to allocate extra shared memory temporarily from the operating system using the method configured with `dynamic_shared_memory_type`, which may be slower due to memory management overheads. Memory that is allocated at startup with `min_dynamic_shared_memory` is affected by the `huge_pages` setting on operating systems where that is supported, and may be more likely to benefit from larger pages on operating systems where

that is managed automatically. The default value is `0` (none). This parameter can only be set at server start.

19.4.2. Disk

`temp_file_limit` (`integer`)

Specifies the maximum amount of disk space that a process can use for temporary files, such as sort and hash temporary files, or the storage file for a held cursor. A transaction attempting to exceed this limit will be canceled. If this value is specified without units, it is taken as kilobytes. `-1` (the default) means no limit. Only superusers and users with the appropriate `SET` privilege can change this setting.

This setting constrains the total space used at any instant by all temporary files used by a given PostgreSQL process. It should be noted that disk space used for explicit temporary tables, as opposed to temporary files used behind-the-scenes in query execution, does *not* count against this limit.

`max_notify_queue_pages` (`integer`)

Specifies the maximum amount of allocated pages for NOTIFY / LISTEN queue. The default value is 1048576. For 8 KB pages it allows to consume up to 8 GB of disk space.

19.4.3. Kernel Resource Usage

`max_files_per_process` (`integer`)

Sets the maximum number of simultaneously open files allowed to each server subprocess. The default is one thousand files. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. But on some platforms (notably, most BSD systems), the kernel will allow individual processes to open many more files than the system can actually support if many processes all try to open that many files. If you find yourself seeing "Too many open files" failures, try reducing this setting. This parameter can only be set at server start.

19.4.4. Cost-based Vacuum Delay

During the execution of VACUUM and ANALYZE commands, the system maintains an internal counter that keeps track of the estimated cost of the various I/O operations that are performed. When the accumulated cost reaches a limit (specified by `vacuum_cost_limit`), the process performing the operation will sleep for a short period of time, as specified by `vacuum_cost_delay`. Then it will reset the counter and continue execution.

The intent of this feature is to allow administrators to reduce the I/O impact of these commands on concurrent database activity. There are many situations where it is not important that maintenance commands like `VACUUM` and `ANALYZE` finish quickly; however, it is usually very important that these commands do not significantly interfere with the ability of the system to perform other database operations. Cost-based vacuum delay provides a way for administrators to achieve this.

This feature is disabled by default for manually issued `VACUUM` commands. To enable it, set the `vacuum_cost_delay` variable to a nonzero value.

`vacuum_cost_delay` (`floating point`)

The amount of time that the process will sleep when the cost limit has been exceeded. If this value is specified without units, it is taken as milliseconds. The default value is zero, which disables the cost-based vacuum delay feature. Positive values enable cost-based vacuuming.

When using cost-based vacuuming, appropriate values for `vacuum_cost_delay` are usually quite small, perhaps less than 1 millisecond. While `vacuum_cost_delay` can be set to frac-

tional-millisecond values, such delays may not be measured accurately on older platforms. On such platforms, increasing VACUUM's throttled resource consumption above what you get at 1ms will require changing the other vacuum cost parameters. You should, nonetheless, keep `vacuum_cost_delay` as small as your platform will consistently measure; large delays are not helpful.

`vacuum_cost_page_hit` (`integer`)

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page. The default value is one.

`vacuum_cost_page_miss` (`integer`)

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content. The default value is 2.

`vacuum_cost_page_dirty` (`integer`)

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again. The default value is 20.

`vacuum_cost_limit` (`integer`)

This is the accumulated cost that will cause the vacuuming process to sleep for `vacuum_cost_delay`. The default is 200.

> **Note**
>
> There are certain operations that hold critical locks and should therefore complete as quickly as possible. Cost-based vacuum delays do not occur during such operations. Therefore it is possible that the cost accumulates far higher than the specified limit. To avoid uselessly long delays in such cases, the actual delay is calculated as `vacuum_cost_delay` * `accumulated_balance` / `vacuum_cost_limit` with a maximum of `vacuum_cost_delay` * 4.

19.4.5. Background Writer

There is a separate server process called the *background writer*, whose function is to issue writes of "dirty" (new or modified) shared buffers. When the number of clean shared buffers appears to be insufficient, the background writer writes some dirty buffers to the file system and marks them as clean. This reduces the likelihood that server processes handling user queries will be unable to find clean buffers and have to write dirty buffers themselves. However, the background writer does cause a net overall increase in I/O load, because while a repeatedly-dirtied page might otherwise be written only once per checkpoint interval, the background writer might write it several times as it is dirtied in the same interval. The parameters discussed in this subsection can be used to tune the behavior for local needs.

`bgwriter_delay` (`integer`)

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the following parameters). It then sleeps for the length of `bgwriter_delay`, and repeats. When there are no dirty buffers in the buffer pool, though, it goes into a longer sleep regardless of `bgwriter_delay`. If this value is specified without units, it is taken as milliseconds. The default value is 200 milliseconds (`200ms`). Note that on some systems, the effective resolution of sleep delays is 10 milliseconds; setting `bgwriter_delay` to a value that is not a multiple of 10 might have the same results as setting

it to the next higher multiple of 10. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`bgwriter_lru_maxpages` (`integer`)

In each round, no more than this many buffers will be written by the background writer. Setting this to zero disables background writing. (Note that checkpoints, which are managed by a separate, dedicated auxiliary process, are unaffected.) The default value is 100 buffers. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`bgwriter_lru_multiplier` (`floating point`)

The number of dirty buffers written in each round is based on the number of new buffers that have been needed by server processes during recent rounds. The average recent need is multiplied by `bgwriter_lru_multiplier` to arrive at an estimate of the number of buffers that will be needed during the next round. Dirty buffers are written until there are that many clean, reusable buffers available. (However, no more than `bgwriter_lru_maxpages` buffers will be written per round.) Thus, a setting of 1.0 represents a "just in time" policy of writing exactly the number of buffers predicted to be needed. Larger values provide some cushion against spikes in demand, while smaller values intentionally leave writes to be done by server processes. The default is 2.0. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`bgwriter_flush_after` (`integer`)

Whenever more than this amount of data has been written by the background writer, attempt to force the OS to issue these writes to the underlying storage. Doing so will limit the amount of dirty data in the kernel's page cache, reducing the likelihood of stalls when an `fsync` is issued at the end of a checkpoint, or when the OS writes data back in larger batches in the background. Often that will result in greatly reduced transaction latency, but there also are some cases, especially with workloads that are bigger than shared_buffers, but smaller than the OS's page cache, where performance might degrade. This setting may have no effect on some platforms. If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The valid range is between `0`, which disables forced writeback, and `2MB`. The default is `512kB` on Linux, `0` elsewhere. (If `BLCKSZ` is not 8kB, the default and maximum values scale proportionally to it.) This parameter can only be set in the `postgresql.conf` file or on the server command line.

Smaller values of `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` reduce the extra I/O load caused by the background writer, but make it more likely that server processes will have to issue writes for themselves, delaying interactive queries.

19.4.6. Asynchronous Behavior

`backend_flush_after` (`integer`)

Whenever more than this amount of data has been written by a single backend, attempt to force the OS to issue these writes to the underlying storage. Doing so will limit the amount of dirty data in the kernel's page cache, reducing the likelihood of stalls when an `fsync` is issued at the end of a checkpoint, or when the OS writes data back in larger batches in the background. Often that will result in greatly reduced transaction latency, but there also are some cases, especially with workloads that are bigger than shared_buffers, but smaller than the OS's page cache, where performance might degrade. This setting may have no effect on some platforms. If this value is specified without units, it is taken as blocks, that is `BLCKSZ` bytes, typically 8kB. The valid range is between `0`, which disables forced writeback, and `2MB`. The default is `0`, i.e., no forced writeback. (If `BLCKSZ` is not 8kB, the maximum value scales proportionally to it.)

`effective_io_concurrency` (`integer`)

Sets the number of concurrent disk I/O operations that PostgreSQL expects can be executed simultaneously. Raising this value will increase the number of I/O operations that any individual

PostgreSQL session attempts to initiate in parallel. The allowed range is 1 to 1000, or zero to disable issuance of asynchronous I/O requests. Currently, this setting only affects bitmap heap scans.

For magnetic drives, a good starting point for this setting is the number of separate drives comprising a RAID 0 stripe or RAID 1 mirror being used for the database. (For RAID 5 the parity drive should not be counted.) However, if the database is often busy with multiple queries issued in concurrent sessions, lower values may be sufficient to keep the disk array busy. A value higher than needed to keep the disks busy will only result in extra CPU overhead. SSDs and other memory-based storage can often process many concurrent requests, so the best value might be in the hundreds.

Asynchronous I/O depends on an effective `posix_fadvise` function, which some operating systems lack. If the function is not present then setting this parameter to anything but zero will result in an error. On some operating systems (e.g., Solaris), the function is present but does not actually do anything.

The default is 1 on supported systems, otherwise 0. This value can be overridden for tables in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`maintenance_io_concurrency` (`integer`)

Similar to `effective_io_concurrency`, but used for maintenance work that is done on behalf of many client sessions.

The default is 10 on supported systems, otherwise 0. This value can be overridden for tables in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`io_combine_limit` (`integer`)

Controls the largest I/O size in operations that combine I/O. The default is 128kB.

`max_worker_processes` (`integer`)

Sets the maximum number of background processes that the cluster can support. This parameter can only be set at server start. The default is 8.

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

When changing this value, consider also adjusting max_parallel_workers, max_parallel_maintenance_workers, and max_parallel_workers_per_gather.

`max_parallel_workers_per_gather` (`integer`)

Sets the maximum number of workers that can be started by a single `Gather` or `Gather Merge` node. Parallel workers are taken from the pool of processes established by max_worker_processes, limited by max_parallel_workers. Note that the requested number of workers may not actually be available at run time. If this occurs, the plan will run with fewer workers than expected, which may be inefficient. The default value is 2. Setting this value to 0 disables parallel query execution.

Note that parallel queries may consume very substantially more resources than non-parallel queries, because each worker process is a completely separate process which has roughly the same impact on the system as an additional user session. This should be taken into account when choosing a value for this setting, as well as when configuring other settings that control resource utilization, such as work_mem. Resource limits such as `work_mem` are applied individually to each worker, which means the total utilization may be much higher across all processes than it would normally be for any single process. For example, a parallel query using 4 workers may use up to 5 times as much CPU time, memory, I/O bandwidth, and so forth as a query which uses no workers at all.

For more information on parallel query, see Chapter 15.

`max_parallel_maintenance_workers` (`integer`)

Sets the maximum number of parallel workers that can be started by a single utility command. Currently, the parallel utility commands that support the use of parallel workers are `CREATE INDEX` when building a B-tree or BRIN index, and `VACUUM` without `FULL` option. Parallel workers are taken from the pool of processes established by max_worker_processes, limited by max_parallel_workers. Note that the requested number of workers may not actually be available at run time. If this occurs, the utility operation will run with fewer workers than expected. The default value is 2. Setting this value to 0 disables the use of parallel workers by utility commands.

Note that parallel utility commands should not consume substantially more memory than equivalent non-parallel operations. This strategy differs from that of parallel query, where resource limits generally apply per worker process. Parallel utility commands treat the resource limit `maintenance_work_mem` as a limit to be applied to the entire utility command, regardless of the number of parallel worker processes. However, parallel utility commands may still consume substantially more CPU resources and I/O bandwidth.

`max_parallel_workers` (`integer`)

Sets the maximum number of workers that the cluster can support for parallel operations. The default value is 8. When increasing or decreasing this value, consider also adjusting max_parallel_maintenance_workers and max_parallel_workers_per_gather. Also, note that a setting for this value which is higher than max_worker_processes will have no effect, since parallel workers are taken from the pool of worker processes established by that setting.

`parallel_leader_participation` (`boolean`)

Allows the leader process to execute the query plan under `Gather` and `Gather Merge` nodes instead of waiting for worker processes. The default is `on`. Setting this value to `off` reduces the likelihood that workers will become blocked because the leader is not reading tuples fast enough, but requires the leader process to wait for worker processes to start up before the first tuples can be produced. The degree to which the leader can help or hinder performance depends on the plan type, number of workers and query duration.

19.5. Write Ahead Log

For additional information on tuning these settings, see Section 28.5.

19.5.1. Settings

`wal_level` (`enum`)

`wal_level` determines how much information is written to the WAL. The default value is `replica`, which writes enough data to support WAL archiving and replication, including running read-only queries on a standby server. `minimal` removes all logging except the information required to recover from a crash or immediate shutdown. Finally, `logical` adds information necessary to support logical decoding. Each level includes the information logged at all lower levels. This parameter can only be set at server start.

The `minimal` level generates the least WAL volume. It logs no row information for permanent relations in transactions that create or rewrite them. This can make operations much faster (see Section 14.4.7). Operations that initiate this optimization include:

```
ALTER ... SET TABLESPACE
CLUSTER
CREATE TABLE
```

`REFRESH MATERIALIZED VIEW` (without `CONCURRENTLY`)
`REINDEX`
`TRUNCATE`

However, minimal WAL does not contain sufficient information for point-in-time recovery, so `replica` or higher must be used to enable continuous archiving (archive_mode) and streaming binary replication. In fact, the server will not even start in this mode if `max_wal_senders` is non-zero. Note that changing `wal_level` to `minimal` makes previous base backups unusable for point-in-time recovery and standby servers.

In `logical` level, the same information is logged as with `replica`, plus information needed to extract logical change sets from the WAL. Using a level of `logical` will increase the WAL volume, particularly if many tables are configured for `REPLICA IDENTITY FULL` and many `UPDATE` and `DELETE` statements are executed.

In releases prior to 9.6, this parameter also allowed the values `archive` and `hot_standby`. These are still accepted but mapped to `replica`.

`fsync` (`boolean`)

If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk, by issuing `fsync()` system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster can recover to a consistent state after an operating system or hardware crash.

While turning off `fsync` is often a performance benefit, this can result in unrecoverable data corruption in the event of a power failure or system crash. Thus it is only advisable to turn off `fsync` if you can easily recreate your entire database from external data.

Examples of safe circumstances for turning off `fsync` include the initial loading of a new database cluster from a backup file, using a database cluster for processing a batch of data after which the database will be thrown away and recreated, or for a read-only database clone which gets recreated frequently and is not used for failover. High quality hardware alone is not a sufficient justification for turning off `fsync`.

For reliable recovery when changing `fsync` off to on, it is necessary to force all modified buffers in the kernel to durable storage. This can be done while the cluster is shutdown or while `fsync` is on by running `initdb --sync-only`, running `sync`, unmounting the file system, or rebooting the server.

In many situations, turning off synchronous_commit for noncritical transactions can provide much of the potential performance benefit of turning off `fsync`, without the attendant risks of data corruption.

`fsync` can only be set in the `postgresql.conf` file or on the server command line. If you turn this parameter off, also consider turning off full_page_writes.

`synchronous_commit` (`enum`)

Specifies how much WAL processing must complete before the database server returns a “success” indication to the client. Valid values are `remote_apply`, `on` (the default), `remote_write`, `local`, and `off`.

If `synchronous_standby_names` is empty, the only meaningful settings are `on` and `off`; `remote_apply`, `remote_write` and `local` all provide the same local synchronization level as `on`. The local behavior of all non-`off` modes is to wait for local flush of WAL to disk. In `off` mode, there is no waiting, so there can be a delay between when success is reported to the client and when the transaction is later guaranteed to be safe against a server crash. (The maximum delay is three times wal_writer_delay.) Unlike fsync, setting this parameter to `off` does not create any risk of database inconsistency: an operating system or database crash might result in some

recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly. So, turning `synchronous_commit` off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. For more discussion see Section 28.4.

If synchronous_standby_names is non-empty, `synchronous_commit` also controls whether transaction commits will wait for their WAL records to be processed on the standby server(s).

When set to `remote_apply`, commits will wait until replies from the current synchronous standby(s) indicate they have received the commit record of the transaction and applied it, so that it has become visible to queries on the standby(s), and also written to durable storage on the standbys. This will cause much larger commit delays than previous settings since it waits for WAL replay. When set to `on`, commits wait until replies from the current synchronous standby(s) indicate they have received the commit record of the transaction and flushed it to durable storage. This ensures the transaction will not be lost unless both the primary and all synchronous standbys suffer corruption of their database storage. When set to `remote_write`, commits will wait until replies from the current synchronous standby(s) indicate they have received the commit record of the transaction and written it to their file systems. This setting ensures data preservation if a standby instance of PostgreSQL crashes, but not if the standby suffers an operating-system-level crash because the data has not necessarily reached durable storage on the standby. The setting `local` causes commits to wait for local flush to disk, but not for replication. This is usually not desirable when synchronous replication is in use, but is provided for completeness.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is therefore possible, and useful, to have some transactions commit synchronously and others asynchronously. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue `SET LOCAL synchronous_commit TO OFF` within the transaction.

Table 19.1 summarizes the capabilities of the `synchronous_commit` settings.

Table 19.1. synchronous_commit Modes

synchronous_commit setting	local durable commit	standby durable commit after PG crash	standby durable commit after OS crash	standby query consistency
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

`wal_sync_method` (`enum`)

Method used for forcing WAL updates out to disk. If `fsync` is off then this setting is irrelevant, since WAL file updates will not be forced out at all. Possible values are:

- `open_datasync` (write WAL files with `open()` option `O_DSYNC`)
- `fdatasync` (call `fdatasync()` at each commit)
- `fsync` (call `fsync()` at each commit)
- `fsync_writethrough` (call `fsync()` at each commit, forcing write-through of any disk write cache)
- `open_sync` (write WAL files with `open()` option `O_SYNC`)

Not all of these choices are available on all platforms. The default is the first method in the above list that is supported by the platform, except that `fdatasync` is the default on Linux and FreeBSD. The default is not necessarily ideal; it might be necessary to change this setting or other aspects of your system configuration in order to create a crash-safe configuration or achieve optimal performance. These aspects are discussed in Section 28.1. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`full_page_writes` (`boolean`)

When this parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint. This is needed because a page write that is in process during an operating system crash might be only partially completed, leading to an on-disk page that contains a mix of old and new data. The row-level change data normally stored in WAL will not be enough to completely restore such a page during post-crash recovery. Storing the full page image guarantees that the page can be correctly restored, but at the price of increasing the amount of data that must be written to WAL. (Because WAL replay always starts from a checkpoint, it is sufficient to do this during the first change of each page after a checkpoint. Therefore, one way to reduce the cost of full-page writes is to increase the checkpoint interval parameters.)

Turning this parameter off speeds normal operation, but might lead to either unrecoverable data corruption, or silent data corruption, after a system failure. The risks are similar to turning off `fsync`, though smaller, and it should be turned off only based on the same circumstances recommended for that parameter.

Turning off this parameter does not affect use of WAL archiving for point-in-time recovery (PITR) (see Section 25.3).

This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `on`.

`wal_log_hints` (`boolean`)

When this parameter is `on`, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint, even for non-critical modifications of so-called hint bits.

If data checksums are enabled, hint bit updates are always WAL-logged and this setting is ignored. You can use this setting to test how much extra WAL-logging would occur if your database had data checksums enabled.

This parameter can only be set at server start. The default value is `off`.

`wal_compression` (`enum`)

This parameter enables compression of WAL using the specified compression method. When enabled, the PostgreSQL server compresses full page images written to WAL when full_page_writes is on or during a base backup. A compressed page image will be decompressed during WAL replay. The supported methods are `pglz`, `lz4` (if PostgreSQL was compiled with `--with-lz4`) and `zstd` (if PostgreSQL was compiled with `--with-zstd`). The default value is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

Enabling compression can reduce the WAL volume without increasing the risk of unrecoverable data corruption, but at the cost of some extra CPU spent on the compression during WAL logging and on the decompression during WAL replay.

`wal_init_zero` (`boolean`)

If set to `on` (the default), this option causes new WAL files to be filled with zeroes. On some file systems, this ensures that space is allocated before we need to write WAL records. However, *Copy-On-Write* (COW) file systems may not benefit from this technique, so the option is given

to skip the unnecessary work. If set to `off`, only the final byte is written when the file is created so that it has the expected size.

`wal_recycle` (`boolean`)

If set to `on` (the default), this option causes WAL files to be recycled by renaming them, avoiding the need to create new ones. On COW file systems, it may be faster to create new ones, so the option is given to disable this behavior.

`wal_buffers` (`integer`)

The amount of shared memory used for WAL data that has not yet been written to disk. The default setting of -1 selects a size equal to 1/32nd (about 3%) of shared_buffers, but not less than `64kB` nor more than the size of one WAL segment, typically `16MB`. This value can be set manually if the automatic choice is too large or too small, but any positive value less than `32kB` will be treated as `32kB`. If this value is specified without units, it is taken as WAL blocks, that is `XLOG_BLCKSZ` bytes, typically 8kB. This parameter can only be set at server start.

The contents of the WAL buffers are written out to disk at every transaction commit, so extremely large values are unlikely to provide a significant benefit. However, setting this value to at least a few megabytes can improve write performance on a busy server where many clients are committing at once. The auto-tuning selected by the default setting of -1 should give reasonable results in most cases.

`wal_writer_delay` (`integer`)

Specifies how often the WAL writer flushes WAL, in time terms. After flushing WAL the writer sleeps for the length of time given by `wal_writer_delay`, unless woken up sooner by an asynchronously committing transaction. If the last flush happened less than `wal_writer_delay` ago and less than `wal_writer_flush_after` worth of WAL has been produced since, then WAL is only written to the operating system, not flushed to disk. If this value is specified without units, it is taken as milliseconds. The default value is 200 milliseconds (`200ms`). Note that on some systems, the effective resolution of sleep delays is 10 milliseconds; setting `wal_writer_delay` to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`wal_writer_flush_after` (`integer`)

Specifies how often the WAL writer flushes WAL, in volume terms. If the last flush happened less than `wal_writer_delay` ago and less than `wal_writer_flush_after` worth of WAL has been produced since, then WAL is only written to the operating system, not flushed to disk. If `wal_writer_flush_after` is set to `0` then WAL data is always flushed immediately. If this value is specified without units, it is taken as WAL blocks, that is `XLOG_BLCKSZ` bytes, typically 8kB. The default is `1MB`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`wal_skip_threshold` (`integer`)

When `wal_level` is `minimal` and a transaction commits after creating or rewriting a permanent relation, this setting determines how to persist the new data. If the data is smaller than this setting, write it to the WAL log; otherwise, use an fsync of affected files. Depending on the properties of your storage, raising or lowering this value might help if such commits are slowing concurrent transactions. If this value is specified without units, it is taken as kilobytes. The default is two megabytes (`2MB`).

`commit_delay` (`integer`)

Setting `commit_delay` adds a time delay before a WAL flush is initiated. This can improve group commit throughput by allowing a larger number of transactions to commit via a single WAL flush, if system load is high enough that additional transactions become ready to commit within

the given interval. However, it also increases latency by up to the commit_delay for each WAL flush. Because the delay is just wasted if no other transactions become ready to commit, a delay is only performed if at least commit_siblings other transactions are active when a flush is about to be initiated. Also, no delays are performed if fsync is disabled. If this value is specified without units, it is taken as microseconds. The default commit_delay is zero (no delay). Only superusers and users with the appropriate SET privilege can change this setting.

In PostgreSQL releases prior to 9.3, commit_delay behaved differently and was much less effective: it affected only commits, rather than all WAL flushes, and waited for the entire configured delay even if the WAL flush was completed sooner. Beginning in PostgreSQL 9.3, the first process that becomes ready to flush waits for the configured interval, while subsequent processes wait only until the leader completes the flush operation.

commit_siblings (integer)

Minimum number of concurrent open transactions to require before performing the commit_delay delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval. The default is five transactions.

19.5.2. Checkpoints

checkpoint_timeout (integer)

Maximum time between automatic WAL checkpoints. If this value is specified without units, it is taken as seconds. The valid range is between 30 seconds and one day. The default is five minutes (5min). Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.

checkpoint_completion_target (floating point)

Specifies the target of checkpoint completion, as a fraction of total time between checkpoints. The default is 0.9, which spreads the checkpoint across almost all of the available interval, providing fairly consistent I/O load while also leaving some time for checkpoint completion overhead. Reducing this parameter is not recommended because it causes the checkpoint to complete faster. This results in a higher rate of I/O during the checkpoint followed by a period of less I/O between the checkpoint completion and the next scheduled checkpoint. This parameter can only be set in the postgresql.conf file or on the server command line.

checkpoint_flush_after (integer)

Whenever more than this amount of data has been written while performing a checkpoint, attempt to force the OS to issue these writes to the underlying storage. Doing so will limit the amount of dirty data in the kernel's page cache, reducing the likelihood of stalls when an fsync is issued at the end of the checkpoint, or when the OS writes data back in larger batches in the background. Often that will result in greatly reduced transaction latency, but there also are some cases, especially with workloads that are bigger than shared_buffers, but smaller than the OS's page cache, where performance might degrade. This setting may have no effect on some platforms. If this value is specified without units, it is taken as blocks, that is BLCKSZ bytes, typically 8kB. The valid range is between 0, which disables forced writeback, and 2MB. The default is 256kB on Linux, 0 elsewhere. (If BLCKSZ is not 8kB, the default and maximum values scale proportionally to it.) This parameter can only be set in the postgresql.conf file or on the server command line.

checkpoint_warning (integer)

Write a message to the server log if checkpoints caused by the filling of WAL segment files happen closer together than this amount of time (which suggests that max_wal_size ought to be raised). If this value is specified without units, it is taken as seconds. The default is 30 seconds (30s). Zero disables the warning. No warnings will be generated if checkpoint_timeout is less than checkpoint_warning. This parameter can only be set in the postgresql.conf file or on the server command line.

`max_wal_size` (`integer`)

Maximum size to let the WAL grow during automatic checkpoints. This is a soft limit; WAL size can exceed `max_wal_size` under special circumstances, such as heavy load, a failing `archive_command` or `archive_library`, or a high `wal_keep_size` setting. If this value is specified without units, it is taken as megabytes. The default is 1 GB. Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`min_wal_size` (`integer`)

As long as WAL disk usage stays below this setting, old WAL files are always recycled for future use at a checkpoint, rather than removed. This can be used to ensure that enough WAL space is reserved to handle spikes in WAL usage, for example when running large batch jobs. If this value is specified without units, it is taken as megabytes. The default is 80 MB. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.5.3. Archiving

`archive_mode` (`enum`)

When `archive_mode` is enabled, completed WAL segments are sent to archive storage by setting archive_command or archive_library. In addition to `off`, to disable, there are two modes: `on`, and `always`. During normal operation, there is no difference between the two modes, but when set to `always` the WAL archiver is enabled also during archive recovery or standby mode. In `always` mode, all files restored from the archive or streamed with streaming replication will be archived (again). See Section 26.2.9 for details.

`archive_mode` is a separate setting from `archive_command` and `archive_library` so that `archive_command` and `archive_library` can be changed without leaving archiving mode. This parameter can only be set at server start. `archive_mode` cannot be enabled when `wal_level` is set to `minimal`.

`archive_command` (`string`)

The local shell command to execute to archive a completed WAL file segment. Any `%p` in the string is replaced by the path name of the file to archive, and any `%f` is replaced by only the file name. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Use `%%` to embed an actual `%` character in the command. It is important for the command to return a zero exit status only if it succeeds. For more information see Section 25.3.1.

This parameter can only be set in the `postgresql.conf` file or on the server command line. It is only used if `archive_mode` was enabled at server start and `archive_library` is set to an empty string. If both `archive_command` and `archive_library` are set, an error will be raised. If `archive_command` is an empty string (the default) while `archive_mode` is enabled (and `archive_library` is set to an empty string), WAL archiving is temporarily disabled, but the server continues to accumulate WAL segment files in the expectation that a command will soon be provided. Setting `archive_command` to a command that does nothing but return true, e.g., `/bin/true` (`REM` on Windows), effectively disables archiving, but also breaks the chain of WAL files needed for archive recovery, so it should only be used in unusual circumstances.

`archive_library` (`string`)

The library to use for archiving completed WAL file segments. If set to an empty string (the default), archiving via shell is enabled, and archive_command is used. If both `archive_command` and `archive_library` are set, an error will be raised. Otherwise, the specified shared library is used for archiving. The WAL archiver process is restarted by the postmaster when this parameter changes. For more information, see Section 25.3.1 and Chapter 49.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`archive_timeout` (`integer`)

The archive_command or archive_library is only invoked for completed WAL segments. Hence, if your server generates little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To limit how old unarchived data can be, you can set `archive_timeout` to force the server to switch to a new WAL segment file periodically. When this parameter is greater than zero, the server will switch to a new segment file whenever this amount of time has elapsed since the last segment file switch, and there has been any database activity, including a single checkpoint (checkpoints are skipped if there is no database activity). Note that archived files that are closed early due to a forced switch are still the same length as completely full files. Therefore, it is unwise to use a very short `archive_timeout` — it will bloat your archive storage. `archive_timeout` settings of a minute or so are usually reasonable. You should consider using streaming replication, instead of archiving, if you want data to be copied off the primary server more quickly than that. If this value is specified without units, it is taken as seconds. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.5.4. Recovery

This section describes the settings that apply to recovery in general, affecting crash recovery, streaming replication and archive-based replication.

`recovery_prefetch` (`enum`)

Whether to try to prefetch blocks that are referenced in the WAL that are not yet in the buffer pool, during recovery. Valid values are `off`, `on` and `try` (the default). The setting `try` enables prefetching only if the operating system provides the `posix_fadvise` function, which is currently used to implement prefetching. Note that some operating systems provide the function, but it doesn't do anything.

Prefetching blocks that will soon be needed can reduce I/O wait times during recovery with some workloads. See also the wal_decode_buffer_size and maintenance_io_concurrency settings, which limit prefetching activity.

`wal_decode_buffer_size` (`integer`)

A limit on how far ahead the server can look in the WAL, to find blocks to prefetch. If this value is specified without units, it is taken as bytes. The default is 512kB.

19.5.5. Archive Recovery

This section describes the settings that apply only for the duration of the recovery. They must be reset for any subsequent recovery you wish to perform.

“Recovery” covers using the server as a standby or for executing a targeted recovery. Typically, standby mode would be used to provide high availability and/or read scalability, whereas a targeted recovery is used to recover from data loss.

To start the server in standby mode, create a file called `standby.signal` in the data directory. The server will enter recovery and will not stop recovery when the end of archived WAL is reached, but will keep trying to continue recovery by connecting to the sending server as specified by the `primary_conninfo` setting and/or by fetching new WAL segments using `restore_command`. For this mode, the parameters from this section and Section 19.6.3 are of interest. Parameters from Section 19.5.6 will also be applied but are typically not useful in this mode.

To start the server in targeted recovery mode, create a file called `recovery.signal` in the data directory. If both `standby.signal` and `recovery.signal` files are created, standby mode takes precedence. Targeted recovery mode ends when the archived WAL is fully replayed, or when `re-`

covery_target is reached. In this mode, the parameters from both this section and Section 19.5.6 will be used.

restore_command (string)

The local shell command to execute to retrieve an archived segment of the WAL file series. This parameter is required for archive recovery, but optional for streaming replication. Any %f in the string is replaced by the name of the file to retrieve from the archive, and any %p is replaced by the copy destination path name on the server. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Any %r is replaced by the name of the file containing the last valid restart point. That is the earliest file that must be kept to allow a restore to be restartable, so this information can be used to truncate the archive to just the minimum required to support restarting from the current restore. %r is typically only used by warm-standby configurations (see Section 26.2). Write %% to embed an actual % character.

It is important for the command to return a zero exit status only if it succeeds. The command *will* be asked for file names that are not present in the archive; it must return nonzero when so asked. Examples:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  #
 Windows
```

An exception is that if the command was terminated by a signal (other than SIGTERM, which is used as part of a database server shutdown) or an error by the shell (such as command not found), then recovery will abort and the server will not start up.

This parameter can only be set in the postgresql.conf file or on the server command line.

archive_cleanup_command (string)

This optional parameter specifies a shell command that will be executed at every restartpoint. The purpose of archive_cleanup_command is to provide a mechanism for cleaning up old archived WAL files that are no longer needed by the standby server. Any %r is replaced by the name of the file containing the last valid restart point. That is the earliest file that must be *kept* to allow a restore to be restartable, and so all files earlier than %r may be safely removed. This information can be used to truncate the archive to just the minimum required to support restart from the current restore. The pg_archivecleanup module is often used in archive_cleanup_command for single-standby configurations, for example:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/
archivedir %r'
```

Note however that if multiple standby servers are restoring from the same archive directory, you will need to ensure that you do not delete WAL files until they are no longer needed by any of the servers. archive_cleanup_command would typically be used in a warm-standby configuration (see Section 26.2). Write %% to embed an actual % character in the command.

If the command returns a nonzero exit status then a warning log message will be written. An exception is that if the command was terminated by a signal or an error by the shell (such as command not found), a fatal error will be raised.

This parameter can only be set in the postgresql.conf file or on the server command line.

recovery_end_command (string)

This parameter specifies a shell command that will be executed once only at the end of recovery. This parameter is optional. The purpose of the recovery_end_command is to provide a mechanism for cleanup following replication or recovery. Any %r is replaced by the name of the file containing the last valid restart point, like in archive_cleanup_command.

If the command returns a nonzero exit status then a warning log message will be written and the database will proceed to start up anyway. An exception is that if the command was terminated by a signal or an error by the shell (such as command not found), the database will not proceed with startup.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.5.6. Recovery Target

By default, recovery will recover to the end of the WAL log. The following parameters can be used to specify an earlier stopping point. At most one of `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time`, or `recovery_target_xid` can be used; if more than one of these is specified in the configuration file, an error will be raised. These parameters can only be set at server start.

`recovery_target = 'immediate'`

This parameter specifies that recovery should end as soon as a consistent state is reached, i.e., as early as possible. When restoring from an online backup, this means the point where taking the backup ended.

Technically, this is a string parameter, but `'immediate'` is currently the only allowed value.

`recovery_target_name` (`string`)

This parameter specifies the named restore point (created with `pg_create_restore_point()`) to which recovery will proceed.

`recovery_target_time` (`timestamp`)

This parameter specifies the time stamp up to which recovery will proceed. The precise stopping point is also influenced by recovery_target_inclusive.

The value of this parameter is a time stamp in the same format accepted by the `timestamp with time zone` data type, except that you cannot use a time zone abbreviation (unless the timezone_abbreviations variable has been set earlier in the configuration file). Preferred style is to use a numeric offset from UTC, or you can write a full time zone name, e.g., `Europe/Helsinki` not `EEST`.

`recovery_target_xid` (`string`)

This parameter specifies the transaction ID up to which recovery will proceed. Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that will be recovered are those that committed before (and optionally including) the specified one. The precise stopping point is also influenced by recovery_target_inclusive.

`recovery_target_lsn` (`pg_lsn`)

This parameter specifies the LSN of the write-ahead log location up to which recovery will proceed. The precise stopping point is also influenced by recovery_target_inclusive. This parameter is parsed using the system data type `pg_lsn`.

The following options further specify the recovery target, and affect what happens when the target is reached:

`recovery_target_inclusive` (`boolean`)

Specifies whether to stop just after the specified recovery target (`on`), or just before the recovery target (`off`). Applies when recovery_target_lsn, recovery_target_time, or recovery_target_xid

is specified. This setting controls whether transactions having exactly the target WAL location (LSN), commit time, or transaction ID, respectively, will be included in the recovery. Default is `on`.

`recovery_target_timeline` (`string`)

Specifies recovering into a particular timeline. The value can be a numeric timeline ID or a special value. The value `current` recovers along the same timeline that was current when the base backup was taken. The value `latest` recovers to the latest timeline found in the archive, which is useful in a standby server. `latest` is the default.

To specify a timeline ID in hexadecimal (for example, if extracted from a WAL file name or history file), prefix it with a `0x`. For instance, if the WAL file name is `00000011000000A10000004F`, then the timeline ID is `0x11` (or 17 decimal).

You usually only need to set this parameter in complex re-recovery situations, where you need to return to a state that itself was reached after a point-in-time recovery. See Section 25.3.6 for discussion.

`recovery_target_action` (`enum`)

Specifies what action the server should take once the recovery target is reached. The default is `pause`, which means recovery will be paused. `promote` means the recovery process will finish and the server will start to accept connections. Finally `shutdown` will stop the server after reaching the recovery target.

The intended use of the `pause` setting is to allow queries to be executed against the database to check if this recovery target is the most desirable point for recovery. The paused state can be resumed by using `pg_wal_replay_resume()` (see Table 9.97), which then causes recovery to end. If this recovery target is not the desired stopping point, then shut down the server, change the recovery target settings to a later target and restart to continue recovery.

The `shutdown` setting is useful to have the instance ready at the exact replay point desired. The instance will still be able to replay more WAL records (and in fact will have to replay WAL records since the last checkpoint next time it is started).

Note that because `recovery.signal` will not be removed when `recovery_target_action` is set to `shutdown`, any subsequent start will end with immediate shutdown unless the configuration is changed or the `recovery.signal` file is removed manually.

This setting has no effect if no recovery target is set. If hot_standby is not enabled, a setting of `pause` will act the same as `shutdown`. If the recovery target is reached while a promotion is ongoing, a setting of `pause` will act the same as `promote`.

In any case, if a recovery target is configured but the archive recovery ends before the target is reached, the server will shut down with a fatal error.

19.5.7. WAL Summarization

These settings control WAL summarization, a feature which must be enabled in order to perform an incremental backup.

`summarize_wal` (`boolean`)

Enables the WAL summarizer process. Note that WAL summarization can be enabled either on a primary or on a standby. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `off`.

The server cannot be started with `summarize_wal=on` if `wal_level` is set to `minimal`. If `summarize_wal=on` is configured after server startup while `wal_level=mini-`

mal, the summarizer will run but refuse to generate summary files for any WAL generated with wal_level=minimal.

wal_summary_keep_time (integer)

Configures the amount of time after which the WAL summarizer automatically removes old WAL summaries. The file timestamp is used to determine which files are old enough to remove. Typically, you should set this comfortably higher than the time that could pass between a backup and a later incremental backup that depends on it. WAL summaries must be available for the entire range of WAL records between the preceding backup and the new one being taken; if not, the incremental backup will fail. If this parameter is set to zero, WAL summaries will not be automatically deleted, but it is safe to manually remove files that you know will not be required for future incremental backups. This parameter can only be set in the postgresql.conf file or on the server command line. If this value is specified without units, it is taken as minutes. The default is 10 days. If summarize_wal = off, existing WAL summaries will not be removed regardless of the value of this parameter, because the WAL summarizer will not run.

19.6. Replication

These settings control the behavior of the built-in *streaming replication* feature (see Section 26.2.5), and the built-in *logical replication* feature (see Chapter 29).

For *streaming replication*, servers will be either a primary or a standby server. Primaries can send data, while standbys are always receivers of replicated data. When cascading replication (see Section 26.2.7) is used, standby servers can also be senders, as well as receivers. Parameters are mainly for sending and standby servers, though some parameters have meaning only on the primary server. Settings may vary across the cluster without problems if that is required.

For *logical replication*, *publishers* (servers that do CREATE PUBLICATION) replicate data to *subscribers* (servers that do CREATE SUBSCRIPTION). Servers can also be publishers and subscribers at the same time. Note, the following sections refer to publishers as "senders". For more details about logical replication configuration settings refer to Section 29.11.

19.6.1. Sending Servers

These parameters can be set on any server that is to send replication data to one or more standby servers. The primary is always a sending server, so these parameters must always be set on the primary. The role and meaning of these parameters does not change after a standby becomes the primary.

max_wal_senders (integer)

Specifies the maximum number of concurrent connections from standby servers or streaming base backup clients (i.e., the maximum number of simultaneously running WAL sender processes). The default is 10. The value 0 means replication is disabled. Abrupt disconnection of a streaming client might leave an orphaned connection slot behind until a timeout is reached, so this parameter should be set slightly higher than the maximum number of expected clients so disconnected clients can immediately reconnect. This parameter can only be set at server start. Also, wal_level must be set to replica or higher to allow connections from standby servers.

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

max_replication_slots (integer)

Specifies the maximum number of replication slots (see Section 26.2.6) that the server can support. The default is 10. This parameter can only be set at server start. Setting it to a lower value than the number of currently existing replication slots will prevent the server from starting. Also, wal_level must be set to replica or higher to allow replication slots to be used.

Note that this parameter also applies on the subscriber side, but with a different meaning.

`wal_keep_size` (`integer`)

Specifies the minimum size of past WAL files kept in the `pg_wal` directory, in case a standby server needs to fetch them for streaming replication. If a standby server connected to the sending server falls behind by more than `wal_keep_size` megabytes, the sending server might remove a WAL segment still needed by the standby, in which case the replication connection will be terminated. Downstream connections will also eventually fail as a result. (However, the standby server can recover by fetching the segment from archive, if WAL archiving is in use.)

This sets only the minimum size of segments retained in `pg_wal`; the system might need to retain more segments for WAL archival or to recover from a checkpoint. If `wal_keep_size` is zero (the default), the system doesn't keep any extra segments for standby purposes, so the number of old WAL segments available to standby servers is a function of the location of the previous checkpoint and status of WAL archiving. If this value is specified without units, it is taken as megabytes. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`max_slot_wal_keep_size` (`integer`)

Specify the maximum size of WAL files that replication slots are allowed to retain in the `pg_wal` directory at checkpoint time. If `max_slot_wal_keep_size` is -1 (the default), replication slots may retain an unlimited amount of WAL files. Otherwise, if restart_lsn of a replication slot falls behind the current LSN by more than the given size, the standby using the slot may no longer be able to continue replication due to removal of required WAL files. You can see the WAL availability of replication slots in pg_replication_slots. If this value is specified without units, it is taken as megabytes. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`wal_sender_timeout` (`integer`)

Terminate replication connections that are inactive for longer than this amount of time. This is useful for the sending server to detect a standby crash or network outage. If this value is specified without units, it is taken as milliseconds. The default value is 60 seconds. A value of zero disables the timeout mechanism.

With a cluster distributed across multiple geographic locations, using different values per location brings more flexibility in the cluster management. A smaller value is useful for faster failure detection with a standby having a low-latency network connection, and a larger value helps in judging better the health of a standby if located on a remote location, with a high-latency network connection.

`track_commit_timestamp` (`boolean`)

Record commit time of transactions. This parameter can only be set in `postgresql.conf` file or on the server command line. The default value is `off`.

`synchronized_standby_slots` (`string`)

A comma-separated list of streaming replication standby server slot names that logical WAL sender processes will wait for. Logical WAL sender processes will send decoded changes to plugins only after the specified replication slots confirm receiving WAL. This guarantees that logical replication failover slots do not consume changes until those changes are received and flushed to corresponding physical standbys. If a logical replication connection is meant to switch to a physical standby after the standby is promoted, the physical replication slot for the standby should be listed here. Note that logical replication will not proceed if the slots specified in the `synchronized_standby_slots` do not exist or are invalidated. Additionally, the replication management functions `pg_replication_slot_advance`, `pg_logical_slot_get_changes`, and `pg_logical_slot_peek_changes`, when used with logical

failover slots, will block until all physical slots specified in `synchronized_standby_slots` have confirmed WAL receipt.

The standbys corresponding to the physical replication slots in `synchronized_standby_slots` must configure `sync_replication_slots = true` so they can receive logical failover slot changes from the primary.

19.6.2. Primary Server

These parameters can be set on the primary server that is to send replication data to one or more standby servers. Note that in addition to these parameters, wal_level must be set appropriately on the primary server, and optionally WAL archiving can be enabled as well (see Section 19.5.3). The values of these parameters on standby servers are irrelevant, although you may wish to set them there in preparation for the possibility of a standby becoming the primary.

`synchronous_standby_names` (`string`)

Specifies a list of standby servers that can support *synchronous replication*, as described in Section 26.2.8. There will be one or more active synchronous standbys; transactions waiting for commit will be allowed to proceed after these standby servers confirm receipt of their data. The synchronous standbys will be those whose names appear in this list, and that are both currently connected and streaming data in real-time (as shown by a state of `streaming` in the `pg_stat_replication` view). Specifying more than one synchronous standby can allow for very high availability and protection against data loss.

The name of a standby server for this purpose is the `application_name` setting of the standby, as set in the standby's connection information. In case of a physical replication standby, this should be set in the `primary_conninfo` setting; the default is the setting of cluster_name if set, else `walreceiver`. For logical replication, this can be set in the connection information of the subscription, and it defaults to the subscription name. For other replication stream consumers, consult their documentation.

This parameter specifies a list of standby servers using either of the following syntaxes:

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

where *`num_sync`* is the number of synchronous standbys that transactions need to wait for replies from, and *`standby_name`* is the name of a standby server. `FIRST` and `ANY` specify the method to choose synchronous standbys from the listed servers.

The keyword `FIRST`, coupled with *`num_sync`*, specifies a priority-based synchronous replication and makes transaction commits wait until their WAL records are replicated to *`num_sync`* synchronous standbys chosen based on their priorities. For example, a setting of `FIRST 3 (s1, s2, s3, s4)` will cause each commit to wait for replies from three higher-priority standbys chosen from standby servers `s1`, `s2`, `s3` and `s4`. The standbys whose names appear earlier in the list are given higher priority and will be considered as synchronous. Other standby servers appearing later in this list represent potential synchronous standbys. If any of the current synchronous standbys disconnects for whatever reason, it will be replaced immediately with the next-highest-priority standby. The keyword `FIRST` is optional.

The keyword `ANY`, coupled with *`num_sync`*, specifies a quorum-based synchronous replication and makes transaction commits wait until their WAL records are replicated to *at least* *`num_sync`* listed standbys. For example, a setting of `ANY 3 (s1, s2, s3, s4)` will cause each commit to proceed as soon as at least any three standbys of `s1`, `s2`, `s3` and `s4` reply.

`FIRST` and `ANY` are case-insensitive. If these keywords are used as the name of a standby server, its *`standby_name`* must be double-quoted.

The third syntax was used before PostgreSQL version 9.6 and is still supported. It's the same as the first syntax with `FIRST` and *`num_sync`* equal to 1. For example, `FIRST 1 (s1, s2)` and `s1, s2` have the same meaning: either `s1` or `s2` is chosen as a synchronous standby.

The special entry `*` matches any standby name.

There is no mechanism to enforce uniqueness of standby names. In case of duplicates one of the matching standbys will be considered as higher priority, though exactly which one is indeterminate.

> **Note**
>
> Each *`standby_name`* should have the form of a valid SQL identifier, unless it is `*`. You can use double-quoting if necessary. But note that *`standby_name`*s are compared to standby application names case-insensitively, whether double-quoted or not.

If no synchronous standby names are specified here, then synchronous replication is not enabled and transaction commits will not wait for replication. This is the default configuration. Even when synchronous replication is enabled, individual transactions can be configured not to wait for replication by setting the synchronous_commit parameter to `local` or `off`.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.6.3. Standby Servers

These settings control the behavior of a standby server that is to receive replication data. Their values on the primary server are irrelevant.

`primary_conninfo` (`string`)

Specifies a connection string to be used for the standby server to connect with a sending server. This string is in the format described in Section 32.1.1. If any option is unspecified in this string, then the corresponding environment variable (see Section 32.15) is checked. If the environment variable is not set either, then defaults are used.

The connection string should specify the host name (or address) of the sending server, as well as the port number if it is not the same as the standby server's default. Also specify a user name corresponding to a suitably-privileged role on the sending server (see Section 26.2.5.1). A password needs to be provided too, if the sender demands password authentication. It can be provided in the `primary_conninfo` string, or in a separate `~/.pgpass` file on the standby server (use `replication` as the database name).

For replication slot synchronization (see Section 47.2.3), it is also necessary to specify a valid `dbname` in the `primary_conninfo` string. This will only be used for slot synchronization. It is ignored for streaming.

This parameter can only be set in the `postgresql.conf` file or on the server command line. If this parameter is changed while the WAL receiver process is running, that process is signaled to shut down and expected to restart with the new setting (except if `primary_conninfo` is an empty string). This setting has no effect if the server is not in standby mode.

`primary_slot_name` (`string`)

Optionally specifies an existing replication slot to be used when connecting to the sending server via streaming replication to control resource removal on the upstream node (see Section 26.2.6). This parameter can only be set in the `postgresql.conf` file or on the server command line. If this parameter is changed while the WAL receiver process is running, that process is signaled

to shut down and expected to restart with the new setting. This setting has no effect if primary_conninfo is not set or the server is not in standby mode.

hot_standby (boolean)

Specifies whether or not you can connect and run queries during recovery, as described in Section 26.4. The default value is on. This parameter can only be set at server start. It only has effect during archive recovery or in standby mode.

max_standby_archive_delay (integer)

When hot standby is active, this parameter determines how long the standby server should wait before canceling standby queries that conflict with about-to-be-applied WAL entries, as described in Section 26.4.2. max_standby_archive_delay applies when WAL data is being read from WAL archive (and is therefore not current). If this value is specified without units, it is taken as milliseconds. The default is 30 seconds. A value of -1 allows the standby to wait forever for conflicting queries to complete. This parameter can only be set in the postgresql.conf file or on the server command line.

Note that max_standby_archive_delay is not the same as the maximum length of time a query can run before cancellation; rather it is the maximum total time allowed to apply any one WAL segment's data. Thus, if one query has resulted in significant delay earlier in the WAL segment, subsequent conflicting queries will have much less grace time.

max_standby_streaming_delay (integer)

When hot standby is active, this parameter determines how long the standby server should wait before canceling standby queries that conflict with about-to-be-applied WAL entries, as described in Section 26.4.2. max_standby_streaming_delay applies when WAL data is being received via streaming replication. If this value is specified without units, it is taken as milliseconds. The default is 30 seconds. A value of -1 allows the standby to wait forever for conflicting queries to complete. This parameter can only be set in the postgresql.conf file or on the server command line.

Note that max_standby_streaming_delay is not the same as the maximum length of time a query can run before cancellation; rather it is the maximum total time allowed to apply WAL data once it has been received from the primary server. Thus, if one query has resulted in significant delay, subsequent conflicting queries will have much less grace time until the standby server has caught up again.

wal_receiver_create_temp_slot (boolean)

Specifies whether the WAL receiver process should create a temporary replication slot on the remote instance when no permanent replication slot to use has been configured (using primary_slot_name). The default is off. This parameter can only be set in the postgresql.conf file or on the server command line. If this parameter is changed while the WAL receiver process is running, that process is signaled to shut down and expected to restart with the new setting.

wal_receiver_status_interval (integer)

Specifies the minimum frequency for the WAL receiver process on the standby to send information about replication progress to the primary or upstream standby, where it can be seen using the pg_stat_replication view. The standby will report the last write-ahead log location it has written, the last position it has flushed to disk, and the last position it has applied. This parameter's value is the maximum amount of time between reports. Updates are sent each time the write or flush positions change, or as often as specified by this parameter if set to a non-zero value. There are additional cases where updates are sent while ignoring this parameter; for example, when processing of the existing WAL completes or when synchronous_commit is set to remote_apply. Thus, the apply position may lag slightly behind the true position. If this value is specified without units, it is taken as seconds. The default value is 10 seconds. This parameter can only be set in the postgresql.conf file or on the server command line.

`hot_standby_feedback` (`boolean`)

Specifies whether or not a hot standby will send feedback to the primary or upstream standby about queries currently executing on the standby. This parameter can be used to eliminate query cancels caused by cleanup records, but can cause database bloat on the primary for some workloads. Feedback messages will not be sent more frequently than once per `wal_receiver_status_interval`. The default value is `off`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If cascaded replication is in use the feedback is passed upstream until it eventually reaches the primary. Standbys make no other use of feedback they receive other than to pass upstream.

`wal_receiver_timeout` (`integer`)

Terminate replication connections that are inactive for longer than this amount of time. This is useful for the receiving standby server to detect a primary node crash or network outage. If this value is specified without units, it is taken as milliseconds. The default value is 60 seconds. A value of zero disables the timeout mechanism. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`wal_retrieve_retry_interval` (`integer`)

Specifies how long the standby server should wait when WAL data is not available from any sources (streaming replication, local `pg_wal` or WAL archive) before trying again to retrieve WAL data. If this value is specified without units, it is taken as milliseconds. The default value is 5 seconds. This parameter can only be set in the `postgresql.conf` file or on the server command line.

This parameter is useful in configurations where a node in recovery needs to control the amount of time to wait for new WAL data to be available. For example, in archive recovery, it is possible to make the recovery more responsive in the detection of a new WAL file by reducing the value of this parameter. On a system with low WAL activity, increasing it reduces the amount of requests necessary to access WAL archives, something useful for example in cloud environments where the number of times an infrastructure is accessed is taken into account.

In logical replication, this parameter also limits how often a failing replication apply worker will be respawned.

`recovery_min_apply_delay` (`integer`)

By default, a standby server restores WAL records from the sending server as soon as possible. It may be useful to have a time-delayed copy of the data, offering opportunities to correct data loss errors. This parameter allows you to delay recovery by a specified amount of time. For example, if you set this parameter to `5min`, the standby will replay each transaction commit only when the system time on the standby is at least five minutes past the commit time reported by the primary. If this value is specified without units, it is taken as milliseconds. The default is zero, adding no delay.

It is possible that the replication delay between servers exceeds the value of this parameter, in which case no delay is added. Note that the delay is calculated between the WAL time stamp as written on primary and the current time on the standby. Delays in transfer because of network lag or cascading replication configurations may reduce the actual wait time significantly. If the system clocks on primary and standby are not synchronized, this may lead to recovery applying records earlier than expected; but that is not a major issue because useful settings of this parameter are much larger than typical time deviations between servers.

The delay occurs only on WAL records for transaction commits. Other records are replayed as quickly as possible, which is not a problem because MVCC visibility rules ensure their effects are not visible until the corresponding commit record is applied.

The delay occurs once the database in recovery has reached a consistent state, until the standby is promoted or triggered. After that the standby will end recovery without further waiting.

WAL records must be kept on the standby until they are ready to be applied. Therefore, longer delays will result in a greater accumulation of WAL files, increasing disk space requirements for the standby's `pg_wal` directory.

This parameter is intended for use with streaming replication deployments; however, if the parameter is specified it will be honored in all cases except crash recovery. `hot_standby_feedback` will be delayed by use of this feature which could lead to bloat on the primary; use both together with care.

> **Warning**
>
> Synchronous replication is affected by this setting when `synchronous_commit` is set to `remote_apply`; every `COMMIT` will need to wait to be applied.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`sync_replication_slots` (`boolean`)

It enables a physical standby to synchronize logical failover slots from the primary server so that logical subscribers can resume replication from the new primary server after failover.

It is disabled by default. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.6.4. Subscribers

These settings control the behavior of a logical replication subscriber. Their values on the publisher are irrelevant. See Section 29.11 for more details.

`max_replication_slots` (`integer`)

Specifies how many replication origins (see Chapter 48) can be tracked simultaneously, effectively limiting how many logical replication subscriptions can be created on the server. Setting it to a lower value than the current number of tracked replication origins (reflected in pg_replication_origin_status) will prevent the server from starting. `max_replication_slots` must be set to at least the number of subscriptions that will be added to the subscriber, plus some reserve for table synchronization.

Note that this parameter also applies on a sending server, but with a different meaning.

`max_logical_replication_workers` (`integer`)

Specifies maximum number of logical replication workers. This includes leader apply workers, parallel apply workers, and table synchronization workers.

Logical replication workers are taken from the pool defined by `max_worker_processes`.

The default value is 4. This parameter can only be set at server start.

`max_sync_workers_per_subscription` (`integer`)

Maximum number of synchronization workers per subscription. This parameter controls the amount of parallelism of the initial data copy during the subscription initialization or when new tables are added.

Currently, there can be only one synchronization worker per table.

The synchronization workers are taken from the pool defined by `max_logical_replication_workers`.

The default value is 2. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`max_parallel_apply_workers_per_subscription` (`integer`)

Maximum number of parallel apply workers per subscription. This parameter controls the amount of parallelism for streaming of in-progress transactions with subscription parameter `streaming = parallel`.

The parallel apply workers are taken from the pool defined by `max_logical_replication_workers`.

The default value is 2. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.7. Query Planning

19.7.1. Planner Method Configuration

These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a *temporary* solution is to use one of these configuration parameters to force the optimizer to choose a different plan. Better ways to improve the quality of the plans chosen by the optimizer include adjusting the planner cost constants (see Section 19.7.2), running `ANALYZE` manually, increasing the value of the default_statistics_target configuration parameter, and increasing the amount of statistics collected for specific columns using `ALTER TABLE SET STATISTICS`.

`enable_async_append` (`boolean`)

Enables or disables the query planner's use of async-aware append plan types. The default is `on`.

`enable_bitmapscan` (`boolean`)

Enables or disables the query planner's use of bitmap-scan plan types. The default is `on`.

`enable_gathermerge` (`boolean`)

Enables or disables the query planner's use of gather merge plan types. The default is `on`.

`enable_group_by_reordering` (`boolean`)

Controls if the query planner will produce a plan which will provide `GROUP BY` keys sorted in the order of keys of a child node of the plan, such as an index scan. When disabled, the query planner will produce a plan with `GROUP BY` keys only sorted to match the `ORDER BY` clause, if any. When enabled, the planner will try to produce a more efficient plan. The default value is `on`.

`enable_hashagg` (`boolean`)

Enables or disables the query planner's use of hashed aggregation plan types. The default is `on`.

`enable_hashjoin` (`boolean`)

Enables or disables the query planner's use of hash-join plan types. The default is `on`.

`enable_incremental_sort` (`boolean`)

Enables or disables the query planner's use of incremental sort steps. The default is `on`.

`enable_indexscan` (`boolean`)

Enables or disables the query planner's use of index-scan and index-only-scan plan types. The default is `on`. Also see enable_indexonlyscan.

`enable_indexonlyscan` (`boolean`)

Enables or disables the query planner's use of index-only-scan plan types (see Section 11.9). The default is `on`. The enable_indexscan setting must also be enabled to have the query planner consider index-only-scans.

`enable_material` (`boolean`)

Enables or disables the query planner's use of materialization. It is impossible to suppress materialization entirely, but turning this variable off prevents the planner from inserting materialize nodes except in cases where it is required for correctness. The default is `on`.

`enable_memoize` (`boolean`)

Enables or disables the query planner's use of memoize plans for caching results from parameterized scans inside nested-loop joins. This plan type allows scans to the underlying plans to be skipped when the results for the current parameters are already in the cache. Less commonly looked up results may be evicted from the cache when more space is required for new entries. The default is `on`.

`enable_mergejoin` (`boolean`)

Enables or disables the query planner's use of merge-join plan types. The default is `on`.

`enable_nestloop` (`boolean`)

Enables or disables the query planner's use of nested-loop join plans. It is impossible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_parallel_append` (`boolean`)

Enables or disables the query planner's use of parallel-aware append plan types. The default is `on`.

`enable_parallel_hash` (`boolean`)

Enables or disables the query planner's use of hash-join plan types with parallel hash. Has no effect if hash-join plans are not also enabled. The default is `on`.

`enable_partition_pruning` (`boolean`)

Enables or disables the query planner's ability to eliminate a partitioned table's partitions from query plans. This also controls the planner's ability to generate query plans which allow the query executor to remove (ignore) partitions during query execution. The default is `on`. See Section 5.12.4 for details.

`enable_partitionwise_join` (`boolean`)

Enables or disables the query planner's use of partitionwise join, which allows a join between partitioned tables to be performed by joining the matching partitions. Partitionwise join currently applies only when the join conditions include all the partition keys, which must be of the same data type and have one-to-one matching sets of child partitions. With this setting enabled, the number of nodes whose memory usage is restricted by `work_mem` appearing in the final plan can increase linearly according to the number of partitions being scanned. This can result in a large increase in overall memory consumption during the execution of the query. Query planning also becomes significantly more expensive in terms of memory and CPU. The default value is `off`.

`enable_partitionwise_aggregate` (`boolean`)

Enables or disables the query planner's use of partitionwise grouping or aggregation, which allows grouping or aggregation on partitioned tables to be performed separately for each partition. If the `GROUP BY` clause does not include the partition keys, only partial aggregation can be performed on a per-partition basis, and finalization must be performed later. With this setting enabled, the number of nodes whose memory usage is restricted by `work_mem` appearing in the final plan can increase linearly according to the number of partitions being scanned. This can result in a large increase in overall memory consumption during the execution of the query. Query planning also becomes significantly more expensive in terms of memory and CPU. The default value is `off`.

`enable_presorted_aggregate` (`boolean`)

Controls if the query planner will produce a plan which will provide rows which are presorted in the order required for the query's `ORDER BY` / `DISTINCT` aggregate functions. When disabled, the query planner will produce a plan which will always require the executor to perform a sort before performing aggregation of each aggregate function containing an `ORDER BY` or `DISTINCT` clause. When enabled, the planner will try to produce a more efficient plan which provides input to the aggregate functions which is presorted in the order they require for aggregation. The default value is `on`.

`enable_seqscan` (`boolean`)

Enables or disables the query planner's use of sequential scan plan types. It is impossible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_sort` (`boolean`)

Enables or disables the query planner's use of explicit sort steps. It is impossible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_tidscan` (`boolean`)

Enables or disables the query planner's use of TID scan plan types. The default is `on`.

19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

> **Note**
>
> Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the following parameters.

Random access to mechanical disk storage is normally much more expensive than four times sequential access. However, a lower default is used (4.0) because the majority of random accesses to disk, such as indexed reads, are assumed to be in cache. The default value can be thought of as modeling random access as 40 times slower than sequential, while expecting 90% of random reads to be cached.

If you believe a 90% cache rate is an incorrect assumption for your workload, you can increase random_page_cost to better reflect the true cost of random storage reads. Correspondingly, if your data is likely to be completely in cache, such as when the database is smaller than the total server memory, decreasing random_page_cost can be appropriate. Storage that has a low random read cost relative to sequential, e.g., solid-state drives, might also be better modeled with a lower value for random_page_cost, e.g., `1.1`.

> **Tip**
>
> Although the system will let you set `random_page_cost` to less than `seq_page_cost`, it is not physically sensible to do so. However, setting them equal makes sense if the database is entirely cached in RAM, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

`cpu_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each row during a query. The default is 0.01.

`cpu_index_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each index entry during an index scan. The default is 0.005.

`cpu_operator_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each operator or function executed during a query. The default is 0.0025.

`parallel_setup_cost` (`floating point`)

Sets the planner's estimate of the cost of launching parallel worker processes. The default is 1000.

`parallel_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of transferring one tuple from a parallel worker process to another process. The default is 0.1.

`min_parallel_table_scan_size` (`integer`)

Sets the minimum amount of table data that must be scanned in order for a parallel scan to be considered. For a parallel sequential scan, the amount of table data scanned is always equal to the

size of the table, but when indexes are used the amount of table data scanned will normally be less. If this value is specified without units, it is taken as blocks, that is BLCKSZ bytes, typically 8kB. The default is 8 megabytes (8MB).

min_parallel_index_scan_size (integer)

Sets the minimum amount of index data that must be scanned in order for a parallel scan to be considered. Note that a parallel index scan typically won't touch the entire index; it is the number of pages which the planner believes will actually be touched by the scan which is relevant. This parameter is also used to decide whether a particular index can participate in a parallel vacuum. See VACUUM. If this value is specified without units, it is taken as blocks, that is BLCKSZ bytes, typically 8kB. The default is 512 kilobytes (512kB).

effective_cache_size (integer)

Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL's shared buffers and the portion of the kernel's disk cache that will be used for PostgreSQL data files, though some data might exist in both places. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes. The system also does not assume data remains in the disk cache between queries. If this value is specified without units, it is taken as blocks, that is BLCKSZ bytes, typically 8kB. The default is 4 gigabytes (4GB). (If BLCKSZ is not 8kB, the default value scales proportionally to it.)

jit_above_cost (floating point)

Sets the query cost above which JIT compilation is activated, if enabled (see Chapter 30). Performing JIT costs planning time but can accelerate query execution. Setting this to -1 disables JIT compilation. The default is 100000.

jit_inline_above_cost (floating point)

Sets the query cost above which JIT compilation attempts to inline functions and operators. Inlining adds planning time, but can improve execution speed. It is not meaningful to set this to less than jit_above_cost. Setting this to -1 disables inlining. The default is 500000.

jit_optimize_above_cost (floating point)

Sets the query cost above which JIT compilation applies expensive optimizations. Such optimization adds planning time, but can improve execution speed. It is not meaningful to set this to less than jit_above_cost, and it is unlikely to be beneficial to set it to more than jit_inline_above_cost. Setting this to -1 disables expensive optimizations. The default is 500000.

19.7.3. Genetic Query Optimizer

The genetic query optimizer (GEQO) is an algorithm that does query planning using heuristic searching. This reduces planning time for complex queries (those joining many relations), at the cost of producing plans that are sometimes inferior to those found by the normal exhaustive-search algorithm. For more information see Chapter 60.

geqo (boolean)

Enables or disables genetic query optimization. This is on by default. It is usually best not to turn it off in production; the geqo_threshold variable provides more granular control of GEQO.

`geqo_threshold` (`integer`)

Use genetic query optimization to plan queries with at least this many `FROM` items involved. (Note that a `FULL OUTER JOIN` construct counts as only one `FROM` item.) The default is 12. For simpler queries it is usually best to use the regular, exhaustive-search planner, but for queries with many tables the exhaustive search takes too long, often longer than the penalty of executing a suboptimal plan. Thus, a threshold on the size of the query is a convenient way to manage use of GEQO.

`geqo_effort` (`integer`)

Controls the trade-off between planning time and query plan quality in GEQO. This variable must be an integer in the range from 1 to 10. The default value is five. Larger values increase the time spent doing query planning, but also increase the likelihood that an efficient query plan will be chosen.

`geqo_effort` doesn't actually do anything directly; it is only used to compute the default values for the other variables that influence GEQO behavior (described below). If you prefer, you can set the other parameters by hand instead.

`geqo_pool_size` (`integer`)

Controls the pool size used by GEQO, that is the number of individuals in the genetic population. It must be at least two, and useful values are typically 100 to 1000. If it is set to zero (the default setting) then a suitable value is chosen based on `geqo_effort` and the number of tables in the query.

`geqo_generations` (`integer`)

Controls the number of generations used by GEQO, that is the number of iterations of the algorithm. It must be at least one, and useful values are in the same range as the pool size. If it is set to zero (the default setting) then a suitable value is chosen based on `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Controls the selection bias used by GEQO. The selection bias is the selective pressure within the population. Values can be from 1.50 to 2.00; the latter is the default.

`geqo_seed` (`floating point`)

Controls the initial value of the random number generator used by GEQO to select random paths through the join order search space. The value can range from zero (the default) to one. Varying the value changes the set of join paths explored, and may result in a better or worse best path being found.

19.7.4. Other Planner Options

`default_statistics_target` (`integer`)

Sets the default statistics target for table columns without a column-specific target set via `ALTER TABLE SET STATISTICS`. Larger values increase the time needed to do `ANALYZE`, but might improve the quality of the planner's estimates. The default is 100. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

`constraint_exclusion` (`enum`)

Controls the query planner's use of table constraints to optimize queries. The allowed values of `constraint_exclusion` are `on` (examine constraints for all tables), `off` (never examine constraints), and `partition` (examine constraints only for inheritance child tables and `UNION`

ALL subqueries). partition is the default setting. It is often used with traditional inheritance trees to improve performance.

When this parameter allows it for a particular table, the planner compares query conditions with the table's CHECK constraints, and omits scanning tables for which the conditions contradict the constraints. For example:

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999))
 INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999))
 INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

With constraint exclusion enabled, this SELECT will not scan child1000 at all, improving performance.

Currently, constraint exclusion is enabled by default only for cases that are often used to implement table partitioning via inheritance trees. Turning it on for all tables imposes extra planning overhead that is quite noticeable on simple queries, and most often will yield no benefit for simple queries. If you have no tables that are partitioned using traditional inheritance, you might prefer to turn it off entirely. (Note that the equivalent feature for partitioned tables is controlled by a separate parameter, enable_partition_pruning.)

Refer to Section 5.12.5 for more information on using constraint exclusion to implement partitioning.

`cursor_tuple_fraction` (floating point)

Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved. The default is 0.1. Smaller values of this setting bias the planner towards using “fast start” plans for cursors, which will retrieve the first few rows quickly while perhaps taking a long time to fetch all rows. Larger values put more emphasis on the total estimated time. At the maximum setting of 1.0, cursors are planned exactly like regular queries, considering only the total estimated time and not how soon the first rows might be delivered.

`from_collapse_limit` (integer)

The planner will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but might yield inferior query plans. The default is eight. For more information see Section 14.3.

Setting this value to geqo_threshold or more may trigger use of the GEQO planner, resulting in non-optimal plans. See Section 19.7.3.

`jit` (boolean)

Determines whether JIT compilation may be used by PostgreSQL, if available (see Chapter 30). The default is on.

`join_collapse_limit` (integer)

The planner will rewrite explicit JOIN constructs (except FULL JOINs) into lists of FROM items whenever a list of no more than this many items would result. Smaller values reduce planning time but might yield inferior query plans.

By default, this variable is set the same as from_collapse_limit, which is appropriate for most uses. Setting it to 1 prevents any reordering of explicit JOINs. Thus, the explicit join order specified in the query will be the actual order in which the relations are joined. Because the query planner does not always choose the optimal join order, advanced users can elect to temporarily

set this variable to 1, and then specify the join order they desire explicitly. For more information see Section 14.3.

Setting this value to geqo_threshold or more may trigger use of the GEQO planner, resulting in non-optimal plans. See Section 19.7.3.

`plan_cache_mode` (`enum`)

Prepared statements (either explicitly prepared or implicitly generated, for example by PL/pgSQL) can be executed using custom or generic plans. Custom plans are made afresh for each execution using its specific set of parameter values, while generic plans do not rely on the parameter values and can be re-used across executions. Thus, use of a generic plan saves planning time, but if the ideal plan depends strongly on the parameter values then a generic plan may be inefficient. The choice between these options is normally made automatically, but it can be overridden with `plan_cache_mode`. The allowed values are `auto` (the default), `force_custom_plan` and `force_generic_plan`. This setting is considered when a cached plan is to be executed, not when it is prepared. For more information see PREPARE.

`recursive_worktable_factor` (`floating point`)

Sets the planner's estimate of the average size of the working table of a recursive query, as a multiple of the estimated size of the initial non-recursive term of the query. This helps the planner choose the most appropriate method for joining the working table to the query's other tables. The default value is `10.0`. A smaller value such as `1.0` can be helpful when the recursion has low “fan-out” from one step to the next, as for example in shortest-path queries. Graph analytics queries may benefit from larger-than-default values.

19.8. Error Reporting and Logging

19.8.1. Where to Log

`log_destination` (`string`)

PostgreSQL supports several methods for logging server messages, including stderr, csvlog, jsonlog, and syslog. On Windows, eventlog is also supported. Set this parameter to a list of desired log destinations separated by commas. The default is to log to stderr only. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If csvlog is included in `log_destination`, log entries are output in “comma separated value” (CSV) format, which is convenient for loading logs into programs. See Section 19.8.4 for details. logging_collector must be enabled to generate CSV-format log output.

If jsonlog is included in `log_destination`, log entries are output in JSON format, which is convenient for loading logs into programs. See Section 19.8.5 for details. logging_collector must be enabled to generate JSON-format log output.

When either stderr, csvlog or jsonlog are included, the file `current_logfiles` is created to record the location of the log file(s) currently in use by the logging collector and the associated logging destination. This provides a convenient way to find the logs currently in use by the instance. Here is an example of this file's content:

```
stderr log/postgresql.log
csvlog log/postgresql.csv
jsonlog log/postgresql.json
```

`current_logfiles` is recreated when a new log file is created as an effect of rotation, and when `log_destination` is reloaded. It is removed when none of stderr, csvlog or jsonlog are included in `log_destination`, and when the logging collector is disabled.

> **Note**
>
> On most Unix systems, you will need to alter the configuration of your system's syslog daemon in order to make use of the syslog option for `log_destination`. PostgreSQL can log to syslog facilities `LOCAL0` through `LOCAL7` (see syslog_facility), but the default syslog configuration on most platforms will discard all such messages. You will need to add something like:
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> to the syslog daemon's configuration file to make it work.
>
> On Windows, when you use the `eventlog` option for `log_destination`, you should register an event source and its library with the operating system so that the Windows Event Viewer can display event log messages cleanly. See Section 18.12 for details.

`logging_collector` (`boolean`)

This parameter enables the *logging collector*, which is a background process that captures log messages sent to stderr and redirects them into log files. This approach is often more useful than logging to syslog, since some types of messages might not appear in syslog output. (One common example is dynamic-linker failure messages; another is error messages produced by scripts such as `archive_command`.) This parameter can only be set at server start.

> **Note**
>
> It is possible to log to stderr without using the logging collector; the log messages will just go to wherever the server's stderr is directed. However, that method is only suitable for low log volumes, since it provides no convenient way to rotate log files. Also, on some platforms not using the logging collector can result in lost or garbled log output, because multiple processes writing concurrently to the same log file can overwrite each other's output.

> **Note**
>
> The logging collector is designed to never lose messages. This means that in case of extremely high load, server processes could be blocked while trying to send additional log messages when the collector has fallen behind. In contrast, syslog prefers to drop messages if it cannot write them, which means it may fail to log some messages in such cases but it will not block the rest of the system.

`log_directory` (`string`)

When `logging_collector` is enabled, this parameter determines the directory in which log files will be created. It can be specified as an absolute path, or relative to the cluster data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `log`.

`log_filename` (`string`)

When `logging_collector` is enabled, this parameter sets the file names of the created log files. The value is treated as a `strftime` pattern, so %-escapes can be used to specify time-varying file names. (Note that if there are any time-zone-dependent %-escapes, the computation is

done in the zone specified by log_timezone.) The supported %-escapes are similar to those listed in the Open Group's strftime [1] specification. Note that the system's `strftime` is not used directly, so platform-specific (nonstandard) extensions do not work. The default is `postgresql-%Y-%m-%d_%H%M%S.log`.

If you specify a file name without escapes, you should plan to use a log rotation utility to avoid eventually filling the entire disk. In releases prior to 8.4, if no % escapes were present, PostgreSQL would append the epoch of the new log file's creation time, but this is no longer the case.

If CSV-format output is enabled in `log_destination`, `.csv` will be appended to the time-stamped log file name to create the file name for CSV-format output. (If `log_filename` ends in `.log`, the suffix is replaced instead.)

If JSON-format output is enabled in `log_destination`, `.json` will be appended to the time-stamped log file name to create the file name for JSON-format output. (If `log_filename` ends in `.log`, the suffix is replaced instead.)

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_file_mode` (`integer`)

On Unix systems this parameter sets the permissions for log files when `logging_collector` is enabled. (On Microsoft Windows this parameter is ignored.) The parameter value is expected to be a numeric mode specified in the format accepted by the `chmod` and `umask` system calls. (To use the customary octal format the number must start with a `0` (zero).)

The default permissions are `0600`, meaning only the server owner can read or write the log files. The other commonly useful setting is `0640`, allowing members of the owner's group to read the files. Note however that to make use of such a setting, you'll need to alter log_directory to store the files somewhere outside the cluster data directory. In any case, it's unwise to make the log files world-readable, since they might contain sensitive data.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_rotation_age` (`integer`)

When `logging_collector` is enabled, this parameter determines the maximum amount of time to use an individual log file, after which a new log file will be created. If this value is specified without units, it is taken as minutes. The default is 24 hours. Set to zero to disable time-based creation of new log files. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_rotation_size` (`integer`)

When `logging_collector` is enabled, this parameter determines the maximum size of an individual log file. After this amount of data has been emitted into a log file, a new log file will be created. If this value is specified without units, it is taken as kilobytes. The default is 10 megabytes. Set to zero to disable size-based creation of new log files. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_truncate_on_rotation` (`boolean`)

When `logging_collector` is enabled, this parameter will cause PostgreSQL to truncate (overwrite), rather than append to, any existing log file of the same name. However, truncation will occur only when a new file is being opened due to time-based rotation, not during server startup or size-based rotation. When off, pre-existing files will be appended to in all cases. For example, using this setting in combination with a `log_filename` like `postgresql-%H.log` would result in generating twenty-four hourly log files and then cyclically overwriting them. This parameter can only be set in the `postgresql.conf` file or on the server command line.

[1] https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

Example: To keep 7 days of logs, one log file per day named `server_log.Mon`, `server_log.Tue`, etc., and automatically overwrite last week's log with this week's log, set `log_filename` to `server_log.%a`, `log_truncate_on_rotation` to `on`, and `log_rotation_age` to `1440`.

Example: To keep 24 hours of logs, one log file per hour, but also rotate sooner if the log file size exceeds 1GB, set `log_filename` to `server_log.%H%M`, `log_truncate_on_rotation` to `on`, `log_rotation_age` to `60`, and `log_rotation_size` to `1000000`. Including `%M` in `log_filename` allows any size-driven rotations that might occur to select a file name different from the hour's initial file name.

`syslog_facility` (`enum`)

When logging to syslog is enabled, this parameter determines the syslog “facility” to be used. You can choose from `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7`; the default is `LOCAL0`. See also the documentation of your system's syslog daemon. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`syslog_ident` (`string`)

When logging to syslog is enabled, this parameter determines the program name used to identify PostgreSQL messages in syslog logs. The default is `postgres`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`syslog_sequence_numbers` (`boolean`)

When logging to syslog and this is on (the default), then each message will be prefixed by an increasing sequence number (such as `[2]`). This circumvents the “--- last message repeated N times ---” suppression that many syslog implementations perform by default. In more modern syslog implementations, repeated message suppression can be configured (for example, `$RepeatedMsgReduction` in rsyslog), so this might not be necessary. Also, you could turn this off if you actually want to suppress repeated messages.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`syslog_split_messages` (`boolean`)

When logging to syslog is enabled, this parameter determines how messages are delivered to syslog. When on (the default), messages are split by lines, and long lines are split so that they will fit into 1024 bytes, which is a typical size limit for traditional syslog implementations. When off, PostgreSQL server log messages are delivered to the syslog service as is, and it is up to the syslog service to cope with the potentially bulky messages.

If syslog is ultimately logging to a text file, then the effect will be the same either way, and it is best to leave the setting on, since most syslog implementations either cannot handle large messages or would need to be specially configured to handle them. But if syslog is ultimately writing into some other medium, it might be necessary or more useful to keep messages logically together.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`event_source` (`string`)

When logging to event log is enabled, this parameter determines the program name used to identify PostgreSQL messages in the log. The default is `PostgreSQL`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.8.2. When to Log

`log_min_messages` (`enum`)

Controls which message levels are written to the server log. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`.

Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log. The default is `WARNING`. Note that `LOG` has a different rank here than in client_min_messages. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_min_error_statement` (`enum`)

Controls which SQL statements that cause an error condition are recorded in the server log. The current SQL statement is included in the log entry for any message of the specified severity or higher. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`. The default is `ERROR`, which means statements causing errors, log messages, fatal errors, or panics will be logged. To effectively turn off logging of failing statements, set this parameter to `PANIC`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_min_duration_statement` (`integer`)

Causes the duration of each completed statement to be logged if the statement ran for at least the specified amount of time. For example, if you set it to `250ms` then all SQL statements that run 250ms or longer will be logged. Enabling this parameter can be helpful in tracking down unoptimized queries in your applications. If this value is specified without units, it is taken as milliseconds. Setting this to zero prints all statement durations. `-1` (the default) disables logging statement durations. Only superusers and users with the appropriate `SET` privilege can change this setting.

This overrides log_min_duration_sample, meaning that queries with duration exceeding this setting are not subject to sampling and are always logged.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note**
>
> When using this option together with log_statement, the text of statements that are logged because of `log_statement` will not be repeated in the duration log message. If you are not using syslog, it is recommended that you log the PID or session ID using log_line_prefix so that you can link the statement message to the later duration message using the process ID or session ID.

`log_min_duration_sample` (`integer`)

Allows sampling the duration of completed statements that ran for at least the specified amount of time. This produces the same kind of log entries as log_min_duration_statement, but only for a subset of the executed statements, with sample rate controlled by log_statement_sample_rate. For example, if you set it to `100ms` then all SQL statements that run 100ms or longer will be considered for sampling. Enabling this parameter can be helpful when the traffic is too high to log all queries. If this value is specified without units, it is taken as milliseconds. Setting this to zero samples all statement durations. `-1` (the default) disables sampling statement durations. Only superusers and users with the appropriate `SET` privilege can change this setting.

This setting has lower priority than `log_min_duration_statement`, meaning that statements with durations exceeding `log_min_duration_statement` are not subject to sampling and are always logged.

Other notes for `log_min_duration_statement` apply also to this setting.

`log_statement_sample_rate` (`floating point`)

Determines the fraction of statements with duration exceeding log_min_duration_sample that will be logged. Sampling is stochastic, for example `0.5` means there is statistically one chance in

two that any given statement will be logged. The default is `1.0`, meaning to log all sampled statements. Setting this to zero disables sampled statement-duration logging, the same as setting `log_min_duration_sample` to `-1`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_transaction_sample_rate` (`floating point`)

Sets the fraction of transactions whose statements are all logged, in addition to statements logged for other reasons. It applies to each new transaction regardless of its statements' durations. Sampling is stochastic, for example `0.1` means there is statistically one chance in ten that any given transaction will be logged. `log_transaction_sample_rate` can be helpful to construct a sample of transactions. The default is `0`, meaning not to log statements from any additional transactions. Setting this to `1` logs all statements of all transactions. Only superusers and users with the appropriate `SET` privilege can change this setting.

> **Note**
>
> Like all statement-logging options, this option can add significant overhead.

`log_startup_progress_interval` (`integer`)

Sets the amount of time after which the startup process will log a message about a long-running operation that is still in progress, as well as the interval between further progress messages for that operation. The default is 10 seconds. A setting of `0` disables the feature. If this value is specified without units, it is taken as milliseconds. This setting is applied separately to each operation. This parameter can only be set in the `postgresql.conf` file or on the server command line.

For example, if syncing the data directory takes 25 seconds and thereafter resetting unlogged relations takes 8 seconds, and if this setting has the default value of 10 seconds, then a messages will be logged for syncing the data directory after it has been in progress for 10 seconds and again after it has been in progress for 20 seconds, but nothing will be logged for resetting unlogged relations.

Table 19.2 explains the message severity levels used by PostgreSQL. If logging output is sent to syslog or Windows' eventlog, the severity levels are translated as shown in the table.

Table 19.2. Message Severity Levels

Severity	Usage	syslog	eventlog
`DEBUG1 .. DEBUG5`	Provides successively-more-detailed information for use by developers.	`DEBUG`	`INFORMATION`
`INFO`	Provides information implicitly requested by the user, e.g., output from `VACUUM VERBOSE`.	`INFO`	`INFORMATION`
`NOTICE`	Provides information that might be helpful to users, e.g., notice of truncation of long identifiers.	`NOTICE`	`INFORMATION`
`WARNING`	Provides warnings of likely problems, e.g., `COMMIT` outside a transaction block.	`NOTICE`	`WARNING`
`ERROR`	Reports an error that caused the current command to abort.	`WARNING`	`ERROR`
`LOG`	Reports information of interest to administrators, e.g., checkpoint activity.	`INFO`	`INFORMATION`

Severity	Usage	syslog	eventlog
FATAL	Reports an error that caused the current session to abort.	ERR	ERROR
PANIC	Reports an error that caused all database sessions to abort.	CRIT	ERROR

19.8.3. What to Log

> **Note**
>
> What you choose to log can have security implications; see Section 24.3.

`application_name` (`string`)

The `application_name` can be any string of less than `NAMEDATALEN` characters (64 characters in a standard build). It is typically set by an application upon connection to the server. The name will be displayed in the `pg_stat_activity` view and included in CSV log entries. It can also be included in regular log entries via the log_line_prefix parameter. Only printable ASCII characters may be used in the `application_name` value. Other characters are replaced with C-style hexadecimal escapes.

`debug_print_parse` (`boolean`)
`debug_print_rewritten` (`boolean`)
`debug_print_plan` (`boolean`)

These parameters enable various debugging output to be emitted. When set, they print the resulting parse tree, the query rewriter output, or the execution plan for each executed query. These messages are emitted at `LOG` message level, so by default they will appear in the server log but will not be sent to the client. You can change that by adjusting client_min_messages and/or log_min_messages. These parameters are off by default.

`debug_pretty_print` (`boolean`)

When set, `debug_pretty_print` indents the messages produced by `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. This results in more readable but much longer output than the “compact” format used when it is off. It is on by default.

`log_autovacuum_min_duration` (`integer`)

Causes each action executed by autovacuum to be logged if it ran for at least the specified amount of time. Setting this to zero logs all autovacuum actions. `-1` disables logging autovacuum actions. If this value is specified without units, it is taken as milliseconds. For example, if you set this to `250ms` then all automatic vacuums and analyzes that run 250ms or longer will be logged. In addition, when this parameter is set to any value other than `-1`, a message will be logged if an autovacuum action is skipped due to a conflicting lock or a concurrently dropped relation. The default is `10min`. Enabling this parameter can be helpful in tracking autovacuum activity. This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`log_checkpoints` (`boolean`)

Causes checkpoints and restartpoints to be logged in the server log. Some statistics are included in the log messages, including the number of buffers written and the time spent writing them. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is on.

`log_connections` (`boolean`)

Causes each attempted connection to the server to be logged, as well as successful completion of both client authentication (if necessary) and authorization. Only superusers and users with the appropriate `SET` privilege can change this parameter at session start, and it cannot be changed at all within a session. The default is `off`.

> **Note**
>
> Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate "connection received" messages do not necessarily indicate a problem.

`log_disconnections` (`boolean`)

Causes session terminations to be logged. The log output provides information similar to `log_connections`, plus the duration of the session. Only superusers and users with the appropriate `SET` privilege can change this parameter at session start, and it cannot be changed at all within a session. The default is `off`.

`log_duration` (`boolean`)

Causes the duration of every completed statement to be logged. The default is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note**
>
> The difference between enabling `log_duration` and setting log_min_duration_statement to zero is that exceeding `log_min_duration_statement` forces the text of the query to be logged, but this option doesn't. Thus, if `log_duration` is `on` and `log_min_duration_statement` has a positive value, all durations are logged but the query text is included only for statements exceeding the threshold. This behavior can be useful for gathering statistics in high-load installations.

`log_error_verbosity` (`enum`)

Controls the amount of detail written in the server log for each message that is logged. Valid values are `TERSE`, `DEFAULT`, and `VERBOSE`, each adding more fields to displayed messages. `TERSE` excludes the logging of `DETAIL`, `HINT`, `QUERY`, and `CONTEXT` error information. `VERBOSE` output includes the `SQLSTATE` error code (see also Appendix A) and the source code file name, function name, and line number that generated the error. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_hostname` (`boolean`)

By default, connection log messages only show the IP address of the connecting host. Turning this parameter on causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_line_prefix` (`string`)

This is a `printf`-style string that is output at the beginning of each log line. `%` characters begin "escape sequences" that are replaced with status information as outlined below. Unrecognized

escapes are ignored. Other characters are copied straight to the log line. Some escapes are only recognized by session processes, and will be treated as empty by background processes such as the main server process. Status information may be aligned either left or right by specifying a numeric literal after the % and before the option. A negative value will cause the status information to be padded on the right with spaces to give it a minimum width, whereas a positive value will pad on the left. Padding can be useful to aid human readability in log files.

This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `'%m [%p] '` which logs a time stamp and the process ID.

Escape	Effect	Session only
`%a`	Application name	yes
`%u`	User name	yes
`%d`	Database name	yes
`%r`	Remote host name or IP address, and remote port	yes
`%h`	Remote host name or IP address	yes
`%b`	Backend type	no
`%p`	Process ID	no
`%P`	Process ID of the parallel group leader, if this process is a parallel query worker	no
`%t`	Time stamp without milliseconds	no
`%m`	Time stamp with milliseconds	no
`%n`	Time stamp with milliseconds (as a Unix epoch)	no
`%i`	Command tag: type of session's current command	yes
`%e`	SQLSTATE error code	no
`%c`	Session ID: see below	no
`%l`	Number of the log line for each session or process, starting at 1	no
`%s`	Process start time stamp	no
`%v`	Virtual transaction ID (procNumber/localXID); see Section 66.1	no
`%x`	Transaction ID (0 if none is assigned); see Section 66.1	no
`%q`	Produces no output, but tells non-session processes to stop at this point in the string; ignored by session processes	no
`%Q`	Query identifier of the current query. Query identifiers are not computed by default, so this field will be zero unless compute_query_id parameter is enabled or a third-party	yes

Escape	Effect	Session only
	module that computes query identifiers is configured.	
%%	Literal %	no

The backend type corresponds to the column `backend_type` in the view `pg_stat_activity`, but additional types can appear in the log that don't show in that view.

The `%c` escape prints a quasi-unique session identifier, consisting of two 4-byte hexadecimal numbers (without leading zeros) separated by a dot. The numbers are the process start time and the process ID, so `%c` can also be used as a space saving way of printing those items. For example, to generate the session identifier from `pg_stat_activity`, use this query:

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer)
 || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

> **Tip**
>
> If you set a nonempty value for `log_line_prefix`, you should usually make its last character be a space, to provide visual separation from the rest of the log line. A punctuation character can be used too.

> **Tip**
>
> Syslog produces its own time stamp and process ID information, so you probably do not want to include those escapes if you are logging to syslog.

> **Tip**
>
> The `%q` escape is useful when including information that is only available in session (backend) context like user or database name. For example:
>
> ```
> log_line_prefix = '%m [%p] %q%u@%d/%a '
> ```

> **Note**
>
> The `%Q` escape always reports a zero identifier for lines output by log_statement because `log_statement` generates output before an identifier can be calculated, including invalid statements for which an identifier cannot be calculated.

`log_lock_waits` (`boolean`)

Controls whether a log message is produced when a session waits longer than deadlock_timeout to acquire a lock. This is useful in determining if lock waits are causing poor performance. The

default is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_recovery_conflict_waits` (`boolean`)

Controls whether a log message is produced when the startup process waits longer than `deadlock_timeout` for recovery conflicts. This is useful in determining if recovery conflicts prevent the recovery from applying WAL.

The default is `off`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_parameter_max_length` (`integer`)

If greater than zero, each bind parameter value logged with a non-error statement-logging message is trimmed to this many bytes. Zero disables logging of bind parameters for non-error statement logs. `-1` (the default) allows bind parameters to be logged in full. If this value is specified without units, it is taken as bytes. Only superusers and users with the appropriate `SET` privilege can change this setting.

This setting only affects log messages printed as a result of log_statement, log_duration, and related settings. Non-zero values of this setting add some overhead, particularly if parameters are sent in binary form, since then conversion to text is required.

`log_parameter_max_length_on_error` (`integer`)

If greater than zero, each bind parameter value reported in error messages is trimmed to this many bytes. Zero (the default) disables including bind parameters in error messages. `-1` allows bind parameters to be printed in full. If this value is specified without units, it is taken as bytes.

Non-zero values of this setting add overhead, as PostgreSQL will need to store textual representations of parameter values in memory at the start of each statement, whether or not an error eventually occurs. The overhead is greater when bind parameters are sent in binary form than when they are sent as text, since the former case requires data conversion while the latter only requires copying the string.

`log_statement` (`enum`)

Controls which SQL statements are logged. Valid values are `none` (off), `ddl`, `mod`, and `all` (all statements). `ddl` logs all data definition statements, such as `CREATE`, `ALTER`, and `DROP` statements. `mod` logs all `ddl` statements, plus data-modifying statements such as `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY FROM`. `PREPARE`, `EXECUTE`, and `EXPLAIN ANALYZE` statements are also logged if their contained command is of an appropriate type. For clients using extended query protocol, logging occurs when an Execute message is received, and values of the Bind parameters are included (with any embedded single-quote marks doubled).

The default is `none`. Only superusers and users with the appropriate `SET` privilege can change this setting.

> **Note**
>
> Statements that contain simple syntax errors are not logged even by the `log_statement = all` setting, because the log message is emitted only after basic parsing has been done to determine the statement type. In the case of extended query protocol, this setting likewise does not log statements that fail before the Execute phase (i.e., during parse analysis or planning). Set `log_min_error_statement` to `ERROR` (or lower) to log such statements.
>
> Logged statements might reveal sensitive data and even contain plaintext passwords.

`log_replication_commands` (`boolean`)

Causes each replication command and `walsender` process's replication slot acquisition/release to be logged in the server log. See Section 53.4 for more information about replication command. The default value is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_temp_files` (`integer`)

Controls logging of temporary file names and sizes. Temporary files can be created for sorts, hashes, and temporary query results. If enabled by this setting, a log entry is emitted for each temporary file, with the file size specified in bytes, when it is deleted. A value of zero logs all temporary file information, while positive values log only files whose size is greater than or equal to the specified amount of data. If this value is specified without units, it is taken as kilobytes. The default setting is -1, which disables such logging. Only superusers and users with the appropriate `SET` privilege can change this setting.

`log_timezone` (`string`)

Sets the time zone used for timestamps written in the server log. Unlike TimeZone, this value is cluster-wide, so that all sessions will report timestamps consistently. The built-in default is `GMT`, but that is typically overridden in `postgresql.conf`; initdb will install a setting there corresponding to its system environment. See Section 8.5.3 for more information. This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.8.4. Using CSV-Format Log Output

Including `csvlog` in the `log_destination` list provides a convenient way to import log files into a database table. This option emits log lines in comma-separated-values (CSV) format, with these columns: time stamp with milliseconds, user name, database name, process ID, client host:port number, session ID, per-session line number, command tag, session start time, virtual transaction ID, regular transaction ID, error severity, SQLSTATE code, error message, error message detail, hint, internal query that led to the error (if any), character count of the error position therein, error context, user query that led to the error (if any and enabled by `log_min_error_statement`), character count of the error position therein, location of the error in the PostgreSQL source code (if `log_error_verbosity` is set to `verbose`), application name, backend type, process ID of parallel group leader, and query id. Here is a sample table definition for storing CSV-format log output:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
```

```
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint,
  PRIMARY KEY (session_id, session_line_num)
);
```

To import a log file into this table, use the `COPY FROM` command:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

It is also possible to access the file as a foreign table, using the supplied file_fdw module.

There are a few things you need to do to simplify importing CSV log files:

1. Set `log_filename` and `log_rotation_age` to provide a consistent, predictable naming scheme for your log files. This lets you predict what the file name will be and know when an individual log file is complete and therefore ready to be imported.

2. Set `log_rotation_size` to 0 to disable size-based log rotation, as it makes the log file name difficult to predict.

3. Set `log_truncate_on_rotation` to `on` so that old log data isn't mixed with the new in the same file.

4. The table definition above includes a primary key specification. This is useful to protect against accidentally importing the same information twice. The `COPY` command commits all of the data it imports at one time, so any error will cause the entire import to fail. If you import a partial log file and later import the file again when it is complete, the primary key violation will cause the import to fail. Wait until the log is complete and closed before importing. This procedure will also protect against accidentally importing a partial line that hasn't been completely written, which would also cause `COPY` to fail.

19.8.5. Using JSON-Format Log Output

Including `jsonlog` in the `log_destination` list provides a convenient way to import log files into many different programs. This option emits log lines in JSON format.

String fields with null values are excluded from output. Additional fields may be added in the future. User applications that process `jsonlog` output should ignore unknown fields.

Each log line is serialized as a JSON object with the set of keys and their associated values shown in Table 19.3.

Table 19.3. Keys and Values of JSON Log Entries

Key name	Type	Description
`timestamp`	string	Time stamp with milliseconds
`user`	string	User name
`dbname`	string	Database name
`pid`	number	Process ID
`remote_host`	string	Client host

Key name	Type	Description
remote_port	number	Client port
session_id	string	Session ID
line_num	number	Per-session line number
ps	string	Current ps display
session_start	string	Session start time
vxid	string	Virtual transaction ID
txid	string	Regular transaction ID
error_severity	string	Error severity
state_code	string	SQLSTATE code
message	string	Error message
detail	string	Error message detail
hint	string	Error message hint
internal_query	string	Internal query that led to the error
internal_position	number	Cursor index into internal query
context	string	Error context
statement	string	Client-supplied query string
cursor_position	number	Cursor index into query string
func_name	string	Error location function name
file_name	string	File name of error location
file_line_num	number	File line number of the error location
application_name	string	Client application name
backend_type	string	Type of backend
leader_pid	number	Process ID of leader for active parallel workers
query_id	number	Query ID

19.8.6. Process Title

These settings control how process titles of server processes are modified. Process titles are typically viewed using programs like ps or, on Windows, Process Explorer. See Section 27.1 for details.

`cluster_name` (`string`)

Sets a name that identifies this database cluster (instance) for various purposes. The cluster name appears in the process title for all server processes in this cluster. Moreover, it is the default application name for a standby connection (see synchronous_standby_names).

The name can be any string of less than `NAMEDATALEN` characters (64 characters in a standard build). Only printable ASCII characters may be used in the `cluster_name` value. Other characters are replaced with C-style hexadecimal escapes. No name is shown if this parameter is set to the empty string `''` (which is the default). This parameter can only be set at server start.

`update_process_title` (`boolean`)

Enables updating of the process title every time a new SQL command is received by the server. This setting defaults to `on` on most platforms, but it defaults to `off` on Windows due to that

platform's larger overhead for updating the process title. Only superusers and users with the appropriate SET privilege can change this setting.

19.9. Run-time Statistics

19.9.1. Cumulative Query and Index Statistics

These parameters control the server-wide cumulative statistics system. When enabled, the data that is collected can be accessed via the pg_stat and pg_statio family of system views. Refer to Chapter 27 for more information.

track_activities (boolean)

Enables the collection of information on the currently executing command of each session, along with its identifier and the time when that command began execution. This parameter is on by default. Note that even when enabled, this information is only visible to superusers, roles with privileges of the pg_read_all_stats role and the user owning the sessions being reported on (including sessions belonging to a role they have the privileges of), so it should not represent a security risk. Only superusers and users with the appropriate SET privilege can change this setting.

track_activity_query_size (integer)

Specifies the amount of memory reserved to store the text of the currently executing command for each active session, for the pg_stat_activity.query field. If this value is specified without units, it is taken as bytes. The default value is 1024 bytes. This parameter can only be set at server start.

track_counts (boolean)

Enables collection of statistics on database activity. This parameter is on by default, because the autovacuum daemon needs the collected information. Only superusers and users with the appropriate SET privilege can change this setting.

track_io_timing (boolean)

Enables timing of database I/O calls. This parameter is off by default, as it will repeatedly query the operating system for the current time, which may cause significant overhead on some platforms. You can use the pg_test_timing tool to measure the overhead of timing on your system. I/O timing information is displayed in pg_stat_database, pg_stat_io, in the output of EXPLAIN when the BUFFERS option is used, in the output of VACUUM when the VERBOSE option is used, by autovacuum for auto-vacuums and auto-analyzes, when log_autovacuum_min_duration is set and by pg_stat_statements. Only superusers and users with the appropriate SET privilege can change this setting.

track_wal_io_timing (boolean)

Enables timing of WAL I/O calls. This parameter is off by default, as it will repeatedly query the operating system for the current time, which may cause significant overhead on some platforms. You can use the pg_test_timing tool to measure the overhead of timing on your system. I/O timing information is displayed in pg_stat_wal. Only superusers and users with the appropriate SET privilege can change this setting.

track_functions (enum)

Enables tracking of function call counts and time used. Specify pl to track only procedural-language functions, all to also track SQL and C language functions. The default is none, which disables function statistics tracking. Only superusers and users with the appropriate SET privilege can change this setting.

> **Note**
>
> SQL-language functions that are simple enough to be "inlined" into the calling query will not be tracked, regardless of this setting.

`stats_fetch_consistency` (`enum`)

Determines the behavior when cumulative statistics are accessed multiple times within a transaction. When set to `none`, each access re-fetches counters from shared memory. When set to `cache`, the first access to statistics for an object caches those statistics until the end of the transaction unless `pg_stat_clear_snapshot()` is called. When set to `snapshot`, the first statistics access caches all statistics accessible in the current database, until the end of the transaction unless `pg_stat_clear_snapshot()` is called. Changing this parameter in a transaction discards the statistics snapshot. The default is `cache`.

> **Note**
>
> `none` is most suitable for monitoring systems. If values are only accessed once, it is the most efficient. `cache` ensures repeat accesses yield the same values, which is important for queries involving e.g. self-joins. `snapshot` can be useful when interactively inspecting statistics, but has higher overhead, particularly if many database objects exist.

19.9.2. Statistics Monitoring

`compute_query_id` (`enum`)

Enables in-core computation of a query identifier. Query identifiers can be displayed in the `pg_stat_activity` view, using `EXPLAIN`, or emitted in the log if configured via the log_line_prefix parameter. The pg_stat_statements extension also requires a query identifier to be computed. Note that an external module can alternatively be used if the in-core query identifier computation method is not acceptable. In this case, in-core computation must be always disabled. Valid values are `off` (always disabled), `on` (always enabled), `auto`, which lets modules such as pg_stat_statements automatically enable it, and `regress` which has the same effect as `auto`, except that the query identifier is not shown in the `EXPLAIN` output in order to facilitate automated regression testing. The default is `auto`.

> **Note**
>
> To ensure that only one query identifier is calculated and displayed, extensions that calculate query identifiers should throw an error if a query identifier has already been computed.

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

For each query, output performance statistics of the respective module to the server log. This is a crude profiling instrument, similar to the Unix `getrusage()` operating system facility. `log_statement_stats` reports total statement statistics, while the others report per-module statistics. `log_statement_stats` cannot be enabled together with any of the per-module options. All of these options are disabled by default. Only superusers and users with the appropriate `SET` privilege can change these settings.

19.10. Automatic Vacuuming

These settings control the behavior of the *autovacuum* feature. Refer to Section 24.1.6 for more information. Note that many of these settings can be overridden on a per-table basis; see Storage Parameters.

`autovacuum` (`boolean`)

Controls whether the server should run the autovacuum launcher daemon. This is on by default; however, track_counts must also be enabled for autovacuum to work. This parameter can only be set in the `postgresql.conf` file or on the server command line; however, autovacuuming can be disabled for individual tables by changing table storage parameters.

Note that even when this parameter is disabled, the system will launch autovacuum processes if necessary to prevent transaction ID wraparound. See Section 24.1.5 for more information.

`autovacuum_max_workers` (`integer`)

Specifies the maximum number of autovacuum processes (other than the autovacuum launcher) that may be running at any one time. The default is three. This parameter can only be set at server start.

`autovacuum_naptime` (`integer`)

Specifies the minimum delay between autovacuum runs on any given database. In each round the daemon examines the database and issues `VACUUM` and `ANALYZE` commands as needed for tables in that database. If this value is specified without units, it is taken as seconds. The default is one minute (`1min`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`autovacuum_vacuum_threshold` (`integer`)

Specifies the minimum number of updated or deleted tuples needed to trigger a `VACUUM` in any one table. The default is 50 tuples. This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`autovacuum_vacuum_insert_threshold` (`integer`)

Specifies the number of inserted tuples needed to trigger a `VACUUM` in any one table. The default is 1000 tuples. If -1 is specified, autovacuum will not trigger a `VACUUM` operation on any tables based on the number of inserts. This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`autovacuum_analyze_threshold` (`integer`)

Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an `ANALYZE` in any one table. The default is 50 tuples. This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`autovacuum_vacuum_scale_factor` (`floating point`)

Specifies a fraction of the table size to add to `autovacuum_vacuum_threshold` when deciding whether to trigger a `VACUUM`. The default is 0.2 (20% of table size). This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`autovacuum_vacuum_insert_scale_factor` (`floating point`)

Specifies a fraction of the table size to add to `autovacuum_vacuum_insert_threshold` when deciding whether to trigger a `VACUUM`. The default is 0.2 (20% of table size). This parameter

can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_analyze_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_freeze_max_age (integer)

Specifies the maximum age (in transactions) that a table's pg_class.relfrozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled.

Vacuum also allows removal of old files from the pg_xact subdirectory, which is why the default is a relatively low 200 million transactions. This parameter can only be set at server start, but the setting can be reduced for individual tables by changing table storage parameters. For more information see Section 24.1.5.

autovacuum_multixact_freeze_max_age (integer)

Specifies the maximum age (in multixacts) that a table's pg_class.relminmxid field can attain before a VACUUM operation is forced to prevent multixact ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled.

Vacuuming multixacts also allows removal of old files from the pg_multixact/members and pg_multixact/offsets subdirectories, which is why the default is a relatively low 400 million multixacts. This parameter can only be set at server start, but the setting can be reduced for individual tables by changing table storage parameters. For more information see Section 24.1.5.1.

autovacuum_vacuum_cost_delay (floating point)

Specifies the cost delay value that will be used in automatic VACUUM operations. If -1 is specified, the regular vacuum_cost_delay value will be used. If this value is specified without units, it is taken as milliseconds. The default value is 2 milliseconds. This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_vacuum_cost_limit (integer)

Specifies the cost limit value that will be used in automatic VACUUM operations. If -1 is specified (which is the default), the regular vacuum_cost_limit value will be used. Note that the value is distributed proportionally among the running autovacuum workers, if there is more than one, so that the sum of the limits for each worker does not exceed the value of this variable. This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

19.11. Client Connection Defaults

19.11.1. Statement Behavior

client_min_messages (enum)

Controls which message levels are sent to the client. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, and ERROR. Each level includes all the

levels that follow it. The later the level, the fewer messages are sent. The default is NOTICE. Note that LOG has a different rank here than in log_min_messages.

INFO level messages are always sent to the client.

search_path (string)

This variable specifies the order in which schemas are searched when an object (table, data type, function, etc.) is referenced by a simple name with no schema specified. When there are objects of identical names in different schemas, the one found first in the search path is used. An object that is not in any of the schemas in the search path can only be referenced by specifying its containing schema with a qualified (dotted) name.

The value for search_path must be a comma-separated list of schema names. Any name that is not an existing schema, or is a schema for which the user does not have USAGE permission, is silently ignored.

If one of the list items is the special name $user, then the schema having the name returned by CURRENT_USER is substituted, if there is such a schema and the user has USAGE permission for it. (If not, $user is ignored.)

The system catalog schema, pg_catalog, is always searched, whether it is mentioned in the path or not. If it is mentioned in the path then it will be searched in the specified order. If pg_catalog is not in the path then it will be searched *before* searching any of the path items.

Likewise, the current session's temporary-table schema, pg_temp_*nnn*, is always searched if it exists. It can be explicitly listed in the path by using the alias pg_temp. If it is not listed in the path then it is searched first (even before pg_catalog). However, the temporary schema is only searched for relation (table, view, sequence, etc.) and data type names. It is never searched for function or operator names.

When objects are created without specifying a particular target schema, they will be placed in the first valid schema named in search_path. An error is reported if the search path is empty.

The default value for this parameter is "$user", public. This setting supports shared use of a database (where no users have private schemas, and all share use of public), private per-user schemas, and combinations of these. Other effects can be obtained by altering the default search path setting, either globally or per-user.

For more information on schema handling, see Section 5.10. In particular, the default configuration is suitable only when the database has a single user or a few mutually-trusting users.

The current effective value of the search path can be examined via the SQL function current_schemas (see Section 9.27). This is not quite the same as examining the value of search_path, since current_schemas shows how the items appearing in search_path were resolved.

row_security (boolean)

This variable controls whether to raise an error in lieu of applying a row security policy. When set to on, policies apply normally. When set to off, queries fail which would otherwise apply at least one policy. The default is on. Change to off where limited row visibility could cause incorrect results; for example, pg_dump makes that change by default. This variable has no effect on roles which bypass every row security policy, to wit, superusers and roles with the BYPASSRLS attribute.

For more information on row security policies, see CREATE POLICY.

default_table_access_method (string)

This parameter specifies the default table access method to use when creating tables or materialized views if the CREATE command does not explicitly specify an access method, or when

`SELECT ... INTO` is used, which does not allow specifying a table access method. The default is `heap`.

`default_tablespace` (`string`)

This variable specifies the default tablespace in which to create objects (tables and indexes) when a `CREATE` command does not explicitly specify a tablespace.

The value is either the name of a tablespace, or an empty string to specify using the default tablespace of the current database. If the value does not match the name of any existing tablespace, PostgreSQL will automatically use the default tablespace of the current database. If a nondefault tablespace is specified, the user must have `CREATE` privilege for it, or creation attempts will fail.

This variable is not used for temporary tables; for them, temp_tablespaces is consulted instead.

This variable is also not used when creating databases. By default, a new database inherits its tablespace setting from the template database it is copied from.

If this parameter is set to a value other than the empty string when a partitioned table is created, the partitioned table's tablespace will be set to that value, which will be used as the default tablespace for partitions created in the future, even if `default_tablespace` has changed since then.

For more information on tablespaces, see Section 22.6.

`default_toast_compression` (`enum`)

This variable sets the default TOAST compression method for values of compressible columns. (This can be overridden for individual columns by setting the `COMPRESSION` column option in `CREATE TABLE` or `ALTER TABLE`.) The supported compression methods are `pglz` and (if PostgreSQL was compiled with `--with-lz4`) `lz4`. The default is `pglz`.

`temp_tablespaces` (`string`)

This variable specifies tablespaces in which to create temporary objects (temp tables and indexes on temp tables) when a `CREATE` command does not explicitly specify a tablespace. Temporary files for purposes such as sorting large data sets are also created in these tablespaces.

The value is a list of names of tablespaces. When there is more than one name in the list, PostgreSQL chooses a random member of the list each time a temporary object is to be created; except that within a transaction, successively created temporary objects are placed in successive tablespaces from the list. If the selected element of the list is an empty string, PostgreSQL will automatically use the default tablespace of the current database instead.

When `temp_tablespaces` is set interactively, specifying a nonexistent tablespace is an error, as is specifying a tablespace for which the user does not have `CREATE` privilege. However, when using a previously set value, nonexistent tablespaces are ignored, as are tablespaces for which the user lacks `CREATE` privilege. In particular, this rule applies when using a value set in `postgresql.conf`.

The default value is an empty string, which results in all temporary objects being created in the default tablespace of the current database.

See also default_tablespace.

`check_function_bodies` (`boolean`)

This parameter is normally on. When set to `off`, it disables validation of the routine body string during CREATE FUNCTION and CREATE PROCEDURE. Disabling validation avoids side effects of the validation process, in particular preventing false positives due to problems such as forward references. Set this parameter to `off` before loading functions on behalf of other users; pg_dump does so automatically.

`default_transaction_isolation` (`enum`)

Each SQL transaction has an isolation level, which can be either “read uncommitted”, “read committed”, “repeatable read”, or “serializable”. This parameter controls the default isolation level of each new transaction. The default is “read committed”.

Consult Chapter 13 and SET TRANSACTION for more information.

`default_transaction_read_only` (`boolean`)

A read-only SQL transaction cannot alter non-temporary tables. This parameter controls the default read-only status of each new transaction. The default is `off` (read/write).

Consult SET TRANSACTION for more information.

`default_transaction_deferrable` (`boolean`)

When running at the `serializable` isolation level, a deferrable read-only SQL transaction may be delayed before it is allowed to proceed. However, once it begins executing it does not incur any of the overhead required to ensure serializability; so serialization code will have no reason to force it to abort because of concurrent updates, making this option suitable for long-running read-only transactions.

This parameter controls the default deferrable status of each new transaction. It currently has no effect on read-write transactions or those operating at isolation levels lower than `serializable`. The default is `off`.

Consult SET TRANSACTION for more information.

`transaction_isolation` (`enum`)

This parameter reflects the current transaction's isolation level. At the beginning of each transaction, it is set to the current value of default_transaction_isolation. Any subsequent attempt to change it is equivalent to a SET TRANSACTION command.

`transaction_read_only` (`boolean`)

This parameter reflects the current transaction's read-only status. At the beginning of each transaction, it is set to the current value of default_transaction_read_only. Any subsequent attempt to change it is equivalent to a SET TRANSACTION command.

`transaction_deferrable` (`boolean`)

This parameter reflects the current transaction's deferrability status. At the beginning of each transaction, it is set to the current value of default_transaction_deferrable. Any subsequent attempt to change it is equivalent to a SET TRANSACTION command.

`session_replication_role` (`enum`)

Controls firing of replication-related triggers and rules for the current session. Possible values are `origin` (the default), `replica` and `local`. Setting this parameter results in discarding any previously cached query plans. Only superusers and users with the appropriate `SET` privilege can change this setting.

The intended use of this setting is that logical replication systems set it to `replica` when they are applying replicated changes. The effect of that will be that triggers and rules (that have not been altered from their default configuration) will not fire on the replica. See the `ALTER TABLE` clauses `ENABLE TRIGGER` and `ENABLE RULE` for more information.

PostgreSQL treats the settings `origin` and `local` the same internally. Third-party replication systems may use these two values for their internal purposes, for example using `local` to designate a session whose changes should not be replicated.

Since foreign keys are implemented as triggers, setting this parameter to `replica` also disables all foreign key checks, which can leave data in an inconsistent state if improperly used.

`statement_timeout` (`integer`)

Abort any statement that takes more than the specified amount of time. If `log_min_error_statement` is set to `ERROR` or lower, the statement that timed out will also be logged. If this value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

The timeout is measured from the time a command arrives at the server until it is completed by the server. If multiple SQL statements appear in a single simple-query message, the timeout is applied to each statement separately. (PostgreSQL versions before 13 usually treated the timeout as applying to the whole query string.) In extended query protocol, the timeout starts running when any query-related message (Parse, Bind, Execute, Describe) arrives, and it is canceled by completion of an Execute or Sync message.

Setting `statement_timeout` in `postgresql.conf` is not recommended because it would affect all sessions.

`transaction_timeout` (`integer`)

Terminate any session that spans longer than the specified amount of time in a transaction. The limit applies both to explicit transactions (started with `BEGIN`) and to an implicitly started transaction corresponding to a single statement. If this value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

If `transaction_timeout` is shorter or equal to `idle_in_transaction_session_timeout` or `statement_timeout` then the longer timeout is ignored.

Setting `transaction_timeout` in `postgresql.conf` is not recommended because it would affect all sessions.

> **Note**
>
> Prepared transactions are not subject to this timeout.

`lock_timeout` (`integer`)

Abort any statement that waits longer than the specified amount of time while attempting to acquire a lock on a table, index, row, or other database object. The time limit applies separately to each lock acquisition attempt. The limit applies both to explicit locking requests (such as `LOCK TABLE`, or `SELECT FOR UPDATE` without `NOWAIT`) and to implicitly-acquired locks. If this value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

Unlike `statement_timeout`, this timeout can only occur while waiting for locks. Note that if `statement_timeout` is nonzero, it is rather pointless to set `lock_timeout` to the same or larger value, since the statement timeout would always trigger first. If `log_min_error_statement` is set to `ERROR` or lower, the statement that timed out will be logged.

Setting `lock_timeout` in `postgresql.conf` is not recommended because it would affect all sessions.

`idle_in_transaction_session_timeout` (`integer`)

Terminate any session that has been idle (that is, waiting for a client query) within an open transaction for longer than the specified amount of time. If this value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

This option can be used to ensure that idle sessions do not hold locks for an unreasonable amount of time. Even when no significant locks are held, an open transaction prevents vacuuming away recently-dead tuples that may be visible only to this transaction; so remaining idle for a long time can contribute to table bloat. See Section 24.1 for more details.

`idle_session_timeout` (`integer`)

Terminate any session that has been idle (that is, waiting for a client query), but not within an open transaction, for longer than the specified amount of time. If this value is specified without units, it is taken as milliseconds. A value of zero (the default) disables the timeout.

Unlike the case with an open transaction, an idle session without a transaction imposes no large costs on the server, so there is less need to enable this timeout than `idle_in_transaction_session_timeout`.

Be wary of enforcing this timeout on connections made through connection-pooling software or other middleware, as such a layer may not react well to unexpected connection closure. It may be helpful to enable this timeout only for interactive sessions, perhaps by applying it only to particular users.

`vacuum_freeze_table_age` (`integer`)

`VACUUM` performs an aggressive scan if the table's `pg_class.relfrozenxid` field has reached the age specified by this setting. An aggressive scan differs from a regular `VACUUM` in that it visits every page that might contain unfrozen XIDs or MXIDs, not just those that might contain dead tuples. The default is 150 million transactions. Although users can set this value anywhere from zero to two billion, `VACUUM` will silently limit the effective value to 95% of autovacuum_freeze_max_age, so that a periodic manual `VACUUM` has a chance to run before an anti-wraparound autovacuum is launched for the table. For more information see Section 24.1.5.

`vacuum_freeze_min_age` (`integer`)

Specifies the cutoff age (in transactions) that `VACUUM` should use to decide whether to trigger freezing of pages that have an older XID. The default is 50 million transactions. Although users can set this value anywhere from zero to one billion, `VACUUM` will silently limit the effective value to half the value of autovacuum_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 24.1.5.

`vacuum_failsafe_age` (`integer`)

Specifies the maximum age (in transactions) that a table's `pg_class.relfrozenxid` field can attain before `VACUUM` takes extraordinary measures to avoid system-wide transaction ID wraparound failure. This is `VACUUM`'s strategy of last resort. The failsafe typically triggers when an autovacuum to prevent transaction ID wraparound has already been running for some time, though it's possible for the failsafe to trigger during any `VACUUM`.

When the failsafe is triggered, any cost-based delay that is in effect will no longer be applied, further non-essential maintenance tasks (such as index vacuuming) are bypassed, and any *Buffer Access Strategy* in use will be disabled resulting in `VACUUM` being free to make use of all of *shared buffers*.

The default is 1.6 billion transactions. Although users can set this value anywhere from zero to 2.1 billion, `VACUUM` will silently adjust the effective value to no less than 105% of autovacuum_freeze_max_age.

`vacuum_multixact_freeze_table_age` (`integer`)

`VACUUM` performs an aggressive scan if the table's `pg_class.relminmxid` field has reached the age specified by this setting. An aggressive scan differs from a regular `VACUUM` in that it visits every page that might contain unfrozen XIDs or MXIDs, not just those that might contain dead tuples. The default is 150 million multixacts. Although users can set this value anywhere from

zero to two billion, VACUUM will silently limit the effective value to 95% of autovacuum_multixact_freeze_max_age, so that a periodic manual VACUUM has a chance to run before an anti-wraparound is launched for the table. For more information see Section 24.1.5.1.

vacuum_multixact_freeze_min_age (integer)

Specifies the cutoff age (in multixacts) that VACUUM should use to decide whether to trigger freezing of pages with an older multixact ID. The default is 5 million multixacts. Although users can set this value anywhere from zero to one billion, VACUUM will silently limit the effective value to half the value of autovacuum_multixact_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 24.1.5.1.

vacuum_multixact_failsafe_age (integer)

Specifies the maximum age (in multixacts) that a table's pg_class.relminmxid field can attain before VACUUM takes extraordinary measures to avoid system-wide multixact ID wraparound failure. This is VACUUM's strategy of last resort. The failsafe typically triggers when an autovacuum to prevent transaction ID wraparound has already been running for some time, though it's possible for the failsafe to trigger during any VACUUM.

When the failsafe is triggered, any cost-based delay that is in effect will no longer be applied, and further non-essential maintenance tasks (such as index vacuuming) are bypassed.

The default is 1.6 billion multixacts. Although users can set this value anywhere from zero to 2.1 billion, VACUUM will silently adjust the effective value to no less than 105% of autovacuum_multixact_freeze_max_age.

bytea_output (enum)

Sets the output format for values of type bytea. Valid values are hex (the default) and escape (the traditional PostgreSQL format). See Section 8.4 for more information. The bytea type always accepts both formats on input, regardless of this setting.

xmlbinary (enum)

Sets how binary values are to be encoded in XML. This applies for example when bytea values are converted to XML by the functions xmlelement or xmlforest. Possible values are base64 and hex, which are both defined in the XML Schema standard. The default is base64. For further information about XML-related functions, see Section 9.15.

The actual choice here is mostly a matter of taste, constrained only by possible restrictions in client applications. Both methods support all possible values, although the hex encoding will be somewhat larger than the base64 encoding.

xmloption (enum)

Sets whether DOCUMENT or CONTENT is implicit when converting between XML and character string values. See Section 8.13 for a description of this. Valid values are DOCUMENT and CONTENT. The default is CONTENT.

According to the SQL standard, the command to set this option is

```
SET XML OPTION { DOCUMENT | CONTENT };
```

This syntax is also available in PostgreSQL.

gin_pending_list_limit (integer)

Sets the maximum size of a GIN index's pending list, which is used when fastupdate is enabled. If the list grows larger than this maximum size, it is cleaned up by moving the entries in it

to the index's main GIN data structure in bulk. If this value is specified without units, it is taken as kilobytes. The default is four megabytes (4MB). This setting can be overridden for individual GIN indexes by changing index storage parameters. See Section 64.4.4.1 and Section 64.4.5 for more information.

createrole_self_grant (string)

If a user who has CREATEROLE but not SUPERUSER creates a role, and if this is set to a non-empty value, the newly-created role will be granted to the creating user with the options specified. The value must be set, inherit, or a comma-separated list of these. The default value is an empty string, which disables the feature.

The purpose of this option is to allow a CREATEROLE user who is not a superuser to automatically inherit, or automatically gain the ability to SET ROLE to, any created users. Since a CREATEROLE user is always implicitly granted ADMIN OPTION on created roles, that user could always execute a GRANT statement that would achieve the same effect as this setting. However, it can be convenient for usability reasons if the grant happens automatically. A superuser automatically inherits the privileges of every role and can always SET ROLE to any role, and this setting can be used to produce a similar behavior for CREATEROLE users for users which they create.

event_triggers (boolean)

Allow temporarily disabling execution of event triggers in order to troubleshoot and repair faulty event triggers. All event triggers will be disabled by setting it to false. Setting the value to true allows all event triggers to fire, this is the default value. Only superusers and users with the appropriate SET privilege can change this setting.

restrict_nonsystem_relation_kind (string)

Set relation kinds for which access to non-system relations is prohibited. The value takes the form of a comma-separated list of relation kinds. Currently, the supported relation kinds are view and foreign-table.

19.11.2. Locale and Formatting

DateStyle (string)

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. For historical reasons, this variable contains two independent components: the output format specification (ISO, Postgres, SQL, or German) and the input/output specification for year/month/day ordering (DMY, MDY, or YMD). These can be set separately or together. The keywords Euro and European are synonyms for DMY; the keywords US, NonEuro, and NonEuropean are synonyms for MDY. See Section 8.5 for more information. The built-in default is ISO, MDY, but initdb will initialize the configuration file with a setting that corresponds to the behavior of the chosen lc_time locale.

IntervalStyle (enum)

Sets the display format for interval values. The value sql_standard will produce output matching SQL standard interval literals. The value postgres (which is the default) will produce output matching PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to ISO. The value postgres_verbose will produce output matching PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to non-ISO output. The value iso_8601 will produce output matching the time interval “format with designators” defined in section 4.4.3.2 of ISO 8601.

The IntervalStyle parameter also affects the interpretation of ambiguous interval input. See Section 8.5.4 for more information.

`TimeZone` (`string`)

Sets the time zone for displaying and interpreting time stamps. The built-in default is `GMT`, but that is typically overridden in `postgresql.conf`; initdb will install a setting there corresponding to its system environment. See Section 8.5.3 for more information.

`timezone_abbreviations` (`string`)

Sets the collection of time zone abbreviations that will be accepted by the server for datetime input. The default is `'Default'`, which is a collection that works in most of the world; there are also `'Australia'` and `'India'`, and other collections can be defined for a particular installation. See Section B.4 for more information.

`extra_float_digits` (`integer`)

This parameter adjusts the number of digits used for textual output of floating-point values, including `float4`, `float8`, and geometric data types.

If the value is 1 (the default) or above, float values are output in shortest-precise format; see Section 8.1.3. The actual number of digits generated depends only on the value being output, not on the value of this parameter. At most 17 digits are required for `float8` values, and 9 for `float4` values. This format is both fast and precise, preserving the original binary float value exactly when correctly read. For historical compatibility, values up to 3 are permitted.

If the value is zero or negative, then the output is rounded to a given decimal precision. The precision used is the standard number of digits for the type (`FLT_DIG` or `DBL_DIG` as appropriate) reduced according to the value of this parameter. (For example, specifying -1 will cause `float4` values to be output rounded to 5 significant digits, and `float8` values rounded to 14 digits.) This format is slower and does not preserve all the bits of the binary float value, but may be more human-readable.

> **Note**
>
> The meaning of this parameter, and its default value, changed in PostgreSQL 12; see Section 8.1.3 for further discussion.

`client_encoding` (`string`)

Sets the client-side encoding (character set). The default is to use the database encoding. The character sets supported by the PostgreSQL server are described in Section 23.3.1.

`lc_messages` (`string`)

Sets the language in which messages are displayed. Acceptable values are system-dependent; see Section 23.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Only superusers and users with the appropriate `SET` privilege can change this setting.

`lc_monetary` (`string`)

Sets the locale to use for formatting monetary amounts, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 23.1 for more information. If

this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_numeric` (`string`)

Sets the locale to use for formatting numbers, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 23.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_time` (`string`)

Sets the locale to use for formatting dates and times, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 23.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`icu_validation_level` (`enum`)

When ICU locale validation problems are encountered, controls which message level is used to report the problem. Valid values are `DISABLED`, `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, and `LOG`.

If set to `DISABLED`, does not report validation problems at all. Otherwise reports problems at the given message level. The default is `WARNING`.

`default_text_search_config` (`string`)

Selects the text search configuration that is used by those variants of the text search functions that do not have an explicit argument specifying the configuration. See Chapter 12 for further information. The built-in default is `pg_catalog.simple`, but initdb will initialize the configuration file with a setting that corresponds to the chosen `lc_ctype` locale, if a configuration matching that locale can be identified.

19.11.3. Shared Library Preloading

Several settings are available for preloading shared libraries into the server, in order to load additional functionality or achieve performance benefits. For example, a setting of `'$libdir/mylib'` would cause `mylib.so` (or on some platforms, `mylib.sl`) to be preloaded from the installation's standard library directory. The differences between the settings are when they take effect and what privileges are required to change them.

PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax `'$libdir/plXXX'` where `XXX` is `pgsql`, `perl`, `tcl`, or `python`.

Only shared libraries specifically intended to be used with PostgreSQL can be loaded this way. Every PostgreSQL-supported library has a “magic block” that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way. You might be able to use operating-system facilities such as `LD_PRELOAD` for that.

In general, refer to the documentation of a specific module for the recommended way to load that module.

`local_preload_libraries` (`string`)

This variable specifies one or more shared libraries that are to be preloaded at connection start. It contains a comma-separated list of library names, where each name is interpreted as for the `LOAD` command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail.

This option can be set by any user. Because of that, the libraries that can be loaded are restricted to those appearing in the `plugins` subdirectory of the installation's standard library directory. (It is the database administrator's responsibility to ensure that only "safe" libraries are installed there.) Entries in `local_preload_libraries` can specify this directory explicitly, for example `$libdir/plugins/mylib`, or just specify the library name — `mylib` would have the same effect as `$libdir/plugins/mylib`.

The intent of this feature is to allow unprivileged users to load debugging or performance-measurement libraries into specific sessions without requiring an explicit `LOAD` command. To that end, it would be typical to set this parameter using the `PGOPTIONS` environment variable on the client or by using `ALTER ROLE SET`.

However, unless a module is specifically designed to be used in this way by non-superusers, this is usually not the right setting to use. Look at session_preload_libraries instead.

`session_preload_libraries` (`string`)

This variable specifies one or more shared libraries that are to be preloaded at connection start. It contains a comma-separated list of library names, where each name is interpreted as for the `LOAD` command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail. Only superusers and users with the appropriate `SET` privilege can change this setting.

The intent of this feature is to allow debugging or performance-measurement libraries to be loaded into specific sessions without an explicit `LOAD` command being given. For example, auto_explain could be enabled for all sessions under a given user name by setting this parameter with `ALTER ROLE SET`. Also, this parameter can be changed without restarting the server (but changes only take effect when a new session is started), so it is easier to add new modules this way, even if they should apply to all sessions.

Unlike shared_preload_libraries, there is no large performance advantage to loading a library at session start rather than when it is first used. There is some advantage, however, when connection pooling is used.

`shared_preload_libraries` (`string`)

This variable specifies one or more shared libraries to be preloaded at server start. It contains a comma-separated list of library names, where each name is interpreted as for the `LOAD` command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. This parameter can only be set at server start. If a specified library is not found, the server will fail to start.

Some libraries need to perform certain operations that can only take place at postmaster start, such as allocating shared memory, reserving light-weight locks, or starting background workers. Those libraries must be loaded at server start through this parameter. See the documentation of each library for details.

Other libraries can also be preloaded. By preloading a shared library, the library startup time is avoided when the library is first used. However, the time to start each new server process might increase slightly, even if that process never uses the library. So this parameter is recommended only for libraries that will be used in most sessions. Also, changing this parameter requires a server restart, so this is not the right setting to use for short-term debugging tasks, say. Use session_preload_libraries for that instead.

> **Note**
>
> On Windows hosts, preloading a library at server start will not reduce the time required to start each new server process; each server process will re-load all preload libraries. How-

ever, shared_preload_libraries is still useful on Windows hosts for libraries that need to perform operations at postmaster start time.

jit_provider (string)

This variable is the name of the JIT provider library to be used (see Section 30.4.2). The default is llvmjit. This parameter can only be set at server start.

If set to a non-existent library, JIT will not be available, but no error will be raised. This allows JIT support to be installed separately from the main PostgreSQL package.

19.11.4. Other Defaults

dynamic_library_path (string)

If a dynamically loadable module needs to be opened and the file name specified in the CREATE FUNCTION or LOAD command does not have a directory component (i.e., the name does not contain a slash), the system will search this path for the required file.

The value for dynamic_library_path must be a list of absolute directory paths separated by colons (or semi-colons on Windows). If a list element starts with the special string $libdir, the compiled-in PostgreSQL package library directory is substituted for $libdir; this is where the modules provided by the standard PostgreSQL distribution are installed. (Use pg_config --pkglibdir to find out the name of this directory.) For example:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/
my_project/lib:$libdir'
```

or, in a Windows environment:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;
$libdir'
```

The default value for this parameter is '$libdir'. If the value is set to an empty string, the automatic path search is turned off.

This parameter can be changed at run time by superusers and users with the appropriate SET privilege, but a setting done that way will only persist until the end of the client connection, so this method should be reserved for development purposes. The recommended way to set this parameter is in the postgresql.conf configuration file.

gin_fuzzy_search_limit (integer)

Soft upper limit of the size of the set returned by GIN index scans. For more information see Section 64.4.5.

19.12. Lock Management

deadlock_timeout (integer)

This is the amount of time to wait on a lock before checking to see if there is a deadlock condition. The check for deadlock is relatively expensive, so the server doesn't run it every time it waits for a lock. We optimistically assume that deadlocks are not common in production applications and just wait on the lock for a while before checking for a deadlock. Increasing this value reduces the amount of time wasted in needless deadlock checks, but slows down reporting of real deadlock errors. If this value is specified without units, it is taken as milliseconds. The default is one second

(1s), which is probably about the smallest value you would want in practice. On a heavily loaded server you might want to raise it. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock. Only superusers and users with the appropriate SET privilege can change this setting.

When log_lock_waits is set, this parameter also determines the amount of time to wait before a log message is issued about the lock wait. If you are trying to investigate locking delays you might want to set a shorter than normal deadlock_timeout.

max_locks_per_transaction (integer)

The shared lock table has space for max_locks_per_transaction objects (e.g., tables) per server process or prepared transaction; hence, no more than this many distinct objects can be locked at any one time. This parameter limits the average number of object locks used by each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table. This is *not* the number of rows that can be locked; that value is unlimited. The default, 64, has historically proven sufficient, but you might need to raise this value if you have queries that touch many different tables in a single transaction, e.g., query of a parent table with many children. This parameter can only be set at server start.

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

max_pred_locks_per_transaction (integer)

The shared predicate lock table has space for max_pred_locks_per_transaction objects (e.g., tables) per server process or prepared transaction; hence, no more than this many distinct objects can be locked at any one time. This parameter limits the average number of object locks used by each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table. This is *not* the number of rows that can be locked; that value is unlimited. The default, 64, has historically proven sufficient, but you might need to raise this value if you have clients that touch many different tables in a single serializable transaction. This parameter can only be set at server start.

max_pred_locks_per_relation (integer)

This controls how many pages or tuples of a single relation can be predicate-locked before the lock is promoted to covering the whole relation. Values greater than or equal to zero mean an absolute limit, while negative values mean max_pred_locks_per_transaction divided by the absolute value of this setting. The default is -2, which keeps the behavior from previous versions of PostgreSQL. This parameter can only be set in the postgresql.conf file or on the server command line.

max_pred_locks_per_page (integer)

This controls how many rows on a single page can be predicate-locked before the lock is promoted to covering the whole page. The default is 2. This parameter can only be set in the postgresql.conf file or on the server command line.

19.13. Version and Platform Compatibility

19.13.1. Previous PostgreSQL Versions

array_nulls (boolean)

This controls whether the array input parser recognizes unquoted NULL as specifying a null array element. By default, this is on, allowing array values containing null values to be entered. However, PostgreSQL versions before 8.2 did not support null values in arrays, and therefore would treat NULL as specifying a normal array element with the string value “NULL”. For backward compatibility with applications that require the old behavior, this variable can be turned off.

Note that it is possible to create array values containing null values even when this variable is `off`.

`backslash_quote` (`enum`)

This controls whether a quote mark can be represented by `\'` in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it (`''`) but PostgreSQL has historically also accepted `\'`. However, use of `\'` creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII \. If client-side code does escaping incorrectly then an SQL-injection attack is possible. This risk can be prevented by making the server reject queries in which a quote mark appears to be escaped by a backslash. The allowed values of `backslash_quote` are `on` (allow `\'` always), `off` (reject always), and `safe_encoding` (allow only if client encoding does not allow ASCII \ within a multibyte character). `safe_encoding` is the default setting.

Note that in a standard-conforming string literal, \ just means \ anyway. This parameter only affects the handling of non-standard-conforming literals, including escape string syntax (`E'...'`).

`escape_string_warning` (`boolean`)

When on, a warning is issued if a backslash (\) appears in an ordinary string literal (`'...'` syntax) and `standard_conforming_strings` is off. The default is `on`.

Applications that wish to use backslash as escape should be modified to use escape string syntax (`E'...'`), because the default behavior of ordinary strings is now to treat backslash as an ordinary character, per SQL standard. This variable can be enabled to help locate code that needs to be changed.

`lo_compat_privileges` (`boolean`)

In PostgreSQL releases prior to 9.0, large objects did not have access privileges and were, therefore, always readable and writable by all users. Setting this variable to `on` disables the new privilege checks, for compatibility with prior releases. The default is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

Setting this variable does not disable all security checks related to large objects — only those for which the default behavior has changed in PostgreSQL 9.0.

`quote_all_identifiers` (`boolean`)

When the database generates SQL, force all identifiers to be quoted, even if they are not (currently) keywords. This will affect the output of `EXPLAIN` as well as the results of functions like `pg_get_viewdef`. See also the `--quote-all-identifiers` option of pg_dump and pg_dumpall.

`standard_conforming_strings` (`boolean`)

This controls whether ordinary string literals (`'...'`) treat backslashes literally, as specified in the SQL standard. Beginning in PostgreSQL 9.1, the default is `on` (prior releases defaulted to `off`). Applications can check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (`E'...'`) is supported. Escape string syntax (Section 4.1.2.2) should be used if an application desires backslashes to be treated as escape characters.

`synchronize_seqscans` (`boolean`)

This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then “wrap around” the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in the row ordering returned by queries that have no `ORDER BY` clause.

Setting this parameter to `off` ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table. The default is `on`.

19.13.2. Platform and Client Compatibility

`transform_null_equals` (`boolean`)

When on, expressions of the form *`expr`* `= NULL` (or `NULL =` *`expr`*) are treated as *`expr`* `IS NULL`, that is, they return true if *`expr`* evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of *`expr`* `= NULL` is to always return null (unknown). Therefore this parameter defaults to `off`.

However, filtered forms in Microsoft Access generate queries that appear to use *`expr`* `= NULL` to test for null values, so if you use that interface to access the database you might want to turn this option on. Since expressions of the form *`expr`* `= NULL` always return the null value (using the SQL standard interpretation), they are not very useful and do not appear often in normal applications so this option does little harm in practice. But new users are frequently confused about the semantics of expressions involving null values, so this option is off by default.

Note that this option only affects the exact form `= NULL`, not other comparison operators or other expressions that are computationally equivalent to some expression involving the equals operator (such as `IN`). Thus, this option is not a general fix for bad programming.

Refer to Section 9.2 for related information.

`allow_alter_system` (`boolean`)

When `allow_alter_system` is set to `off`, an error is returned if the `ALTER SYSTEM` command is executed. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default value is `on`.

Note that this setting must not be regarded as a security feature. It only disables the `ALTER SYSTEM` command. It does not prevent a superuser from changing the configuration using other SQL commands. A superuser has many ways of executing shell commands at the operating system level, and can therefore modify `postgresql.auto.conf` regardless of the value of this setting.

Turning this setting off is intended for environments where the configuration of PostgreSQL is managed by some external tool. In such environments, a well intentioned superuser might *mistakenly* use `ALTER SYSTEM` to change the configuration instead of using the external tool. This might result in unintended behavior, such as the external tool overwriting the change at some later point in time when it updates the configuration. Setting this parameter to `off` can help avoid such mistakes.

This parameter only controls the use of `ALTER SYSTEM`. The settings stored in `postgresql.auto.conf` take effect even if `allow_alter_system` is set to `off`.

19.14. Error Handling

`exit_on_error` (`boolean`)

If on, any error will terminate the current session. By default, this is set to off, so that only FATAL errors will terminate the session.

`restart_after_crash` (`boolean`)

When set to on, which is the default, PostgreSQL will automatically reinitialize after a backend crash. Leaving this value set to on is normally the best way to maximize the availability of the

database. However, in some circumstances, such as when PostgreSQL is being invoked by clusterware, it may be useful to disable the restart so that the clusterware can gain control and take any actions it deems appropriate.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`data_sync_retry` (`boolean`)

When set to off, which is the default, PostgreSQL will raise a PANIC-level error on failure to flush modified data files to the file system. This causes the database server to crash. This parameter can only be set at server start.

On some operating systems, the status of data in the kernel's page cache is unknown after a write-back failure. In some cases it might have been entirely forgotten, making it unsafe to retry; the second attempt may be reported as successful, when in fact the data has been lost. In these circumstances, the only way to avoid data loss is to recover from the WAL after any failure is reported, preferably after investigating the root cause of the failure and replacing any faulty hardware.

If set to on, PostgreSQL will instead report an error but continue to run so that the data flushing operation can be retried in a later checkpoint. Only set it to on after investigating the operating system's treatment of buffered data in case of write-back failure.

`recovery_init_sync_method` (`enum`)

When set to `fsync`, which is the default, PostgreSQL will recursively open and synchronize all files in the data directory before crash recovery begins. The search for files will follow symbolic links for the WAL directory and each configured tablespace (but not any other symbolic links). This is intended to make sure that all WAL and data files are durably stored on disk before replaying changes. This applies whenever starting a database cluster that did not shut down cleanly, including copies created with pg_basebackup.

On Linux, `syncfs` may be used instead, to ask the operating system to synchronize the file systems that contain the data directory, the WAL files and each tablespace (but not any other file systems that may be reachable through symbolic links). This may be a lot faster than the `fsync` setting, because it doesn't need to open each file one by one. On the other hand, it may be slower if a file system is shared by other applications that modify a lot of files, since those files will also be written to disk. Furthermore, on versions of Linux before 5.8, I/O errors encountered while writing data to disk may not be reported to PostgreSQL, and relevant error messages may appear only in kernel logs.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

19.15. Preset Options

The following “parameters” are read-only. As such, they have been excluded from the sample `postgresql.conf` file. These options report various aspects of PostgreSQL behavior that might be of interest to certain applications, particularly administrative front-ends. Most of them are determined when PostgreSQL is compiled or when it is installed.

`block_size` (`integer`)

Reports the size of a disk block. It is determined by the value of `BLCKSZ` when building the server. The default value is 8192 bytes. The meaning of some configuration variables (such as shared_buffers) is influenced by `block_size`. See Section 19.4 for information.

`data_checksums` (`boolean`)

Reports whether data checksums are enabled for this cluster. See data checksums for more information.

`data_directory_mode` (`integer`)

On Unix systems this parameter reports the permissions the data directory (defined by data_directory) had at server startup. (On Microsoft Windows this parameter will always display `0700`.) See group access for more information.

`debug_assertions` (`boolean`)

Reports whether PostgreSQL has been built with assertions enabled. That is the case if the macro `USE_ASSERT_CHECKING` is defined when PostgreSQL is built (accomplished e.g., by the `configure` option `--enable-cassert`). By default PostgreSQL is built without assertions.

`huge_pages_status` (`enum`)

Reports the state of huge pages in the current instance: `on`, `off`, or `unknown` (if displayed with `postgres -C`). This parameter is useful to determine whether allocation of huge pages was successful under `huge_pages=try`. See huge_pages for more information.

`integer_datetimes` (`boolean`)

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times. As of PostgreSQL 10, this is always `on`.

`in_hot_standby` (`boolean`)

Reports whether the server is currently in hot standby mode. When this is `on`, all transactions are forced to be read-only. Within a session, this can change only if the server is promoted to be primary. See Section 26.4 for more information.

`max_function_args` (`integer`)

Reports the maximum number of function arguments. It is determined by the value of `FUNC_MAX_ARGS` when building the server. The default value is 100 arguments.

`max_identifier_length` (`integer`)

Reports the maximum identifier length. It is determined as one less than the value of `NAMEDATALEN` when building the server. The default value of `NAMEDATALEN` is 64; therefore the default `max_identifier_length` is 63 bytes, which can be less than 63 characters when using multibyte encodings.

`max_index_keys` (`integer`)

Reports the maximum number of index keys. It is determined by the value of `INDEX_MAX_KEYS` when building the server. The default value is 32 keys.

`segment_size` (`integer`)

Reports the number of blocks (pages) that can be stored within a file segment. It is determined by the value of `RELSEG_SIZE` when building the server. The maximum size of a segment file in bytes is equal to `segment_size` multiplied by `block_size`; by default this is 1GB.

`server_encoding` (`string`)

Reports the database encoding (character set). It is determined when the database is created. Ordinarily, clients need only be concerned with the value of client_encoding.

`server_version` (`string`)

Reports the version number of the server. It is determined by the value of `PG_VERSION` when building the server.

`server_version_num` (`integer`)

Reports the version number of the server as an integer. It is determined by the value of `PG_VERSION_NUM` when building the server.

`shared_memory_size` (`integer`)

Reports the size of the main shared memory area, rounded up to the nearest megabyte.

`shared_memory_size_in_huge_pages` (`integer`)

Reports the number of huge pages that are needed for the main shared memory area based on the specified huge_page_size. If huge pages are not supported, this will be `-1`.

This setting is supported only on Linux. It is always set to `-1` on other platforms. For more details about using huge pages on Linux, see Section 18.4.5.

`ssl_library` (`string`)

Reports the name of the SSL library that this PostgreSQL server was built with (even if SSL is not currently configured or in use on this instance), for example `OpenSSL`, or an empty string if none.

`wal_block_size` (`integer`)

Reports the size of a WAL disk block. It is determined by the value of `XLOG_BLCKSZ` when building the server. The default value is 8192 bytes.

`wal_segment_size` (`integer`)

Reports the size of write ahead log segments. The default value is 16MB. See Section 28.5 for more information.

19.16. Customized Options

This feature was designed to allow parameters not normally known to PostgreSQL to be added by add-on modules (such as procedural languages). This allows extension modules to be configured in the standard ways.

Custom options have two-part names: an extension name, then a dot, then the parameter name proper, much like qualified names in SQL. An example is `plpgsql.variable_conflict`.

Because custom options may need to be set in processes that have not loaded the relevant extension module, PostgreSQL will accept a setting for any two-part parameter name. Such variables are treated as placeholders and have no function until the module that defines them is loaded. When an extension module is loaded, it will add its variable definitions and convert any placeholder values according to those definitions. If there are any unrecognized placeholders that begin with its extension name, warnings are issued and those placeholders are removed.

19.17. Developer Options

The following parameters are intended for developer testing, and should never be used on a production database. However, some of them can be used to assist with the recovery of severely damaged databases. As such, they have been excluded from the sample `postgresql.conf` file. Note that many of these parameters require special source compilation flags to work at all.

`allow_in_place_tablespaces` (`boolean`)

Allows tablespaces to be created as directories inside `pg_tblspc`, when an empty location string is provided to the `CREATE TABLESPACE` command. This is intended to allow testing replication

scenarios where primary and standby servers are running on the same machine. Such directories are likely to confuse backup tools that expect to find only symbolic links in that location. Only superusers and users with the appropriate SET privilege can change this setting.

allow_system_table_mods (boolean)

Allows modification of the structure of system tables as well as certain other risky actions on system tables. This is otherwise not allowed even for superusers. Ill-advised use of this setting can cause irretrievable data loss or seriously corrupt the database system. Only superusers and users with the appropriate SET privilege can change this setting.

backtrace_functions (string)

This parameter contains a comma-separated list of C function names. If an error is raised and the name of the internal C function where the error happens matches a value in the list, then a backtrace is written to the server log together with the error message. This can be used to debug specific areas of the source code.

Backtrace support is not available on all platforms, and the quality of the backtraces depends on compilation options.

Only superusers and users with the appropriate SET privilege can change this setting.

debug_discard_caches (integer)

When set to 1, each system catalog cache entry is invalidated at the first possible opportunity, whether or not anything that would render it invalid really occurred. Caching of system catalogs is effectively disabled as a result, so the server will run extremely slowly. Higher values run the cache invalidation recursively, which is even slower and only useful for testing the caching logic itself. The default value of 0 selects normal catalog caching behavior.

This parameter can be very helpful when trying to trigger hard-to-reproduce bugs involving concurrent catalog changes, but it is otherwise rarely needed. See the source code files inval.c and pg_config_manual.h for details.

This parameter is supported when DISCARD_CACHES_ENABLED was defined at compile time (which happens automatically when using the configure option --enable-cassert). In production builds, its value will always be 0 and attempts to set it to another value will raise an error.

debug_io_direct (string)

Ask the kernel to minimize caching effects for relation data and WAL files using O_DIRECT (most Unix-like systems), F_NOCACHE (macOS) or FILE_FLAG_NO_BUFFERING (Windows).

May be set to an empty string (the default) to disable use of direct I/O, or a comma-separated list of operations that should use direct I/O. The valid options are data for main data files, wal for WAL files, and wal_init for WAL files when being initially allocated.

Some operating systems and file systems do not support direct I/O, so non-default settings may be rejected at startup or cause errors.

Currently this feature reduces performance, and is intended for developer testing only.

debug_parallel_query (enum)

Allows the use of parallel queries for testing purposes even in cases where no performance benefit is expected. The allowed values of debug_parallel_query are off (use parallel mode only when it is expected to improve performance), on (force parallel query for all queries for which it is thought to be safe), and regress (like on, but with additional behavior changes as explained below).

More specifically, setting this value to `on` will add a `Gather` node to the top of any query plan for which this appears to be safe, so that the query runs inside of a parallel worker. Even when a parallel worker is not available or cannot be used, operations such as starting a subtransaction that would be prohibited in a parallel query context will be prohibited unless the planner believes that this will cause the query to fail. If failures or unexpected results occur when this option is set, some functions used by the query may need to be marked `PARALLEL UNSAFE` (or, possibly, `PARALLEL RESTRICTED`).

Setting this value to `regress` has all of the same effects as setting it to `on` plus some additional effects that are intended to facilitate automated regression testing. Normally, messages from a parallel worker include a context line indicating that, but a setting of `regress` suppresses this line so that the output is the same as in non-parallel execution. Also, the `Gather` nodes added to plans by this setting are hidden in `EXPLAIN` output so that the output matches what would be obtained if this setting were turned `off`.

`ignore_system_indexes` (`boolean`)

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes. This parameter cannot be changed after session start.

`post_auth_delay` (`integer`)

The amount of time to delay when a new server process is started, after it conducts the authentication procedure. This is intended to give developers an opportunity to attach to the server process with a debugger. If this value is specified without units, it is taken as seconds. A value of zero (the default) disables the delay. This parameter cannot be changed after session start.

`pre_auth_delay` (`integer`)

The amount of time to delay just after a new server process is forked, before it conducts the authentication procedure. This is intended to give developers an opportunity to attach to the server process with a debugger to trace down misbehavior in authentication. If this value is specified without units, it is taken as seconds. A value of zero (the default) disables the delay. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`trace_notify` (`boolean`)

Generates a great amount of debugging output for the `LISTEN` and `NOTIFY` commands. client_min_messages or log_min_messages must be `DEBUG1` or lower to send this output to the client or server logs, respectively.

`trace_sort` (`boolean`)

If on, emit information about resource usage during sort operations. This parameter is only available if the `TRACE_SORT` macro was defined when PostgreSQL was compiled. (However, `TRACE_SORT` is currently defined by default.)

`trace_locks` (`boolean`)

If on, emit information about lock usage. Information dumped includes the type of lock operation, the type of lock and the unique identifier of the object being locked or unlocked. Also included are bit masks for the lock types already granted on this object as well as for the lock types awaited on this object. For each lock type a count of the number of granted locks and waiting locks is also dumped as well as the totals. An example of the log file output is shown here:

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
```

```
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844)
 id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844)
 id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

Details of the structure being dumped may be found in `src/include/storage/lock.h`.

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`trace_lwlocks` (`boolean`)

If on, emit information about lightweight lock usage. Lightweight locks are intended primarily to provide mutual exclusion of access to shared-memory data structures.

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`trace_userlocks` (`boolean`)

If on, emit information about user lock usage. Output is the same as for `trace_locks`, only for advisory locks.

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`trace_lock_oidmin` (`integer`)

If set, do not trace locks for tables below this OID (used to avoid output on system tables).

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`trace_lock_table` (`integer`)

Unconditionally trace locks on this table (OID).

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`debug_deadlocks` (`boolean`)

If set, dumps information about all current locks when a deadlock timeout occurs.

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`log_btree_build_stats` (`boolean`)

If set, logs system resource usage statistics (memory and CPU) on various B-tree operations.

This parameter is only available if the `BTREE_BUILD_STATS` macro was defined when PostgreSQL was compiled.

`wal_consistency_checking` (`string`)

This parameter is intended to be used to check for bugs in the WAL redo routines. When enabled, full-page images of any buffers modified in conjunction with the WAL record are added to the record. If the record is subsequently replayed, the system will first apply each record and then test whether the buffers modified by the record match the stored images. In certain cases (such as hint bits), minor variations are acceptable, and will be ignored. Any unexpected differences will result in a fatal error, terminating recovery.

The default value of this setting is the empty string, which disables the feature. It can be set to `all` to check all records, or to a comma-separated list of resource managers to check only records originating from those resource managers. Currently, the supported resource managers are `heap`, `heap2`, `btree`, `hash`, `gin`, `gist`, `sequence`, `spgist`, `brin`, and `generic`. Extensions may define additional resource managers. Only superusers and users with the appropriate `SET` privilege can change this setting.

`wal_debug` (`boolean`)

If on, emit WAL-related debugging output. This parameter is only available if the `WAL_DEBUG` macro was defined when PostgreSQL was compiled.

`ignore_checksum_failure` (`boolean`)

Only has effect if data checksums are enabled.

Detection of a checksum failure during a read normally causes PostgreSQL to report an error, aborting the current transaction. Setting `ignore_checksum_failure` to on causes the system to ignore the failure (but still report a warning), and continue processing. This behavior may *cause crashes, propagate or hide corruption, or other serious problems*. However, it may allow you to get past the error and retrieve undamaged tuples that might still be present in the table if the block header is still sane. If the header is corrupt an error will be reported even if this option is enabled. The default setting is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`zero_damaged_pages` (`boolean`)

Detection of a damaged page header normally causes PostgreSQL to report an error, aborting the current transaction. Setting `zero_damaged_pages` to on causes the system to instead report a warning, zero out the damaged page in memory, and continue processing. This behavior *will destroy data*, namely all the rows on the damaged page. However, it does allow you to get past the error and retrieve rows from any undamaged pages that might be present in the table. It is useful for recovering data if corruption has occurred due to a hardware or software error. You should generally not set this on until you have given up hope of recovering data from the damaged pages of a table. Zeroed-out pages are not forced to disk so it is recommended to recreate the table or the index before turning this parameter off again. The default setting is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`ignore_invalid_pages` (`boolean`)

If set to `off` (the default), detection of WAL records having references to invalid pages during recovery causes PostgreSQL to raise a PANIC-level error, aborting the recovery. Setting `ignore_invalid_pages` to `on` causes the system to ignore invalid page references in WAL records (but still report a warning), and continue the recovery. This behavior may *cause crashes, data loss, propagate or hide corruption, or other serious problems*. However, it may allow you to get past the PANIC-level error, to finish the recovery, and to cause the server to start up. The parameter can only be set at server start. It only has effect during recovery or in standby mode.

`jit_debugging_support` (`boolean`)

If LLVM has the required functionality, register generated functions with GDB. This makes debugging easier. The default setting is `off`. This parameter can only be set at server start.

`jit_dump_bitcode` (`boolean`)

Writes the generated LLVM IR out to the file system, inside data_directory. This is only useful for working on the internals of the JIT implementation. The default setting is `off`. Only superusers and users with the appropriate `SET` privilege can change this setting.

`jit_expressions` (`boolean`)

Determines whether expressions are JIT compiled, when JIT compilation is activated (see Section 30.2). The default is `on`.

`jit_profiling_support` (`boolean`)

If LLVM has the required functionality, emit the data needed to allow perf to profile functions generated by JIT. This writes out files to `~/.debug/jit/`; the user is responsible for performing cleanup when desired. The default setting is `off`. This parameter can only be set at server start.

`jit_tuple_deforming` (`boolean`)

Determines whether tuple deforming is JIT compiled, when JIT compilation is activated (see Section 30.2). The default is `on`.

`remove_temp_files_after_crash` (`boolean`)

When set to `on`, which is the default, PostgreSQL will automatically remove temporary files after a backend crash. If disabled, the files will be retained and may be used for debugging, for example. Repeated crashes may however result in accumulation of useless files. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`send_abort_for_crash` (`boolean`)

By default, after a backend crash the postmaster will stop remaining child processes by sending them SIGQUIT signals, which permits them to exit more-or-less gracefully. When this option is set to `on`, SIGABRT is sent instead. That normally results in production of a core dump file for each such child process. This can be handy for investigating the states of other processes after a crash. It can also consume lots of disk space in the event of repeated crashes, so do not enable this on systems you are not monitoring carefully. Beware that no support exists for cleaning up the core file(s) automatically. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`send_abort_for_kill` (`boolean`)

By default, after attempting to stop a child process with SIGQUIT, the postmaster will wait five seconds and then send SIGKILL to force immediate termination. When this option is set to `on`, SIGABRT is sent instead of SIGKILL. That normally results in production of a core dump file for each such child process. This can be handy for investigating the states of "stuck" child processes. It can also consume lots of disk space in the event of repeated crashes, so do not enable this on systems you are not monitoring carefully. Beware that no support exists for cleaning up the core file(s) automatically. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`debug_logical_replication_streaming` (`enum`)

The allowed values are `buffered` and `immediate`. The default is `buffered`. This parameter is intended to be used to test logical decoding and replication of large transactions. The effect of `debug_logical_replication_streaming` is different for the publisher and subscriber:

On the publisher side, `debug_logical_replication_streaming` allows streaming or serializing changes immediately in logical decoding. When set to `immediate`, stream each change if the `streaming` option of `CREATE SUBSCRIPTION` is enabled, otherwise, serialize

each change. When set to `buffered`, the decoding will stream or serialize changes when `logical_decoding_work_mem` is reached.

On the subscriber side, if the `streaming` option is set to `parallel`, `debug_logical_replication_streaming` can be used to direct the leader apply worker to send changes to the shared memory queue or to serialize all changes to the file. When set to `buffered`, the leader sends changes to parallel apply workers via a shared memory queue. When set to `immediate`, the leader serializes all changes to files and notifies the parallel apply workers to read and apply them at the end of the transaction.

19.18. Short Options

For convenience there are also single letter command-line option switches available for some parameters. They are described in Table 19.4. Some of these options exist for historical reasons, and their presence as a single-letter option does not necessarily indicate an endorsement to use the option heavily.

Table 19.4. Short Option Key

Short Option	Equivalent
`-B` *x*	`shared_buffers = x`
`-d` *x*	`log_min_messages = DEBUG`*x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fo`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop = off`, `enable_indexonlyscan = off`, `enable_seqscan = off`, `enable_tidscan = off`
`-F`	`fsync = off`
`-h` *x*	`listen_addresses = x`
`-i`	`listen_addresses = '*'`
`-k` *x*	`unix_socket_directories = x`
`-l`	`ssl = on`
`-N` *x*	`max_connections = x`
`-O`	`allow_system_table_mods = on`
`-p` *x*	`port = x`
`-P`	`ignore_system_indexes = on`
`-s`	`log_statement_stats = on`
`-S` *x*	`work_mem = x`
`-tpa`, `-tpl`, `-te`	`log_parser_stats = on`, `log_planner_stats = on`, `log_executor_stats = on`
`-W` *x*	`post_auth_delay = x`

Chapter 20. 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 21 for more information. Therefore, it is essential to restrict which database users can connect.

> **Note**
>
> As explained in Chapter 21, 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.

20.1. 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 `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`, calling the SQL function `pg_reload_conf()`, or using `kill -HUP`) to make it re-read the file.

> **Note**
>
> The preceding statement is not true on Microsoft Windows: there, any changes in the `pg_hba.conf` file are immediately applied by subsequent new connections.

The system view `pg_hba_file_rules` can be helpful for pre-testing changes to the `pg_hba.conf` file, or for diagnosing problems if loading of the file did not have the desired effects. Rows in the view with non-null `error` fields indicate problems in the corresponding lines of the file.

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. A record can be continued onto the next line by ending the line with a backslash. (Backslashes are not special except at the end of a line.) 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. Backslash line continuation applies even within quoted text or comments.

Each authentication 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.

Each record can be an include directive or an authentication record. Include directives specify files that can be included, that contain additional records. The records will be inserted in place of the include directives. Include directives only contain two fields: `include`, `include_if_exists` or `include_dir` directive and the file or directory to be included. The file or directory can be a relative or absolute path, and can be double-quoted. For the `include_dir` form, all files not starting with a `.` and ending with `.conf` will be included. Multiple files within an include directory are processed in file name order (according to C locale rules, i.e., numbers before letters, and uppercase letters before lowercase ones).

A record can have several 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]
hostgssenc         database  user  address     auth-method  [auth-
options]
hostnogssenc       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]
hostgssenc         database  user  IP-address  IP-mask      auth-
method  [auth-options]
hostnogssenc       database  user  IP-address  IP-mask      auth-
method  [auth-options]
include            file
include_if_exists  file
include_dir        directory
```

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 SSL or non-SSL connection attempts as well as GSSAPI encrypted or non-GSSAPI encrypted 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 by setting the ssl configuration parameter (see Section 18.9 for more information). Otherwise, the `hostssl` record is ignored except for logging a warning that it cannot match any connections.

`hostnossl`

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

`hostgssenc`

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

To make use of this option the server must be built with GSSAPI support. Otherwise, the `hostgssenc` record is ignored except for logging a warning that it cannot match any connections.

`hostnogssenc`

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

`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 physical replication connection is requested, however, it doesn't match with logical replication connections. Note that physical replication connections do not specify any particular database whereas logical replication connections do specify it. Otherwise, this is the name of a specific PostgreSQL database or a regular expression. Multiple database names and/or regular expressions can be supplied by separating them with commas.

If the database name starts with a slash (`/`), the remainder of the name is treated as a regular expression. (See Section 9.7.3.1 for details of PostgreSQL's regular expression syntax.)

A separate file containing database names and/or regular expressions 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, a regular expression (when starting with a slash (`/`), 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 and/or regular expressions can be supplied by separating them with commas.

If the user name starts with a slash (/), the remainder of the name is treated as a regular expression. (See Section 9.7.3.1 for details of PostgreSQL's regular expression syntax.)

A separate file containing user names and/or regular expressions 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.

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.

These fields do not apply to `local` records.

> **Note**
>
> 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 com-

plicates 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 do not apply to `local` 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 20.3. All the options are lower case and treated case sensitively, so even acronyms like `ldap` must be specified as lower case.

`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 20.4 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.

`scram-sha-256`

Perform SCRAM-SHA-256 authentication to verify the user's password. See Section 20.5 for details.

`md5`

Perform SCRAM-SHA-256 or MD5 authentication to verify the user's password. See Section 20.5 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 20.5 for details.

`gss`

Use GSSAPI to authenticate the user. This is only available for TCP/IP connections. See Section 20.6 for details. It can be used in conjunction with GSSAPI encryption.

`sspi`

Use SSPI to authenticate the user. This is only available on Windows. See Section 20.7 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 20.8 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 20.9 for details.

`ldap`

Authenticate using an LDAP server. See Section 20.10 for details.

`radius`

Authenticate using a RADIUS server. See Section 20.11 for details.

`cert`

Authenticate using SSL client certificates. See Section 20.12 for details.

`pam`

Authenticate using the Pluggable Authentication Modules (PAM) service provided by the operating system. See Section 20.13 for details.

`bsd`

Authenticate using the BSD Authentication service provided by the operating system. See Section 20.14 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.

In addition to the method-specific options listed below, there is a method-independent authentication option `clientcert`, which can be specified in any `hostssl` record. This option can be set to `verify-ca` or `verify-full`. Both options require the client to present a valid (trusted) SSL certificate, while `verify-full` additionally enforces that the `cn` (Common Name) in the certificate matches the username or an applicable mapping. This behavior is similar to the `cert` authentication method (see Section 20.12) but enables pairing the verification of client certificates with any authentication method that supports `hostssl` entries.

On any record using client certificate authentication (i.e. one using the `cert` authentication method or one using the `clientcert` option), you can specify which part of the client certificate credentials to match using the `clientname` option. This option can have one of two values. If you specify `clientname=CN`, which is the default, the username is matched against the certificate's `Common Name (CN)`. If instead you specify `clientname=DN` the username is matched against the entire `Distinguished Name (DN)` of the certificate. This option is probably best used in conjunction with a username map. The comparison is done with the `DN` in RFC 2253[1] format. To see the `DN` of a client certificate in this format, do

[1] https://datatracker.ietf.org/doc/html/rfc2253

```
openssl x509 -in myclient.crt -noout -subject -nameopt RFC2253 |
 sed "s/^subject=//"
```

Care needs to be taken when using this option, especially when using regular expression matching against the DN.

`include`

This line will be replaced by the contents of the given file.

`include_if_exists`

This line will be replaced by the content of the given file if the file exists. Otherwise, a message is logged to indicate that the file has been skipped.

`include_dir`

This line will be replaced by the contents of all the files found in the directory, if they don't start with a `.` and end with `.conf`, processed in file name order (according to C locale rules, i.e., numbers before letters, and uppercase letters before lowercase ones).

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.

> **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 20.1. See the next section for details on the different authentication methods.

Example 20.1. 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

# The same using a regular expression for DATABASE, that allows
 connection
# to any databases with a name beginning with "db" and finishing
 with a
# number using two to four digits (like "db1234" or "db12").
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    "/^db\d{2,4}$"  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
 scram-sha-256
```

```
# Allow any user from hosts in the example.com domain to connect to
# any database if the user's password is correctly supplied.
#
# Require SCRAM authentication for most users, but make an
 exception
# for user 'mike', who uses an older client that doesn't support
 SCRAM
# authentication.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com
 scram-sha-256

# In the absence of preceding "host" lines, these three lines will
# reject all connections from 192.168.54.1 (since that entry will
 be
# matched first), but allow GSSAPI-encrypted connections from
 anywhere else
# on the Internet.  The zero mask causes no bits of the host IP
 address to
# be considered, so it matches any host.  Unencrypted GSSAPI
 connections
# (which "fall through" to the third line since "hostgssenc" only
 matches
# encrypted GSSAPI connections) are allowed, but only from
 192.168.12.10.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             192.168.54.1/32
 reject
hostgssenc all          all             0.0.0.0/0               gss
host    all             all             192.168.12.10/32        gss

# Allow users from 192.168.x.x hosts to connect to any database, if
# they pass the ident check.  If, for example, ident says the user
 is
# "bryanh" and he requests to connect as PostgreSQL user "guest1",
 the
# connection is allowed if there is an entry in pg_ident.conf for
 map
# "omicron" that says "bryanh" is allowed to connect as "guest1".
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             192.168.0.0/16
 ident map=omicron

# If these are the only four lines for local connections, they will
# allow local users to connect only to their own databases
 (databases
# with the same name as their database user name) except for users
 whose
# name end with "helpdesk", administrators and members of role
 "support",
```

```
# who can connect to all databases.  The file $PGDATA/admins
 contains a
# list of names of administrators.  Passwords are required in all
 cases.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
local   sameuser        all                                     md5
local   all             /^.*helpdesk$                           md5
local   all             @admins                                 md5
local   all             +support                                md5

# The last two lines above can be combined into a single line:
local   all             @admins,+support                        md5

# The database column can also use lists and file names:
local   db1,db2,@demodbs  all                                   md5
```

20.2. User Name Maps

When using an external authentication system such as Ident or GSSAPI, the name of the operating system user that initiated the connection might not be the same as the database user (role) that is to be used. In this case, a user name map can be applied to map the operating system user name to a database user. To use user name mapping, specify `map=`*`map-name`* in the options field in `pg_hba.conf`. This option is supported for all authentication methods that receive external user names. Since different mappings might be needed for different connections, the name of the map to be used is specified in the *`map-name`* parameter in `pg_hba.conf` to indicate which map to use for each individual connection.

User name maps are defined in the ident map file, which by default is named `pg_ident.conf` and is stored in the cluster's data directory. (It is possible to place the map file elsewhere, however; see the ident_file configuration parameter.) The ident map file contains lines of the general forms:

```
map-name system-username database-username
include file
include_if_exists file
include_dir directory
```

Comments, whitespace and line continuations are handled in the same way as in `pg_hba.conf`. The *`map-name`* is an arbitrary name that will be used to refer to this mapping in `pg_hba.conf`. The other two fields specify an operating system user name and a matching database user name. The same *`map-name`* can be used repeatedly to specify multiple user-mappings within a single map.

As for `pg_hba.conf`, the lines in this file can be include directives, following the same rules.

The `pg_ident.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`, calling the SQL function `pg_reload_conf()`, or using `kill -HUP`) to make it re-read the file.

The system view `pg_ident_file_mappings` can be helpful for pre-testing changes to the `pg_ident.conf` file, or for diagnosing problems if loading of the file did not have the desired effects. Rows in the view with non-null `error` fields indicate problems in the corresponding lines of the file.

There is no restriction regarding how many database users a given operating system user can correspond to, nor vice versa. Thus, entries in a map should be thought of as meaning “this operating system user is allowed to connect as this database user”, rather than implying that they are equivalent. The connection will be allowed if there is any map entry that pairs the user name obtained from the external authentication system with the database user name that the user has requested to connect as. The

value `all` can be used as the *`database-username`* to specify that if the *`system-username`* matches, then this user is allowed to log in as any of the existing database users. Quoting `all` makes the keyword lose its special meaning.

If the *`database-username`* begins with a `+` character, then the operating system user can login as any user belonging to that role, similarly to how user names beginning with `+` are treated in `pg_hba.conf`. Thus, a `+` mark 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. Quoting a username starting with a `+` makes the `+` lose its special meaning.

If the *`system-username`* field starts with a slash (`/`), the remainder of the field is treated as a regular expression. (See Section 9.7.3.1 for details of PostgreSQL's regular expression syntax.) The regular expression can include a single capture, or parenthesized subexpression, which can then be referenced in the *`database-username`* field as `\1` (backslash-one). This allows the mapping of multiple user names in a single line, which is particularly useful for simple syntax substitutions. For example, these entries

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

will remove the domain part for users with system user names that end with `@mydomain.com`, and allow any user whose system name ends with `@otherdomain.com` to log in as `guest`. Quoting a *`database-username`* containing `\1` *does not* make `\1` lose its special meaning.

If the *`database-username`* field starts with a slash (`/`), the remainder of the field is treated as a regular expression (see Section 9.7.3.1 for details of PostgreSQL's regular expression syntax). It is not possible to use `\1` to use a capture from regular expression on *`system-username`* for a regular expression on *`database-username`*.

> **Tip**
>
> Keep in mind that by default, a regular expression can match just part of a string. It's usually wise to use `^` and `$`, as shown in the above example, to force the match to be to the entire system user name.

A `pg_ident.conf` file that could be used in conjunction with the `pg_hba.conf` file in Example 20.1 is shown in Example 20.2. In this example, anyone logged in to a machine on the 192.168 network that does not have the operating system user name `bryanh`, `ann`, or `robert` would not be granted access. Unix user `robert` would only be allowed access when he tries to connect as PostgreSQL user `bob`, not as `robert` or anyone else. `ann` would only be allowed to connect as `ann`. User `bryanh` would be allowed to connect as either `bryanh` or as `guest1`.

Example 20.2. An Example `pg_ident.conf` File

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# bob has user name robert on these machines
omicron         robert                  bob
# bryanh can also connect as guest1
omicron         bryanh                  guest1
```

20.3. Authentication Methods

PostgreSQL provides various methods for authenticating users:

- Trust authentication, which simply trusts that users are who they say they are.
- Password authentication, which requires that users send a password.
- GSSAPI authentication, which relies on a GSSAPI-compatible security library. Typically this is used to access an authentication server such as a Kerberos or Microsoft Active Directory server.
- SSPI authentication, which uses a Windows-specific protocol similar to GSSAPI.
- Ident authentication, which relies on an "Identification Protocol" (RFC 1413[2]) service on the client's machine. (On local Unix-socket connections, this is treated as peer authentication.)
- Peer authentication, which relies on operating system facilities to identify the process at the other end of a local connection. This is not supported for remote connections.
- LDAP authentication, which relies on an LDAP authentication server.
- RADIUS authentication, which relies on a RADIUS authentication server.
- Certificate authentication, which requires an SSL connection and authenticates users by checking the SSL certificate they send.
- PAM authentication, which relies on a PAM (Pluggable Authentication Modules) library.
- BSD authentication, which relies on the BSD Authentication framework (currently available only on OpenBSD).

Peer authentication is usually recommendable for local connections, though trust authentication might be sufficient in some circumstances. Password authentication is the easiest choice for remote connections. All the other options require some kind of external security infrastructure (usually an authentication server or a certificate authority for issuing SSL certificates), or are platform-specific.

The following sections describe each of these authentication methods in more detail.

20.4. Trust Authentication

When `trust` authentication is specified, PostgreSQL assumes that anyone who can connect to the server is authorized to access the database with whatever database user name they specify (even superuser names). Of course, restrictions made in the `database` and `user` columns still apply. This method should only be used when there is adequate operating-system-level protection on connections to the server.

`trust` authentication is appropriate and very convenient for local connections on a single-user workstation. It is usually *not* appropriate by itself on a multiuser machine. However, you might be able to use `trust` even on a multiuser machine, if you restrict access to the server's Unix-domain socket file using file-system permissions. To do this, set the `unix_socket_permissions` (and possibly `unix_socket_group`) configuration parameters as described in Section 19.3. Or you could set the `unix_socket_directories` configuration parameter to place the socket file in a suitably restricted directory.

Setting file-system permissions only helps for Unix-socket connections. Local TCP/IP connections are not restricted by file-system permissions. Therefore, if you want to use file-system permissions for local security, remove the `host ... 127.0.0.1 ...` line from `pg_hba.conf`, or change it to a non-`trust` authentication method.

`trust` authentication is only suitable for TCP/IP connections if you trust every user on every machine that is allowed to connect to the server by the `pg_hba.conf` lines that specify `trust`. It is seldom reasonable to use `trust` for any TCP/IP connections other than those from localhost (127.0.0.1).

[2] https://datatracker.ietf.org/doc/html/rfc1413

20.5. Password Authentication

There are several password-based authentication methods. These methods operate similarly but differ in how the users' passwords are stored on the server and how the password provided by a client is sent across the connection.

`scram-sha-256`

The method `scram-sha-256` performs SCRAM-SHA-256 authentication, as described in RFC 7677[3]. It is a challenge-response scheme that prevents password sniffing on untrusted connections and supports storing passwords on the server in a cryptographically hashed form that is thought to be secure.

This is the most secure of the currently provided methods, but it is not supported by older client libraries.

`md5`

The method `md5` uses a custom less secure challenge-response mechanism. It prevents password sniffing and avoids storing passwords on the server in plain text but provides no protection if an attacker manages to steal the password hash from the server. Also, the MD5 hash algorithm is nowadays no longer considered secure against determined attacks.

To ease transition from the `md5` method to the newer SCRAM method, if `md5` is specified as a method in `pg_hba.conf` but the user's password on the server is encrypted for SCRAM (see below), then SCRAM-based authentication will automatically be chosen instead.

`password`

The method `password` sends the password in clear-text and is therefore vulnerable to password "sniffing" attacks. It should always be avoided if possible. If the connection is protected by SSL encryption then `password` can be used safely, though. (Though SSL certificate authentication might be a better choice if one is depending on using SSL).

PostgreSQL database passwords are separate from operating system user passwords. The password for each database user is stored in the `pg_authid` system catalog. Passwords can be managed with the SQL commands CREATE ROLE and ALTER ROLE, e.g., **`CREATE ROLE foo WITH LOGIN PASSWORD 'secret'`**, or the psql command `\password`. If no password has been set up for a user, the stored password is null and password authentication will always fail for that user.

The availability of the different password-based authentication methods depends on how a user's password on the server is encrypted (or hashed, more accurately). This is controlled by the configuration parameter password_encryption at the time the password is set. If a password was encrypted using the `scram-sha-256` setting, then it can be used for the authentication methods `scram-sha-256` and `password` (but password transmission will be in plain text in the latter case). The authentication method specification `md5` will automatically switch to using the `scram-sha-256` method in this case, as explained above, so it will also work. If a password was encrypted using the `md5` setting, then it can be used only for the `md5` and `password` authentication method specifications (again, with the password transmitted in plain text in the latter case). (Previous PostgreSQL releases supported storing the password on the server in plain text. This is no longer possible.) To check the currently stored password hashes, see the system catalog `pg_authid`.

To upgrade an existing installation from `md5` to `scram-sha-256`, after having ensured that all client libraries in use are new enough to support SCRAM, set `password_encryption = 'scram-sha-256'` in `postgresql.conf`, make all users set new passwords, and change the authentication method specifications in `pg_hba.conf` to `scram-sha-256`.

[3] https://datatracker.ietf.org/doc/html/rfc7677

20.6. GSSAPI Authentication

GSSAPI is an industry-standard protocol for secure authentication defined in RFC 2743[4]. PostgreSQL supports GSSAPI for authentication, communications encryption, or both. GSSAPI provides automatic authentication (single sign-on) for systems that support it. The authentication itself is secure. If GSSAPI encryption or SSL encryption is used, the data sent along the database connection will be encrypted; otherwise, it will not.

GSSAPI support has to be enabled when PostgreSQL is built; see Chapter 17 for more information.

When GSSAPI uses Kerberos, it uses a standard service principal (authentication identity) name in the format *`servicename/hostname@realm`*. The principal name used by a particular installation is not encoded in the PostgreSQL server in any way; rather it is specified in the *keytab* file that the server reads to determine its identity. If multiple principals are listed in the keytab file, the server will accept any one of them. The server's realm name is the preferred realm specified in the Kerberos configuration file(s) accessible to the server.

When connecting, the client must know the principal name of the server it intends to connect to. The *`servicename`* part of the principal is ordinarily `postgres`, but another value can be selected via libpq's krbsrvname connection parameter. The *`hostname`* part is the fully qualified host name that libpq is told to connect to. The realm name is the preferred realm specified in the Kerberos configuration file(s) accessible to the client.

The client will also have a principal name for its own identity (and it must have a valid ticket for this principal). To use GSSAPI for authentication, the client principal must be associated with a PostgreSQL database user name. The `pg_ident.conf` configuration file can be used to map principals to user names; for example, `pgusername@realm` could be mapped to just `pgusername`. Alternatively, you can use the full `username@realm` principal as the role name in PostgreSQL without any mapping.

PostgreSQL also supports mapping client principals to user names by just stripping the realm from the principal. This method is supported for backwards compatibility and is strongly discouraged as it is then impossible to distinguish different users with the same user name but coming from different realms. To enable this, set `include_realm` to 0. For simple single-realm installations, doing that combined with setting the `krb_realm` parameter (which checks that the principal's realm matches exactly what is in the `krb_realm` parameter) is still secure; but this is a less capable approach compared to specifying an explicit mapping in `pg_ident.conf`.

The location of the server's keytab file is specified by the krb_server_keyfile configuration parameter. For security reasons, it is recommended to use a separate keytab just for the PostgreSQL server rather than allowing the server to read the system keytab file. Make sure that your server keytab file is readable (and preferably only readable, not writable) by the PostgreSQL server account. (See also Section 18.1.)

The keytab file is generated using the Kerberos software; see the Kerberos documentation for details. The following example shows doing this using the kadmin tool of MIT Kerberos:

```
kadmin% addprinc -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

The following authentication options are supported for the GSSAPI authentication method:

`include_realm`

If set to 0, the realm name from the authenticated user principal is stripped off before being passed through the user name mapping (Section 20.2). This is discouraged and is primarily available for

[4] https://datatracker.ietf.org/doc/html/rfc2743

backwards compatibility, as it is not secure in multi-realm environments unless `krb_realm` is also used. It is recommended to leave `include_realm` set to the default (1) and to provide an explicit mapping in `pg_ident.conf` to convert principal names to PostgreSQL user names.

`map`

Allows mapping from client principals to database user names. See Section 20.2 for details. For a GSSAPI/Kerberos principal, such as `username@EXAMPLE.COM` (or, less commonly, `username/hostbased@EXAMPLE.COM`), the user name used for mapping is `username@EXAMPLE.COM` (or `username/hostbased@EXAMPLE.COM`, respectively), unless `include_realm` has been set to 0, in which case `username` (or `username/hostbased`) is what is seen as the system user name when mapping.

`krb_realm`

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

In addition to these settings, which can be different for different `pg_hba.conf` entries, there is the server-wide krb_caseins_users configuration parameter. If that is set to true, client principals are matched to user map entries case-insensitively. `krb_realm`, if set, is also matched case-insensitively.

20.7. SSPI Authentication

SSPI is a Windows technology for secure authentication with single sign-on. PostgreSQL will use SSPI in `negotiate` mode, which will use Kerberos when possible and automatically fall back to NTLM in other cases. SSPI and GSSAPI interoperate as clients and servers, e.g., an SSPI client can authenticate to an GSSAPI server. It is recommended to use SSPI on Windows clients and servers and GSSAPI on non-Windows platforms.

When using Kerberos authentication, SSPI works the same way GSSAPI does; see Section 20.6 for details.

The following configuration options are supported for SSPI:

`include_realm`

If set to 0, the realm name from the authenticated user principal is stripped off before being passed through the user name mapping (Section 20.2). This is discouraged and is primarily available for backwards compatibility, as it is not secure in multi-realm environments unless `krb_realm` is also used. It is recommended to leave `include_realm` set to the default (1) and to provide an explicit mapping in `pg_ident.conf` to convert principal names to PostgreSQL user names.

`compat_realm`

If set to 1, the domain's SAM-compatible name (also known as the NetBIOS name) is used for the `include_realm` option. This is the default. If set to 0, the true realm name from the Kerberos user principal name is used.

Do not disable this option unless your server runs under a domain account (this includes virtual service accounts on a domain member system) and all clients authenticating through SSPI are also using domain accounts, or authentication will fail.

`upn_username`

If this option is enabled along with `compat_realm`, the user name from the Kerberos UPN is used for authentication. If it is disabled (the default), the SAM-compatible user name is used. By default, these two names are identical for new user accounts.

Note that libpq uses the SAM-compatible name if no explicit user name is specified. If you use libpq or a driver based on it, you should leave this option disabled or explicitly specify user name in the connection string.

`map`

Allows for mapping between system and database user names. See Section 20.2 for details. For an SSPI/Kerberos principal, such as `username@EXAMPLE.COM` (or, less commonly, `username/hostbased@EXAMPLE.COM`), the user name used for mapping is `username@EXAMPLE.COM` (or `username/hostbased@EXAMPLE.COM`, respectively), unless `include_realm` has been set to 0, in which case `username` (or `username/hostbased`) is what is seen as the system user name when mapping.

`krb_realm`

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

20.8. Ident Authentication

The ident authentication method works by obtaining the client's operating system user name from an ident server and using it as the allowed database user name (with an optional user name mapping). This is only supported on TCP/IP connections.

> **Note**
>
> When ident is specified for a local (non-TCP/IP) connection, peer authentication (see Section 20.9) will be used instead.

The following configuration options are supported for `ident`:

`map`

Allows for mapping between system and database user names. See Section 20.2 for details.

The "Identification Protocol" is described in RFC 1413[5]. Virtually every Unix-like operating system ships with an ident server that listens on TCP port 113 by default. The basic functionality of an ident server is to answer questions like "What user initiated the connection that goes out of your port *X* and connects to my port *Y*?". Since PostgreSQL knows both *X* and *Y* when a physical connection is established, it can interrogate the ident server on the host of the connecting client and can theoretically determine the operating system user for any given connection.

The drawback of this procedure is that it depends on the integrity of the client: if the client machine is untrusted or compromised, an attacker could run just about any program on port 113 and return any user name they choose. This authentication method is therefore only appropriate for closed networks where each client machine is under tight control and where the database and system administrators operate in close contact. In other words, you must trust the machine running the ident server. Heed the warning:

> The Identification Protocol is not intended as an authorization or access control protocol.
>
> —RFC 1413

Some ident servers have a nonstandard option that causes the returned user name to be encrypted, using a key that only the originating machine's administrator knows. This option *must not* be used

[5] https://datatracker.ietf.org/doc/html/rfc1413

when using the ident server with PostgreSQL, since PostgreSQL does not have any way to decrypt the returned string to determine the actual user name.

20.9. Peer Authentication

The peer authentication method works by obtaining the client's operating system user name from the kernel and using it as the allowed database user name (with optional user name mapping). This method is only supported on local connections.

The following configuration options are supported for `peer`:

`map`

Allows for mapping between system and database user names. See Section 20.2 for details.

Peer authentication is only available on operating systems providing the `getpeereid()` function, the `SO_PEERCRED` socket parameter, or similar mechanisms. Currently that includes Linux, most flavors of BSD including macOS, and Solaris.

20.10. LDAP Authentication

This authentication method operates similarly to `password` except that it uses LDAP as the password verification method. LDAP is used only to validate the user name/password pairs. Therefore the user must already exist in the database before LDAP can be used for authentication.

LDAP authentication can operate in two modes. In the first mode, which we will call the simple bind mode, the server will bind to the distinguished name constructed as *`prefix username suffix`*. Typically, the *`prefix`* parameter is used to specify `cn=`, or *`DOMAIN`*`\` in an Active Directory environment. *`suffix`* is used to specify the remaining part of the DN in a non-Active Directory environment.

In the second mode, which we will call the search+bind mode, the server first binds to the LDAP directory with a fixed user name and password, specified with *`ldapbinddn`* and *`ldapbindpasswd`*, and performs a search for the user trying to log in to the database. If no user and password is configured, an anonymous bind will be attempted to the directory. The search will be performed over the subtree at *`ldapbasedn`*, and will try to do an exact match of the attribute specified in *`ldapsearchattribute`*. Once the user has been found in this search, the server re-binds to the directory as this user, using the password specified by the client, to verify that the login is correct. This mode is the same as that used by LDAP authentication schemes in other software, such as Apache `mod_authnz_ldap` and `pam_ldap`. This method allows for significantly more flexibility in where the user objects are located in the directory, but will cause two additional requests to the LDAP server to be made.

The following configuration options are used in both modes:

`ldapserver`

Names or IP addresses of LDAP servers to connect to. Multiple servers may be specified, separated by spaces.

`ldapport`

Port number on LDAP server to connect to. If no port is specified, the LDAP library's default port setting will be used.

`ldapscheme`

Set to `ldaps` to use LDAPS. This is a non-standard way of using LDAP over SSL, supported by some LDAP server implementations. See also the `ldaptls` option for an alternative.

`ldaptls`

Set to 1 to make the connection between PostgreSQL and the LDAP server use TLS encryption. This uses the `StartTLS` operation per RFC 4513[6]. See also the `ldapscheme` option for an alternative.

Note that using `ldapscheme` or `ldaptls` only encrypts the traffic between the PostgreSQL server and the LDAP server. The connection between the PostgreSQL server and the PostgreSQL client will still be unencrypted unless SSL is used there as well.

The following options are used in simple bind mode only:

`ldapprefix`

String to prepend to the user name when forming the DN to bind as, when doing simple bind authentication.

`ldapsuffix`

String to append to the user name when forming the DN to bind as, when doing simple bind authentication.

The following options are used in search+bind mode only:

`ldapbasedn`

Root DN to begin the search for the user in, when doing search+bind authentication.

`ldapbinddn`

DN of user to bind to the directory with to perform the search when doing search+bind authentication.

`ldapbindpasswd`

Password for user to bind to the directory with to perform the search when doing search+bind authentication.

`ldapsearchattribute`

Attribute to match against the user name in the search when doing search+bind authentication. If no attribute is specified, the `uid` attribute will be used.

`ldapsearchfilter`

The search filter to use when doing search+bind authentication. Occurrences of `$username` will be replaced with the user name. This allows for more flexible search filters than `ldapsearchattribute`.

`ldapurl`

An RFC 4516[7] LDAP URL. This is an alternative way to write some of the other LDAP options in a more compact and standard form. The format is

`ldap[s]://`*`host`*`[:`*`port`*`]/`*`basedn`*`[?[`*`attribute`*`][?[`*`scope`*`][?[`*`filter`*`]]]]`

`scope` must be one of `base`, `one`, `sub`, typically the last. (The default is `base`, which is normally not useful in this application.) *`attribute`* can nominate a single attribute, in which case

[6] https://datatracker.ietf.org/doc/html/rfc4513
[7] https://datatracker.ietf.org/doc/html/rfc4516

it is used as a value for `ldapsearchattribute`. If *`attribute`* is empty then *`filter`* can be used as a value for `ldapsearchfilter`.

The URL scheme `ldaps` chooses the LDAPS method for making LDAP connections over SSL, equivalent to using `ldapscheme=ldaps`. To use encrypted LDAP connections using the `StartTLS` operation, use the normal URL scheme `ldap` and specify the `ldaptls` option in addition to `ldapurl`.

For non-anonymous binds, `ldapbinddn` and `ldapbindpasswd` must be specified as separate options.

LDAP URLs are currently only supported with OpenLDAP, not on Windows.

It is an error to mix configuration options for simple bind with options for search+bind.

When using search+bind mode, the search can be performed using a single attribute specified with `ldapsearchattribute`, or using a custom search filter specified with `ldapsearchfilter`. Specifying `ldapsearchattribute=foo` is equivalent to specifying `ldapsearchfilter="(foo=$username)"`. If neither option is specified the default is `ldapsearchattribute=uid`.

If PostgreSQL was compiled with OpenLDAP as the LDAP client library, the `ldapserver` setting may be omitted. In that case, a list of host names and ports is looked up via RFC 2782[8] DNS SRV records. The name `_ldap._tcp.DOMAIN` is looked up, where `DOMAIN` is extracted from `ldapbasedn`.

Here is an example for a simple-bind LDAP configuration:

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn="
 ldapsuffix=", dc=example, dc=net"
```

When a connection to the database server as database user `someuser` is requested, PostgreSQL will attempt to bind to the LDAP server using the DN `cn=someuser, dc=example, dc=net` and the password provided by the client. If that connection succeeds, the database access is granted.

Here is an example for a search+bind configuration:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example,
 dc=net" ldapsearchattribute=uid
```

When a connection to the database server as database user `someuser` is requested, PostgreSQL will attempt to bind anonymously (since `ldapbinddn` was not specified) to the LDAP server, perform a search for `(uid=someuser)` under the specified base DN. If an entry is found, it will then attempt to bind using that found information and the password supplied by the client. If that second bind succeeds, the database access is granted.

Here is the same search+bind configuration written as a URL:

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?
uid?sub"
```

Some other software that supports authentication against LDAP uses the same URL format, so it will be easier to share the configuration.

Here is an example for a search+bind configuration that uses `ldapsearchfilter` instead of `ldapsearchattribute` to allow authentication by user ID or email address:

[8] https://datatracker.ietf.org/doc/html/rfc2782

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example,
 dc=net" ldapsearchfilter="(|(uid=$username)(mail=$username))"
```

Here is an example for a search+bind configuration that uses DNS SRV discovery to find the host name(s) and port(s) for the LDAP service for the domain name `example.net`:

```
host ... ldap ldapbasedn="dc=example,dc=net"
```

> **Tip**
>
> Since LDAP often uses commas and spaces to separate the different parts of a DN, it is often necessary to use double-quoted parameter values when configuring LDAP options, as shown in the examples.

20.11. RADIUS Authentication

This authentication method operates similarly to `password` except that it uses RADIUS as the password verification method. RADIUS is used only to validate the user name/password pairs. Therefore the user must already exist in the database before RADIUS can be used for authentication.

When using RADIUS authentication, an Access Request message will be sent to the configured RADIUS server. This request will be of type `Authenticate Only`, and include parameters for `user name`, `password` (encrypted) and `NAS Identifier`. The request will be encrypted using a secret shared with the server. The RADIUS server will respond to this request with either `Access Accept` or `Access Reject`. There is no support for RADIUS accounting.

Multiple RADIUS servers can be specified, in which case they will be tried sequentially. If a negative response is received from a server, the authentication will fail. If no response is received, the next server in the list will be tried. To specify multiple servers, separate the server names with commas and surround the list with double quotes. If multiple servers are specified, the other RADIUS options can also be given as comma-separated lists, to provide individual values for each server. They can also be specified as a single value, in which case that value will apply to all servers.

The following configuration options are supported for RADIUS:

`radiusservers`

The DNS names or IP addresses of the RADIUS servers to connect to. This parameter is required.

`radiussecrets`

The shared secrets used when talking securely to the RADIUS servers. This must have exactly the same value on the PostgreSQL and RADIUS servers. It is recommended that this be a string of at least 16 characters. This parameter is required.

> **Note**
>
> The encryption vector used will only be cryptographically strong if PostgreSQL is built with support for OpenSSL. In other cases, the transmission to the RADIUS server should only be considered obfuscated, not secured, and external security measures should be applied if necessary.

`radiusports`

The port numbers to connect to on the RADIUS servers. If no port is specified, the default RADIUS port (`1812`) will be used.

`radiusidentifiers`

The strings to be used as `NAS Identifier` in the RADIUS requests. This parameter can be used, for example, to identify which database cluster the user is attempting to connect to, which can be useful for policy matching on the RADIUS server. If no identifier is specified, the default `postgresql` will be used.

If it is necessary to have a comma or whitespace in a RADIUS parameter value, that can be done by putting double quotes around the value, but it is tedious because two layers of double-quoting are now required. An example of putting whitespace into RADIUS secret strings is:

```
host ... radius radiusservers="server1,server2"
 radiussecrets="""secret one"",""secret two"""
```

20.12. Certificate Authentication

This authentication method uses SSL client certificates to perform authentication. It is therefore only available for SSL connections; see Section 18.9.2 for SSL configuration instructions. When using this authentication method, the server will require that the client provide a valid, trusted certificate. No password prompt will be sent to the client. The `cn` (Common Name) attribute of the certificate will be compared to the requested database user name, and if they match the login will be allowed. User name mapping can be used to allow `cn` to be different from the database user name.

The following configuration options are supported for SSL certificate authentication:

`map`

Allows for mapping between system and database user names. See Section 20.2 for details.

It is redundant to use the `clientcert` option with `cert` authentication because `cert` authentication is effectively `trust` authentication with `clientcert=verify-full`.

20.13. PAM Authentication

This authentication method operates similarly to `password` except that it uses PAM (Pluggable Authentication Modules) as the authentication mechanism. The default PAM service name is `postgresql`. PAM is used only to validate user name/password pairs and optionally the connected remote host name or IP address. Therefore the user must already exist in the database before PAM can be used for authentication. For more information about PAM, please read the Linux-PAM Page[9].

The following configuration options are supported for PAM:

`pamservice`

PAM service name.

`pam_use_hostname`

Determines whether the remote IP address or the host name is provided to PAM modules through the `PAM_RHOST` item. By default, the IP address is used. Set this option to 1 to use the resolved host name instead. Host name resolution can lead to login delays. (Most PAM configurations

[9] https://www.kernel.org/pub/linux/libs/pam/

don't use this information, so it is only necessary to consider this setting if a PAM configuration was specifically created to make use of it.)

> **Note**
>
> If PAM is set up to read `/etc/shadow`, authentication will fail because the PostgreSQL server is started by a non-root user. However, this is not an issue when PAM is configured to use LDAP or other authentication methods.

20.14. BSD Authentication

This authentication method operates similarly to `password` except that it uses BSD Authentication to verify the password. BSD Authentication is used only to validate user name/password pairs. Therefore the user's role must already exist in the database before BSD Authentication can be used for authentication. The BSD Authentication framework is currently only available on OpenBSD.

BSD Authentication in PostgreSQL uses the `auth-postgresql` login type and authenticates with the `postgresql` login class if that's defined in `login.conf`. By default that login class does not exist, and PostgreSQL will use the default login class.

> **Note**
>
> To use BSD Authentication, the PostgreSQL user account (that is, the operating system user running the server) must first be added to the `auth` group. The `auth` group exists by default on OpenBSD systems.

20.15. Authentication Problems

Authentication failures and related problems generally manifest themselves through error messages like the following:

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user
 "andym", database "testdb"
```

This is what you are most likely to get if you succeed in contacting the server, but it does not want to talk to you. As the message suggests, the server refused the connection request because it found no matching entry in its `pg_hba.conf` configuration file.

```
FATAL:  password authentication failed for user "andym"
```

Messages like this indicate that you contacted the server, and it is willing to talk to you, but not until you pass the authorization method specified in the `pg_hba.conf` file. Check the password you are providing, or check your Kerberos or ident software if the complaint mentions one of those authentication types.

```
FATAL:  user "andym" does not exist
```

The indicated database user name was not found.

```
FATAL:  database "testdb" does not exist
```

The database you are trying to connect to does not exist. Note that if you do not specify a database name, it defaults to the database user name.

> **Tip**
>
> The server log might contain more information about an authentication failure than is reported to the client. If you are confused about the reason for a failure, check the server log.

Chapter 21. Database Roles

PostgreSQL manages database access permissions using the concept of *roles*. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant *membership* in a role to another role, thus allowing the member role to use privileges assigned to another role.

The concept of roles subsumes the concepts of “users” and “groups”. In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both.

This chapter describes how to create and manage roles. More information about the effects of role privileges on various database objects can be found in Section 5.8.

21.1. Database Roles

Database roles are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database roles are global across a database cluster installation (and not per individual database). To create a role use the `CREATE ROLE` SQL command:

```
CREATE ROLE name;
```

name follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as `LOGIN`, to the command. More details appear below.) To remove an existing role, use the analogous `DROP ROLE` command:

```
DROP ROLE name;
```

For convenience, the programs createuser and dropuser are provided as wrappers around these SQL commands that can be called from the shell command line:

```
createuser name
dropuser name
```

To determine the set of existing roles, examine the `pg_roles` system catalog, for example:

```
SELECT rolname FROM pg_roles;
```

or to see just those capable of logging in:

```
SELECT rolname FROM pg_roles WHERE rolcanlogin;
```

The psql program's `\du` meta-command is also useful for listing the existing roles.

In order to bootstrap the database system, a freshly initialized system always contains one predefined login-capable role. This role is always a “superuser”, and it will have the same name as the operating system user that initialized the database cluster with `initdb` unless a different name is specified. This role is often named `postgres`. In order to create more roles you first have to connect as this initial role.

Every connection to the database server is made using the name of some particular role, and this role determines the initial access privileges for commands issued in that connection. The role name to use for a particular database connection is indicated by the client that is initiating the connection request in an application-specific fashion. For example, the `psql` program uses the `-U` command line option to indicate the role to connect as. Many applications assume the name of the current operating system user by default (including `createuser` and `psql`). Therefore it is often convenient to maintain a naming correspondence between roles and operating system users.

The set of database roles a given client connection can connect as is determined by the client authentication setup, as explained in Chapter 20. (Thus, a client is not limited to connect as the role matching its operating system user, just as a person's login name need not match his or her real name.) Since the role identity determines the set of privileges available to a connected client, it is important to carefully configure privileges when setting up a multiuser environment.

21.2. Role Attributes

A database role can have a number of attributes that define its privileges and interact with the client authentication system.

login privilege

Only roles that have the `LOGIN` attribute can be used as the initial role name for a database connection. A role with the `LOGIN` attribute can be considered the same as a “database user”. To create a role with login privilege, use either:

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

(`CREATE USER` is equivalent to `CREATE ROLE` except that `CREATE USER` includes `LOGIN` by default, while `CREATE ROLE` does not.)

superuser status

A database superuser bypasses all permission checks, except the right to log in. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use `CREATE ROLE name SUPERUSER`. You must do this as a role that is already a superuser.

database creation

A role must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEDB`.

role creation

A role must be explicitly given permission to create more roles (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEROLE`. A role with `CREATEROLE` privilege can alter and drop roles which have been granted to the `CREATEROLE` user with the `ADMIN` option. Such a grant occurs automatically when a `CREATEROLE` user that is not a superuser creates a new role, so that by default, a `CREATEROLE` user can alter and drop the roles which they have created. Altering a role includes most changes that can be made using `ALTER ROLE`, including, for example, changing passwords. It also includes modifications to a role that can be made using the `COMMENT` and `SECURITY LABEL` commands.

However, `CREATEROLE` does not convey the ability to create `SUPERUSER` roles, nor does it convey any power over `SUPERUSER` roles that already exist. Furthermore, `CREATEROLE` does not convey the power to create `REPLICATION` users, nor the ability to grant or revoke the `REPLICATION` privilege, nor the ability to modify the role properties of such users. However, it

does allow ALTER ROLE ... SET and ALTER ROLE ... RENAME to be used on REPLICATION roles, as well as the use of COMMENT ON ROLE, SECURITY LABEL ON ROLE, and DROP ROLE. Finally, CREATEROLE does not confer the ability to grant or revoke the BYPASSRLS privilege.

initiating replication

A role must explicitly be given permission to initiate streaming replication (except for superusers, since those bypass all permission checks). A role used for streaming replication must have LOGIN permission as well. To create such a role, use CREATE ROLE *name* REPLICATION LOGIN.

password

A password is only significant if the client authentication method requires the user to supply a password when connecting to the database. The password and md5 authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with CREATE ROLE *name* PASSWORD '*string*'.

inheritance of privileges

A role inherits the privileges of roles it is a member of, by default. However, to create a role which does not inherit privileges by default, use CREATE ROLE *name* NOINHERIT. Alternatively, inheritance can be overridden for individual grants by using WITH INHERIT TRUE or WITH INHERIT FALSE.

bypassing row-level security

A role must be explicitly given permission to bypass every row-level security (RLS) policy (except for superusers, since those bypass all permission checks). To create such a role, use CREATE ROLE *name* BYPASSRLS as a superuser.

connection limit

Connection limit can specify how many concurrent connections a role can make. -1 (the default) means no limit. Specify connection limit upon role creation with CREATE ROLE *name* CONNECTION LIMIT '*integer*'.

A role's attributes can be modified after creation with ALTER ROLE. See the reference pages for the CREATE ROLE and ALTER ROLE commands for details.

A role can also have role-specific defaults for many of the run-time configuration settings described in Chapter 19. For example, if for some reason you want to disable index scans (hint: not a good idea) anytime you connect, you can use:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

This will save the setting (but not set it immediately). In subsequent connections by this role it will appear as though SET enable_indexscan TO off had been executed just before the session started. You can still alter this setting during the session; it will only be the default. To remove a role-specific default setting, use ALTER ROLE *rolename* RESET *varname*. Note that role-specific defaults attached to roles without LOGIN privilege are fairly useless, since they will never be invoked.

When a non-superuser creates a role using the CREATEROLE privilege, the created role is automatically granted back to the creating user, just as if the bootstrap superuser had executed the command GRANT created_user TO creating_user WITH ADMIN TRUE, SET FALSE, INHERIT FALSE. Since a CREATEROLE user can only exercise special privileges with regard to an existing role if they have ADMIN OPTION on it, this grant is just sufficient to allow a CREATEROLE user to administer the roles they created. However, because it is created with INHERIT FALSE, SET FALSE, the CREATEROLE user doesn't inherit the privileges of the created role, nor can it access

the privileges of that role using SET ROLE. However, since any user who has ADMIN OPTION on a role can grant membership in that role to any other user, the CREATEROLE user can gain access to the created role by simply granting that role back to themselves with the INHERIT and/or SET options. Thus, the fact that privileges are not inherited by default nor is SET ROLE granted by default is a safeguard against accidents, not a security feature. Also note that, because this automatic grant is granted by the bootstrap superuser, it cannot be removed or changed by the CREATEROLE user; however, any superuser could revoke it, modify it, and/or issue additional such grants to other CREATEROLE users. Whichever CREATEROLE users have ADMIN OPTION on a role at any given time can administer it.

21.3. Role Membership

It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In PostgreSQL this is done by creating a role that represents the group, and then granting *membership* in the group role to individual user roles.

To set up a group role, first create the role:

```
CREATE ROLE name;
```

Typically a role being used as a group would not have the LOGIN attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the GRANT and REVOKE commands:

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

You can grant membership to other group roles, too (since there isn't really any distinction between group roles and non-group roles). The database will not let you set up circular membership loops. Also, it is not permitted to grant membership in a role to PUBLIC.

The members of a group role can use the privileges of the role in two ways. First, member roles that have been granted membership with the SET option can do SET ROLE to temporarily “become” the group role. In this state, the database session has access to the privileges of the group role rather than the original login role, and any database objects created are considered owned by the group role not the login role. Second, member roles that have been granted membership with the INHERIT option automatically have use of the privileges of those directly or indirectly a member of, though the chain stops at memberships lacking the inherit option. As an example, suppose we have done:

```
CREATE ROLE joe LOGIN;
CREATE ROLE admin;
CREATE ROLE wheel;
CREATE ROLE island;
GRANT admin TO joe WITH INHERIT TRUE;
GRANT wheel TO admin WITH INHERIT FALSE;
GRANT island TO joe WITH INHERIT TRUE, SET FALSE;
```

Immediately after connecting as role joe, a database session will have use of privileges granted directly to joe plus any privileges granted to admin and island, because joe “inherits” those privileges. However, privileges granted to wheel are not available, because even though joe is indirectly a member of wheel, the membership is via admin which was granted using WITH INHERIT FALSE. After:

```
SET ROLE admin;
```

the session would have use of only those privileges granted to admin, and not those granted to joe or island. After:

```
SET ROLE wheel;
```

the session would have use of only those privileges granted to wheel, and not those granted to either joe or admin. The original privilege state can be restored with any of:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> **Note**
>
> The SET ROLE command always allows selecting any role that the original login role is directly or indirectly a member of, provided that there is a chain of membership grants each of which has SET TRUE (which is the default). Thus, in the above example, it is not necessary to become admin before becoming wheel. On the other hand, it is not possible to become island at all; joe can only access those privileges via inheritance.

> **Note**
>
> In the SQL standard, there is a clear distinction between users and roles, and users do not automatically inherit privileges while roles do. This behavior can be obtained in PostgreSQL by giving roles being used as SQL roles the INHERIT attribute, while giving roles being used as SQL users the NOINHERIT attribute. However, PostgreSQL defaults to giving all roles the INHERIT attribute, for backward compatibility with pre-8.1 releases in which users always had use of permissions granted to groups they were members of.

The role attributes LOGIN, SUPERUSER, CREATEDB, and CREATEROLE can be thought of as special privileges, but they are never inherited as ordinary privileges on database objects are. You must actually SET ROLE to a specific role having one of these attributes in order to make use of the attribute. Continuing the above example, we might choose to grant CREATEDB and CREATEROLE to the admin role. Then a session connecting as role joe would not have these privileges immediately, only after doing SET ROLE admin.

To destroy a group role, use DROP ROLE:

```
DROP ROLE name;
```

Any memberships in the group role are automatically revoked (but the member roles are not otherwise affected).

21.4. Dropping Roles

Because roles can own database objects and can hold privileges to access other objects, dropping a role is often not just a matter of a quick DROP ROLE. Any objects owned by the role must first be dropped or reassigned to other owners; and any permissions granted to the role must be revoked.

Ownership of objects can be transferred one at a time using ALTER commands, for example:

```
ALTER TABLE bobs_table OWNER TO alice;
```

Alternatively, the `REASSIGN OWNED` command can be used to reassign ownership of all objects owned by the role-to-be-dropped to a single other role. Because `REASSIGN OWNED` cannot access objects in other databases, it is necessary to run it in each database that contains objects owned by the role. (Note that the first such `REASSIGN OWNED` will change the ownership of any shared-across-databases objects, that is databases or tablespaces, that are owned by the role-to-be-dropped.)

Once any valuable objects have been transferred to new owners, any remaining objects owned by the role-to-be-dropped can be dropped with the `DROP OWNED` command. Again, this command cannot access objects in other databases, so it is necessary to run it in each database that contains objects owned by the role. Also, `DROP OWNED` will not drop entire databases or tablespaces, so it is necessary to do that manually if the role owns any databases or tablespaces that have not been transferred to new owners.

`DROP OWNED` also takes care of removing any privileges granted to the target role for objects that do not belong to it. Because `REASSIGN OWNED` does not touch such objects, it's typically necessary to run both `REASSIGN OWNED` and `DROP OWNED` (in that order!) to fully remove the dependencies of a role to be dropped.

In short then, the most general recipe for removing a role that has been used to own objects is:

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- repeat the above commands in each database of the cluster
DROP ROLE doomed_role;
```

When not all owned objects are to be transferred to the same successor owner, it's best to handle the exceptions manually and then perform the above steps to mop up.

If `DROP ROLE` is attempted while dependent objects still remain, it will issue messages identifying which objects need to be reassigned or dropped.

21.5. Predefined Roles

PostgreSQL provides a set of predefined roles that provide access to certain, commonly needed, privileged capabilities and information. Administrators (including roles that have the `CREATEROLE` privilege) can `GRANT` these roles to users and/or other roles in their environment, providing those users with access to the specified capabilities and information.

The predefined roles are described in Table 21.1. Note that the specific permissions for each of the roles may change in the future as additional capabilities are added. Administrators should monitor the release notes for changes.

Table 21.1. Predefined Roles

Role	Allowed Access
pg_read_all_data	Read all data (tables, views, sequences), as if having `SELECT` rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute `BYPASSRLS` set. If RLS is being used, an administrator may wish to set `BYPASSRLS` on roles which this role is GRANTed to.
pg_write_all_data	Write all data (tables, views, sequences), as if having `INSERT`, `UPDATE`, and `DELETE` rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role

Role	Allowed Access
	does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.
pg_read_all_settings	Read all configuration variables, even those normally visible only to superusers.
pg_read_all_stats	Read all pg_stat_* views and use various statistics related extensions, even those normally visible only to superusers.
pg_stat_scan_tables	Execute monitoring functions that may take ACCESS SHARE locks on tables, potentially for a long time.
pg_monitor	Read/execute various monitoring views and functions. This role is a member of pg_read_all_settings, pg_read_all_stats and pg_stat_scan_tables.
pg_database_owner	None. Membership consists, implicitly, of the current database owner.
pg_signal_backend	Signal another backend to cancel a query or terminate its session.
pg_read_server_files	Allow reading files from any location the database can access on the server with COPY and other file-access functions.
pg_write_server_files	Allow writing to files in any location the database can access on the server with COPY and other file-access functions.
pg_execute_server_program	Allow executing programs on the database server as the user the database runs as with COPY and other functions which allow executing a server-side program.
pg_checkpoint	Allow executing the CHECKPOINT command.
pg_maintain	Allow executing VACUUM, ANALYZE, CLUSTER, REFRESH MATERIALIZED VIEW, REINDEX, and LOCK TABLE on all relations, as if having MAINTAIN rights on those objects, even without having it explicitly.
pg_use_reserved_connections	Allow use of connection slots reserved via reserved_connections.
pg_create_subscription	Allow users with CREATE permission on the database to issue CREATE SUBSCRIPTION.

The pg_monitor, pg_read_all_settings, pg_read_all_stats and pg_stat_scan_tables roles are intended to allow administrators to easily configure a role for the purpose of monitoring the database server. They grant a set of common privileges allowing the role to read various useful configuration settings, statistics and other system information normally restricted to superusers.

The pg_database_owner role has one implicit, situation-dependent member, namely the owner of the current database. Like any role, it can own objects or receive grants of access privileges. Consequently, once pg_database_owner has rights within a template database, each owner of a database instantiated from that template will exercise those rights. pg_database_owner cannot be a member of any role, and it cannot have non-implicit members. Initially, this role owns the public schema, so each database owner governs local use of the schema.

The pg_signal_backend role is intended to allow administrators to enable trusted, but non-superuser, roles to send signals to other backends. Currently this role enables sending of signals for canceling a query on another backend or terminating its session. A user granted this role cannot however send signals to a backend owned by a superuser. See Section 9.28.2.

The pg_read_server_files, pg_write_server_files and pg_execute_server_program roles are intended to allow administrators to have trusted, but non-superuser, roles which are able to access files and run programs on the database server as the user the database runs as. As these roles are able to access any file on the server file system, they bypass all database-level

permission checks when accessing files directly and they could be used to gain superuser-level access, therefore great care should be taken when granting these roles to users.

Care should be taken when granting these roles to ensure they are only used where needed and with the understanding that these roles grant access to privileged information.

Administrators can grant access to these roles to users using the `GRANT` command, for example:

```
GRANT pg_signal_backend TO admin_user;
```

21.6. Function Security

Functions, triggers and row-level security policies allow users to insert code into the backend server that other users might execute unintentionally. Hence, these mechanisms permit users to “Trojan horse” others with relative ease. The strongest protection is tight control over who can define objects. Where that is infeasible, write queries referring only to objects having trusted owners. Remove from `search_path` any schemas that permit untrusted users to create objects.

Functions run inside the backend server process with the operating system permissions of the database server daemon. If the programming language used for the function allows unchecked memory accesses, it is possible to change the server's internal data structures. Hence, among many other things, such functions can circumvent any system access controls. Function languages that allow such access are considered “untrusted”, and PostgreSQL allows only superusers to create functions written in those languages.

Chapter 22. Managing Databases

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQL objects ("database objects"). This chapter describes the properties of databases, and how to create, manage, and destroy them.

22.1. Overview

A small number of objects, like role, database, and tablespace names, are defined at the cluster level and stored in the `pg_global` tablespace. Inside the cluster are multiple databases, which are isolated from each other but can access cluster-level objects. Inside each database are multiple schemas, which contain objects like tables and functions. So the full hierarchy is: cluster, database, schema, table (or some other kind of object, such as a function).

When connecting to the database server, a client must specify the database name in its connection request. It is not possible to access more than one database per connection. However, clients can open multiple connections to the same database, or different databases. Database-level security has two components: access control (see Section 20.1), managed at the connection level, and authorization control (see Section 5.8), managed via the grant system. Foreign data wrappers (see postgres_fdw) allow for objects within one database to act as proxies for objects in other database or clusters. The older dblink module (see dblink) provides a similar capability. By default, all users can connect to all databases using all connection methods.

If one PostgreSQL server cluster is planned to contain unrelated projects or users that should be, for the most part, unaware of each other, it is recommended to put them into separate databases and adjust authorizations and access controls accordingly. If the projects or users are interrelated, and thus should be able to use each other's resources, they should be put in the same database but probably into separate schemas; this provides a modular structure with namespace isolation and authorization control. More information about managing schemas is in Section 5.10.

While multiple databases can be created within a single cluster, it is advised to consider carefully whether the benefits outweigh the risks and limitations. In particular, the impact that having a shared WAL (see Chapter 28) has on backup and recovery options. While individual databases in the cluster are isolated when considered from the user's perspective, they are closely bound from the database administrator's point-of-view.

Databases are created with the `CREATE DATABASE` command (see Section 22.2) and destroyed with the `DROP DATABASE` command (see Section 22.5). To determine the set of existing databases, examine the `pg_database` system catalog, for example

```
SELECT datname FROM pg_database;
```

The psql program's `\l` meta-command and `-l` command-line option are also useful for listing the existing databases.

> **Note**
>
> The SQL standard calls databases "catalogs", but there is no difference in practice.

22.2. Creating a Database

In order to create a database, the PostgreSQL server must be up and running (see Section 18.3).

Databases are created with the SQL command CREATE DATABASE:

```
CREATE DATABASE name;
```

where *name* follows the usual rules for SQL identifiers. The current role automatically becomes the owner of the new database. It is the privilege of the owner of a database to remove it later (which also removes all the objects in it, even if they have a different owner).

The creation of databases is a restricted operation. See Section 21.2 for how to grant permission.

Since you need to be connected to the database server in order to execute the `CREATE DATABASE` command, the question remains how the *first* database at any given site can be created. The first database is always created by the `initdb` command when the data storage area is initialized. (See Section 18.2.) This database is called `postgres`. So to create the first "ordinary" database you can connect to `postgres`.

Two additional databases, `template1` and `template0`, are also created during database cluster initialization. Whenever a new database is created within the cluster, `template1` is essentially cloned. This means that any changes you make in `template1` are propagated to all subsequently created databases. Because of this, avoid creating objects in `template1` unless you want them propagated to every newly created database. `template0` is meant as a pristine copy of the original contents of `template1`. It can be cloned instead of `template1` when it is important to make a database without any such site-local additions. More details appear in Section 22.3.

As a convenience, there is a program you can execute from the shell to create new databases, `createdb`.

```
createdb dbname
```

`createdb` does no magic. It connects to the `postgres` database and issues the `CREATE DATABASE` command, exactly as described above. The createdb reference page contains the invocation details. Note that `createdb` without any arguments will create a database with the current user name.

> **Note**
>
> Chapter 20 contains information about how to restrict who can connect to a given database.

Sometimes you want to create a database for someone else, and have them become the owner of the new database, so they can configure and manage it themselves. To achieve that, use one of the following commands:

```
CREATE DATABASE dbname OWNER rolename;
```

from the SQL environment, or:

```
createdb -O rolename dbname
```

from the shell. Only the superuser is allowed to create a database for someone else (that is, for a role you are not a member of).

22.3. Template Databases

`CREATE DATABASE` actually works by copying an existing database. By default, it copies the standard system database named `template1`. Thus that database is the "template" from which new data-

bases are made. If you add objects to `template1`, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language PL/Perl in `template1`, it will automatically be available in user databases without any extra action being taken when those databases are created.

However, `CREATE DATABASE` does not copy database-level `GRANT` permissions attached to the source database. The new database has default database-level permissions.

There is a second standard system database named `template0`. This database contains the same data as the initial contents of `template1`, that is, only the standard objects predefined by your version of PostgreSQL. `template0` should never be changed after the database cluster has been initialized. By instructing `CREATE DATABASE` to copy `template0` instead of `template1`, you can create a "pristine" user database (one where no user-defined objects exist and where the system objects have not been altered) that contains none of the site-local additions in `template1`. This is particularly handy when restoring a `pg_dump` dump: the dump script should be restored in a pristine database to ensure that one recreates the correct contents of the dumped database, without conflicting with objects that might have been added to `template1` later on.

Another common reason for copying `template0` instead of `template1` is that new encoding and locale settings can be specified when copying `template0`, whereas a copy of `template1` must use the same settings it does. This is because `template1` might contain encoding-specific or locale-specific data, while `template0` is known not to.

To create a database by copying `template0`, use:

```
CREATE DATABASE dbname TEMPLATE template0;
```

from the SQL environment, or:

```
createdb -T template0 dbname
```

from the shell.

It is possible to create additional template databases, and indeed one can copy any database in a cluster by specifying its name as the template for `CREATE DATABASE`. It is important to understand, however, that this is not (yet) intended as a general-purpose "`COPY DATABASE`" facility. The principal limitation is that no other sessions can be connected to the source database while it is being copied. `CREATE DATABASE` will fail if any other connection exists when it starts; during the copy operation, new connections to the source database are prevented.

Two useful flags exist in `pg_database` for each database: the columns `datistemplate` and `datallowconn`. `datistemplate` can be set to indicate that a database is intended as a template for `CREATE DATABASE`. If this flag is set, the database can be cloned by any user with `CREATEDB` privileges; if it is not set, only superusers and the owner of the database can clone it. If `datallowconn` is false, then no new connections to that database will be allowed (but existing sessions are not terminated simply by setting the flag false). The `template0` database is normally marked `datallowconn = false` to prevent its modification. Both `template0` and `template1` should always be marked with `datistemplate = true`.

> **Note**
>
> `template1` and `template0` do not have any special status beyond the fact that the name `template1` is the default source database name for `CREATE DATABASE`. For example, one could drop `template1` and recreate it from `template0` without any ill effects. This course of action might be advisable if one has carelessly added a bunch of junk in `template1`. (To delete `template1`, it must have `pg_database.datistemplate = false`.)

> The `postgres` database is also created when a database cluster is initialized. This database is meant as a default database for users and applications to connect to. It is simply a copy of `template1` and can be dropped and recreated if necessary.

22.4. Database Configuration

Recall from Chapter 19 that the PostgreSQL server provides a large number of run-time configuration variables. You can set database-specific default values for many of these settings.

For example, if for some reason you want to disable the GEQO optimizer for a given database, you'd ordinarily have to either disable it for all databases or make sure that every connecting client is careful to issue `SET geqo TO off`. To make this setting the default within a particular database, you can execute the command:

```
ALTER DATABASE mydb SET geqo TO off;
```

This will save the setting (but not set it immediately). In subsequent connections to this database it will appear as though `SET geqo TO off;` had been executed just before the session started. Note that users can still alter this setting during their sessions; it will only be the default. To undo any such setting, use `ALTER DATABASE` *`dbname`* `RESET` *`varname`*.

22.5. Destroying a Database

Databases are destroyed with the command DROP DATABASE:

```
DROP DATABASE name;
```

Only the owner of the database, or a superuser, can drop a database. Dropping a database removes all objects that were contained within the database. The destruction of a database cannot be undone.

You cannot execute the `DROP DATABASE` command while connected to the victim database. You can, however, be connected to any other database, including the `template1` database. `template1` would be the only option for dropping the last user database of a given cluster.

For convenience, there is also a shell program to drop databases, dropdb:

```
dropdb dbname
```

(Unlike `createdb`, it is not the default action to drop the database with the current user name.)

22.6. Tablespaces

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very

fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

> **Warning**
>
> Even though located outside the main PostgreSQL data directory, tablespaces are an integral part of the database cluster and *cannot* be treated as an autonomous collection of data files. They are dependent on metadata contained in the main data directory, and therefore cannot be attached to a different database cluster or backed up individually. Similarly, if you lose a tablespace (file deletion, disk failure, etc.), the database cluster might become unreadable or unable to start. Placing a tablespace on a temporary file system like a RAM disk risks the reliability of the entire cluster.

To define a tablespace, use the CREATE TABLESPACE command, for example::

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

The location must be an existing, empty directory that is owned by the PostgreSQL operating system user. All objects subsequently created within the tablespace will be stored in files underneath this directory. The location must not be on removable or transient storage, as the cluster might fail to function if the tablespace is missing or lost.

> **Note**
>
> There is usually not much point in making more than one tablespace per logical file system, since you cannot control the location of individual files within a logical file system. However, PostgreSQL does not enforce any such limitation, and indeed it is not directly aware of the file system boundaries on your system. It just stores files in the directories you tell it to use.

Creation of the tablespace itself must be done as a database superuser, but after that you can allow ordinary database users to use it. To do that, grant them the `CREATE` privilege on it.

Tables, indexes, and entire databases can be assigned to particular tablespaces. To do so, a user with the `CREATE` privilege on a given tablespace must pass the tablespace name as a parameter to the relevant command. For example, the following creates a table in the tablespace `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Alternatively, use the default_tablespace parameter:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

When `default_tablespace` is set to anything but an empty string, it supplies an implicit `TABLESPACE` clause for `CREATE TABLE` and `CREATE INDEX` commands that do not have an explicit one.

There is also a temp_tablespaces parameter, which determines the placement of temporary tables and indexes, as well as temporary files that are used for purposes such as sorting large data sets. This can be a list of tablespace names, rather than only one, so that the load associated with temporary objects can be spread over multiple tablespaces. A random member of the list is picked each time a temporary object is to be created.

The tablespace associated with a database is used to store the system catalogs of that database. Furthermore, it is the default tablespace used for tables, indexes, and temporary files created within the database, if no TABLESPACE clause is given and no other selection is specified by default_tablespace or temp_tablespaces (as appropriate). If a database is created without specifying a tablespace for it, it uses the same tablespace as the template database it is copied from.

Two tablespaces are automatically created when the database cluster is initialized. The pg_global tablespace is used only for shared system catalogs. The pg_default tablespace is the default tablespace of the template1 and template0 databases (and, therefore, will be the default tablespace for other databases as well, unless overridden by a TABLESPACE clause in CREATE DATABASE).

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

To remove an empty tablespace, use the DROP TABLESPACE command.

To determine the set of existing tablespaces, examine the pg_tablespace system catalog, for example

```
SELECT spcname FROM pg_tablespace;
```

The psql program's \db meta-command is also useful for listing the existing tablespaces.

The directory $PGDATA/pg_tblspc contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster. Although not recommended, it is possible to adjust the tablespace layout by hand by redefining these links. Under no circumstances perform this operation while the server is running. Note that in PostgreSQL 9.1 and earlier you will also need to update the pg_tablespace catalog with the new locations. (If you do not, pg_dump will continue to output the old tablespace locations.)

Chapter 23. Localization

This chapter describes the available localization features from the point of view of the administrator. PostgreSQL supports two localization facilities:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, translated messages, and other aspects. This is covered in Section 23.1 and Section 23.2.
- Providing a number of different character sets to support storing text in all kinds of languages, and providing character set translation between client and server. This is covered in Section 23.3.

23.1. Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. PostgreSQL uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your system.

23.1.1. Overview

Locale support is automatically initialized when a database cluster is created using `initdb`. `initdb` will initialize the database cluster with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your database cluster then there is nothing else you need to do. If you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `initdb` exactly which locale to use by specifying the `--locale` option. For example:

```
initdb --locale=sv_SE
```

This example for Unix systems sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might include `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be used for a locale then the specifications can take the form *`language_territory.codeset`*. For example, `fr_BE.UTF-8` represents the French language (fr) as spoken in Belgium (BE), with a UTF-8 character set encoding.

What locales are available on your system under what names depends on what was provided by the operating system vendor and what was installed. On most Unix systems, the command `locale -a` will provide a list of available locales. Windows uses more verbose locale names, such as `German_Germany` or `Swedish_Sweden.1252`, but the principles are the same.

Occasionally it is useful to mix rules from several locales, e.g., use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only certain aspects of the localization rules:

`LC_COLLATE`	String sort order
`LC_CTYPE`	Character classification (What is a letter? Its upper-case equivalent?)
`LC_MESSAGES`	Language of messages
`LC_MONETARY`	Formatting of currency amounts
`LC_NUMERIC`	Formatting of numbers
`LC_TIME`	Formatting of dates and times

The category names translate into names of `initdb` options to override the locale choice for a specific category. For instance, to set the locale to French Canadian, but use U.S. rules for formatting currency, use `initdb --locale=fr_CA --lc-monetary=en_US`.

If you want the system to behave as if it had no locale support, use the special locale name `C`, or equivalently `POSIX`.

Some locale categories must have their values fixed when the database is created. You can use different settings for different databases, but once a database is created, you cannot change them for that database anymore. `LC_COLLATE` and `LC_CTYPE` are these categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns would become corrupt. (But you can alleviate this restriction using collations, as discussed in Section 23.2.) The default values for these categories are determined when `initdb` is run, and those values are used when new databases are created, unless specified otherwise in the `CREATE DATABASE` command.

The other locale categories can be changed whenever desired by setting the server configuration parameters that have the same name as the locale categories (see Section 19.11.2 for details). The values that are chosen by `initdb` are actually only written into the configuration file `postgresql.conf` to serve as defaults when the server is started. If you remove these assignments from `postgresql.conf` then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings before starting the server. A consequence of this is that if client and server are set up in different locales, messages might appear in different languages depending on where they originated.

> **Note**
>
> When we speak of inheriting the locale from the execution environment, this means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: `LC_ALL`, `LC_COLLATE` (or the variable corresponding to the respective category), `LANG`. If none of these environment variables are set then the locale defaults to `C`.
>
> Some message localization libraries also look at the environment variable `LANGUAGE` which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the documentation about gettext.

To enable messages to be translated to the user's preferred language, NLS must have been selected at build time (`configure --enable-nls`). All other locale support is built in automatically.

23.1.2. Behavior

The locale settings influence the following SQL features:

- Sort order in queries using `ORDER BY` or the standard comparison operators on textual data
- The `upper`, `lower`, and `initcap` functions
- Pattern matching operators (`LIKE`, `SIMILAR TO`, and POSIX-style regular expressions); locales affect both case insensitive matching and the classification of characters by character-class regular expressions
- The `to_char` family of functions
- The ability to use indexes with `LIKE` clauses

The drawback of using locales other than `C` or `POSIX` in PostgreSQL is its performance impact. It slows character handling and prevents ordinary indexes from being used by `LIKE`. For this reason use locales only if you actually need them.

As a workaround to allow PostgreSQL to use indexes with `LIKE` clauses under a non-C locale, several custom operator classes exist. These allow the creation of an index that performs a strict character-by-

character comparison, ignoring locale comparison rules. Refer to Section 11.10 for more information. Another approach is to create indexes using the C collation, as discussed in Section 23.2.

23.1.3. Selecting Locales

Locales can be selected in different scopes depending on requirements. The above overview showed how locales are specified using `initdb` to set the defaults for the entire cluster. The following list shows where locales can be selected. Each item provides the defaults for the subsequent items, and each lower item allows overriding the defaults on a finer granularity.

1. As explained above, the environment of the operating system provides the defaults for the locales of a newly initialized database cluster. In many cases, this is enough: if the operating system is configured for the desired language/territory, by default PostgreSQL will also behave according to that locale.

2. As shown above, command-line options for `initdb` specify the locale settings for a newly initialized database cluster. Use this if the operating system does not have the locale configuration you want for your database system.

3. A locale can be selected separately for each database. The SQL command `CREATE DATABASE` and its command-line equivalent `createdb` have options for that. Use this for example if a database cluster houses databases for multiple tenants with different requirements.

4. Locale settings can be made for individual table columns. This uses an SQL object called *collation* and is explained in Section 23.2. Use this for example to sort data in different languages or customize the sort order of a particular table.

5. Finally, locales can be selected for an individual query. Again, this uses SQL collation objects. This could be used to change the sort order based on run-time choices or for ad-hoc experimentation.

23.1.4. Locale Providers

A locale provider specifies which library defines the locale behavior for collations and character classifications.

The commands and tools that select the locale settings, as described above, each have an option to select the locale provider. Here is an example to initialize a database cluster using the ICU provider:

```
initdb --locale-provider=icu --icu-locale=en
```

See the description of the respective commands and programs for details. Note that you can mix locale providers at different granularities, for example use `libc` by default for the cluster but have one database that uses the `icu` provider, and then have collation objects using either provider within those databases.

Regardless of the locale provider, the operating system is still used to provide some locale-aware behavior, such as messages (see lc_messages).

The available locale providers are listed below:

`builtin`

The `builtin` provider uses built-in operations. Only the `C` and `C.UTF-8` locales are supported for this provider.

The `C` locale behavior is identical to the `C` locale in the libc provider. When using this locale, the behavior may depend on the database encoding.

The `C.UTF-8` locale is available only for when the database encoding is `UTF-8`, and the behavior is based on Unicode. The collation uses the code point values only. The regular expression

character classes are based on the "POSIX Compatible" semantics, and the case mapping is the "simple" variant.

`icu`

The `icu` provider uses the external ICU library. PostgreSQL must have been configured with support.

ICU provides collation and character classification behavior that is independent of the operating system and database encoding, which is preferable if you expect to transition to other platforms without any change in results. `LC_COLLATE` and `LC_CTYPE` can be set independently of the ICU locale.

> **Note**
>
> For the ICU provider, results may depend on the version of the ICU library used, as it is updated to reflect changes in natural language over time.

`libc`

The `libc` provider uses the operating system's C library. The collation and character classification behavior is controlled by the settings `LC_COLLATE` and `LC_CTYPE`, so they cannot be set independently.

> **Note**
>
> The same locale name may have different behavior on different platforms when using the libc provider.

23.1.5. ICU Locales

23.1.5.1. ICU Locale Names

The ICU format for the locale name is a Language Tag.

```
CREATE COLLATION mycollation1 (provider = icu, locale = 'ja-JP');
CREATE COLLATION mycollation2 (provider = icu, locale = 'fr');
```

23.1.5.2. Locale Canonicalization and Validation

When defining a new ICU collation object or database with ICU as the provider, the given locale name is transformed ("canonicalized") into a language tag if not already in that form. For instance,

```
CREATE COLLATION mycollation3 (provider = icu, locale = 'en-US-u-
kn-true');
NOTICE:  using standard form "en-US-u-kn" for locale "en-US-u-kn-
true"
CREATE COLLATION mycollation4 (provider = icu, locale =
 'de_DE.utf8');
NOTICE:  using standard form "de-DE" for locale "de_DE.utf8"
```

If you see this notice, ensure that the `provider` and `locale` are the expected result. For consistent results when using the ICU provider, specify the canonical language tag instead of relying on the transformation.

A locale with no language name, or the special language name `root`, is transformed to have the language `und` ("undefined").

ICU can transform most libc locale names, as well as some other formats, into language tags for easier transition to ICU. If a libc locale name is used in ICU, it may not have precisely the same behavior as in libc.

If there is a problem interpreting the locale name, or if the locale name represents a language or region that ICU does not recognize, you will see the following warning:

```
CREATE COLLATION nonsense (provider = icu, locale = 'nonsense');
WARNING:  ICU locale "nonsense" has unknown language "nonsense"
HINT:  To disable ICU locale validation, set parameter
 icu_validation_level to DISABLED.
CREATE COLLATION
```

icu_validation_level controls how the message is reported. Unless set to `ERROR`, the collation will still be created, but the behavior may not be what the user intended.

23.1.5.3. Language Tag

A language tag, defined in BCP 47, is a standardized identifier used to identify languages, regions, and other information about a locale.

Basic language tags are simply *`language-region`*; or even just *`language`*. The *`language`* is a language code (e.g. `fr` for French), and *`region`* is a region code (e.g. `CA` for Canada). Examples: `ja-JP`, `de`, or `fr-CA`.

Collation settings may be included in the language tag to customize collation behavior. ICU allows extensive customization, such as sensitivity (or insensitivity) to accents, case, and punctuation; treatment of digits within text; and many other options to satisfy a variety of uses.

To include this additional collation information in a language tag, append `-u`, which indicates there are additional collation settings, followed by one or more *`-key-value`* pairs. The *`key`* is the key for a collation setting and *`value`* is a valid value for that setting. For boolean settings, the *`-key`* may be specified without a corresponding *`-value`*, which implies a value of `true`.

For example, the language tag `en-US-u-kn-ks-level2` means the locale with the English language in the US region, with collation settings `kn` set to `true` and `ks` set to `level2`. Those settings mean the collation will be case-insensitive and treat a sequence of digits as a single number:

```
CREATE COLLATION mycollation5 (provider = icu, deterministic =
 false, locale = 'en-US-u-kn-ks-level2');
SELECT 'aB' = 'Ab' COLLATE mycollation5 as result;
 result
--------
 t
(1 row)

SELECT 'N-45' < 'N-123' COLLATE mycollation5 as result;
 result
--------
 t
(1 row)
```

See Section 23.2.3 for details and additional examples of using language tags with custom collation information for the locale.

23.1.6. Problems

If locale support doesn't work according to the explanation above, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you can use the command `locale -a` if your operating system provides it.

Check that PostgreSQL is actually using the locale that you think it is. The `LC_COLLATE` and `LC_CTYPE` settings are determined when a database is created, and cannot be changed except by creating a new database. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are initially determined by the environment the server is started in, but can be changed on-the-fly. You can check the active locale settings using the `SHOW` command.

The directory `src/test/locale` in the source distribution contains a test suite for PostgreSQL's locale support.

Client applications that handle server-side errors by parsing the text of the error message will obviously have problems when the server's messages are in a different language. Authors of such applications are advised to make use of the error code scheme instead.

Maintaining catalogs of message translations requires the on-going efforts of many volunteers that want to see PostgreSQL speak their preferred language well. If messages in your language are currently not available or not fully translated, your assistance would be appreciated. If you want to help, refer to Chapter 55 or write to the developers' mailing list.

23.2. Collation Support

The collation feature allows specifying the sort order and character classification behavior of data per-column, or even per-operation. This alleviates the restriction that the `LC_COLLATE` and `LC_CTYPE` settings of a database cannot be changed after its creation.

23.2.1. Concepts

Conceptually, every expression of a collatable data type has a collation. (The built-in collatable data types are `text`, `varchar`, and `char`. User-defined base types can also be marked collatable, and of course a *domain* over a collatable data type is collatable.) If the expression is a column reference, the collation of the expression is the defined collation of the column. If the expression is a constant, the collation is the default collation of the data type of the constant. The collation of a more complex expression is derived from the collations of its inputs, as described below.

The collation of an expression can be the “default” collation, which means the locale settings defined for the database. It is also possible for an expression's collation to be indeterminate. In such cases, ordering operations and other operations that need to know the collation will fail.

When the database system has to perform an ordering or a character classification, it uses the collation of the input expression. This happens, for example, with `ORDER BY` clauses and function or operator calls such as `<`. The collation to apply for an `ORDER BY` clause is simply the collation of the sort key. The collation to apply for a function or operator call is derived from the arguments, as described below. In addition to comparison operators, collations are taken into account by functions that convert between lower and upper case letters, such as `lower`, `upper`, and `initcap`; by pattern matching operators; and by `to_char` and related functions.

For a function or operator call, the collation that is derived by examining the argument collations is used at run time for performing the specified operation. If the result of the function or operator call is of a collatable data type, the collation is also used at parse time as the defined collation of the function or operator expression, in case there is a surrounding expression that requires knowledge of its collation.

The *collation derivation* of an expression can be implicit or explicit. This distinction affects how collations are combined when multiple different collations appear in an expression. An explicit collation

derivation occurs when a `COLLATE` clause is used; all other collation derivations are implicit. When multiple collations need to be combined, for example in a function call, the following rules are used:

1. If any input expression has an explicit collation derivation, then all explicitly derived collations among the input expressions must be the same, otherwise an error is raised. If any explicitly derived collation is present, that is the result of the collation combination.

2. Otherwise, all input expressions must have the same implicit collation derivation or the default collation. If any non-default collation is present, that is the result of the collation combination. Otherwise, the result is the default collation.

3. If there are conflicting non-default implicit collations among the input expressions, then the combination is deemed to have indeterminate collation. This is not an error condition unless the particular function being invoked requires knowledge of the collation it should apply. If it does, an error will be raised at run-time.

For example, consider this table definition:

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

Then in

```
SELECT a < 'foo' FROM test1;
```

the `<` comparison is performed according to `de_DE` rules, because the expression combines an implicitly derived collation with the default collation. But in

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

the comparison is performed using `fr_FR` rules, because the explicit collation derivation overrides the implicit one. Furthermore, given

```
SELECT a < b FROM test1;
```

the parser cannot determine which collation to apply, since the `a` and `b` columns have conflicting implicit collations. Since the `<` operator does need to know which collation to use, this will result in an error. The error can be resolved by attaching an explicit collation specifier to either input expression, thus:

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

or equivalently

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

On the other hand, the structurally similar case

```
SELECT a || b FROM test1;
```

does not result in an error, because the `||` operator does not care about collations: its result is the same regardless of the collation.

The collation assigned to a function or operator's combined input expressions is also considered to apply to the function or operator's result, if the function or operator delivers a result of a collatable data type. So, in

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

the ordering will be done according to `de_DE` rules. But this query:

```
SELECT * FROM test1 ORDER BY a || b;
```

results in an error, because even though the `||` operator doesn't need to know a collation, the `ORDER BY` clause does. As before, the conflict can be resolved with an explicit collation specifier:

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

23.2.2. Managing Collations

A collation is an SQL schema object that maps an SQL name to locales provided by libraries installed in the operating system. A collation definition has a *provider* that specifies which library supplies the locale data. One standard provider name is `libc`, which uses the locales provided by the operating system C library. These are the locales used by most tools provided by the operating system. Another provider is `icu`, which uses the external ICU library. ICU locales can only be used if support for ICU was configured when PostgreSQL was built.

A collation object provided by `libc` maps to a combination of `LC_COLLATE` and `LC_CTYPE` settings, as accepted by the `setlocale()` system library call. (As the name would suggest, the main purpose of a collation is to set `LC_COLLATE`, which controls the sort order. But it is rarely necessary in practice to have an `LC_CTYPE` setting that is different from `LC_COLLATE`, so it is more convenient to collect these under one concept than to create another infrastructure for setting `LC_CTYPE` per expression.) Also, a `libc` collation is tied to a character set encoding (see Section 23.3). The same collation name may exist for different encodings.

A collation object provided by `icu` maps to a named collator provided by the ICU library. ICU does not support separate “collate” and “ctype” settings, so they are always the same. Also, ICU collations are independent of the encoding, so there is always only one ICU collation of a given name in a database.

23.2.2.1. Standard Collations

On all platforms, the following collations are supported:

`unicode`

This SQL standard collation sorts using the Unicode Collation Algorithm with the Default Unicode Collation Element Table. It is available in all encodings. ICU support is required to use this collation, and behavior may change if Postgres is built with a different version of ICU. (This collation has the same behavior as the ICU root locale; see `und-x-icu` (for “undefined”).)

`ucs_basic`

This SQL standard collation sorts using the Unicode code point values rather than natural language order, and only the ASCII letters “`A`” through “`Z`” are treated as letters. The behavior is efficient and stable across all versions. Only available for encoding `UTF8`. (This collation has the same behavior as the libc locale specification `C` in `UTF8` encoding.)

`pg_c_utf8`

This collation sorts by Unicode code point values rather than natural language order. For the functions `lower`, `initcap`, and `upper`, it uses Unicode simple case mapping. For pattern matching

(including regular expressions), it uses the POSIX Compatible variant of Unicode Compatibility Properties[1]. Behavior is efficient and stable within a Postgres major version. This collation is only available for encoding `UTF8`.

`C` (equivalent to `POSIX`)

The `C` and `POSIX` collations are based on "traditional C" behavior. They sort by byte values rather than natural language order, and only the ASCII letters "`A`" through "`Z`" are treated as letters. The behavior is efficient and stable across all versions for a given database encoding, but behavior may vary between different database encodings.

`default`

The `default` collation selects the locale specified at database creation time.

Additional collations may be available depending on operating system support. The efficiency and stability of these additional collations depend on the collation provider, the provider version, and the locale.

23.2.2.2. Predefined Collations

If the operating system provides support for using multiple locales within a single program (`newlocale` and related functions), or if support for ICU is configured, then when a database cluster is initialized, `initdb` populates the system catalog `pg_collation` with collations based on all the locales it finds in the operating system at the time.

To inspect the currently available locales, use the query `SELECT * FROM pg_collation`, or the command `\dOS+` in psql.

23.2.2.2.1. libc Collations

For example, the operating system might provide a locale named `de_DE.utf8`. `initdb` would then create a collation named `de_DE.utf8` for encoding `UTF8` that has both `LC_COLLATE` and `LC_CTYPE` set to `de_DE.utf8`. It will also create a collation with the `.utf8` tag stripped off the name. So you could also use the collation under the name `de_DE`, which is less cumbersome to write and makes the name less encoding-dependent. Note that, nevertheless, the initial set of collation names is platform-dependent.

The default set of collations provided by `libc` map directly to the locales installed in the operating system, which can be listed using the command `locale -a`. In case a `libc` collation is needed that has different values for `LC_COLLATE` and `LC_CTYPE`, or if new locales are installed in the operating system after the database system was initialized, then a new collation may be created using the CREATE COLLATION command. New operating system locales can also be imported en masse using the `pg_import_system_collations()` function.

Within any particular database, only collations that use that database's encoding are of interest. Other entries in `pg_collation` are ignored. Thus, a stripped collation name such as `de_DE` can be considered unique within a given database even though it would not be unique globally. Use of the stripped collation names is recommended, since it will make one fewer thing you need to change if you decide to change to another database encoding. Note however that the `default`, `C`, and `POSIX` collations can be used regardless of the database encoding.

PostgreSQL considers distinct collation objects to be incompatible even when they have identical properties. Thus for example,

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

will draw an error even though the `C` and `POSIX` collations have identical behaviors. Mixing stripped and non-stripped collation names is therefore not recommended.

[1] https://www.unicode.org/reports/tr18/#Compatibility_Properties

23.2.2.2.2. ICU Collations

With ICU, it is not sensible to enumerate all possible locale names. ICU uses a particular naming system for locales, but there are many more ways to name a locale than there are actually distinct locales. `initdb` uses the ICU APIs to extract a set of distinct locales to populate the initial set of collations. Collations provided by ICU are created in the SQL environment with names in BCP 47 language tag format, with a "private use" extension `-x-icu` appended, to distinguish them from libc locales.

Here are some example collations that might be created:

`de-x-icu`

German collation, default variant

`de-AT-x-icu`

German collation for Austria, default variant

(There are also, say, `de-DE-x-icu` or `de-CH-x-icu`, but as of this writing, they are equivalent to `de-x-icu`.)

`und-x-icu` (for "undefined")

ICU "root" collation. Use this to get a reasonable language-agnostic sort order.

Some (less frequently used) encodings are not supported by ICU. When the database encoding is one of these, ICU collation entries in `pg_collation` are ignored. Attempting to use one will draw an error along the lines of "collation "de-x-icu" for encoding "WIN874" does not exist".

23.2.2.3. Creating New Collation Objects

If the standard and predefined collations are not sufficient, users can create their own collation objects using the SQL command CREATE COLLATION.

The standard and predefined collations are in the schema `pg_catalog`, like all predefined objects. User-defined collations should be created in user schemas. This also ensures that they are saved by `pg_dump`.

23.2.2.3.1. libc Collations

New libc collations can be created like this:

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

The exact values that are acceptable for the `locale` clause in this command depend on the operating system. On Unix-like systems, the command `locale -a` will show a list.

Since the predefined libc collations already include all collations defined in the operating system when the database instance is initialized, it is not often necessary to manually create new ones. Reasons might be if a different naming system is desired (in which case see also Section 23.2.2.3.3) or if the operating system has been upgraded to provide new locale definitions (in which case see also `pg_import_system_collations()`).

23.2.2.3.2. ICU Collations

ICU collations can be created like:

```
CREATE COLLATION german (provider = icu, locale = 'de-DE');
```

ICU locales are specified as a BCP 47 Language Tag, but can also accept most libc-style locale names. If possible, libc-style locale names are transformed into language tags.

New ICU collations can customize collation behavior extensively by including collation attributes in the language tag. See Section 23.2.3 for details and examples.

23.2.2.3.3. Copying Collations

The command CREATE COLLATION can also be used to create a new collation from an existing collation, which can be useful to be able to use operating-system-independent collation names in applications, create compatibility names, or use an ICU-provided collation under a more readable name. For example:

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

23.2.2.4. Nondeterministic Collations

A collation is either *deterministic* or *nondeterministic*. A deterministic collation uses deterministic comparisons, which means that it considers strings to be equal only if they consist of the same byte sequence. Nondeterministic comparison may determine strings to be equal even if they consist of different bytes. Typical situations include case-insensitive comparison, accent-insensitive comparison, as well as comparison of strings in different Unicode normal forms. It is up to the collation provider to actually implement such insensitive comparisons; the deterministic flag only determines whether ties are to be broken using bytewise comparison. See also Unicode Technical Standard 10[2] for more information on the terminology.

To create a nondeterministic collation, specify the property `deterministic = false` to `CREATE COLLATION`, for example:

```
CREATE COLLATION ndcoll (provider = icu, locale = 'und',
 deterministic = false);
```

This example would use the standard Unicode collation in a nondeterministic way. In particular, this would allow strings in different normal forms to be compared correctly. More interesting examples make use of the ICU customization facilities explained above. For example:

```
CREATE COLLATION case_insensitive (provider = icu, locale = 'und-u-
ks-level2', deterministic = false);
CREATE COLLATION ignore_accents (provider = icu, locale = 'und-u-
ks-level1-kc-true', deterministic = false);
```

All standard and predefined collations are deterministic, all user-defined collations are deterministic by default. While nondeterministic collations give a more “correct” behavior, especially when considering the full power of Unicode and its many special cases, they also have some drawbacks. Foremost, their use leads to a performance penalty. Note, in particular, that B-tree cannot use deduplication with indexes that use a nondeterministic collation. Also, certain operations are not possible with nondeterministic collations, such as pattern matching operations. Therefore, they should be used only in cases where they are specifically wanted.

> **Tip**
>
> To deal with text in different Unicode normalization forms, it is also an option to use the functions/expressions `normalize` and `is normalized` to preprocess or check the strings, instead of using nondeterministic collations. There are different trade-offs for each approach.

[2] https://www.unicode.org/reports/tr10

23.2.3. ICU Custom Collations

ICU allows extensive control over collation behavior by defining new collations with collation settings as a part of the language tag. These settings can modify the collation order to suit a variety of needs. For instance:

```
-- ignore differences in accents and case
CREATE COLLATION ignore_accent_case (provider = icu, deterministic
 = false, locale = 'und-u-ks-level1');
SELECT 'Å' = 'A' COLLATE ignore_accent_case; -- true
SELECT 'z' = 'Z' COLLATE ignore_accent_case; -- true

-- upper case letters sort before lower case.
CREATE COLLATION upper_first (provider = icu, locale = 'und-u-kf-
upper');
SELECT 'B' < 'b' COLLATE upper_first; -- true

-- treat digits numerically and ignore punctuation
CREATE COLLATION num_ignore_punct (provider = icu, deterministic =
 false, locale = 'und-u-ka-shifted-kn');
SELECT 'id-45' < 'id-123' COLLATE num_ignore_punct; -- true
SELECT 'w;x*y-z' = 'wxyz' COLLATE num_ignore_punct; -- true
```

Many of the available options are described in Section 23.2.3.2, or see Section 23.2.3.5 for more details.

23.2.3.1. ICU Comparison Levels

Comparison of two strings (collation) in ICU is determined by a multi-level process, where textual features are grouped into "levels". Treatment of each level is controlled by the collation settings. Higher levels correspond to finer textual features.

Table 23.1 shows which textual feature differences are considered significant when determining equality at the given level. The Unicode character `U+2063` is an invisible separator, and as seen in the table, is ignored for at all levels of comparison less than `identic`.

Table 23.1. ICU Collation Levels

Level	Description	`'f' = 'f'`	`'ab' = U&'a\2063b'`	`'x-y' = 'x_y'`	`'g' = 'G'`	`'n' = 'ñ'`	`'y' = 'z'`
level1	Base Character	`true`	`true`	`true`	`true`	`true`	`false`
level2	Accents	`true`	`true`	`true`	`true`	`false`	`false`
level3	Case/Variants	`true`	`true`	`true`	`false`	`false`	`false`
level4	Punctuation[a]	`true`	`true`	`false`	`false`	`false`	`false`
identic	All	`true`	`false`	`false`	`false`	`false`	`false`

[a] only with `ka-shifted`; see Table 23.2

At every level, even with full normalization off, basic normalization is performed. For example, `'á'` may be composed of the code points `U&'\0061\0301'` or the single code point `U&'\00E1'`, and those sequences will be considered equal even at the `identic` level. To treat any difference in code point representation as distinct, use a collation created with `deterministic` set to `true`.

23.2.3.1.1. Collation Level Examples

```
CREATE COLLATION level3 (provider = icu, deterministic = false,
 locale = 'und-u-ka-shifted-ks-level3');
CREATE COLLATION level4 (provider = icu, deterministic = false,
 locale = 'und-u-ka-shifted-ks-level4');
CREATE COLLATION identic (provider = icu, deterministic = false,
 locale = 'und-u-ka-shifted-ks-identic');

-- invisible separator ignored at all levels except identic
SELECT 'ab' = U&'a\2063b' COLLATE level4; -- true
SELECT 'ab' = U&'a\2063b' COLLATE identic; -- false

-- punctuation ignored at level3 but not at level 4
SELECT 'x-y' = 'x_y' COLLATE level3; -- true
SELECT 'x-y' = 'x_y' COLLATE level4; -- false
```

23.2.3.2. Collation Settings for an ICU Locale

Table 23.2 shows the available collation settings, which can be used as part of a language tag to customize a collation.

Table 23.2. ICU Collation Settings

Key	Values	Default	Description
`co`	`emoji`, `phonebk`, `standard`, `...`	`standard`	Collation type. See Section 23.2.3.5 for additional options and details.
`ka`	`noignore`, `shifted`	`noignore`	If set to `shifted`, causes some characters (e.g. punctuation or space) to be ignored in comparison. Key `ks` must be set to `level3` or lower to take effect. Set key `kv` to control which character classes are ignored.
`kb`	`true`, `false`	`false`	Backwards comparison for the level 2 differences. For example, locale `und-u-kb` sorts `'àe'` before `'aé'`.
`kc`	`true`, `false`	`false`	Separates case into a "level 2.5" that falls between accents and other level 3 features. If set to `true` and `ks` is set to `level1`, will ignore accents but take case into account.
`kf`	`upper`, `lower`, `false`	`false`	If set to `upper`, upper case sorts before lower case. If set to `lower`, lower case sorts before upper case. If set to `false`, the sort depends on the rules of the locale.
`kn`	`true`, `false`	`false`	If set to `true`, numbers within a string are treated as a single numeric value rather than a sequence of digits. For example, `'id-45'` sorts before `'id-123'`.
`kk`	`true`, `false`	`false`	Enable full normalization; may affect performance. Basic normalization is performed even when set to `false`. Locales for languages that require full normalization typically enable it by default. Full normalization is important in some cases, such as when multiple accents are applied to a single character. For example, the code point

Key	Values	Default	Description
			sequences `U&'\0065\0323\0302'` and `U&'\0065\0302\0323'` represent an `e` with circumflex and dot-below accents applied in different orders. With full normalization on, these code point sequences are treated as equal; otherwise they are unequal.
`kr`	`space`, `punct`, `symbol`, `currency`, `digit`, *`script-id`*		Set to one or more of the valid values, or any BCP 47 *`script-id`*, e.g. `latn` ("Latin") or `grek` ("Greek"). Multiple values are separated by "`-`". Redefines the ordering of classes of characters; those characters belonging to a class earlier in the list sort before characters belonging to a class later in the list. For instance, the value `digit-currency-space` (as part of a language tag like `und-u-kr-digit-currency-space`) sorts punctuation before digits and spaces.
`ks`	`level1`, `level2`, `level3`, `level4`, `identic`	`level3`	Sensitivity (or "strength") when determining equality, with `level1` the least sensitive to differences and `identic` the most sensitive to differences. See Table 23.1 for details.
`kv`	`space`, `punct`, `symbol`, `currency`	`punct`	Classes of characters ignored during comparison at level 3. Setting to a later value includes earlier values; e.g. `symbol` also includes `punct` and `space` in the characters to be ignored. Key `ka` must be set to `shifted` and key `ks` must be set to `level3` or lower to take effect.

Defaults may depend on locale. The above table is not meant to be complete. See Section 23.2.3.5 for additional options and details.

> **Note**
>
> For many collation settings, you must create the collation with `deterministic` set to `false` for the setting to have the desired effect (see Section 23.2.2.4). Additionally, some settings only take effect when the key `ka` is set to `shifted` (see Table 23.2).

23.2.3.3. Collation Settings Examples

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
'de-u-co-phonebk');
```

German collation with phone book collation type

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale =
'und-u-co-emoji');
```

Root collation with Emoji collation type, per Unicode Technical Standard #51

```
CREATE COLLATION latinlast (provider = icu, locale = 'en-u-kr-grek-
latn');
```

Sort Greek letters before Latin ones. (The default is Latin before Greek.)

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-up-
per');
```

Sort upper-case letters before lower-case letters. (The default is lower-case letters first.)

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-kr-
grek-latn');
```

Combines both of the above options.

23.2.3.4. ICU Tailoring Rules

If the options provided by the collation settings shown above are not sufficient, the order of collation elements can be changed with tailoring rules, whose syntax is detailed at https://unicode-org.github.io/icu/userguide/collation/customization/.

This small example creates a collation based on the root locale with a tailoring rule:

```
CREATE COLLATION custom (provider = icu, locale = 'und', rules =
 '&V << w <<< W');
```

With this rule, the letter “W” is sorted after “V”, but is treated as a secondary difference similar to an accent. Rules like this are contained in the locale definitions of some languages. (Of course, if a locale definition already contains the desired rules, then they don't need to be specified again explicitly.)

Here is a more complex example. The following statement sets up a collation named `ebcdic` with rules to sort US-ASCII characters in the order of the EBCDIC encoding.

```
CREATE COLLATION ebcdic (provider = icu, locale = 'und',
rules = $$
& ' ' < '.' < '<' < '(' < '+' < \|
< '&' < '!' < '$' < '*' < ')' < ';'
< '-' < '/' < ',' < '%' < '_' < '>' < '?'
< '`' < ':' < '#' < '@' < \' < '=' < '"'
<*a-r < '~' <*s-z < '^' < '[' < ']'
< '{' <*A-I < '}' <*J-R < '\' <*S-Z <*0-9
$$);

SELECT c
FROM (VALUES ('a'), ('b'), ('A'), ('B'), ('1'), ('2'), ('!'),
 ('^')) AS x(c)
ORDER BY c COLLATE ebcdic;
 c
---
 !
 a
 b
 ^
 A
 B
 1
 2
```

23.2.3.5. External References for ICU

This section (Section 23.2.3) is only a brief overview of ICU behavior and language tags. Refer to the following documents for technical details, additional options, and new behavior:

- Unicode Technical Standard #35[3]
- BCP 47[4]
- CLDR repository[5]
- https://unicode-org.github.io/icu/userguide/locale/
- https://unicode-org.github.io/icu/userguide/collation/

23.3. Character Set Support

The character set support in PostgreSQL allows you to store text in a variety of character sets (also called encodings), including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your PostgreSQL database cluster using `initdb`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

An important restriction, however, is that each database's character set must be compatible with the database's `LC_CTYPE` (character classification) and `LC_COLLATE` (string sort order) locale settings. For `C` or `POSIX` locale, any character set is allowed, but for other libc-provided locales there is only one character set that will work correctly. (On Windows, however, UTF-8 encoding can be used with any locale.) If you have ICU support configured, ICU-provided locales can be used with most but not all server-side encodings.

23.3.1. Supported Character Sets

Table 23.3 shows the character sets available for use in PostgreSQL.

Table 23.3. PostgreSQL Character Sets

Name	Description	Language	Server?	ICU?	Bytes/ Char	Aliases
`BIG5`	Big Five	Traditional Chinese	No	No	1–2	`WIN950`, `Windows950`
`EUC_CN`	Extended UNIX Code-CN	Simplified Chinese	Yes	Yes	1–3	
`EUC_JP`	Extended UNIX Code-JP	Japanese	Yes	Yes	1–3	
`EUC_JIS_2004`	Extended UNIX Code-JP, JIS X 0213	Japanese	Yes	No	1–3	
`EUC_KR`	Extended UNIX Code-KR	Korean	Yes	Yes	1–3	
`EUC_TW`	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	Yes	1–4	

[3] https://www.unicode.org/reports/tr35/tr35-collation.html
[4] https://www.rfc-editor.org/info/bcp47
[5] https://github.com/unicode-org/cldr/blob/master/common/bcp47/collation.xml

Name	Description	Language	Server?	ICU?	Bytes/ Char	Aliases
GB18030	National Standard	Chinese	No	No	1–4	
GBK	Extended National Standard	Simplified Chinese	No	No	1–2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	Yes	1	
JOHAB	JOHAB	Korean (Hangul)	No	No	1–3	
KOI8R	KOI8-R	Cyrillic (Russian)	Yes	Yes	1	KOI8
KOI8U	KOI8-U	Cyrillic (Ukrainian)	Yes	Yes	1	
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	No	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	No	1–4	
SJIS	Shift JIS	Japanese	No	No	1–2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japanese	No	No	1–2	

Name	Description	Language	Server?	ICU?	Bytes/ Char	Aliases
SQL_ASCII	unspecified (see text)	*any*	Yes	No	1	
UHC	Unified Hangul Code	Korean	No	No	1–2	WIN949, Windows949
UTF8	Unicode, 8-bit	*all*	Yes	Yes	1–4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	No	1	
WIN1250	Windows CP1250	Central European	Yes	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	Yes	1	ABC, TCVN, TCVN5712, VSCII

Not all client APIs support all the listed character sets. For example, the PostgreSQL JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.

The SQL_ASCII setting behaves considerably differently from the other settings. When the server character set is SQL_ASCII, the server interprets byte values 0–127 according to the ASCII standard, while byte values 128–255 are taken as uninterpreted characters. No encoding conversion will be done when the setting is SQL_ASCII. Thus, this setting is not so much a declaration that a specific encoding is in use, as a declaration of ignorance about the encoding. In most cases, if you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting because PostgreSQL will be unable to help you by converting or validating non-ASCII characters.

23.3.2. Setting the Character Set

initdb defines the default character set (encoding) for a PostgreSQL cluster. For example,

```
initdb -E EUC_JP
```

sets the default character set to EUC_JP (Extended Unix Code for Japanese). You can use --encoding instead of -E if you prefer longer option strings. If no -E or --encoding option is given, initdb attempts to determine the appropriate encoding to use based on the specified or default locale.

You can specify a non-default encoding at database creation time, provided that the encoding is compatible with the selected locale:

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-
ctype=ko_KR.euckr korean
```

This will create a database named `korean` that uses the character set `EUC_KR`, and locale `ko_KR`. Another way to accomplish this is to use this SQL command:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR'
 LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr'
 TEMPLATE=template0;
```

Notice that the above commands specify copying the `template0` database. When copying any other database, the encoding and locale settings cannot be changed from those of the source database, because that might result in corrupt data. For more information see Section 22.3.

The encoding for a database is stored in the system catalog `pg_database`. You can see it by using the `psql -l` option or the `\l` command.

```
$ psql -l
                                        List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |
      Access Privileges
-----------+----------+-----------+-------------+-------------
+-----------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 {=c/hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 {=c/hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

Important

On most modern operating systems, PostgreSQL can determine which character set is implied by the `LC_CTYPE` setting, and it will enforce that only the matching database encoding is used. On older systems it is your responsibility to ensure that you use the encoding expected by the locale you have selected. A mistake in this area is likely to lead to strange behavior of locale-dependent operations such as sorting.

PostgreSQL will allow superusers to create databases with `SQL_ASCII` encoding even when `LC_CTYPE` is not `C` or `POSIX`. As noted above, `SQL_ASCII` does not enforce that the data stored in the database has any particular encoding, and so this choice poses risks of locale-dependent misbehavior. Using this combination of settings is deprecated and may someday be forbidden altogether.

23.3.3. Automatic Character Set Conversion Between Server and Client

PostgreSQL supports automatic character set conversion between server and client for many combinations of character sets (Section 23.3.4 shows which ones).

To enable automatic character set conversion, you have to tell PostgreSQL the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the `\encoding` command in psql. `\encoding` allows you to change client encoding on the fly. For example, to change the encoding to `SJIS`, type:

  ```
  \encoding SJIS
  ```

- libpq (Section 32.11) has functions to control the client encoding.

- Using `SET client_encoding TO`. Setting the client encoding can be done with this SQL command:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  Also you can use the standard SQL syntax `SET NAMES` for this purpose:

  ```
  SET NAMES 'value';
  ```

  To query the current client encoding:

  ```
  SHOW client_encoding;
  ```

  To return to the default encoding:

  ```
  RESET client_encoding;
  ```

- Using `PGCLIENTENCODING`. If the environment variable `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

- Using the configuration variable client_encoding. If the `client_encoding` variable is set, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose `EUC_JP` for the server and `LATIN1` for the client, and some Japanese characters are returned that do not have a representation in `LATIN1` — an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. (However, if the server's character set is not `SQL_ASCII`, the server will still check that incoming data is valid for that encoding; so the net effect is as though the client character set were the same as the server's.) Just as for the server, use of `SQL_ASCII` is unwise unless you are working with all-ASCII data.

23.3.4. Available Character Set Conversions

PostgreSQL allows conversion between any two character sets for which a conversion function is listed in the `pg_conversion` system catalog. PostgreSQL comes with some predefined conversions, as summarized in Table 23.4 and shown in more detail in Table 23.5. You can create a new conversion using the SQL command CREATE CONVERSION. (To be used for automatic client/server conversions, a conversion must be marked as "default" for its character set pair.)

Table 23.4. Built-in Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	*not supported as a server encoding*
EUC_CN	*EUC_CN*, MULE_INTERNAL, UTF8
EUC_JP	*EUC_JP*, MULE_INTERNAL, SJIS, UTF8
EUC_JIS_2004	*EUC_JIS_2004*, SHIFT_JIS_2004, UTF8
EUC_KR	*EUC_KR*, MULE_INTERNAL, UTF8
EUC_TW	*EUC_TW*, BIG5, MULE_INTERNAL, UTF8
GB18030	*not supported as a server encoding*
GBK	*not supported as a server encoding*
ISO_8859_5	*ISO_8859_5*, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	*ISO_8859_6*, UTF8
ISO_8859_7	*ISO_8859_7*, UTF8
ISO_8859_8	*ISO_8859_8*, UTF8
JOHAB	*not supported as a server encoding*
KOI8R	*KOI8R*, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	*KOI8U*, UTF8
LATIN1	*LATIN1*, MULE_INTERNAL, UTF8
LATIN2	*LATIN2*, MULE_INTERNAL, UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL, UTF8
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*not supported as a server encoding*
SHIFT_JIS_2004	*not supported as a server encoding*
SQL_ASCII	*any (no conversion will be performed)*
UHC	*not supported as a server encoding*
UTF8	*all supported encodings*
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251

Server Character Set	Available Client Character Sets
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

Table 23.5. All Built-in Character Set Conversions

Conversion Name [a]	Source Encoding	Destination Encoding
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL

Conversion Name [a]	Source Encoding	Destination Encoding
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR

Conversion Name [a]	Source Encoding	Destination Encoding
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866

Conversion Name [a]	Source Encoding	Destination Encoding
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] The conversion names follow a standard naming scheme: The official name of the source encoding with all non-alphanumeric characters replaced by underscores, followed by _to_, followed by the similarly processed destination encoding name. Therefore, these names sometimes deviate from the customary encoding names shown in Table 23.3.

23.3.5. Further Reading

These are good sources to start learning about various kinds of encoding systems.

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

Contains detailed explanations of EUC_JP, EUC_CN, EUC_KR, EUC_TW.

https://www.unicode.org/

The web site of the Unicode Consortium.

RFC 3629[6]

UTF-8 (8-bit UCS/Unicode Transformation Format) is defined here.

[6] https://datatracker.ietf.org/doc/html/rfc3629

Chapter 24. Routine Database Maintenance Tasks

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are *required*, but they are repetitive in nature and can easily be automated using standard tools such as cron scripts or Windows' Task Scheduler. It is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is the creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.). The backup and recovery mechanisms available in PostgreSQL are discussed at length in Chapter 25.

The other main category of maintenance task is periodic “vacuuming” of the database. This activity is discussed in Section 24.1. Closely related to this is updating the statistics that will be used by the query planner, as discussed in Section 24.1.3.

Another task that might need periodic attention is log file management. This is discussed in Section 24.3.

check_postgres[1] is available for monitoring database health and reporting unusual conditions. check_postgres integrates with Nagios and MRTG, but can be run standalone too.

PostgreSQL is low-maintenance compared to some other database management systems. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

24.1. Routine Vacuuming

PostgreSQL databases require periodic maintenance known as *vacuuming*. For many installations, it is sufficient to let vacuuming be performed by the *autovacuum daemon*, which is described in Section 24.1.6. You might need to adjust the autovacuuming parameters described there to obtain best results for your situation. Some database administrators will want to supplement or replace the daemon's activities with manually-managed `VACUUM` commands, which typically are executed according to a schedule by cron or Task Scheduler scripts. To set up manually-managed vacuuming properly, it is essential to understand the issues discussed in the next few subsections. Administrators who rely on autovacuuming may still wish to skim this material to help them understand and adjust autovacuuming.

24.1.1. Vacuuming Basics

PostgreSQL's `VACUUM` command has to process each table on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.

2. To update data statistics used by the PostgreSQL query planner.

3. To update the visibility map, which speeds up index-only scans.

4. To protect against loss of very old data due to *transaction ID wraparound* or *multixact ID wraparound*.

Each of these reasons dictates performing `VACUUM` operations of varying frequency and scope, as explained in the following subsections.

[1] https://bucardo.org/check_postgres/

There are two variants of VACUUM: standard VACUUM and VACUUM FULL. VACUUM FULL can reclaim more disk space but runs much more slowly. Also, the standard form of VACUUM can run in parallel with production database operations. (Commands such as SELECT, INSERT, UPDATE, and DELETE will continue to function normally, though you will not be able to modify the definition of a table with commands such as ALTER TABLE while it is being vacuumed.) VACUUM FULL requires an ACCESS EXCLUSIVE lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard VACUUM and avoid VACUUM FULL.

VACUUM creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming — see Section 19.4.4.

24.1.2. Recovering Disk Space

In PostgreSQL, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control (MVCC, see Chapter 13): the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running VACUUM.

The standard form of VACUUM removes dead row versions in tables and indexes and marks the space available for future reuse. However, it will not return the space to the operating system, except in the special case where one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. In contrast, VACUUM FULL actively compacts tables by writing a complete new version of the table file with no dead space. This minimizes the size of the table, but can take a long time. It also requires extra disk space for the new copy of the table, until the operation completes.

The usual goal of routine vacuuming is to do standard VACUUMs often enough to avoid needing VACUUM FULL. The autovacuum daemon attempts to work this way, and in fact will never issue VACUUM FULL. In this approach, the idea is not to keep tables at their minimum size, but to maintain steady-state usage of disk space: each table occupies space equivalent to its minimum size plus however much space gets used up between vacuum runs. Although VACUUM FULL can be used to shrink a table back to its minimum size and return the disk space to the operating system, there is not much point in this if the table will just grow again in the future. Thus, moderately-frequent standard VACUUM runs are a better approach than infrequent VACUUM FULL runs for maintaining heavily-updated tables.

Some administrators prefer to schedule vacuuming themselves, for example doing all the work at night when load is low. The difficulty with doing vacuuming according to a fixed schedule is that if a table has an unexpected spike in update activity, it may get bloated to the point that VACUUM FULL is really necessary to reclaim space. Using the autovacuum daemon alleviates this problem, since the daemon schedules vacuuming dynamically in response to update activity. It is unwise to disable the daemon completely unless you have an extremely predictable workload. One possible compromise is to set the daemon's parameters so that it will only react to unusually heavy update activity, thus keeping things from getting out of hand, while scheduled VACUUMs are expected to do the bulk of the work when the load is typical.

For those not using autovacuum, a typical approach is to schedule a database-wide VACUUM once a day during a low-usage period, supplemented by more frequent vacuuming of heavily-updated tables as necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don't forget to VACUUM each one; the program vacuumdb might be helpful.

> **Tip**
>
> Plain VACUUM may not be satisfactory when a table contains large numbers of dead row versions as a result of massive update or delete activity. If you have such a table and you need to

reclaim the excess disk space it occupies, you will need to use VACUUM FULL, or alternatively CLUSTER or one of the table-rewriting variants of ALTER TABLE. These commands rewrite an entire new copy of the table and build new indexes for it. All these options require an ACCESS EXCLUSIVE lock. Note that they also temporarily use extra disk space approximately equal to the size of the table, since the old copies of the table and indexes can't be released until the new ones are complete.

Tip

If you have a table whose entire contents are deleted on a periodic basis, consider doing it with TRUNCATE rather than using DELETE followed by VACUUM. TRUNCATE removes the entire content of the table immediately, without requiring a subsequent VACUUM or VACUUM FULL to reclaim the now-unused disk space. The disadvantage is that strict MVCC semantics are violated.

24.1.3. Updating Planner Statistics

The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the ANALYZE command, which can be invoked by itself or as an optional step in VACUUM. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue ANALYZE commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually-scheduled ANALYZE operations, particularly if it is known that update activity on a table will not affect the statistics of “interesting” columns. The daemon schedules ANALYZE strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.

Tuples changed in partitions and inheritance children do not trigger analyze on the parent table. If the parent table is empty or rarely changed, it may never be processed by autovacuum, and the statistics for the inheritance tree as a whole won't be collected. It is necessary to run ANALYZE on the parent table manually in order to keep the statistics up to date.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a timestamp column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database, because it is a fast operation. ANALYZE uses a statistically random sampling of the rows of a table rather than reading every single row.

Tip

Although per-column tweaking of ANALYZE frequency might not be very productive, you might find it worthwhile to do per-column adjustment of the level of detail of the statistics collected by ANALYZE. Columns that are heavily used in WHERE clauses and have highly

irregular data distributions might require a finer-grain data histogram than other columns. See `ALTER TABLE SET STATISTICS`, or change the database-wide default using the default_statistics_target configuration parameter.

Also, by default there is limited information available about the selectivity of functions. However, if you create a statistics object or an expression index that uses a function call, useful statistics will be gathered about the function, which can greatly improve query plans that use the expression index.

Tip

The autovacuum daemon does not issue `ANALYZE` commands for foreign tables, since it has no means of determining how often that might be useful. If your queries require statistics on foreign tables for proper planning, it's a good idea to run manually-managed `ANALYZE` commands on those tables on a suitable schedule.

Tip

The autovacuum daemon does not issue `ANALYZE` commands for partitioned tables. Inheritance parents will only be analyzed if the parent itself is changed - changes to child tables do not trigger autoanalyze on the parent table. If your queries require statistics on parent tables for proper planning, it is necessary to periodically run a manual `ANALYZE` on those tables to keep the statistics up to date.

24.1.4. Updating the Visibility Map

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up.

Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table. Since PostgreSQL indexes don't contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An *index-only scan*, on the other hand, checks the visibility map first. If it's known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

24.1.5. Preventing Transaction ID Wraparound Failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is “in the future” and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer *transaction ID wraparound*: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that `VACUUM` will mark rows as *frozen*, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions.

Normal XIDs are compared using modulo-2^{32} arithmetic. This means that for every normal XID, there are two billion XIDs that are “older” and two billion that are “newer”; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be “in the past” for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent this, PostgreSQL reserves a special XID, `FrozenTransactionId`, which does not follow the normal XID comparison rules and is always considered older than every normal XID. Frozen row versions are treated as if the inserting XID were `FrozenTransactionId`, so that they will appear to be “in the past” to all normal transactions regardless of wraparound issues, and so such row versions will be valid until deleted, no matter how long that is.

> **Note**
>
> In PostgreSQL versions before 9.4, freezing was implemented by actually replacing a row's insertion XID with `FrozenTransactionId`, which was visible in the row's `xmin` system column. Newer versions just set a flag bit, preserving the row's original `xmin` for possible forensic use. However, rows with `xmin` equal to `FrozenTransactionId` (2) may still be found in databases pg_upgrade'd from pre-9.4 versions.
>
> Also, system catalogs may contain rows with `xmin` equal to `BootstrapTransactionId` (1), indicating that they were inserted during the first phase of initdb. Like `FrozenTransactionId`, this special XID is treated as older than every normal XID.

vacuum_freeze_min_age controls how old an XID value has to be before rows bearing that XID will be frozen. Increasing this setting may avoid unnecessary work if the rows that would otherwise be frozen will soon be modified again, but decreasing this setting increases the number of transactions that can elapse before the table must be vacuumed again.

`VACUUM` uses the visibility map to determine which pages of a table must be scanned. Normally, it will skip pages that don't have any dead row versions even if those pages might still have row versions with old XID values. Therefore, normal `VACUUM`s won't always freeze every old row version in the table. When that happens, `VACUUM` will eventually need to perform an *aggressive vacuum*, which will freeze all eligible unfrozen XID and MXID values, including those from all-visible but not all-frozen pages. In practice most tables require periodic aggressive vacuuming. vacuum_freeze_table_age controls when `VACUUM` does that: all-visible but not all-frozen pages are scanned if the number of transactions that have passed since the last such scan is greater than `vacuum_freeze_table_age` minus `vacuum_freeze_min_age`. Setting `vacuum_freeze_table_age` to 0 forces `VACUUM` to always use its aggressive strategy.

The maximum time that a table can go unvacuumed is two billion transactions minus the `vacuum_freeze_min_age` value at the time of the last aggressive vacuum. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, autovacuum is invoked on any table that might contain unfrozen rows with XIDs older than the age specified by the configuration parameter autovacuum_freeze_max_age. (This will happen even if autovacuum is disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every `autovacuum_freeze_max_age` minus `vacuum_freeze_min_age` transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need to vacuum for space reclamation, so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing `autovacuum_freeze_max_age` or decreasing `vacuum_freeze_min_age`.

The effective maximum for `vacuum_freeze_table_age` is 0.95 * `autovacuum_freeze_max_age`; a setting higher than that will be capped to the maximum. A value higher than `autovacuum_freeze_max_age` wouldn't make sense because an anti-wraparound autovac-

uum would be triggered at that point anyway, and the 0.95 multiplier leaves some breathing room to run a manual VACUUM before that happens. As a rule of thumb, vacuum_freeze_table_age should be set to a value somewhat below autovacuum_freeze_max_age, leaving enough gap so that a regularly scheduled VACUUM or an autovacuum triggered by normal delete and update activity is run in that window. Setting it too close could lead to anti-wraparound autovacuums, even though the table was recently vacuumed to reclaim space, whereas lower values lead to more frequent aggressive vacuuming.

The sole disadvantage of increasing autovacuum_freeze_max_age (and vacuum_freeze_table_age along with it) is that the pg_xact and pg_commit_ts subdirectories of the database cluster will take more space, because it must store the commit status and (if track_commit_timestamp is enabled) timestamp of all transactions back to the autovacuum_freeze_max_age horizon. The commit status uses two bits per transaction, so if autovacuum_freeze_max_age is set to its maximum allowed value of two billion, pg_xact can be expected to grow to about half a gigabyte and pg_commit_ts to about 20GB. If this is trivial compared to your total database size, setting autovacuum_freeze_max_age to its maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for pg_xact and pg_commit_ts storage. (The default, 200 million transactions, translates to about 50MB of pg_xact storage and about 2GB of pg_commit_ts storage.)

One disadvantage of decreasing vacuum_freeze_min_age is that it might cause VACUUM to do useless work: freezing a row version is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more.

To track the age of the oldest unfrozen XIDs in a database, VACUUM stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table's pg_class row contains the oldest remaining unfrozen XID at the end of the most recent VACUUM that successfully advanced relfrozenxid (typically the most recent aggressive VACUUM). Similarly, the datfrozenxid column of a database's pg_database row is a lower bound on the unfrozen XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this information is to execute queries such as:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction's XID.

> **Tip**
>
> When the VACUUM command's VERBOSE parameter is specified, VACUUM prints various statistics about the table. This includes information about how relfrozenxid and relminmxid advanced, and the number of newly frozen pages. The same details appear in the server log when autovacuum logging (controlled by log_autovacuum_min_duration) reports on a VACUUM operation executed by autovacuum.

VACUUM normally only scans pages that have been modified since the last vacuum, but relfrozenxid can only be advanced when every page of the table that might contain unfrozen XIDs is scanned. This happens when relfrozenxid is more than vacuum_freeze_table_age transactions old, when VACUUM's FREEZE option is used, or when all pages that are not already all-frozen

happen to require vacuuming to remove dead row versions. When VACUUM scans every page in the table that is not already all-frozen, it should set age(relfrozenxid) to a value just a little more than the vacuum_freeze_min_age setting that was used (more by the number of transactions started since the VACUUM started). VACUUM will set relfrozenxid to the oldest XID that remains in the table, so it's possible that the final value will be much more recent than strictly required. If no relfrozenxid-advancing VACUUM is issued on the table until autovacuum_freeze_max_age is reached, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database's oldest XIDs reach forty million transactions from the wraparound point:

```
WARNING:  database "mydb" must be vacuumed within 39985967
 transactions
HINT:  To avoid XID assignment failures, execute a database-wide
 VACUUM in that database.
```

(A manual VACUUM should fix the problem, as suggested by the hint; but note that the VACUUM should be performed by a superuser, else it will fail to process system catalogs, which prevent it from being able to advance the database's datfrozenxid.) If these warnings are ignored, the system will refuse to assign new XIDs once there are fewer than three million transactions left until wraparound:

```
ERROR:  database is not accepting commands that assign new XIDs to
 avoid wraparound data loss in database "mydb"
HINT:  Execute a database-wide VACUUM in that database.
```

In this condition any transactions already in progress can continue, but only read-only transactions can be started. Operations that modify database records or truncate relations will fail. The VACUUM command can still be run normally. Note that, contrary to what was sometimes recommended in earlier releases, it is not necessary or desirable to stop the postmaster or enter single user-mode in order to restore normal operation. Instead, follow these steps:

1. Resolve old prepared transactions. You can find these by checking pg_prepared_xacts for rows where age(transactionid) is large. Such transactions should be committed or rolled back.

2. End long-running open transactions. You can find these by checking pg_stat_activity for rows where age(backend_xid) or age(backend_xmin) is large. Such transactions should be committed or rolled back, or the session can be terminated using pg_terminate_backend.

3. Drop any old replication slots. Use pg_stat_replication to find slots where age(xmin) or age(catalog_xmin) is large. In many cases, such slots were created for replication to servers that no longer exist, or that have been down for a long time. If you drop a slot for a server that still exists and might still try to connect to that slot, that replica may need to be rebuilt.

4. Execute VACUUM in the target database. A database-wide VACUUM is simplest; to reduce the time required, it as also possible to issue manual VACUUM commands on the tables where relminxid is oldest. Do not use VACUUM FULL in this scenario, because it requires an XID and will therefore fail, except in super-user mode, where it will instead consume an XID and thus increase the risk of transaction ID wraparound. Do not use VACUUM FREEZE either, because it will do more than the minimum amount of work required to restore normal operation.

5. Once normal operation is restored, ensure that autovacuum is properly configured in the target database in order to avoid future problems.

> **Note**
>
> In earlier versions, it was sometimes necessary to stop the postmaster and VACUUM the database in a single-user mode. In typical scenarios, this is no longer necessary, and should be

avoided whenever possible, since it involves taking the system down. It is also riskier, since it disables transaction ID wraparound safeguards that are designed to prevent data loss. The only reason to use single-user mode in this scenario is if you wish to TRUNCATE or DROP unneeded tables to avoid needing to VACUUM them. The three-million-transaction safety margin exists to let the administrator do this. See the postgres reference page for details about using single-user mode.

24.1.5.1. Multixacts and Wraparound

Multixact IDs are used to support row locking by multiple transactions. Since there is only limited space in a tuple header to store lock information, that information is encoded as a "multiple transaction ID", or multixact ID for short, whenever there is more than one transaction concurrently locking a row. Information about which transaction IDs are included in any particular multixact ID is stored separately in the `pg_multixact` subdirectory, and only the multixact ID appears in the `xmax` field in the tuple header. Like transaction IDs, multixact IDs are implemented as a 32-bit counter and corresponding storage, all of which requires careful aging management, storage cleanup, and wraparound handling. There is a separate storage area which holds the list of members in each multixact, which also uses a 32-bit counter and which must also be managed.

Whenever VACUUM scans any part of a table, it will replace any multixact ID it encounters which is older than vacuum_multixact_freeze_min_age by a different value, which can be the zero value, a single transaction ID, or a newer multixact ID. For each table, `pg_class.relminmxid` stores the oldest possible multixact ID still appearing in any tuple of that table. If this value is older than vacuum_multixact_freeze_table_age, an aggressive vacuum is forced. As discussed in the previous section, an aggressive vacuum means that only those pages which are known to be all-frozen will be skipped. `mxid_age()` can be used on `pg_class.relminmxid` to find its age.

Aggressive VACUUMs, regardless of what causes them, are *guaranteed* to be able to advance the table's `relminmxid`. Eventually, as all tables in all databases are scanned and their oldest multixact values are advanced, on-disk storage for older multixacts can be removed.

As a safety device, an aggressive vacuum scan will occur for any table whose multixact-age is greater than autovacuum_multixact_freeze_max_age. Also, if the storage occupied by multixacts members exceeds 2GB, aggressive vacuum scans will occur more often for all tables, starting with those that have the oldest multixact-age. Both of these kinds of aggressive scans will occur even if autovacuum is nominally disabled.

Similar to the XID case, if autovacuum fails to clear old MXIDs from a table, the system will begin to emit warning messages when the database's oldest MXIDs reach forty million transactions from the wraparound point. And, just as in the XID case, if these warnings are ignored, the system will refuse to generate new MXIDs once there are fewer than three million left until wraparound.

Normal operation when MXIDs are exhausted can be restored in much the same way as when XIDs are exhausted. Follow the same steps in the previous section, but with the following differences:

1. Running transactions and prepared transactions can be ignored if there is no chance that they might appear in a multixact.

2. MXID information is not directly visible in system views such as `pg_stat_activity`; however, looking for old XIDs is still a good way of determining which transactions are causing MXID wraparound problems.

3. XID exhaustion will block all write transactions, but MXID exhaustion will only block a subset of write transactions, specifically those that involve row locks that require an MXID.

24.1.6. The Autovacuum Daemon

PostgreSQL has an optional but highly recommended feature called *autovacuum*, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless track_counts is set to `true`. In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The “autovacuum daemon” actually consists of multiple processes. There is a persistent daemon process, called the *autovacuum launcher*, which is in charge of starting *autovacuum worker* processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every autovacuum_naptime seconds. (Therefore, if the installation has *N* databases, a new worker will be launched every `autovacuum_naptime`/*N* seconds.) A maximum of autovacuum_max_workers worker processes are allowed to run at the same time. If there are more than `autovacuum_max_workers` databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute VACUUM and/or ANALYZE as needed. log_autovacuum_min_duration can be set to monitor autovacuum workers' activity.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker becomes available. There is no limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers. Note that the number of running workers does not count towards max_connections or superuser_reserved_connections limits.

Tables whose `relfrozenxid` value is more than autovacuum_freeze_max_age transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last VACUUM exceeds the “vacuum threshold”, the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = vacuum base threshold + vacuum scale factor *
 number of tuples
```

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is `pg_class.reltuples`.

The table is also vacuumed if the number of tuples inserted since the last vacuum has exceeded the defined insert threshold, which is defined as:

```
vacuum insert threshold = vacuum base insert threshold + vacuum
 insert scale factor * number of tuples
```

where the vacuum insert base threshold is autovacuum_vacuum_insert_threshold, and vacuum insert scale factor is autovacuum_vacuum_insert_scale_factor. Such vacuums may allow portions of the table to be marked as *all visible* and also allow tuples to be frozen, which can reduce the work required in subsequent vacuums. For tables which receive `INSERT` operations but no or almost no `UPDATE`/`DELETE` operations, it may be beneficial to lower the table's autovacuum_freeze_min_age as this may allow tuples to be frozen by earlier vacuums. The number of obsolete tuples and the number of inserted tuples are obtained from the cumulative statistics system; it is an eventually-consistent count updated by each `UPDATE`, `DELETE` and `INSERT` operation. If the `relfrozenxid` value of the table is more than `vacuum_freeze_table_age` transactions old, an aggressive vacuum is performed to freeze old tuples and advance `relfrozenxid`; otherwise, only pages that have been modified since the last vacuum are scanned.

For analyze, a similar condition is used: the threshold, defined as:

```
analyze threshold = analyze base threshold + analyze scale factor *
 number of tuples
```

is compared to the total number of tuples inserted, updated, or deleted since the last ANALYZE.

Partitioned tables do not directly store tuples and consequently are not processed by autovacuum. (Autovacuum does process table partitions just like other tables.) Unfortunately, this means that autovacuum does not run ANALYZE on partitioned tables, and this can cause suboptimal plans for queries that reference partitioned table statistics. You can work around this problem by manually running ANALYZE on partitioned tables when they are first populated, and again whenever the distribution of data in their partitions changes significantly.

Temporary tables cannot be accessed by autovacuum. Therefore, appropriate vacuum and analyze operations should be performed via session SQL commands.

The default thresholds and scale factors are taken from `postgresql.conf`, but it is possible to override them (and many other autovacuum control parameters) on a per-table basis; see Storage Parameters for more information. If a setting has been changed via a table's storage parameters, that value is used when processing that table; otherwise the global settings are used. See Section 19.10 for more details on the global settings.

When multiple workers are running, the autovacuum cost delay parameters (see Section 19.4.4) are "balanced" among all the running workers, so that the total I/O impact on the system is the same regardless of the number of workers actually running. However, any workers processing tables whose per-table `autovacuum_vacuum_cost_delay` or `autovacuum_vacuum_cost_limit` storage parameters have been set are not considered in the balancing algorithm.

Autovacuum workers generally don't block other commands. If a process attempts to acquire a lock that conflicts with the SHARE UPDATE EXCLUSIVE lock held by autovacuum, lock acquisition will interrupt the autovacuum. For conflicting lock modes, see Table 13.2. However, if the autovacuum is running to prevent transaction ID wraparound (i.e., the autovacuum query name in the `pg_stat_activity` view ends with `(to prevent wraparound)`), the autovacuum is not automatically interrupted.

> **Warning**
>
> Regularly running commands that acquire locks conflicting with a SHARE UPDATE EXCLUSIVE lock (e.g., ANALYZE) can effectively prevent autovacuums from ever completing.

24.2. Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command or a series of individual rebuilding steps.

B-tree index pages that have become completely empty are reclaimed for re-use. However, there is still a possibility of inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. Therefore, a usage pattern in which most, but not all, keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

The potential for bloat in non-B-tree indexes has not been well researched. It is a good idea to periodically monitor the index's physical size when using any non-B-tree index type.

Also, for B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

REINDEX can be used safely and easily in all cases. This command requires an ACCESS EXCLUSIVE lock by default, hence it is often preferable to execute it with its CONCURRENTLY option, which requires only a SHARE UPDATE EXCLUSIVE lock.

24.3. Log File Maintenance

It is a good idea to save the database server's log output somewhere, rather than just discarding it via /dev/null. The log output is invaluable when diagnosing problems.

> **Note**
>
> The server log can contain sensitive information and needs to be protected, no matter how or where it is stored, or the destination to which it is routed. For example, some DDL statements might contain plaintext passwords or other authentication details. Logged statements at the ERROR level might show the SQL source code for applications and might also contain some parts of data rows. Recording data, events and related information is the intended function of this facility, so this is not a leakage or a bug. Please ensure the server logs are visible only to appropriately authorized people.

Log output tends to be voluminous (especially at higher debug levels) so you won't want to save it indefinitely. You need to *rotate* the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the stderr of postgres into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This might be acceptable if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server's stderr output to some type of log rotation program. There is a built-in log rotation facility, which you can use by setting the configuration parameter logging_collector to true in postgresql.conf. The control parameters for this program are described in Section 19.8.1. You can also use this approach to capture the log data in machine readable CSV (comma-separated values) format.

Alternatively, you might prefer to use an external log rotation program if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. One way to do this is to pipe the server's stderr output to the desired program. If you start the server with pg_ctl, then stderr is already redirected to stdout, so you just need a pipe command, for example:

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

You can combine these approaches by setting up logrotate to collect log files produced by PostgreSQL built-in logging collector. In this case, the logging collector defines the names and location of the log files, while logrotate periodically archives these files. When initiating log rotation, logrotate must ensure that the application sends further output to the new file. This is commonly done with a postrotate script that sends a SIGHUP signal to the application, which then reopens the log file. In PostgreSQL, you can run pg_ctl with the logrotate option instead. When the server receives this command, the server either switches to a new log file or reopens the existing file, depending on the logging configuration (see Section 19.8.1).

> **Note**
>
> When using static log file names, the server might fail to reopen the log file if the max open file limit is reached or a file table overflow occurs. In this case, log messages are sent to the old log

file until a successful log rotation. If logrotate is configured to compress the log file and delete it, the server may lose the messages logged in this time frame. To avoid this issue, you can configure the logging collector to dynamically assign log file names and use a `prerotate` script to ignore open log files.

Another production-grade approach to managing log output is to send it to syslog and let syslog deal with file rotation. To do this, set the configuration parameter `log_destination` to `syslog` (to log to syslog only) in `postgresql.conf`. Then you can send a `SIGHUP` signal to the syslog daemon whenever you want to force it to start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it might truncate or drop messages just when you need them the most. Also, on Linux, syslog will flush each message to disk, yielding poor performance. (You can use a "`-`" at the start of the file name in the syslog configuration file to disable syncing.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-useful log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

pgBadger[2] is an external project that does sophisticated log file analysis. check_postgres[3] provides Nagios alerts when important messages appear in the log files, as well as detection of many other extraordinary conditions.

[2] https://pgbadger.darold.net/
[3] https://bucardo.org/check_postgres/

Chapter 25. Backup and Restore

As with everything that contains valuable data, PostgreSQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a clear understanding of the underlying techniques and assumptions.

There are three fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup
- Continuous archiving

Each has its own strengths and weaknesses; each is discussed in turn in the following sections.

25.1. SQL Dump

The idea behind this dump method is to generate a file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program pg_dump for this purpose. The basic usage of this command is:

```
pg_dump dbname > dumpfile
```

As you see, pg_dump writes its result to the standard output. We will see below how this can be useful. While the above command creates a text file, pg_dump can create files in other formats that allow for parallelism and more fine-grained control of object restoration.

pg_dump is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can perform this backup procedure from any remote host that has access to the database. But remember that pg_dump does not operate with special permissions. In particular, it must have read access to all tables that you want to back up, so in order to back up the entire database you almost always have to run it as a database superuser. (If you do not have sufficient privileges to back up the entire database, you can still back up portions of the database to which you do have access using options such as `-n schema` or `-t table`.)

To specify which database server pg_dump should contact, use the command line options `-h host` and `-p port`. The default host is the local host or whatever your `PGHOST` environment variable specifies. Similarly, the default port is indicated by the `PGPORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

Like any other PostgreSQL client application, pg_dump will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that pg_dump connections are subject to the normal client authentication mechanisms (which are described in Chapter 20).

An important advantage of pg_dump over the other backup methods described later is that pg_dump's output can generally be re-loaded into newer versions of PostgreSQL, whereas file-level backups and continuous archiving are both extremely server-version-specific. pg_dump is also the only method that will work when transferring a database to a different machine architecture, such as going from a 32-bit to a 64-bit server.

Dumps created by pg_dump are internally consistent, meaning, the dump represents a snapshot of the database at the time pg_dump began running. pg_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as most forms of `ALTER TABLE`.)

25.1.1. Restoring the Dump

Text files created by pg_dump are intended to be read by the psql program using its default settings. The general command form to restore a text dump is

```
psql -X dbname < dumpfile
```

where *dumpfile* is the file output by the pg_dump command. The database *dbname* will not be created by this command, so you must create it yourself from `template0` before executing psql (e.g., with `createdb -T template0 dbname`). To ensure psql runs with its default settings, use the `-X` (`--no-psqlrc`) option. psql supports options similar to pg_dump for specifying the database server to connect to and the user name to use. See the psql reference page for more information.

Non-text file dumps should be restored using the pg_restore utility.

Before restoring an SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

By default, the psql script will continue to execute after an SQL error is encountered. You might wish to run psql with the `ON_ERROR_STOP` variable set to alter that behavior and have psql exit with an exit status of 3 if an SQL error occurs:

```
psql -X --set ON_ERROR_STOP=on dbname < dumpfile
```

Either way, you will only have a partially restored database. Alternatively, you can specify that the whole dump should be restored as a single transaction, so the restore is either fully completed or fully rolled back. This mode can be specified by passing the `-1` or `--single-transaction` command-line options to psql. When using this mode, be aware that even a minor error can rollback a restore that has already run for many hours. However, that might still be preferable to manually cleaning up a complex database after a partially restored dump.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

```
pg_dump -h host1 dbname | psql -X -h host2 dbname
```

> **Important**
>
> The dumps produced by pg_dump are relative to `template0`. This means that any languages, procedures, etc. added via `template1` will also be dumped by pg_dump. As a result, when restoring, if you are using a customized `template1`, you must create the empty database from `template0`, as in the example above.

After restoring a backup, it is wise to run `ANALYZE` on each database so the query optimizer has useful statistics; see Section 24.1.3 and Section 24.1.6 for more information. For more advice on how to load large amounts of data into PostgreSQL efficiently, refer to Section 14.4.

25.1.2. Using pg_dumpall

pg_dump dumps only a single database at a time, and it does not dump information about roles or tablespaces (because those are cluster-wide rather than per-database). To support convenient dumping of the entire contents of a database cluster, the pg_dumpall program is provided. pg_dumpall backs

up each database in a given cluster, and also preserves cluster-wide data such as role and tablespace definitions. The basic usage of this command is:

```
pg_dumpall > dumpfile
```

The resulting dump can be restored with psql:

```
psql -X -f dumpfile postgres
```

(Actually, you can specify any existing database name to start from, but if you are loading into an empty cluster then `postgres` should usually be used.) It is always necessary to have database superuser access when restoring a pg_dumpall dump, as that is required to restore the role and tablespace information. If you use tablespaces, make sure that the tablespace paths in the dump are appropriate for the new installation.

pg_dumpall works by emitting commands to re-create roles, tablespaces, and empty databases, then invoking pg_dump for each database. This means that while each database will be internally consistent, the snapshots of different databases are not synchronized.

Cluster-wide data can be dumped alone using the pg_dumpall `--globals-only` option. This is necessary to fully backup the cluster if running the pg_dump command on individual databases.

25.1.3. Handling Large Databases

Some operating systems have maximum file size limits that cause problems when creating large pg_dump output files. Fortunately, pg_dump can write to the standard output, so you can use standard Unix tools to work around this potential problem. There are several possible methods:

Use compressed dumps. You can use your favorite compression program, for example gzip:

```
pg_dump dbname | gzip > filename.gz
```

Reload with:

```
gunzip -c filename.gz | psql dbname
```

or:

```
cat filename.gz | gunzip | psql dbname
```

Use `split`. The `split` command allows you to split the output into smaller files that are acceptable in size to the underlying file system. For example, to make 2 gigabyte chunks:

```
pg_dump dbname | split -b 2G - filename
```

Reload with:

```
cat filename* | psql dbname
```

If using GNU split, it is possible to use it and gzip together:

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

It can be restored using `zcat`.

Use pg_dump's custom dump format. If PostgreSQL was built on a system with the zlib compression library installed, the custom dump format will compress data as it writes it to the output file. This will produce dump file sizes similar to using `gzip`, but it has the added advantage that tables can be restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

A custom-format dump is not a script for psql, but instead must be restored with pg_restore, for example:

```
pg_restore -d dbname filename
```

See the pg_dump and pg_restore reference pages for details.

For very large databases, you might need to combine `split` with one of the other two approaches.

Use pg_dump's parallel dump feature. To speed up the dump of a large database, you can use pg_dump's parallel mode. This will dump multiple tables at the same time. You can control the degree of parallelism with the `-j` parameter. Parallel dumps are only supported for the "directory" archive format.

```
pg_dump -j num -F d -f out.dir dbname
```

You can use `pg_restore -j` to restore a dump in parallel. This will work for any archive of either the "custom" or the "directory" archive mode, whether or not it has been created with `pg_dump -j`.

25.2. File System Level Backup

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database; Section 18.2 explains where these files are located. You can use whatever method you prefer for doing file system backups; for example:

```
tar -cf backup.tar /usr/local/pgsql/data
```

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server *must* be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will *not* work (in part because `tar` and similar tools do not take an atomic snapshot of the state of the file system, but also because of internal buffering within the server). Information about stopping the server can be found in Section 18.5. Needless to say, you also need to shut down the server before restoring the data.

2. If you have dug into the details of the file system layout of the database, you might be tempted to try to back up or restore only certain individual tables or databases from their respective files or directories. This will *not* work because the information contained in these files is not usable without the commit log files, `pg_xact/*`, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the associated `pg_xact` data because that would render all other tables in the database cluster useless. So file system backups only work for complete backup and restoration of an entire database cluster.

An alternative file-system backup approach is to make a “consistent snapshot” of the data directory, if the file system supports that functionality (and you are willing to trust that it is implemented correctly). The typical procedure is to make a “frozen snapshot” of the volume containing the database, then copy the whole data directory (not just parts, see above) from the snapshot to a backup device, then release the frozen snapshot. This will work even while the database server is running. However, a backup created in this way saves the database files in a state as if the database server was not properly shut down; therefore, when you start the database server on the backed-up data, it will think the previous server instance crashed and will replay the WAL log. This is not a problem; just be aware of it (and be sure to include the WAL files in your backup). You can perform a `CHECKPOINT` before taking the snapshot to reduce recovery time.

If your database is spread across multiple file systems, there might not be any way to obtain exactly-simultaneous frozen snapshots of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots *must* be simultaneous. Read your file system documentation very carefully before trusting the consistent-snapshot technique in such situations.

If simultaneous snapshots are not possible, one option is to shut down the database server long enough to establish all the frozen snapshots. Another option is to perform a continuous archiving base backup (Section 25.3.2) because such backups are immune to file system changes during the backup. This requires enabling continuous archiving just during the backup process; restore is done using continuous archive recovery (Section 25.3.5).

Another option is to use rsync to perform a file system backup. This is done by first running rsync while the database server is running, then shutting down the database server long enough to do an `rsync --checksum`. (`--checksum` is necessary because `rsync` only has file modification-time granularity of one second.) The second rsync will be quicker than the first, because it has relatively little data to transfer, and the end result will be consistent because the server was down. This method allows a file system backup to be performed with minimal downtime.

Note that a file system backup will typically be larger than an SQL dump. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.) However, taking a file system backup might be faster.

25.3. Continuous Archiving and Point-in-Time Recovery (PITR)

At all times, PostgreSQL maintains a *write ahead log* (WAL) in the `pg_wal/` subdirectory of the cluster's data directory. The log records every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by “replaying” the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the file system backup and then replay from the backed-up WAL files to bring the system to a current state. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we do not need a file system snapshot capability, just tar or a similar archiving tool.

- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.

- It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique

supports *point-in-time recovery*: it is possible to restore the database to its state at any time since your base backup was taken.

- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a *warm standby* system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

> **Note**
>
> pg_dump and pg_dumpall do not produce file-system-level backups and cannot be used as part of a continuous-archiving solution. Such dumps are *logical* and do not contain enough information to be used by WAL replay.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup might be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called “online backup” by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files *before* you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.

25.3.1. Setting Up WAL Archiving

In an abstract sense, a running PostgreSQL system produces an indefinitely long sequence of WAL records. The system physically divides this sequence into WAL *segment files*, which are normally 16MB apiece (although the segment size can be altered during initdb). The segment files are given numeric names that reflect their position in the abstract WAL sequence. When not using WAL archiving, the system normally creates just a few segment files and then “recycles” them by renaming no-longer-needed segment files to higher segment numbers. It's assumed that segment files whose contents precede the last checkpoint are no longer of interest and can be recycled.

When archiving WAL data, we need to capture the contents of each segment file once it is filled, and save that data somewhere before the segment file is recycled for reuse. Depending on the application and the available hardware, there could be many different ways of “saving the data somewhere”: we could copy the segment files to an NFS-mounted directory on another machine, write them onto a tape drive (ensuring that you have a way of identifying the original name of each file), or batch them together and burn them onto CDs, or something else entirely. To provide the database administrator with flexibility, PostgreSQL tries not to make any assumptions about how the archiving will be done. Instead, PostgreSQL lets the administrator specify a shell command or an archive library to be executed to copy a completed segment file to wherever it needs to go. This could be as simple as a shell command that uses `cp`, or it could invoke a complex C function — it's all up to you.

To enable WAL archiving, set the wal_level configuration parameter to `replica` or higher, archive_mode to `on`, specify the shell command to use in the archive_command configuration parameter or specify the library to use in the archive_library configuration parameter. In practice these settings will always be placed in the `postgresql.conf` file.

In `archive_command`, `%p` is replaced by the path name of the file to archive, while `%f` is replaced by only the file name. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Use `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /
mnt/server/archivedir/%f'  # Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  #
 Windows
```

which will copy archivable WAL segments to the directory `/mnt/server/archivedir`. (This is an example, not a recommendation, and might not work on all platforms.) After the `%p` and `%f` parameters have been replaced, the actual command executed might look like this:

```
test ! -f /mnt/server/archivedir/000000010000000A900000065
 && cp pg_wal/000000010000000A900000065 /mnt/server/
archivedir/000000010000000A900000065
```

A similar command will be generated for each new file to be archived.

The archive command will be executed under the ownership of the same user that the PostgreSQL server is running as. Since the series of WAL files being archived contains effectively everything in your database, you will want to be sure that the archived data is protected from prying eyes; for example, archive into a directory that does not have group or world read access.

It is important that the archive command return zero exit status if and only if it succeeds. Upon getting a zero result, PostgreSQL will assume that the file has been successfully archived, and will remove or recycle it. However, a nonzero status tells PostgreSQL that the file was not archived; it will try again periodically until it succeeds.

Another way to archive is to use a custom archive module as the `archive_library`. Since such modules are written in `C`, creating your own may require considerably more effort than writing a shell command. However, archive modules can be more performant than archiving via shell, and they will have access to many useful server resources. For more information about archive modules, see Chapter 49.

When the archive command is terminated by a signal (other than SIGTERM that is used as part of a server shutdown) or an error by the shell with an exit status greater than 125 (such as command not found), or if the archive function emits an `ERROR` or `FATAL`, the archiver process aborts and gets restarted by the postmaster. In such cases, the failure is not reported in pg_stat_archiver.

Archive commands and libraries should generally be designed to refuse to overwrite any pre-existing archive file. This is an important safety feature to preserve the integrity of your archive in case of administrator error (such as sending the output of two different servers to the same archive directory). It is advisable to test your proposed archive library to ensure that it does not overwrite an existing file.

In rare cases, PostgreSQL may attempt to re-archive a WAL file that was previously archived. For example, if the system crashes before the server makes a durable record of archival success, the server will attempt to archive the file again after restarting (provided archiving is still enabled). When an archive command or library encounters a pre-existing file, it should return a zero status or `true`, respectively, if the WAL file has identical contents to the pre-existing archive and the pre-existing archive is fully persisted to storage. If a pre-existing file contains different contents than the WAL file being archived, the archive command or library *must* return a nonzero status or `false`, respectively.

The example command above for Unix avoids overwriting a pre-existing archive by including a separate `test` step. On some Unix platforms, `cp` has switches such as `-i` that can be used to do the same thing less verbosely, but you should not rely on these without verifying that the right exit status is returned. (In particular, GNU `cp` will return status zero when `-i` is used and the target file already exists, which is *not* the desired behavior.)

While designing your archiving setup, consider what will happen if the archive command or library fails repeatedly because some aspect requires operator intervention or the archive runs out of space. For example, this could occur if you write to tape without an autochanger; when the tape fills, nothing further can be archived until the tape is swapped. You should ensure that any error condition or request

to a human operator is reported appropriately so that the situation can be resolved reasonably quickly. The `pg_wal/` directory will continue to fill with WAL segment files until the situation is resolved. (If the file system containing `pg_wal/` fills up, PostgreSQL will do a PANIC shutdown. No committed transactions will be lost, but the database will remain offline until you free some space.)

The speed of the archive command or library is unimportant as long as it can keep up with the average rate at which your server generates WAL data. Normal operation continues even if the archiving process falls a little behind. If archiving falls significantly behind, this will increase the amount of data that would be lost in the event of a disaster. It will also mean that the `pg_wal/` directory will contain large numbers of not-yet-archived segment files, which could eventually exceed available disk space. You are advised to monitor the archiving process to ensure that it is working as you intend.

In writing your archive command or library, you should assume that the file names to be archived can be up to 64 characters long and can contain any combination of ASCII letters, digits, and dots. It is not necessary to preserve the original relative path (`%p`) but it is necessary to preserve the file name (`%f`).

Note that although WAL archiving will allow you to restore any modifications made to the data in your PostgreSQL database, it will not restore changes made to configuration files (that is, `postgresql.conf`, `pg_hba.conf` and `pg_ident.conf`), since those are edited manually rather than through SQL operations. You might wish to keep the configuration files in a location that will be backed up by your regular file system backup procedures. See Section 19.2 for how to relocate the configuration files.

The archive command or function is only invoked on completed WAL segments. Hence, if your server generates only little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set archive_timeout to force the server to switch to a new WAL segment file at least that often. Note that archived files that are archived early due to a forced switch are still the same length as completely full files. It is therefore unwise to set a very short `archive_timeout` — it will bloat your archive storage. `archive_timeout` settings of a minute or so are usually reasonable.

Also, you can force a segment switch manually with `pg_switch_wal` if you want to ensure that a just-finished transaction is archived as soon as possible. Other utility functions related to WAL management are listed in Table 9.95.

When `wal_level` is `minimal` some SQL commands are optimized to avoid WAL logging, as described in Section 14.4.7. If archiving or streaming replication were turned on during execution of one of these statements, WAL would not contain enough information for archive recovery. (Crash recovery is unaffected.) For this reason, `wal_level` can only be changed at server start. However, `archive_command` and `archive_library` can be changed with a configuration file reload. If you are archiving via shell and wish to temporarily stop archiving, one way to do it is to set `archive_command` to the empty string (`''`). This will cause WAL files to accumulate in `pg_wal/` until a working `archive_command` is re-established.

25.3.2. Making a Base Backup

The easiest way to perform a base backup is to use the pg_basebackup tool. It can create a base backup either as regular files or as a tar archive. If more flexibility than pg_basebackup can provide is required, you can also make a base backup using the low level API (see Section 25.3.4).

It is not necessary to be concerned about the amount of time it takes to make a base backup. However, if you normally run the server with `full_page_writes` disabled, you might notice a drop in performance while the backup runs since `full_page_writes` is effectively forced on during backup mode.

To make use of the backup, you will need to keep all the WAL segment files generated during and after the file system backup. To aid you in doing this, the base backup process creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need for the file system backup. For example, if the starting WAL

file is `000000010000123400000055CD` the backup history file will be named something like `000000010000123400000055CD.007C9330.backup`. (The second part of the file name stands for an exact position within the WAL file, and can ordinarily be ignored.) Once you have safely archived the file system backup and the WAL segment files used during the backup (as specified in the backup history file), all archived WAL segments with names numerically less are no longer needed to recover the file system backup and can be deleted. However, you should consider keeping several backup sets to be absolutely certain that you can recover your data.

The backup history file is just a small text file. It contains the label string you gave to pg_basebackup, as well as the starting and ending times and WAL segments of the backup. If you used the label to identify the associated dump file, then the archived history file is enough to tell you which dump file to restore.

Since you have to keep around all the archived WAL files back to your last base backup, the interval between base backups should usually be chosen based on how much storage you want to expend on archived WAL files. You should also consider how long you are prepared to spend recovering, if recovery should be necessary — the system will have to replay all those WAL segments, and that could take awhile if it has been a long time since the last base backup.

25.3.3. Making an Incremental Backup

You can use pg_basebackup to take an incremental backup by specifying the `--incremental` option. You must supply, as an argument to `--incremental`, the backup manifest to an earlier backup from the same server. In the resulting backup, non-relation files will be included in their entirety, but some relation files may be replaced by smaller incremental files which contain only the blocks which have been changed since the earlier backup and enough metadata to reconstruct the current version of the file.

To figure out which blocks need to be backed up, the server uses WAL summaries, which are stored in the data directory, inside the directory `pg_wal/summaries`. If the required summary files are not present, an attempt to take an incremental backup will fail. The summaries present in this directory must cover all LSNs from the start LSN of the prior backup to the start LSN of the current backup. Since the server looks for WAL summaries just after establishing the start LSN of the current backup, the necessary summary files probably won't be instantly present on disk, but the server will wait for any missing files to show up. This also helps if the WAL summarization process has fallen behind. However, if the necessary files have already been removed, or if the WAL summarizer doesn't catch up quickly enough, the incremental backup will fail.

When restoring an incremental backup, it will be necessary to have not only the incremental backup itself but also all earlier backups that are required to supply the blocks omitted from the incremental backup. See pg_combinebackup for further information about this requirement. Note that there are restrictions on the use of `pg_combinebackup` when the checksum status of the cluster has been changed; see pg_combinebackup limitations.

Note that all of the requirements for making use of a full backup also apply to an incremental backup. For instance, you still need all of the WAL segment files generated during and after the file system backup, and any relevant WAL history files. And you still need to create a `recovery.signal` (or `standby.signal`) and perform recovery, as described in Section 25.3.5. The requirement to have earlier backups available at restore time and to use `pg_combinebackup` is an additional requirement on top of everything else. Keep in mind that PostgreSQL has no built-in mechanism to figure out which backups are still needed as a basis for restoring later incremental backups. You must keep track of the relationships between your full and incremental backups on your own, and be certain not to remove earlier backups if they might be needed when restoring later incremental backups.

Incremental backups typically only make sense for relatively large databases where a significant portion of the data does not change, or only changes slowly. For a small database, it's simpler to ignore the existence of incremental backups and simply take full backups, which are simpler to manage. For a large database all of which is heavily modified, incremental backups won't be much smaller than full backups.

An incremental backup is only possible if replay would begin from a later checkpoint than for the previous backup upon which it depends. If you take the incremental backup on the primary, this condition is always satisfied, because each backup triggers a new checkpoint. On a standby, replay begins from the most recent restartpoint. Therefore, an incremental backup of a standby server can fail if there has been very little activity since the previous backup, since no new restartpoint might have been created.

25.3.4. Making a Base Backup Using the Low Level API

Instead of taking a full or incremental base backup using pg_basebackup, you can take a base backup using the low-level API. This procedure contains a few more steps than the pg_basebackup method, but is relatively simple. It is very important that these steps are executed in sequence, and that the success of a step is verified before proceeding to the next step.

Multiple backups are able to be run concurrently (both those started using this backup API and those started using pg_basebackup).

1. Ensure that WAL archiving is enabled and working.

2. Connect to the server (it does not matter which database) as a user with rights to run `pg_backup_start` (superuser, or a user who has been granted `EXECUTE` on the function) and issue the command:

   ```
   SELECT pg_backup_start(label => 'label', fast => false);
   ```

   where `label` is any string you want to use to uniquely identify this backup operation. The connection calling `pg_backup_start` must be maintained until the end of the backup, or the backup will be automatically aborted.

   Online backups are always started at the beginning of a checkpoint. By default, `pg_backup_start` will wait for the next regularly scheduled checkpoint to complete, which may take a long time (see the configuration parameters checkpoint_timeout and checkpoint_completion_target). This is usually preferable as it minimizes the impact on the running system. If you want to start the backup as soon as possible, pass `true` as the second parameter to `pg_backup_start` and it will request an immediate checkpoint, which will finish as fast as possible using as much I/O as possible.

3. Perform the backup, using any convenient file-system-backup tool such as tar or cpio (not pg_dump or pg_dumpall). It is neither necessary nor desirable to stop normal operation of the database while you do this. See Section 25.3.4.1 for things to consider during this backup.

4. In the same connection as before, issue the command:

   ```
   SELECT * FROM pg_backup_stop(wait_for_archive => true);
   ```

   This terminates backup mode. On a primary, it also performs an automatic switch to the next WAL segment. On a standby, it is not possible to automatically switch WAL segments, so you may wish to run `pg_switch_wal` on the primary to perform a manual switch. The reason for the switch is to arrange for the last WAL segment file written during the backup interval to be ready to archive.

   `pg_backup_stop` will return one row with three values. The second of these fields should be written to a file named `backup_label` in the root directory of the backup. The third field should be written to a file named `tablespace_map` unless the field is empty. These files are vital to the backup working and must be written byte for byte without modification, which may require opening the file in binary mode.

5. Once the WAL segment files active during the backup are archived, you are done. The file identified by `pg_backup_stop`'s first return value is the last segment that is required to

form a complete set of backup files. On a primary, if archive_mode is enabled and the wait_for_archive parameter is true, pg_backup_stop does not return until the last segment has been archived. On a standby, archive_mode must be always in order for pg_backup_stop to wait. Archiving of these files happens automatically since you have already configured archive_command or archive_library. In most cases this happens quickly, but you are advised to monitor your archive system to ensure there are no delays. If the archive process has fallen behind because of failures of the archive command or library, it will keep retrying until the archive succeeds and the backup is complete. If you wish to place a time limit on the execution of pg_backup_stop, set an appropriate statement_timeout value, but make note that if pg_backup_stop terminates because of this your backup may not be valid.

If the backup process monitors and ensures that all WAL segment files required for the backup are successfully archived then the wait_for_archive parameter (which defaults to true) can be set to false to have pg_backup_stop return as soon as the stop backup record is written to the WAL. By default, pg_backup_stop will wait until all WAL has been archived, which can take some time. This option must be used with caution: if WAL archiving is not monitored correctly then the backup might not include all of the WAL files and will therefore be incomplete and not able to be restored.

25.3.4.1. Backing Up the Data Directory

Some file system backup tools emit warnings or errors if the files they are trying to copy change while the copy proceeds. When taking a base backup of an active database, this situation is normal and not an error. However, you need to ensure that you can distinguish complaints of this sort from real errors. For example, some versions of rsync return a separate exit code for “vanished source files”, and you can write a driver script to accept this exit code as a non-error case. Also, some versions of GNU tar return an error code indistinguishable from a fatal error if a file was truncated while tar was copying it. Fortunately, GNU tar versions 1.16 and later exit with 1 if a file was changed during the backup, and 2 for other errors. With GNU tar version 1.23 and later, you can use the warning options --warning=no-file-changed --warning=no-file-removed to hide the related warning messages.

Be certain that your backup includes all of the files under the database cluster directory (e.g., /usr/local/pgsql/data). If you are using tablespaces that do not reside underneath this directory, be careful to include them as well (and be sure that your backup archives symbolic links as links, otherwise the restore will corrupt your tablespaces).

You should, however, omit from the backup the files within the cluster's pg_wal/ subdirectory. This slight adjustment is worthwhile because it reduces the risk of mistakes when restoring. This is easy to arrange if pg_wal/ is a symbolic link pointing to someplace outside the cluster directory, which is a common setup anyway for performance reasons. You might also want to exclude postmaster.pid and postmaster.opts, which record information about the running postmaster, not about the postmaster which will eventually use this backup. (These files can confuse pg_ctl.)

It is often a good idea to also omit from the backup the files within the cluster's pg_replslot/ directory, so that replication slots that exist on the primary do not become part of the backup. Otherwise, the subsequent use of the backup to create a standby may result in indefinite retention of WAL files on the standby, and possibly bloat on the primary if hot standby feedback is enabled, because the clients that are using those replication slots will still be connecting to and updating the slots on the primary, not the standby. Even if the backup is only intended for use in creating a new primary, copying the replication slots isn't expected to be particularly useful, since the contents of those slots will likely be badly out of date by the time the new primary comes on line.

The contents of the directories pg_dynshmem/, pg_notify/, pg_serial/, pg_snapshots/, pg_stat_tmp/, and pg_subtrans/ (but not the directories themselves) can be omitted from the backup as they will be initialized on postmaster startup.

Any file or directory beginning with pgsql_tmp can be omitted from the backup. These files are removed on postmaster start and the directories will be recreated as needed.

`pg_internal.init` files can be omitted from the backup whenever a file of that name is found. These files contain relation cache data that is always rebuilt when recovering.

The backup label file includes the label string you gave to `pg_backup_start`, as well as the time at which `pg_backup_start` was run, and the name of the starting WAL file. In case of confusion it is therefore possible to look inside a backup file and determine exactly which backup session the dump file came from. The tablespace map file includes the symbolic link names as they exist in the directory `pg_tblspc/` and the full path of each symbolic link. These files are not merely for your information; their presence and contents are critical to the proper operation of the system's recovery process.

It is also possible to make a backup while the server is stopped. In this case, you obviously cannot use `pg_backup_start` or `pg_backup_stop`, and you will therefore be left to your own devices to keep track of which backup is which and how far back the associated WAL files go. It is generally better to follow the continuous archiving procedure above.

25.3.5. Recovering Using a Continuous Archive Backup

Okay, the worst has happened and you need to recover from your backup. Here is the procedure:

1. Stop the server, if it's running.
2. If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you should at least save the contents of the cluster's `pg_wal` subdirectory, as it might contain WAL files which were not archived before the system went down.
3. Remove all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.
4. If you're restoring a full backup, you can restore the database files directly into the target directories. Be sure that they are restored with the right ownership (the database system user, not `root`!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in `pg_tblspc/` were correctly restored.
5. If you're restoring an incremental backup, you'll need to restore the incremental backup and all earlier backups upon which it directly or indirectly depends to the machine where you are performing the restore. These backups will need to be placed in separate directories, not the target directories where you want the running server to end up. Once this is done, use pg_combinebackup to pull data from the full backup and all of the subsequent incremental backups and write out a synthetic full backup to the target directories. As above, verify that permissions and tablespace links are correct.
6. Remove any files present in `pg_wal/`; these came from the file system backup and are therefore probably obsolete rather than current. If you didn't archive `pg_wal/` at all, then recreate it with proper permissions, being careful to ensure that you re-establish it as a symbolic link if you had it set up that way before.
7. If you have unarchived WAL segment files that you saved in step 2, copy them into `pg_wal/`. (It is best to copy them, not move them, so you still have the unmodified files if a problem occurs and you have to start over.)
8. Set recovery configuration settings in `postgresql.conf` (see Section 19.5.5) and create a file `recovery.signal` in the cluster data directory. You might also want to temporarily modify `pg_hba.conf` to prevent ordinary users from connecting until you are sure the recovery was successful.
9. Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply be restarted and it will continue recovery. Upon completion of the recovery process,

the server will remove `recovery.signal` (to prevent accidentally re-entering recovery mode later) and then commence normal database operations.

10. Inspect the contents of the database to ensure you have recovered to the desired state. If not, return to step 1. If all is well, allow your users to connect by restoring `pg_hba.conf` to normal.

The key part of all this is to set up a recovery configuration that describes how you want to recover and how far the recovery should run. The one thing that you absolutely must specify is the `restore_command`, which tells PostgreSQL how to retrieve archived WAL file segments. Like the `archive_command`, this is a shell command string. It can contain `%f`, which is replaced by the name of the desired WAL file, and `%p`, which is replaced by the path name to copy the WAL file to. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Write `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

which will copy previously archived WAL segments from the directory `/mnt/server/archivedir`. Of course, you can use something much more complicated, perhaps even a shell script that requests the operator to mount an appropriate tape.

It is important that the command return nonzero exit status on failure. The command *will* be called requesting files that are not present in the archive; it must return nonzero when so asked. This is not an error condition. An exception is that if the command was terminated by a signal (other than SIGTERM, which is used as part of a database server shutdown) or an error by the shell (such as command not found), then recovery will abort and the server will not start up.

Not all of the requested files will be WAL segment files; you should also expect requests for files with a suffix of `.history`. Also be aware that the base name of the `%p` path will be different from `%f`; do not expect them to be interchangeable.

WAL segments that cannot be found in the archive will be sought in `pg_wal/`; this allows use of recent un-archived segments. However, segments that are available from the archive will be used in preference to files in `pg_wal/`.

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as possible given the available WAL segments). Therefore, a normal recovery will end with a “file not found” message, the exact text of the error message depending upon your choice of `restore_command`. You may also see an error message at the start of recovery for a file named something like `00000001.history`. This is also normal and does not indicate a problem in simple recovery situations; see Section 25.3.6 for discussion.

If you want to recover to some previous point in time (say, right before the junior DBA dropped your main transaction table), just specify the required stopping point. You can specify the stop point, known as the “recovery target”, either by date/time, named restore point or by completion of a specific transaction ID. As of this writing only the date/time and named restore point options are very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

> **Note**
>
> The stop point must be after the ending time of the base backup, i.e., the end time of `pg_backup_stop`. You cannot use a base backup to recover to a time when that backup was in progress. (To recover to such a time, you must go back to your previous base backup and roll forward from there.)

If recovery finds corrupted WAL data, recovery will halt at that point and the server will not start. In such a case the recovery process could be re-run from the beginning, specifying a “recovery target” before the point of corruption so that recovery can complete normally. If recovery fails for an external

reason, such as a system crash or if the WAL archive has become inaccessible, then the recovery can simply be restarted and it will restart almost from where it failed. Recovery restart works much like checkpointing in normal operation: the server periodically forces all its state to disk, and then updates the `pg_control` file to indicate that the already-processed WAL data need not be scanned again.

25.3.6. Timelines

The ability to restore the database to a previous point in time creates some complexities that are akin to science-fiction stories about time travel and parallel universes. For example, in the original history of the database, suppose you dropped a critical table at 5:15PM on Tuesday evening, but didn't realize your mistake until Wednesday noon. Unfazed, you get out your backup, restore to the point-in-time 5:14PM Tuesday evening, and are up and running. In *this* history of the database universe, you never dropped the table. But suppose you later realize this wasn't such a great idea, and would like to return to sometime Wednesday morning in the original history. You won't be able to if, while your database was up-and-running, it overwrote some of the WAL segment files that led up to the time you now wish you could get back to. Thus, to avoid this, you need to distinguish the series of WAL records generated after you've done a point-in-time recovery from those that were generated in the original database history.

To deal with this problem, PostgreSQL has a notion of *timelines*. Whenever an archive recovery completes, a new timeline is created to identify the series of WAL records generated after that recovery. The timeline ID number is part of WAL segment file names so a new timeline does not overwrite the WAL data generated by previous timelines. For example, in the WAL file name `000000010000123400000055CD`, the leading `00000001` is the timeline ID in hexadecimal. (Note that in other contexts, such as server log messages, timeline IDs are usually printed in decimal.)

It is in fact possible to archive many different timelines. While that might seem like a useless feature, it's often a lifesaver. Consider the situation where you aren't quite sure what point-in-time to recover to, and so have to do several point-in-time recoveries by trial and error until you find the best place to branch off from the old history. Without timelines this process would soon generate an unmanageable mess. With timelines, you can recover to *any* prior state, including states in timeline branches that you abandoned earlier.

Every time a new timeline is created, PostgreSQL creates a “timeline history” file that shows which timeline it branched off from and when. These history files are necessary to allow the system to pick the right WAL segment files when recovering from an archive that contains multiple timelines. Therefore, they are archived into the WAL archive area just like WAL segment files. The history files are just small text files, so it's cheap and appropriate to keep them around indefinitely (unlike the segment files which are large). You can, if you like, add comments to a history file to record your own notes about how and why this particular timeline was created. Such comments will be especially valuable when you have a thicket of different timelines as a result of experimentation.

The default behavior of recovery is to recover to the latest timeline found in the archive. If you wish to recover to the timeline that was current when the base backup was taken or into a specific child timeline (that is, you want to return to some state that was itself generated after a recovery attempt), you need to specify `current` or the target timeline ID in recovery_target_timeline. You cannot recover into timelines that branched off earlier than the base backup.

25.3.7. Tips and Examples

Some tips for configuring continuous archiving are given here.

25.3.7.1. Standalone Hot Backups

It is possible to use PostgreSQL's backup facilities to produce standalone hot backups. These are backups that cannot be used for point-in-time recovery, yet are typically much faster to backup and restore than pg_dump dumps. (They are also much larger than pg_dump dumps, so in some cases the speed advantage might be negated.)

As with base backups, the easiest way to produce a standalone hot backup is to use the pg_basebackup tool. If you include the `-X` parameter when calling it, all the write-ahead log required to use the backup will be included in the backup automatically, and no special action is required to restore the backup.

25.3.7.2. Compressed Archive Logs

If archive storage size is a concern, you can use gzip to compress the archive files:

```
archive_command = 'gzip < %p > /mnt/server/archivedir/%f.gz'
```

You will then need to use gunzip during recovery:

```
restore_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'
```

25.3.7.3. `archive_command` Scripts

Many people choose to use scripts to define their `archive_command`, so that their `postgresql.conf` entry looks very simple:

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Using a separate script file is advisable any time you want to use more than a single command in the archiving process. This allows all complexity to be managed within the script, which can be written in a popular scripting language such as bash or perl.

Examples of requirements that might be solved within a script include:

- Copying data to secure off-site data storage
- Batching WAL files so that they are transferred every three hours, rather than one at a time
- Interfacing with other backup and recovery software
- Interfacing with monitoring software to report errors

> **Tip**
>
> When using an `archive_command` script, it's desirable to enable logging_collector. Any messages written to stderr from the script will then appear in the database server log, allowing complex configurations to be diagnosed easily if they fail.

25.3.8. Caveats

At this writing, there are several limitations of the continuous archiving technique. These will probably be fixed in future releases:

- If a `CREATE DATABASE` command is executed while a base backup is being taken, and then the template database that the `CREATE DATABASE` copied is modified while the base backup is still in progress, it is possible that recovery will cause those modifications to be propagated into the created database as well. This is of course undesirable. To avoid this risk, it is best not to modify any template databases while taking a base backup.
- `CREATE TABLESPACE` commands are WAL-logged with the literal absolute path, and will therefore be replayed as tablespace creations with the same absolute path. This might be undesirable if

the WAL is being replayed on a different machine. It can be dangerous even if the WAL is being replayed on the same machine, but into a new data directory: the replay will still overwrite the contents of the original tablespace. To avoid potential gotchas of this sort, the best practice is to take a new base backup after creating or dropping tablespaces.

It should also be noted that the default WAL format is fairly bulky since it includes many disk page snapshots. These page snapshots are designed to support crash recovery, since we might need to fix partially-written disk pages. Depending on your system hardware and software, the risk of partial writes might be small enough to ignore, in which case you can significantly reduce the total volume of archived WAL files by turning off page snapshots using the full_page_writes parameter. (Read the notes and warnings in Chapter 28 before you do so.) Turning off page snapshots does not prevent use of the WAL for PITR operations. An area for future development is to compress archived WAL data by removing unnecessary page copies even when `full_page_writes` is on. In the meantime, administrators might wish to reduce the number of page snapshots included in WAL by increasing the checkpoint interval parameters as much as feasible.

Chapter 26. High Availability, Load Balancing, and Replication

Database servers can work together to allow a second server to take over quickly if the primary server fails (high availability), or to allow several computers to serve the same data (load balancing). Ideally, database servers could work together seamlessly. Web servers serving static web pages can be combined quite easily by merely load-balancing web requests to multiple machines. In fact, read-only database servers can be combined relatively easily too. Unfortunately, most database servers have a read/write mix of requests, and read/write servers are much harder to combine. This is because though read-only data needs to be placed on each server only once, a write to any server has to be propagated to all servers so that future read requests to those servers return consistent results.

This synchronization problem is the fundamental difficulty for servers working together. Because there is no single solution that eliminates the impact of the sync problem for all use cases, there are multiple solutions. Each solution addresses this problem in a different way, and minimizes its impact for a specific workload.

Some solutions deal with synchronization by allowing only one server to modify the data. Servers that can modify data are called read/write, *master* or *primary* servers. Servers that track changes in the primary are called *standby* or *secondary* servers. A standby server that cannot be connected to until it is promoted to a primary server is called a *warm standby* server, and one that can accept connections and serves read-only queries is called a *hot standby* server.

Some solutions are synchronous, meaning that a data-modifying transaction is not considered committed until all servers have committed the transaction. This guarantees that a failover will not lose any data and that all load-balanced servers will return consistent results no matter which server is queried. In contrast, asynchronous solutions allow some delay between the time of a commit and its propagation to the other servers, opening the possibility that some transactions might be lost in the switch to a backup server, and that load balanced servers might return slightly stale results. Asynchronous communication is used when synchronous would be too slow.

Solutions can also be categorized by their granularity. Some solutions can deal only with an entire database server, while others allow control at the per-table or per-database level.

Performance must be considered in any choice. There is usually a trade-off between functionality and performance. For example, a fully synchronous solution over a slow network might cut performance by more than half, while an asynchronous one might have a minimal performance impact.

The remainder of this section outlines various failover, replication, and load balancing solutions.

26.1. Comparison of Different Solutions

Shared Disk Failover

Shared disk failover avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it were recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior (see Section 18.2.2.1). One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

File System (Block Device) Replication

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the primary. DRBD is a popular file system replication solution for Linux.

Write-Ahead Log Shipping

Warm and hot standby servers can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the standby contains almost all of the data of the main server, and can be quickly made the new primary database server. This can be synchronous or asynchronous and can only be done for the entire database server.

A standby server can be implemented using file-based log shipping (Section 26.2) or streaming replication (see Section 26.2.5), or a combination of both. For information on hot standby, see Section 26.4.

Logical Replication

Logical replication allows a database server to send a stream of data modifications to another server. PostgreSQL logical replication constructs a stream of logical data modifications from the WAL. Logical replication allows replication of data changes on a per-table basis. In addition, a server that is publishing its own changes can also subscribe to changes from another server, allowing data to flow in multiple directions. For more information on logical replication, see Chapter 29. Through the logical decoding interface (Chapter 47), third-party extensions can also provide similar functionality.

Trigger-Based Primary-Standby Replication

A trigger-based replication setup typically funnels data modification queries to a designated primary server. Operating on a per-table basis, the primary server sends data changes (typically) asynchronously to the standby servers. Standby servers can answer queries while the primary is running, and may allow some local data changes or write activity. This form of replication is often used for offloading large analytical or data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple standby servers. Because it updates the standby server asynchronously (in batches), there is possible data loss during fail over.

SQL-Based Replication Middleware

With SQL-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries must be sent to all servers, so that every server receives any changes. But read-only queries can be sent to just one server, allowing the read workload to be distributed among them.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences can have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast rather than actual data changes. If this is unacceptable, either the middleware or the application must determine such values from a single source and then use those values in write queries. Care must also be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED). Pgpool-II and Continuent Tungsten are examples of this type of replication.

Asynchronous Multimaster Replication

For servers that are not regularly connected or have slow communication links, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous mul-

timaster replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules. Bucardo is an example of this type of replication.

Synchronous Multimaster Replication

In synchronous multimaster replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking and commit delays, leading to poor performance. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multimaster replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between primary and standby servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.

PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED) can be used to implement this in application code or middleware.

Table 26.1 summarizes the capabilities of the various solutions listed above.

Table 26.1. High Availability, Load Balancing, and Replication Feature Matrix

Feature	Shared Disk	File System Repl.	Write-Ahead Log Shipping	Logical Repl.	Trigger-Based Repl.	SQL Repl. Middle-ware	Async. MM Repl.	Sync. MM Repl.
Popular examples	NAS	DRBD	built-in streaming repl.	built-in logical repl., pglogical	Londiste, Slony	pgpool-II	Bucardo	
Comm. method	shared disk	disk blocks	WAL	logical decoding	table rows	SQL	table rows	table rows and row locks
No special hardware required		•	•	•	•	•	•	•
Allows multiple primary servers				•		•	•	•
No overhead on primary	•		•	•		•		
No waiting for multiple servers	•		with sync off	with sync off	•		•	
Primary failure will never lose data	•	•	with sync on	with sync on		•		•
Replicas accept read-only queries			with hot standby	•	•	•	•	•

Feature	Shared Disk	File System Repl.	Write-Ahead Log Shipping	Logical Repl.	Trigger-Based Repl.	SQL Repl. Middle-ware	Async. MM Repl.	Sync. MM Repl.
Per-table granularity				•	•		•	•
No conflict resolution necessary	•	•	•		•	•		•

There are a few solutions that do not fit into the above categories:

Data Partitioning

Data partitioning splits tables into data sets. Each set can be modified by only one server. For example, data can be partitioned by offices, e.g., London and Paris, with a server in each office. If queries combining London and Paris data are necessary, an application can query both servers, or primary/standby replication can be used to keep a read-only copy of the other office's data on each server.

Multiple-Server Parallel Query Execution

Many of the above solutions allow multiple servers to handle multiple queries, but none allow a single query to use multiple servers to complete faster. This solution allows multiple servers to work concurrently on a single query. It is usually accomplished by splitting the data among servers and having each server execute its part of the query and return results to a central server where they are combined and returned to the user. This can be implemented using the PL/Proxy tool set.

It should also be noted that because PostgreSQL is open source and easily extended, a number of companies have taken PostgreSQL and created commercial closed-source solutions with unique failover, replication, and load balancing capabilities. These are not discussed here.

26.2. Log-Shipping Standby Servers

Continuous archiving can be used to create a *high availability* (HA) cluster configuration with one or more *standby servers* ready to take over operations if the primary server fails. This capability is widely referred to as *warm standby* or *log shipping*.

The primary and standby server work together to provide this capability, though the servers are only loosely coupled. The primary server operates in continuous archiving mode, while each standby server operates in continuous recovery mode, reading the WAL files from the primary. No changes to the database tables are required to enable this capability, so it offers low administration overhead compared to some other replication solutions. This configuration also has relatively low performance impact on the primary server.

Directly moving WAL records from one database server to another is typically described as log shipping. PostgreSQL implements file-based log shipping by transferring WAL records one file (WAL segment) at a time. WAL files (16MB) can be shipped easily and cheaply over any distance, whether it be to an adjacent system, another system at the same site, or another system on the far side of the globe. The bandwidth required for this technique varies according to the transaction rate of the primary server. Record-based log shipping is more granular and streams WAL changes incrementally over a network connection (see Section 26.2.5).

It should be noted that log shipping is asynchronous, i.e., the WAL records are shipped after transaction commit. As a result, there is a window for data loss should the primary server suffer a catastrophic

failure; transactions not yet shipped will be lost. The size of the data loss window in file-based log shipping can be limited by use of the `archive_timeout` parameter, which can be set as low as a few seconds. However such a low setting will substantially increase the bandwidth required for file shipping. Streaming replication (see Section 26.2.5) allows a much smaller window of data loss.

Recovery performance is sufficiently good that the standby will typically be only moments away from full availability once it has been activated. As a result, this is called a warm standby configuration which offers high availability. Restoring a server from an archived base backup and rollforward will take considerably longer, so that technique only offers a solution for disaster recovery, not high availability. A standby server can also be used for read-only queries, in which case it is called a *hot standby* server. See Section 26.4 for more information.

26.2.1. Planning

It is usually wise to create the primary and standby servers so that they are as similar as possible, at least from the perspective of the database server. In particular, the path names associated with tablespaces will be passed across unmodified, so both primary and standby servers must have the same mount paths for tablespaces if that feature is used. Keep in mind that if CREATE TABLESPACE is executed on the primary, any new mount point needed for it must be created on the primary and all standby servers before the command is executed. Hardware need not be exactly the same, but experience shows that maintaining two identical systems is easier than maintaining two dissimilar ones over the lifetime of the application and system. In any case the hardware architecture must be the same — shipping from, say, a 32-bit to a 64-bit system will not work.

In general, log shipping between servers running different major PostgreSQL release levels is not possible. It is the policy of the PostgreSQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

26.2.2. Standby Server Operation

A server enters standby mode if a `standby.signal` file exists in the data directory when the server is started.

In standby mode, the server continuously applies WAL received from the primary server. The standby server can read WAL from a WAL archive (see restore_command) or directly from the primary over a TCP connection (streaming replication). The standby server will also attempt to restore any WAL found in the standby cluster's `pg_wal` directory. That typically happens after a server restart, when the standby replays again WAL that was streamed from the primary before the restart, but you can also manually copy files to `pg_wal` at any time to have them replayed.

At startup, the standby begins by restoring all WAL available in the archive location, calling `restore_command`. Once it reaches the end of WAL available there and `restore_command` fails, it tries to restore any WAL available in the `pg_wal` directory. If that fails, and streaming replication has been configured, the standby tries to connect to the primary server and start streaming WAL from the last valid record found in archive or `pg_wal`. If that fails or streaming replication is not configured, or if the connection is later disconnected, the standby goes back to step 1 and tries to restore the file from the archive again. This loop of retries from the archive, `pg_wal`, and via streaming replication goes on until the server is stopped or is promoted.

Standby mode is exited and the server switches to normal operation when `pg_ctl promote` is run, or `pg_promote()` is called. Before failover, any WAL immediately available in the archive or in `pg_wal` will be restored, but no attempt is made to connect to the primary.

26.2.3. Preparing the Primary for Standby Servers

Set up continuous archiving on the primary to an archive directory accessible from the standby, as described in Section 25.3. The archive location should be accessible from the standby even when the primary is down, i.e., it should reside on the standby server itself or another trusted server, not on the primary server.

If you want to use streaming replication, set up authentication on the primary server to allow replication connections from the standby server(s); that is, create a role and provide a suitable entry or entries in `pg_hba.conf` with the database field set to `replication`. Also ensure `max_wal_senders` is set to a sufficiently large value in the configuration file of the primary server. If replication slots will be used, ensure that `max_replication_slots` is set sufficiently high as well.

Take a base backup as described in Section 25.3.2 to bootstrap the standby server.

26.2.4. Setting Up a Standby Server

To set up the standby server, restore the base backup taken from primary server (see Section 25.3.5). Create a file `standby.signal` in the standby's cluster data directory. Set restore_command to a simple command to copy files from the WAL archive. If you plan to have multiple standby servers for high availability purposes, make sure that `recovery_target_timeline` is set to `latest` (the default), to make the standby server follow the timeline change that occurs at failover to another standby.

> **Note**
>
> restore_command should return immediately if the file does not exist; the server will retry the command again if necessary.

If you want to use streaming replication, fill in primary_conninfo with a libpq connection string, including the host name (or IP address) and any additional details needed to connect to the primary server. If the primary needs a password for authentication, the password needs to be specified in primary_conninfo as well.

If you're setting up the standby server for high availability purposes, set up WAL archiving, connections and authentication like the primary server, because the standby server will work as a primary server after failover.

If you're using a WAL archive, its size can be minimized using the archive_cleanup_command parameter to remove files that are no longer required by the standby server. The pg_archivecleanup utility is designed specifically to be used with `archive_cleanup_command` in typical single-standby configurations, see pg_archivecleanup. Note however, that if you're using the archive for backup purposes, you need to retain files needed to recover from at least the latest base backup, even if they're no longer needed by the standby.

A simple example of configuration is:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass options=''-c wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

You can have any number of standby servers, but if you use streaming replication, make sure you set `max_wal_senders` high enough in the primary to allow them to be connected simultaneously.

26.2.5. Streaming Replication

Streaming replication allows a standby server to stay more up-to-date than is possible with file-based log shipping. The standby connects to the primary, which streams WAL records to the standby as they're generated, without waiting for the WAL file to be filled.

Streaming replication is asynchronous by default (see Section 26.2.8), in which case there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby. This delay is however much smaller than with file-based log shipping, typically under one second assuming the standby is powerful enough to keep up with the load. With streaming replication, `archive_timeout` is not required to reduce the data loss window.

If you use streaming replication without file-based continuous archiving, the server might recycle old WAL segments before the standby has received them. If this occurs, the standby will need to be reinitialized from a new base backup. You can avoid this by setting `wal_keep_size` to a value large enough to ensure that WAL segments are not recycled too early, or by configuring a replication slot for the standby. If you set up a WAL archive that's accessible from the standby, these solutions are not required, since the standby can always use the archive to catch up provided it retains enough segments.

To use streaming replication, set up a file-based log-shipping standby server as described in Section 26.2. The step that turns a file-based log-shipping standby into streaming replication standby is setting the `primary_conninfo` setting to point to the primary server. Set listen_addresses and authentication options (see `pg_hba.conf`) on the primary so that the standby server can connect to the `replication` pseudo-database on the primary server (see Section 26.2.5.1).

On systems that support the keepalive socket option, setting tcp_keepalives_idle, tcp_keepalives_interval and tcp_keepalives_count helps the primary promptly notice a broken connection.

Set the maximum number of concurrent connections from the standby servers (see max_wal_senders for details).

When the standby is started and `primary_conninfo` is set correctly, the standby will connect to the primary after replaying all WAL files available in the archive. If the connection is established successfully, you will see a `walreceiver` in the standby, and a corresponding `walsender` process in the primary.

26.2.5.1. Authentication

It is very important that the access privileges for replication be set up so that only trusted users can read the WAL stream, because it is easy to extract privileged information from it. Standby servers must authenticate to the primary as an account that has the `REPLICATION` privilege or a superuser. It is recommended to create a dedicated user account with `REPLICATION` and `LOGIN` privileges for replication. While `REPLICATION` privilege gives very high permissions, it does not allow the user to modify any data on the primary system, which the `SUPERUSER` privilege does.

Client authentication for replication is controlled by a `pg_hba.conf` record specifying `replication` in the *`database`* field. For example, if the standby is running on host IP `192.168.1.100` and the account name for replication is `foo`, the administrator can add the following line to the `pg_hba.conf` file on the primary:

```
# Allow the user "foo" from host 192.168.1.100 to connect to the
 primary
# as a replication standby if the user's password is correctly
 supplied.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    replication     foo             192.168.1.100/32        md5
```

The host name and port number of the primary, connection user name, and password are specified in the primary_conninfo. The password can also be set in the `~/.pgpass` file on the standby (spec-

ify `replication` in the *`database`* field). For example, if the primary is running on host IP `192.168.1.50`, port `5432`, the account name for replication is `foo`, and the password is `foopass`, the administrator can add the following line to the `postgresql.conf` file on the standby:

```
# The standby connects to the primary that is running on host
 192.168.1.50
# and port 5432 as the user "foo" whose password is "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass'
```

26.2.5.2. Monitoring

An important health indicator of streaming replication is the amount of WAL records generated in the primary, but not yet applied in the standby. You can calculate this lag by comparing the current WAL write location on the primary with the last WAL location received by the standby. These locations can be retrieved using `pg_current_wal_lsn` on the primary and `pg_last_wal_receive_lsn` on the standby, respectively (see Table 9.95 and Table 9.96 for details). The last WAL receive location in the standby is also displayed in the process status of the WAL receiver process, displayed using the `ps` command (see Section 27.1 for details).

You can retrieve a list of WAL sender processes via the `pg_stat_replication` view. Large differences between `pg_current_wal_lsn` and the view's `sent_lsn` field might indicate that the primary server is under heavy load, while differences between `sent_lsn` and `pg_last_wal_receive_lsn` on the standby might indicate network delay, or that the standby is under heavy load.

On a hot standby, the status of the WAL receiver process can be retrieved via the `pg_stat_wal_receiver` view. A large difference between `pg_last_wal_replay_lsn` and the view's `flushed_lsn` indicates that WAL is being received faster than it can be replayed.

26.2.6. Replication Slots

Replication slots provide an automated way to ensure that the primary server does not remove WAL segments until they have been received by all standbys, and that the primary does not remove rows which could cause a recovery conflict even when the standby is disconnected.

In lieu of using replication slots, it is possible to prevent the removal of old WAL segments using wal_keep_size, or by storing the segments in an archive using archive_command or archive_library. A disadvantage of these methods is that they often result in retaining more WAL segments than required, whereas replication slots retain only the number of segments known to be needed.

Similarly, hot_standby_feedback on its own, without also using a replication slot, provides protection against relevant rows being removed by vacuum, but provides no protection during any time period when the standby is not connected.

> **Caution**
>
> Beware that replication slots can cause the server to retain so many WAL segments that they fill up the space allocated for `pg_wal`. max_slot_wal_keep_size can be used to limit the size of WAL files retained by replication slots.

26.2.6.1. Querying and Manipulating Replication Slots

Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character.

Existing replication slots and their state can be seen in the `pg_replication_slots` view.

Slots can be created and dropped either via the streaming replication protocol (see Section 53.4) or via SQL functions (see Section 9.28.6).

26.2.6.2. Configuration Example

You can create a replication slot like this:

```
postgres=# SELECT * FROM
 pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM
 pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

To configure the standby to use this slot, `primary_slot_name` should be configured on the standby. Here is a simple example:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass'
primary_slot_name = 'node_a_slot'
```

26.2.7. Cascading Replication

The cascading replication feature allows a standby server to accept replication connections and stream WAL records to other standbys, acting as a relay. This can be used to reduce the number of direct connections to the primary and also to minimize inter-site bandwidth overheads.

A standby acting as both a receiver and a sender is known as a cascading standby. Standbys that are more directly connected to the primary are known as upstream servers, while those standby servers further away are downstream servers. Cascading replication does not place limits on the number or arrangement of downstream servers, though each standby connects to only one upstream server which eventually links to a single primary server.

A cascading standby sends not only WAL records received from the primary but also those restored from the archive. So even if the replication connection in some upstream connection is terminated, streaming replication continues downstream for as long as new WAL records are available.

Cascading replication is currently asynchronous. Synchronous replication (see Section 26.2.8) settings have no effect on cascading replication at present.

Hot standby feedback propagates upstream, whatever the cascaded arrangement.

If an upstream standby server is promoted to become the new primary, downstream servers will continue to stream from the new primary if `recovery_target_timeline` is set to `'latest'` (the default).

To use cascading replication, set up the cascading standby so that it can accept replication connections (that is, set max_wal_senders and hot_standby, and configure host-based authentication). You will also need to set `primary_conninfo` in the downstream standby to point to the cascading standby.

26.2.8. Synchronous Replication

PostgreSQL streaming replication is asynchronous by default. If the primary server crashes then some transactions that were committed may not have been replicated to the standby server, causing data loss. The amount of data loss is proportional to the replication delay at the time of failover.

Synchronous replication offers the ability to confirm that all changes made by a transaction have been transferred to one or more synchronous standby servers. This extends that standard level of durability offered by a transaction commit. This level of protection is referred to as 2-safe replication in computer science theory, and group-1-safe (group-safe and 1-safe) when `synchronous_commit` is set to `remote_write`.

When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the write-ahead log on disk of both the primary and standby server. The only possibility that data can be lost is if both the primary and the standby suffer crashes at the same time. This can provide a much higher level of durability, though only if the sysadmin is cautious about the placement and management of the two servers. Waiting for confirmation increases the user's confidence that the changes will not be lost in the event of server crashes but it also necessarily increases the response time for the requesting transaction. The minimum wait time is the round-trip time between primary and standby.

Read-only transactions and transaction rollbacks need not wait for replies from standby servers. Subtransaction commits do not wait for responses from standby servers, only top-level commits. Long running actions such as data loading or index building do not wait until the very final commit message. All two-phase commit actions require commit waits, including both prepare and commit.

A synchronous standby can be a physical replication standby or a logical replication subscriber. It can also be any other physical or logical WAL replication stream consumer that knows how to send the appropriate feedback messages. Besides the built-in physical and logical replication systems, this includes special programs such as `pg_receivewal` and `pg_recvlogical` as well as some third-party replication systems and custom programs. Check the respective documentation for details on synchronous replication support.

26.2.8.1. Basic Configuration

Once streaming replication has been configured, configuring synchronous replication requires only one additional configuration step: synchronous_standby_names must be set to a non-empty value. `synchronous_commit` must also be set to `on`, but since this is the default value, typically no change is required. (See Section 19.5.1 and Section 19.6.2.) This configuration will cause each commit to wait for confirmation that the standby has written the commit record to durable storage. `synchronous_commit` can be set by individual users, so it can be configured in the configuration file, for particular users or databases, or dynamically by applications, in order to control the durability guarantee on a per-transaction basis.

After a commit record has been written to disk on the primary, the WAL record is then sent to the standby. The standby sends reply messages each time a new batch of WAL data is written to disk, unless `wal_receiver_status_interval` is set to zero on the standby. In the case that `synchronous_commit` is set to `remote_apply`, the standby sends reply messages when the commit record is replayed, making the transaction visible. If the standby is chosen as a synchronous standby, according to the setting of `synchronous_standby_names` on the primary, the reply messages from that standby will be considered along with those from other synchronous standbys to decide when to release transactions waiting for confirmation that the commit record has been received. These parameters allow the administrator to specify which standby servers should be synchronous standbys. Note that the configuration of synchronous replication is mainly on the primary. Named standbys must be directly connected to the primary; the primary knows nothing about downstream standby servers using cascaded replication.

Setting `synchronous_commit` to `remote_write` will cause each commit to wait for confirmation that the standby has received the commit record and written it out to its own operating system, but not for the data to be flushed to disk on the standby. This setting provides a weaker guarantee of durability than `on` does: the standby could lose the data in the event of an operating system crash,

though not a PostgreSQL crash. However, it's a useful setting in practice because it can decrease the response time for the transaction. Data loss could only occur if both the primary and the standby crash and the database of the primary gets corrupted at the same time.

Setting `synchronous_commit` to `remote_apply` will cause each commit to wait until the current synchronous standbys report that they have replayed the transaction, making it visible to user queries. In simple cases, this allows for load balancing with causal consistency.

Users will stop waiting if a fast shutdown is requested. However, as when using asynchronous replication, the server will not fully shutdown until all outstanding WAL records are transferred to the currently connected standby servers.

26.2.8.2. Multiple Synchronous Standbys

Synchronous replication supports one or more synchronous standby servers; transactions will wait until all the standby servers which are considered as synchronous confirm receipt of their data. The number of synchronous standbys that transactions must wait for replies from is specified in `synchronous_standby_names`. This parameter also specifies a list of standby names and the method (`FIRST` and `ANY`) to choose synchronous standbys from the listed ones.

The method `FIRST` specifies a priority-based synchronous replication and makes transaction commits wait until their WAL records are replicated to the requested number of synchronous standbys chosen based on their priorities. The standbys whose names appear earlier in the list are given higher priority and will be considered as synchronous. Other standby servers appearing later in this list represent potential synchronous standbys. If any of the current synchronous standbys disconnects for whatever reason, it will be replaced immediately with the next-highest-priority standby.

An example of `synchronous_standby_names` for a priority-based multiple synchronous standbys is:

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

In this example, if four standby servers `s1`, `s2`, `s3` and `s4` are running, the two standbys `s1` and `s2` will be chosen as synchronous standbys because their names appear early in the list of standby names. `s3` is a potential synchronous standby and will take over the role of synchronous standby when either of `s1` or `s2` fails. `s4` is an asynchronous standby since its name is not in the list.

The method `ANY` specifies a quorum-based synchronous replication and makes transaction commits wait until their WAL records are replicated to *at least* the requested number of synchronous standbys in the list.

An example of `synchronous_standby_names` for a quorum-based multiple synchronous standbys is:

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

In this example, if four standby servers `s1`, `s2`, `s3` and `s4` are running, transaction commits will wait for replies from at least any two standbys of `s1`, `s2` and `s3`. `s4` is an asynchronous standby since its name is not in the list.

The synchronous states of standby servers can be viewed using the `pg_stat_replication` view.

26.2.8.3. Planning for Performance

Synchronous replication usually requires carefully planned and placed standby servers to ensure applications perform acceptably. Waiting doesn't utilize system resources, but transaction locks continue to be held until the transfer is confirmed. As a result, incautious use of synchronous replication

will reduce performance for database applications because of increased response times and higher contention.

PostgreSQL allows the application developer to specify the durability level required via replication. This can be specified for the system overall, though it can also be specified for specific users or connections, or even individual transactions.

For example, an application workload might consist of: 10% of changes are important customer details, while 90% of changes are less important data that the business can more easily survive if it is lost, such as chat messages between users.

With synchronous replication options specified at the application level (on the primary) we can offer synchronous replication for the most important changes, without slowing down the bulk of the total workload. Application level options are an important and practical tool for allowing the benefits of synchronous replication for high performance applications.

You should consider that the network bandwidth must be higher than the rate of generation of WAL data.

26.2.8.4. Planning for High Availability

`synchronous_standby_names` specifies the number and names of synchronous standbys that transaction commits made when `synchronous_commit` is set to `on`, `remote_apply` or `remote_write` will wait for responses from. Such transaction commits may never be completed if any one of the synchronous standbys should crash.

The best solution for high availability is to ensure you keep as many synchronous standbys as requested. This can be achieved by naming multiple potential synchronous standbys using `synchronous_standby_names`.

In a priority-based synchronous replication, the standbys whose names appear earlier in the list will be used as synchronous standbys. Standbys listed after these will take over the role of synchronous standby if one of current ones should fail.

In a quorum-based synchronous replication, all the standbys appearing in the list will be used as candidates for synchronous standbys. Even if one of them should fail, the other standbys will keep performing the role of candidates of synchronous standby.

When a standby first attaches to the primary, it will not yet be properly synchronized. This is described as `catchup` mode. Once the lag between standby and primary reaches zero for the first time we move to real-time `streaming` state. The catch-up duration may be long immediately after the standby has been created. If the standby is shut down, then the catch-up period will increase according to the length of time the standby has been down. The standby is only able to become a synchronous standby once it has reached `streaming` state. This state can be viewed using the `pg_stat_replication` view.

If primary restarts while commits are waiting for acknowledgment, those waiting transactions will be marked fully committed once the primary database recovers. There is no way to be certain that all standbys have received all outstanding WAL data at time of the crash of the primary. Some transactions may not show as committed on the standby, even though they show as committed on the primary. The guarantee we offer is that the application will not receive explicit acknowledgment of the successful commit of a transaction until the WAL data is known to be safely received by all the synchronous standbys.

If you really cannot keep as many synchronous standbys as requested then you should decrease the number of synchronous standbys that transaction commits must wait for responses from in `synchronous_standby_names` (or disable it) and reload the configuration file on the primary server.

If the primary is isolated from remaining standby servers you should fail over to the best candidate of those other remaining standby servers.

If you need to re-create a standby server while transactions are waiting, make sure that the functions `pg_backup_start()` and `pg_backup_stop()` are run in a session with `synchronous_commit` = `off`, otherwise those requests will wait forever for the standby to appear.

26.2.9. Continuous Archiving in Standby

When continuous WAL archiving is used in a standby, there are two different scenarios: the WAL archive can be shared between the primary and the standby, or the standby can have its own WAL archive. When the standby has its own WAL archive, set `archive_mode` to `always`, and the standby will call the archive command for every WAL segment it receives, whether it's by restoring from the archive or by streaming replication. The shared archive can be handled similarly, but the `archive_command` or `archive_library` must test if the file being archived exists already, and if the existing file has identical contents. This requires more care in the `archive_command` or `archive_library`, as it must be careful to not overwrite an existing file with different contents, but return success if the exactly same file is archived twice. And all that must be done free of race conditions, if two servers attempt to archive the same file at the same time.

If `archive_mode` is set to `on`, the archiver is not enabled during recovery or standby mode. If the standby server is promoted, it will start archiving after the promotion, but will not archive any WAL or timeline history files that it did not generate itself. To get a complete series of WAL files in the archive, you must ensure that all WAL is archived, before it reaches the standby. This is inherently true with file-based log shipping, as the standby can only restore files that are found in the archive, but not if streaming replication is enabled. When a server is not in recovery mode, there is no difference between `on` and `always` modes.

26.3. Failover

If the primary server fails then the standby server should begin failover procedures.

If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be restarted immediately, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server instance should be created.

If the primary server fails and the standby server becomes the new primary, and then the old primary restarts, you must have a mechanism for informing the old primary that it is no longer the primary. This is sometimes known as STONITH (Shoot The Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which will lead to confusion and ultimately data loss.

Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to prevent some cases of inappropriate failover, but the additional complexity might not be worthwhile unless it is set up with sufficient care and rigorous testing.

PostgreSQL does not provide the system software required to identify a failure on the primary and notify the standby database server. Many such tools exist and are well integrated with the operating system facilities required for successful failover, such as IP address migration.

Once failover to the standby occurs, there is only a single server in operation. This is known as a degenerate state. The former standby is now the primary, but the former primary is down and might stay down. To return to normal operation, a standby server must be recreated, either on the former primary system when it comes up, or on a third, possibly new, system. The pg_rewind utility can be used to speed up this process on large clusters. Once complete, the primary and standby can be considered to have switched roles. Some people choose to use a third server to provide backup for the new primary until the new standby server is recreated, though clearly this complicates the system configuration and operational processes.

So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

If you have opted for logical replication slot synchronization (see Section 47.2.3), then before switching to the standby server, it is recommended to check if the logical slots synchronized on the standby server are ready for failover. This can be done by following the steps described in Section 29.3.

To trigger failover of a log-shipping standby server, run `pg_ctl promote` or call `pg_promote()`. If you're setting up reporting servers that are only used to offload read-only queries from the primary, not for high availability purposes, you don't need to promote.

26.4. Hot Standby

Hot standby is the term used to describe the ability to connect to the server and run read-only queries while the server is in archive recovery or standby mode. This is useful both for replication purposes and for restoring a backup to a desired state with great precision. The term hot standby also refers to the ability of the server to move from recovery through to normal operation while users continue running queries and/or keep their connections open.

Running queries in hot standby mode is similar to normal query operation, though there are several usage and administrative differences explained below.

26.4.1. User's Overview

When the hot_standby parameter is set to true on a standby server, it will begin accepting connections once the recovery has brought the system to a consistent state. All such connections are strictly read-only; not even temporary tables may be written.

The data on the standby takes some time to arrive from the primary server so there will be a measurable delay between primary and standby. Running the same query nearly simultaneously on both primary and standby might therefore return differing results. We say that data on the standby is *eventually consistent* with the primary. Once the commit record for a transaction is replayed on the standby, the changes made by that transaction will be visible to any new snapshots taken on the standby. Snapshots may be taken at the start of each query or at the start of each transaction, depending on the current transaction isolation level. For more details, see Section 13.2.

Transactions started during hot standby may issue the following commands:

- Query access: `SELECT`, `COPY TO`
- Cursor commands: `DECLARE`, `FETCH`, `CLOSE`
- Settings: `SHOW`, `SET`, `RESET`
- Transaction management commands:
  - `BEGIN`, `END`, `ABORT`, `START TRANSACTION`
  - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`
  - `EXCEPTION` blocks and other internal subtransactions
- `LOCK TABLE`, though only when explicitly in one of these modes: `ACCESS SHARE`, `ROW SHARE` or `ROW EXCLUSIVE`.
- Plans and resources: `PREPARE`, `EXECUTE`, `DEALLOCATE`, `DISCARD`

- Plugins and extensions: `LOAD`
- `UNLISTEN`

Transactions started during hot standby will never be assigned a transaction ID and cannot write to the system write-ahead log. Therefore, the following actions will produce error messages:

- Data Manipulation Language (DML): `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `COPY FROM`, `TRUNCATE`. Note that there are no allowed actions that result in a trigger being executed during recovery. This restriction applies even to temporary tables, because table rows cannot be read or written without assigning a transaction ID, which is currently not possible in a hot standby environment.
- Data Definition Language (DDL): `CREATE`, `DROP`, `ALTER`, `COMMENT`. This restriction applies even to temporary tables, because carrying out these operations would require updating the system catalog tables.
- `SELECT ... FOR SHARE | UPDATE`, because row locks cannot be taken without updating the underlying data files.
- Rules on `SELECT` statements that generate DML commands.
- `LOCK` that explicitly requests a mode higher than `ROW EXCLUSIVE MODE`.
- `LOCK` in short default form, since it requests `ACCESS EXCLUSIVE MODE`.
- Transaction management commands that explicitly set non-read-only state:
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE`, `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- Two-phase commit commands: `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` because even read-only transactions need to write WAL in the prepare phase (the first phase of two phase commit).
- Sequence updates: `nextval()`, `setval()`
- `LISTEN`, `NOTIFY`

In normal operation, “read-only” transactions are allowed to use `LISTEN` and `NOTIFY`, so hot standby sessions operate under slightly tighter restrictions than ordinary read-only sessions. It is possible that some of these restrictions might be loosened in a future release.

During hot standby, the parameter `transaction_read_only` is always true and may not be changed. But as long as no attempt is made to modify the database, connections during hot standby will act much like any other database connection. If failover or switchover occurs, the database will switch to normal processing mode. Sessions will remain connected while the server changes mode. Once hot standby finishes, it will be possible to initiate read-write transactions (even from a session begun during hot standby).

Users can determine whether hot standby is currently active for their session by issuing `SHOW in_hot_standby`. (In server versions before 14, the `in_hot_standby` parameter did not exist; a workable substitute method for older servers is `SHOW transaction_read_only`.) In addition, a set of functions (Table 9.96) allow users to access information about the standby server. These allow you to write programs that are aware of the current state of the database. These can be used to monitor the progress of recovery, or to allow you to write complex programs that restore the database to particular states.

26.4.2. Handling Query Conflicts

The primary and standby servers are in many ways loosely connected. Actions on the primary will have an effect on the standby. As a result, there is potential for negative interactions or conflicts between them. The easiest conflict to understand is performance: if a huge data load is taking place on the primary then this will generate a similar stream of WAL records on the standby, so standby queries may contend for system resources, such as I/O.

There are also additional types of conflict that can occur with hot standby. These conflicts are *hard conflicts* in the sense that queries might need to be canceled and, in some cases, sessions disconnected to resolve them. The user is provided with several ways to handle these conflicts. Conflict cases include:

- Access Exclusive locks taken on the primary server, including both explicit `LOCK` commands and various DDL actions, conflict with table accesses in standby queries.
- Dropping a tablespace on the primary conflicts with standby queries using that tablespace for temporary work files.
- Dropping a database on the primary conflicts with sessions connected to that database on the standby.
- Application of a vacuum cleanup record from WAL conflicts with standby transactions whose snapshots can still “see” any of the rows to be removed.
- Application of a vacuum cleanup record from WAL conflicts with queries accessing the target page on the standby, whether or not the data to be removed is visible.

On the primary server, these cases simply result in waiting; and the user might choose to cancel either of the conflicting actions. However, on the standby there is no choice: the WAL-logged action already occurred on the primary so the standby must not fail to apply it. Furthermore, allowing WAL application to wait indefinitely may be very undesirable, because the standby's state will become increasingly far behind the primary's. Therefore, a mechanism is provided to forcibly cancel standby queries that conflict with to-be-applied WAL records.

An example of the problem situation is an administrator on the primary server running `DROP TABLE` on a table that is currently being queried on the standby server. Clearly the standby query cannot continue if the `DROP TABLE` is applied on the standby. If this situation occurred on the primary, the `DROP TABLE` would wait until the other query had finished. But when `DROP TABLE` is run on the primary, the primary doesn't have information about what queries are running on the standby, so it will not wait for any such standby queries. The WAL change records come through to the standby while the standby query is still running, causing a conflict. The standby server must either delay application of the WAL records (and everything after them, too) or else cancel the conflicting query so that the `DROP TABLE` can be applied.

When a conflicting query is short, it's typically desirable to allow it to complete by delaying WAL application for a little bit; but a long delay in WAL application is usually not desirable. So the cancel mechanism has parameters, max_standby_archive_delay and max_standby_streaming_delay, that define the maximum allowed delay in WAL application. Conflicting queries will be canceled once it has taken longer than the relevant delay setting to apply any newly-received WAL data. There are two parameters so that different delay values can be specified for the case of reading WAL data from an archive (i.e., initial recovery from a base backup or “catching up” a standby server that has fallen far behind) versus reading WAL data via streaming replication.

In a standby server that exists primarily for high availability, it's best to set the delay parameters relatively short, so that the server cannot fall far behind the primary due to delays caused by standby queries. However, if the standby server is meant for executing long-running queries, then a high or even infinite delay value may be preferable. Keep in mind however that a long-running query could cause other sessions on the standby server to not see recent changes on the primary, if it delays application of WAL records.

Once the delay specified by max_standby_archive_delay or max_standby_streaming_delay has been exceeded, conflicting queries will be canceled. This usually results just in a cancellation error, although in the case of replaying a DROP DATABASE the entire conflicting session will be terminated. Also, if the conflict is over a lock held by an idle transaction, the conflicting session is terminated (this behavior might change in the future).

Canceled queries may be retried immediately (after beginning a new transaction, of course). Since query cancellation depends on the nature of the WAL records being replayed, a query that was canceled may well succeed if it is executed again.

Keep in mind that the delay parameters are compared to the elapsed time since the WAL data was received by the standby server. Thus, the grace period allowed to any one query on the standby is never more than the delay parameter, and could be considerably less if the standby has already fallen behind as a result of waiting for previous queries to complete, or as a result of being unable to keep up with a heavy update load.

The most common reason for conflict between standby queries and WAL replay is “early cleanup”. Normally, PostgreSQL allows cleanup of old row versions when there are no transactions that need to see them to ensure correct visibility of data according to MVCC rules. However, this rule can only be applied for transactions executing on the primary. So it is possible that cleanup on the primary will remove row versions that are still visible to a transaction on the standby.

Row version cleanup isn't the only potential cause of conflicts with standby queries. All index-only scans (including those that run on standbys) must use an MVCC snapshot that “agrees” with the visibility map. Conflicts are therefore required whenever VACUUM sets a page as all-visible in the visibility map containing one or more rows *not* visible to all standby queries. So even running VACUUM against a table with no updated or deleted rows requiring cleanup might lead to conflicts.

Users should be clear that tables that are regularly and heavily updated on the primary server will quickly cause cancellation of longer running queries on the standby. In such cases the setting of a finite value for max_standby_archive_delay or max_standby_streaming_delay can be considered similar to setting statement_timeout.

Remedial possibilities exist if the number of standby-query cancellations is found to be unacceptable. The first option is to set the parameter hot_standby_feedback, which prevents VACUUM from removing recently-dead rows and so cleanup conflicts do not occur. If you do this, you should note that this will delay cleanup of dead rows on the primary, which may result in undesirable table bloat. However, the cleanup situation will be no worse than if the standby queries were running directly on the primary server, and you are still getting the benefit of off-loading execution onto the standby. If standby servers connect and disconnect frequently, you might want to make adjustments to handle the period when hot_standby_feedback feedback is not being provided. For example, consider increasing max_standby_archive_delay so that queries are not rapidly canceled by conflicts in WAL archive files during disconnected periods. You should also consider increasing max_standby_streaming_delay to avoid rapid cancellations by newly-arrived streaming WAL entries after reconnection.

The number of query cancels and the reason for them can be viewed using the pg_stat_database_conflicts system view on the standby server. The pg_stat_database system view also contains summary information.

Users can control whether a log message is produced when WAL replay is waiting longer than deadlock_timeout for conflicts. This is controlled by the log_recovery_conflict_waits parameter.

26.4.3. Administrator's Overview

If hot_standby is on in postgresql.conf (the default value) and there is a standby.signal file present, the server will run in hot standby mode. However, it may take some time for hot standby connections to be allowed, because the server will not accept connections until it has completed sufficient recovery to provide a consistent state against which queries can run. During this period,

clients that attempt to connect will be refused with an error message. To confirm the server has come up, either loop trying to connect from the application, or look for these messages in the server logs:

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read-only connections
```

Consistency information is recorded once per checkpoint on the primary. It is not possible to enable hot standby when reading WAL written during a period when `wal_level` was not set to `replica` or `logical` on the primary. Reaching a consistent state can also be delayed in the presence of both of these conditions:

- A write transaction has more than 64 subtransactions
- Very long-lived write transactions

If you are running file-based log shipping ("warm standby"), you might need to wait until the next WAL file arrives, which could be as long as the `archive_timeout` setting on the primary.

The settings of some parameters determine the size of shared memory for tracking transaction IDs, locks, and prepared transactions. These shared memory structures must be no smaller on a standby than on the primary in order to ensure that the standby does not run out of shared memory during recovery. For example, if the primary had used a prepared transaction but the standby had not allocated any shared memory for tracking prepared transactions, then recovery could not continue until the standby's configuration is changed. The parameters affected are:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

The easiest way to ensure this does not become a problem is to have these parameters set on the standbys to values equal to or greater than on the primary. Therefore, if you want to increase these values, you should do so on all standby servers first, before applying the changes to the primary server. Conversely, if you want to decrease these values, you should do so on the primary server first, before applying the changes to all standby servers. Keep in mind that when a standby is promoted, it becomes the new reference for the required parameter settings for the standbys that follow it. Therefore, to avoid this becoming a problem during a switchover or failover, it is recommended to keep these settings the same on all standby servers.

The WAL tracks changes to these parameters on the primary. If a hot standby processes WAL that indicates that the current value on the primary is higher than its own value, it will log a warning and pause recovery, for example:

```
WARNING:  hot standby is not possible because of insufficient
 parameter settings
DETAIL:  max_connections = 80 is a lower setting than on the
 primary server, where its value was 100.
LOG:  recovery has paused
DETAIL:  If recovery is unpaused, the server will shut down.
```

```
HINT:  You can then restart the server after making the necessary
 configuration changes.
```

At that point, the settings on the standby need to be updated and the instance restarted before recovery can continue. If the standby is not a hot standby, then when it encounters the incompatible parameter change, it will shut down immediately without pausing, since there is then no value in keeping it up.

It is important that the administrator select appropriate settings for max_standby_archive_delay and max_standby_streaming_delay. The best choices vary depending on business priorities. For example if the server is primarily tasked as a High Availability server, then you will want low delay settings, perhaps even zero, though that is a very aggressive setting. If the standby server is tasked as an additional server for decision support queries then it might be acceptable to set the maximum delay values to many hours, or even -1 which means wait forever for queries to complete.

Transaction status "hint bits" written on the primary are not WAL-logged, so data on the standby will likely re-write the hints again on the standby. Thus, the standby server will still perform disk writes even though all users are read-only; no changes occur to the data values themselves. Users will still write large sort temporary files and re-generate relcache info files, so no part of the database is truly read-only during hot standby mode. Note also that writes to remote databases using dblink module, and other operations outside the database using PL functions will still be possible, even though the transaction is read-only locally.

The following types of administration commands are not accepted during recovery mode:

- Data Definition Language (DDL): e.g., `CREATE INDEX`
- Privilege and Ownership: `GRANT`, `REVOKE`, `REASSIGN`
- Maintenance commands: `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Again, note that some of these commands are actually allowed during "read only" mode transactions on the primary.

As a result, you cannot create additional indexes that exist solely on the standby, nor statistics that exist solely on the standby. If these administration commands are needed, they should be executed on the primary, and eventually those changes will propagate to the standby.

`pg_cancel_backend()` and `pg_terminate_backend()` will work on user backends, but not the startup process, which performs recovery. `pg_stat_activity` does not show recovering transactions as active. As a result, `pg_prepared_xacts` is always empty during recovery. If you wish to resolve in-doubt prepared transactions, view `pg_prepared_xacts` on the primary and issue commands to resolve transactions there or resolve them after the end of recovery.

`pg_locks` will show locks held by backends, as normal. `pg_locks` also shows a virtual transaction managed by the startup process that owns all `AccessExclusiveLocks` held by transactions being replayed by recovery. Note that the startup process does not acquire locks to make database changes, and thus locks other than `AccessExclusiveLocks` do not show in `pg_locks` for the Startup process; they are just presumed to exist.

The Nagios plugin check_pgsql will work, because the simple information it checks for exists. The check_postgres monitoring script will also work, though some reported values could give different or confusing results. For example, last vacuum time will not be maintained, since no vacuum occurs on the standby. Vacuums running on the primary do still send their changes to the standby.

WAL file control commands will not work during recovery, e.g., `pg_backup_start`, `pg_switch_wal` etc.

Dynamically loadable modules work, including `pg_stat_statements`.

Advisory locks work normally in recovery, including deadlock detection. Note that advisory locks are never WAL logged, so it is impossible for an advisory lock on either the primary or the standby

to conflict with WAL replay. Nor is it possible to acquire an advisory lock on the primary and have it initiate a similar advisory lock on the standby. Advisory locks relate only to the server on which they are acquired.

Trigger-based replication systems such as Slony, Londiste and Bucardo won't run on the standby at all, though they will run happily on the primary server as long as the changes are not sent to standby servers to be applied. WAL replay is not trigger-based so you cannot relay from the standby to any system that requires additional database writes or relies on the use of triggers.

New OIDs cannot be assigned, though some UUID generators may still work as long as they do not rely on writing new status to the database.

Currently, temporary table creation is not allowed during read-only transactions, so in some cases existing scripts will not run correctly. This restriction might be relaxed in a later release. This is both an SQL standard compliance issue and a technical issue.

`DROP TABLESPACE` can only succeed if the tablespace is empty. Some standby users may be actively using the tablespace via their `temp_tablespaces` parameter. If there are temporary files in the tablespace, all active queries are canceled to ensure that temporary files are removed, so the tablespace can be removed and WAL replay can continue.

Running `DROP DATABASE` or `ALTER DATABASE ... SET TABLESPACE` on the primary will generate a WAL entry that will cause all users connected to that database on the standby to be forcibly disconnected. This action occurs immediately, whatever the setting of `max_standby_streaming_delay`. Note that `ALTER DATABASE ... RENAME` does not disconnect users, which in most cases will go unnoticed, though might in some cases cause a program confusion if it depends in some way upon database name.

In normal (non-recovery) mode, if you issue `DROP USER` or `DROP ROLE` for a role with login capability while that user is still connected then nothing happens to the connected user — they remain connected. The user cannot reconnect however. This behavior applies in recovery also, so a `DROP USER` on the primary does not disconnect that user on the standby.

The cumulative statistics system is active during recovery. All scans, reads, blocks, index usage, etc., will be recorded normally on the standby. However, WAL replay will not increment relation and database specific counters. I.e. replay will not increment `pg_stat_all_tables` columns (like `n_tup_ins`), nor will reads or writes performed by the startup process be tracked in the `pg_statio_` views, nor will associated `pg_stat_database` columns be incremented.

Autovacuum is not active during recovery. It will start normally at the end of recovery.

The checkpointer process and the background writer process are active during recovery. The checkpointer process will perform restartpoints (similar to checkpoints on the primary) and the background writer process will perform normal block cleaning activities. This can include updates of the hint bit information stored on the standby server. The `CHECKPOINT` command is accepted during recovery, though it performs a restartpoint rather than a new checkpoint.

26.4.4. Hot Standby Parameter Reference

Various parameters have been mentioned above in Section 26.4.2 and Section 26.4.3.

On the primary, the wal_level parameter can be used. max_standby_archive_delay and max_standby_streaming_delay have no effect if set on the primary.

On the standby, parameters hot_standby, max_standby_archive_delay and max_standby_streaming_delay can be used.

26.4.5. Caveats

There are several limitations of hot standby. These can and probably will be fixed in future releases:

- Full knowledge of running transactions is required before snapshots can be taken. Transactions that use large numbers of subtransactions (currently greater than 64) will delay the start of read-only connections until the completion of the longest running write transaction. If this situation occurs, explanatory messages will be sent to the server log.

- Valid starting points for standby queries are generated at each checkpoint on the primary. If the standby is shut down while the primary is in a shutdown state, it might not be possible to re-enter hot standby until the primary is started up, so that it generates further starting points in the WAL logs. This situation isn't a problem in the most common situations where it might happen. Generally, if the primary is shut down and not available anymore, that's likely due to a serious failure that requires the standby being converted to operate as the new primary anyway. And in situations where the primary is being intentionally taken down, coordinating to make sure the standby becomes the new primary smoothly is also standard procedure.

- At the end of recovery, `AccessExclusiveLocks` held by prepared transactions will require twice the normal number of lock table entries. If you plan on running either a large number of concurrent prepared transactions that normally take `AccessExclusiveLocks`, or you plan on having one large transaction that takes many `AccessExclusiveLocks`, you are advised to select a larger value of `max_locks_per_transaction`, perhaps as much as twice the value of the parameter on the primary server. You need not consider this at all if your setting of `max_prepared_transactions` is 0.

- The Serializable transaction isolation level is not yet available in hot standby. (See Section 13.2.3 and Section 13.4.1 for details.) An attempt to set a transaction to the serializable isolation level in hot standby mode will generate an error.

Chapter 27. Monitoring Database Activity

A database administrator frequently wonders, “What is the system doing right now?” This chapter discusses how to find that out.

Several tools are available for monitoring database activity and analyzing performance. Most of this chapter is devoted to describing PostgreSQL's cumulative statistics system, but one should not neglect regular Unix monitoring programs such as `ps`, `top`, `iostat`, and `vmstat`. Also, once one has identified a poorly-performing query, further investigation might be needed using PostgreSQL's `EXPLAIN` command. Section 14.1 discusses `EXPLAIN` and other methods for understanding the behavior of an individual query.

27.1. Standard Unix Tools

On most Unix platforms, PostgreSQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00
 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00
 postgres: background writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00
 postgres: checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00
 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00
 postgres: autovacuum launcher
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00
 postgres: joe runbug 127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00
 postgres: tgl regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00
 postgres: tgl regression [local] idle in transaction
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the primary server process. The command arguments shown for it are the same ones used when it was launched. The next four processes are background worker processes automatically launched by the primary process. (The “autovacuum launcher” process will not be present if you have set the system not to run autovacuum.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

```
postgres: user database host activity
```

The user, database, and (client) host items remain the same for the life of the client connection, but the activity indicator changes. The activity can be `idle` (i.e., waiting for a client command), `idle in transaction` (waiting for client inside a `BEGIN` block), or a command type name such as `SELECT`. Also, `waiting` is appended if the server process is presently waiting on a lock held by another session. In the above example we can infer that process 15606 is waiting for process 15610 to complete its transaction and thereby release some lock. (Process 15610 must be the blocker, because there is no other active session. In more complicated cases it would be necessary to look into the `pg_locks` system view to determine who is blocking whom.)

If cluster_name has been configured the cluster name will also be shown in `ps` output:

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00
 postgres: server1: background writer
...
```

If you have turned off update_process_title then the activity indicator is not updated; the process title is set only once when a new process is launched. On some platforms this saves a measurable amount of per-command overhead; on others it's insignificant.

> **Tip**
>
> Solaris requires special handling. You must use `/usr/ucb/ps`, rather than `/bin/ps`. You also must use two `w` flags, not just one. In addition, your original invocation of the `postgres` command must have a shorter `ps` status display than that provided by each server process. If you fail to do all three things, the `ps` output for each server process will be the original `postgres` command line.

27.2. The Cumulative Statistics System

PostgreSQL's *cumulative statistics system* supports collection and reporting of information about server activity. Presently, accesses to tables and indexes in both disk-block and individual-row terms are counted. The total number of rows in each table, and information about vacuum and analyze actions for each table are also counted. If enabled, calls to user-defined functions and the total time spent in each one are counted as well.

PostgreSQL also supports reporting dynamic information about exactly what is going on in the system right now, such as the exact command currently being executed by other server processes, and which other connections exist in the system. This facility is independent of the cumulative statistics system.

27.2.1. Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration parameters that are normally set in `postgresql.conf`. (See Chapter 19 for details about setting configuration parameters.)

The parameter track_activities enables monitoring of the current command being executed by any server process.

The parameter track_counts controls whether cumulative statistics are collected about table and index accesses.

The parameter track_functions enables tracking of usage of user-defined functions.

The parameter track_io_timing enables monitoring of block read, write, extend, and fsync times.

The parameter track_wal_io_timing enables monitoring of WAL write and fsync times.

Normally these parameters are set in `postgresql.conf` so that they apply to all server processes, but it is possible to turn them on or off in individual sessions using the SET command. (To prevent

ordinary users from hiding their activity from the administrator, only superusers are allowed to change these parameters with SET.)

Cumulative statistics are collected in shared memory. Every PostgreSQL process collects statistics locally, then updates the shared data at appropriate intervals. When a server, including a physical replica, shuts down cleanly, a permanent copy of the statistics data is stored in the pg_stat subdirectory, so that statistics can be retained across server restarts. In contrast, when starting from an unclean shutdown (e.g., after an immediate shutdown, a server crash, starting from a base backup, and point-in-time recovery), all statistics counters are reset.

27.2.2. Viewing Statistics

Several predefined views, listed in Table 27.1, are available to show the current state of the system. There are also several other views, listed in Table 27.2, available to show the accumulated statistics. Alternatively, one can build custom views using the underlying cumulative statistics functions, as discussed in Section 27.2.26.

When using the cumulative statistics views and functions to monitor collected data, it is important to realize that the information does not update instantaneously. Each individual server process flushes out accumulated statistics to shared memory just before going idle, but not more frequently than once per PGSTAT_MIN_INTERVAL milliseconds (1 second unless altered while building the server); so a query or transaction still in progress does not affect the displayed totals and the displayed information lags behind actual activity. However, current-query information collected by track_activities is always up-to-date.

Another important point is that when a server process is asked to display any of the accumulated statistics, accessed values are cached until the end of its current transaction in the default configuration. So the statistics will show static information as long as you continue the current transaction. Similarly, information about the current queries of all sessions is collected when any such information is first requested within a transaction, and the same information will be displayed throughout the transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. When analyzing statistics interactively, or with expensive queries, the time delta between accesses to individual statistics can lead to significant skew in the cached statistics. To minimize skew, stats_fetch_consistency can be set to snapshot, at the price of increased memory usage for caching not-needed statistics data. Conversely, if it's known that statistics are only accessed once, caching accessed statistics is unnecessary and can be avoided by setting stats_fetch_consistency to none. You can invoke pg_stat_clear_snapshot() to discard the current transaction's statistics snapshot or cached values (if any). The next use of statistical information will (when in snapshot mode) cause a new snapshot to be built or (when in cache mode) accessed statistics to be cached.

A transaction can also see its own statistics (not yet flushed out to the shared memory statistics) in the views pg_stat_xact_all_tables, pg_stat_xact_sys_tables, pg_stat_xact_user_tables, and pg_stat_xact_user_functions. These numbers do not act as stated above; instead they update continuously throughout the transaction.

Some of the information in the dynamic statistics views shown in Table 27.1 is security restricted. Ordinary users can only see all the information about their own sessions (sessions belonging to a role that they are a member of). In rows about other sessions, many columns will be null. Note, however, that the existence of a session and its general properties such as its sessions user and database are visible to all users. Superusers and roles with privileges of built-in role pg_read_all_stats (see also Section 21.5) can see all the information about all sessions.

Table 27.1. Dynamic Statistics Views

View Name	Description
pg_stat_activity	One row per server process, showing information related to the current activity of that process,

View Name	Description
	such as state and current query. See pg_stat_activity for details.
pg_stat_replication	One row per WAL sender process, showing statistics about replication to that sender's connected standby server. See pg_stat_replication for details.
pg_stat_wal_receiver	Only one row, showing statistics about the WAL receiver from that receiver's connected server. See pg_stat_wal_receiver for details.
pg_stat_recovery_prefetch	Only one row, showing statistics about blocks prefetched during recovery. See pg_stat_recovery_prefetch for details.
pg_stat_subscription	At least one row per subscription, showing information about the subscription workers. See pg_stat_subscription for details.
pg_stat_ssl	One row per connection (regular and replication), showing information about SSL used on this connection. See pg_stat_ssl for details.
pg_stat_gssapi	One row per connection (regular and replication), showing information about GSSAPI authentication and encryption used on this connection. See pg_stat_gssapi for details.
pg_stat_progress_analyze	One row for each backend (including autovacuum worker processes) running ANALYZE, showing current progress. See Section 27.4.1.
pg_stat_progress_create_index	One row for each backend running CREATE INDEX or REINDEX, showing current progress. See Section 27.4.4.
pg_stat_progress_vacuum	One row for each backend (including autovacuum worker processes) running VACUUM, showing current progress. See Section 27.4.5.
pg_stat_progress_cluster	One row for each backend running CLUSTER or VACUUM FULL, showing current progress. See Section 27.4.2.
pg_stat_progress_basebackup	One row for each WAL sender process streaming a base backup, showing current progress. See Section 27.4.6.
pg_stat_progress_copy	One row for each backend running COPY, showing current progress. See Section 27.4.3.

Table 27.2. Collected Statistics Views

View Name	Description
pg_stat_archiver	One row only, showing statistics about the WAL archiver process's activity. See pg_stat_archiver for details.
pg_stat_bgwriter	One row only, showing statistics about the background writer process's activity. See pg_stat_bgwriter for details.

View Name	Description
pg_stat_checkpointer	One row only, showing statistics about the checkpointer process's activity. See pg_stat_checkpointer for details.
pg_stat_database	One row per database, showing database-wide statistics. See pg_stat_database for details.
pg_stat_database_conflicts	One row per database, showing database-wide statistics about query cancels due to conflict with recovery on standby servers. See pg_stat_database_conflicts for details.
pg_stat_io	One row for each combination of backend type, context, and target object containing cluster-wide I/O statistics. See pg_stat_io for details.
pg_stat_replication_slots	One row per replication slot, showing statistics about the replication slot's usage. See pg_stat_replication_slots for details.
pg_stat_slru	One row per SLRU, showing statistics of operations. See pg_stat_slru for details.
pg_stat_subscription_stats	One row per subscription, showing statistics about errors. See pg_stat_subscription_stats for details.
pg_stat_wal	One row only, showing statistics about WAL activity. See pg_stat_wal for details.
pg_stat_all_tables	One row for each table in the current database, showing statistics about accesses to that specific table. See pg_stat_all_tables for details.
pg_stat_sys_tables	Same as pg_stat_all_tables, except that only system tables are shown.
pg_stat_user_tables	Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_xact_all_tables	Similar to pg_stat_all_tables, but counts actions taken so far within the current transaction (which are *not* yet included in pg_stat_all_tables and related views). The columns for numbers of live and dead rows and vacuum and analyze actions are not present in this view.
pg_stat_xact_sys_tables	Same as pg_stat_xact_all_tables, except that only system tables are shown.
pg_stat_xact_user_tables	Same as pg_stat_xact_all_tables, except that only user tables are shown.
pg_stat_all_indexes	One row for each index in the current database, showing statistics about accesses to that specific index. See pg_stat_all_indexes for details.
pg_stat_sys_indexes	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.

View Name	Description
pg_stat_user_indexes	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_stat_user_functions	One row for each tracked function, showing statistics about executions of that function. See pg_stat_user_functions for details.
pg_stat_xact_user_functions	Similar to pg_stat_user_functions, but counts only calls during the current transaction (which are *not* yet included in pg_stat_user_functions).
pg_statio_all_tables	One row for each table in the current database, showing statistics about I/O on that specific table. See pg_statio_all_tables for details.
pg_statio_sys_tables	Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables	Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes	One row for each index in the current database, showing statistics about I/O on that specific index. See pg_statio_all_indexes for details.
pg_statio_sys_indexes	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
pg_statio_user_indexes	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences	One row for each sequence in the current database, showing statistics about I/O on that specific sequence. See pg_statio_all_sequences for details.
pg_statio_sys_sequences	Same as pg_statio_all_sequences, except that only system sequences are shown. (Presently, no system sequences are defined, so this view is always empty.)
pg_statio_user_sequences	Same as pg_statio_all_sequences, except that only user sequences are shown.

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

The pg_stat_io and pg_statio_ set of views are useful for determining the effectiveness of the buffer cache. They can be used to calculate a cache hit ratio. Note that while PostgreSQL's I/O statistics capture most instances in which the kernel was invoked in order to perform I/O, they do not differentiate between data which had to be fetched from disk and that which already resided in the kernel page cache. Users are advised to use the PostgreSQL statistics views in combination with operating system utilities for a more complete picture of their database's I/O performance.

27.2.3. pg_stat_activity

The pg_stat_activity view will have one row per server process, showing information related to the current activity of that process.

Table 27.3. `pg_stat_activity` View

Column Type Description
`datid` `oid` OID of the database this backend is connected to
`datname` `name` Name of the database this backend is connected to
`pid` `integer` Process ID of this backend
`leader_pid` `integer` Process ID of the parallel group leader if this process is a parallel query worker, or process ID of the leader apply worker if this process is a parallel apply worker. `NULL` indicates that this process is a parallel group leader or leader apply worker, or does not participate in any parallel operation.
`usesysid` `oid` OID of the user logged into this backend
`usename` `name` Name of the user logged into this backend
`application_name` `text` Name of the application that is connected to this backend
`client_addr` `inet` IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.
`client_hostname` `text` Host name of the connected client, as reported by a reverse DNS lookup of `client_addr`. This field will only be non-null for IP connections, and only when log_hostname is enabled.
`client_port` `integer` TCP port number that the client is using for communication with this backend, or `-1` if a Unix socket is used. If this field is null, it indicates that this is an internal server process.
`backend_start` `timestamp with time zone` Time when this process was started. For client backends, this is the time the client connected to the server.
`xact_start` `timestamp with time zone` Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the `query_start` column.
`query_start` `timestamp with time zone` Time when the currently active query was started, or if `state` is not `active`, when the last query was started
`state_change` `timestamp with time zone` Time when the `state` was last changed
`wait_event_type` `text` The type of event for which the backend is waiting, if any; otherwise NULL. See Table 27.4.
`wait_event` `text` Wait event name if backend is currently waiting, otherwise NULL. See Table 27.5 through Table 27.13.
`state` `text`

Column Type Description
Current overall state of this backend. Possible values are: • `active`: The backend is executing a query. • `idle`: The backend is waiting for a new client command. • `idle in transaction`: The backend is in a transaction, but is not currently executing a query. • `idle in transaction (aborted)`: This state is similar to `idle in transaction`, except one of the statements in the transaction caused an error. • `fastpath function call`: The backend is executing a fast-path function. • `disabled`: This state is reported if track_activities is disabled in this backend.
`backend_xid` `xid` Top-level transaction identifier of this backend, if any; see Section 66.1.
`backend_xmin` `xid` The current backend's `xmin` horizon.
`query_id` `bigint` Identifier of this backend's most recent query. If `state` is `active` this field shows the identifier of the currently executing query. In all other states, it shows the identifier of last query that was executed. Query identifiers are not computed by default so this field will be null unless compute_query_id parameter is enabled or a third-party module that computes query identifiers is configured.
`query` `text` Text of this backend's most recent query. If `state` is `active` this field shows the currently executing query. In all other states, it shows the last query that was executed. By default the query text is truncated at 1024 bytes; this value can be changed via the parameter track_activity_query_size.
`backend_type` `text` Type of current backend. Possible types are `autovacuum launcher`, `autovacuum worker`, `logical replication launcher`, `logical replication worker`, `parallel worker`, `background writer`, `client backend`, `checkpointer`, `archiver`, `standalone backend`, `startup`, `walreceiver`, `walsender`, `walwriter` and `walsummarizer`. In addition, background workers registered by extensions may have additional types.

> **Note**
>
> The `wait_event` and `state` columns are independent. If a backend is in the `active` state, it may or may not be `waiting` on some event. If the state is `active` and `wait_event` is non-null, it means that a query is being executed, but is being blocked somewhere in the system.

Table 27.4. Wait Event Types

Wait Event Type	Description
`Activity`	The server process is idle. This event type indicates a process waiting for activity in its main processing loop. `wait_event` will identify the specific wait point; see Table 27.5.

Wait Event Type	Description
`BufferPin`	The server process is waiting for exclusive access to a data buffer. Buffer pin waits can be protracted if another process holds an open cursor that last read data from the buffer in question. See Table 27.6.
`Client`	The server process is waiting for activity on a socket connected to a user application. Thus, the server expects something to happen that is independent of its internal processes. `wait_event` will identify the specific wait point; see Table 27.7.
`Extension`	The server process is waiting for some condition defined by an extension module. See Table 27.8.
`InjectionPoint`	The server process is waiting for an injection point to reach an outcome defined in a test. See Section 36.10.13 for more details. This type has no predefined wait points.
`IO`	The server process is waiting for an I/O operation to complete. `wait_event` will identify the specific wait point; see Table 27.9.
`IPC`	The server process is waiting for some interaction with another server process. `wait_event` will identify the specific wait point; see Table 27.10.
`Lock`	The server process is waiting for a heavyweight lock. Heavyweight locks, also known as lock manager locks or simply locks, primarily protect SQL-visible objects such as tables. However, they are also used to ensure mutual exclusion for certain internal operations such as relation extension. `wait_event` will identify the type of lock awaited; see Table 27.11.
`LWLock`	The server process is waiting for a lightweight lock. Most such locks protect a particular data structure in shared memory. `wait_event` will contain a name identifying the purpose of the lightweight lock. (Some locks have specific names; others are part of a group of locks each with a similar purpose.) See Table 27.12.
`Timeout`	The server process is waiting for a timeout to expire. `wait_event` will identify the specific wait point; see Table 27.13.

Table 27.5. Wait Events of Type `Activity`

`Activity` Wait Event	Description
`ArchiverMain`	Waiting in main loop of archiver process.
`AutovacuumMain`	Waiting in main loop of autovacuum launcher process.
`BgwriterHibernate`	Waiting in background writer process, hibernating.

`Activity` Wait Event	Description
`BgwriterMain`	Waiting in main loop of background writer process.
`CheckpointerMain`	Waiting in main loop of checkpointer process.
`LogicalApplyMain`	Waiting in main loop of logical replication apply process.
`LogicalLauncherMain`	Waiting in main loop of logical replication launcher process.
`LogicalParallelApplyMain`	Waiting in main loop of logical replication parallel apply process.
`RecoveryWalStream`	Waiting in main loop of startup process for WAL to arrive, during streaming recovery.
`ReplicationSlotsyncMain`	Waiting in main loop of slot sync worker.
`ReplicationSlotsyncShutdown`	Waiting for slot sync worker to shut down.
`SysloggerMain`	Waiting in main loop of syslogger process.
`WalReceiverMain`	Waiting in main loop of WAL receiver process.
`WalSenderMain`	Waiting in main loop of WAL sender process.
`WalSummarizerWal`	Waiting in WAL summarizer for more WAL to be generated.
`WalWriterMain`	Waiting in main loop of WAL writer process.

Table 27.6. Wait Events of Type `Bufferpin`

`BufferPin` Wait Event	Description
`BufferPin`	Waiting to acquire an exclusive pin on a buffer.

Table 27.7. Wait Events of Type `Client`

`Client` Wait Event	Description
`ClientRead`	Waiting to read data from the client.
`ClientWrite`	Waiting to write data to the client.
`GssOpenServer`	Waiting to read data from the client while establishing a GSSAPI session.
`LibpqwalreceiverConnect`	Waiting in WAL receiver to establish connection to remote server.
`LibpqwalreceiverReceive`	Waiting in WAL receiver to receive data from remote server.
`SslOpenServer`	Waiting for SSL while attempting connection.
`WaitForStandbyConfirmation`	Waiting for WAL to be received and flushed by the physical standby.
`WalSenderWaitForWal`	Waiting for WAL to be flushed in WAL sender process.
`WalSenderWriteData`	Waiting for any activity when processing replies from WAL receiver in WAL sender process.

Table 27.8. Wait Events of Type `Extension`

`Extension` Wait Event	Description
`Extension`	Waiting in an extension.

Table 27.9. Wait Events of Type `Io`

`IO` Wait Event	Description
`BasebackupRead`	Waiting for base backup to read from a file.
`BasebackupSync`	Waiting for data written by a base backup to reach durable storage.
`BasebackupWrite`	Waiting for base backup to write to a file.
`BuffileRead`	Waiting for a read from a buffered file.
`BuffileTruncate`	Waiting for a buffered file to be truncated.
`BuffileWrite`	Waiting for a write to a buffered file.
`ControlFileRead`	Waiting for a read from the `pg_control` file.
`ControlFileSync`	Waiting for the `pg_control` file to reach durable storage.
`ControlFileSyncUpdate`	Waiting for an update to the `pg_control` file to reach durable storage.
`ControlFileWrite`	Waiting for a write to the `pg_control` file.
`ControlFileWriteUpdate`	Waiting for a write to update the `pg_control` file.
`CopyFileRead`	Waiting for a read during a file copy operation.
`CopyFileWrite`	Waiting for a write during a file copy operation.
`DataFileExtend`	Waiting for a relation data file to be extended.
`DataFileFlush`	Waiting for a relation data file to reach durable storage.
`DataFileImmediateSync`	Waiting for an immediate synchronization of a relation data file to durable storage.
`DataFilePrefetch`	Waiting for an asynchronous prefetch from a relation data file.
`DataFileRead`	Waiting for a read from a relation data file.
`DataFileSync`	Waiting for changes to a relation data file to reach durable storage.
`DataFileTruncate`	Waiting for a relation data file to be truncated.
`DataFileWrite`	Waiting for a write to a relation data file.
`DsmAllocate`	Waiting for a dynamic shared memory segment to be allocated.
`DsmFillZeroWrite`	Waiting to fill a dynamic shared memory backing file with zeroes.
`LockFileAddtodatadirRead`	Waiting for a read while adding a line to the data directory lock file.
`LockFileAddtodatadirSync`	Waiting for data to reach durable storage while adding a line to the data directory lock file.
`LockFileAddtodatadirWrite`	Waiting for a write while adding a line to the data directory lock file.
`LockFileCreateRead`	Waiting to read while creating the data directory lock file.
`LockFileCreateSync`	Waiting for data to reach durable storage while creating the data directory lock file.

IO Wait Event	Description
LockFileCreateWrite	Waiting for a write while creating the data directory lock file.
LockFileRecheckdatadirRead	Waiting for a read during recheck of the data directory lock file.
LogicalRewriteCheckpointSync	Waiting for logical rewrite mappings to reach durable storage during a checkpoint.
LogicalRewriteMappingSync	Waiting for mapping data to reach durable storage during a logical rewrite.
LogicalRewriteMappingWrite	Waiting for a write of mapping data during a logical rewrite.
LogicalRewriteSync	Waiting for logical rewrite mappings to reach durable storage.
LogicalRewriteTruncate	Waiting for truncate of mapping data during a logical rewrite.
LogicalRewriteWrite	Waiting for a write of logical rewrite mappings.
RelationMapRead	Waiting for a read of the relation map file.
RelationMapReplace	Waiting for durable replacement of a relation map file.
RelationMapWrite	Waiting for a write to the relation map file.
ReorderBufferRead	Waiting for a read during reorder buffer management.
ReorderBufferWrite	Waiting for a write during reorder buffer management.
ReorderLogicalMappingRead	Waiting for a read of a logical mapping during reorder buffer management.
ReplicationSlotRead	Waiting for a read from a replication slot control file.
ReplicationSlotRestoreSync	Waiting for a replication slot control file to reach durable storage while restoring it to memory.
ReplicationSlotSync	Waiting for a replication slot control file to reach durable storage.
ReplicationSlotWrite	Waiting for a write to a replication slot control file.
SlruFlushSync	Waiting for SLRU data to reach durable storage during a checkpoint or database shutdown.
SlruRead	Waiting for a read of an SLRU page.
SlruSync	Waiting for SLRU data to reach durable storage following a page write.
SlruWrite	Waiting for a write of an SLRU page.
SnapbuildRead	Waiting for a read of a serialized historical catalog snapshot.
SnapbuildSync	Waiting for a serialized historical catalog snapshot to reach durable storage.
SnapbuildWrite	Waiting for a write of a serialized historical catalog snapshot.
TimelineHistoryFileSync	Waiting for a timeline history file received via streaming replication to reach durable storage.

IO Wait Event	Description
TimelineHistoryFileWrite	Waiting for a write of a timeline history file received via streaming replication.
TimelineHistoryRead	Waiting for a read of a timeline history file.
TimelineHistorySync	Waiting for a newly created timeline history file to reach durable storage.
TimelineHistoryWrite	Waiting for a write of a newly created timeline history file.
TwophaseFileRead	Waiting for a read of a two phase state file.
TwophaseFileSync	Waiting for a two phase state file to reach durable storage.
TwophaseFileWrite	Waiting for a write of a two phase state file.
VersionFileSync	Waiting for the version file to reach durable storage while creating a database.
VersionFileWrite	Waiting for the version file to be written while creating a database.
WalsenderTimelineHistoryRead	Waiting for a read from a timeline history file during a walsender timeline command.
WalBootstrapSync	Waiting for WAL to reach durable storage during bootstrapping.
WalBootstrapWrite	Waiting for a write of a WAL page during bootstrapping.
WalCopyRead	Waiting for a read when creating a new WAL segment by copying an existing one.
WalCopySync	Waiting for a new WAL segment created by copying an existing one to reach durable storage.
WalCopyWrite	Waiting for a write when creating a new WAL segment by copying an existing one.
WalInitSync	Waiting for a newly initialized WAL file to reach durable storage.
WalInitWrite	Waiting for a write while initializing a new WAL file.
WalRead	Waiting for a read from a WAL file.
WalSummaryRead	Waiting for a read from a WAL summary file.
WalSummaryWrite	Waiting for a write to a WAL summary file.
WalSync	Waiting for a WAL file to reach durable storage.
WalSyncMethodAssign	Waiting for data to reach durable storage while assigning a new WAL sync method.
WalWrite	Waiting for a write to a WAL file.

Table 27.10. Wait Events of Type Ipc

IPC Wait Event	Description
AppendReady	Waiting for subplan nodes of an Append plan node to be ready.
ArchiveCleanupCommand	Waiting for archive_cleanup_command to complete.
ArchiveCommand	Waiting for archive_command to complete.

IPC Wait Event	Description
BackendTermination	Waiting for the termination of another backend.
BackupWaitWalArchive	Waiting for WAL files required for a backup to be successfully archived.
BgworkerShutdown	Waiting for background worker to shut down.
BgworkerStartup	Waiting for background worker to start up.
BtreePage	Waiting for the page number needed to continue a parallel B-tree scan to become available.
BufferIo	Waiting for buffer I/O to complete.
CheckpointDelayComplete	Waiting for a backend that blocks a checkpoint from completing.
CheckpointDelayStart	Waiting for a backend that blocks a checkpoint from starting.
CheckpointDone	Waiting for a checkpoint to complete.
CheckpointStart	Waiting for a checkpoint to start.
ExecuteGather	Waiting for activity from a child process while executing a Gather plan node.
HashBatchAllocate	Waiting for an elected Parallel Hash participant to allocate a hash table.
HashBatchElect	Waiting to elect a Parallel Hash participant to allocate a hash table.
HashBatchLoad	Waiting for other Parallel Hash participants to finish loading a hash table.
HashBuildAllocate	Waiting for an elected Parallel Hash participant to allocate the initial hash table.
HashBuildElect	Waiting to elect a Parallel Hash participant to allocate the initial hash table.
HashBuildHashInner	Waiting for other Parallel Hash participants to finish hashing the inner relation.
HashBuildHashOuter	Waiting for other Parallel Hash participants to finish partitioning the outer relation.
HashGrowBatchesDecide	Waiting to elect a Parallel Hash participant to decide on future batch growth.
HashGrowBatchesElect	Waiting to elect a Parallel Hash participant to allocate more batches.
HashGrowBatchesFinish	Waiting for an elected Parallel Hash participant to decide on future batch growth.
HashGrowBatchesReallocate	Waiting for an elected Parallel Hash participant to allocate more batches.
HashGrowBatchesRepartition	Waiting for other Parallel Hash participants to finish repartitioning.
HashGrowBucketsElect	Waiting to elect a Parallel Hash participant to allocate more buckets.
HashGrowBucketsReallocate	Waiting for an elected Parallel Hash participant to finish allocating more buckets.
HashGrowBucketsReinsert	Waiting for other Parallel Hash participants to finish inserting tuples into new buckets.

IPC Wait Event	Description
LogicalApplySendData	Waiting for a logical replication leader apply process to send data to a parallel apply process.
LogicalParallelApplyStateChange	Waiting for a logical replication parallel apply process to change state.
LogicalSyncData	Waiting for a logical replication remote server to send data for initial table synchronization.
LogicalSyncStateChange	Waiting for a logical replication remote server to change state.
MessageQueueInternal	Waiting for another process to be attached to a shared message queue.
MessageQueuePutMessage	Waiting to write a protocol message to a shared message queue.
MessageQueueReceive	Waiting to receive bytes from a shared message queue.
MessageQueueSend	Waiting to send bytes to a shared message queue.
MultixactCreation	Waiting for a multixact creation to complete.
ParallelBitmapScan	Waiting for parallel bitmap scan to become initialized.
ParallelCreateIndexScan	Waiting for parallel CREATE INDEX workers to finish heap scan.
ParallelFinish	Waiting for parallel workers to finish computing.
ProcarrayGroupUpdate	Waiting for the group leader to clear the transaction ID at transaction end.
ProcSignalBarrier	Waiting for a barrier event to be processed by all backends.
Promote	Waiting for standby promotion.
RecoveryConflictSnapshot	Waiting for recovery conflict resolution for a vacuum cleanup.
RecoveryConflictTablespace	Waiting for recovery conflict resolution for dropping a tablespace.
RecoveryEndCommand	Waiting for recovery_end_command to complete.
RecoveryPause	Waiting for recovery to be resumed.
ReplicationOriginDrop	Waiting for a replication origin to become inactive so it can be dropped.
ReplicationSlotDrop	Waiting for a replication slot to become inactive so it can be dropped.
RestoreCommand	Waiting for restore_command to complete.
SafeSnapshot	Waiting to obtain a valid snapshot for a READ ONLY DEFERRABLE transaction.
SyncRep	Waiting for confirmation from a remote server during synchronous replication.
WalReceiverExit	Waiting for the WAL receiver to exit.
WalReceiverWaitStart	Waiting for startup process to send initial data for streaming replication.

IPC Wait Event	Description
WalSummaryReady	Waiting for a new WAL summary to be generated.
XactGroupUpdate	Waiting for the group leader to update transaction status at transaction end.

Table 27.11. Wait Events of Type Lock

Lock Wait Event	Description
advisory	Waiting to acquire an advisory user lock.
applytransaction	Waiting to acquire a lock on a remote transaction being applied by a logical replication subscriber.
extend	Waiting to extend a relation.
frozenid	Waiting to update pg_database.datfrozenxid and pg_database.datminmxid.
object	Waiting to acquire a lock on a non-relation database object.
page	Waiting to acquire a lock on a page of a relation.
relation	Waiting to acquire a lock on a relation.
spectoken	Waiting to acquire a speculative insertion lock.
transactionid	Waiting for a transaction to finish.
tuple	Waiting to acquire a lock on a tuple.
userlock	Waiting to acquire a user lock.
virtualxid	Waiting to acquire a virtual transaction ID lock; see Section 66.1.

Table 27.12. Wait Events of Type Lwlock

LWLock Wait Event	Description
AddinShmemInit	Waiting to manage an extension's space allocation in shared memory.
AutoFile	Waiting to update the postgresql.auto.conf file.
Autovacuum	Waiting to read or update the current state of autovacuum workers.
AutovacuumSchedule	Waiting to ensure that a table selected for autovacuum still needs vacuuming.
BackgroundWorker	Waiting to read or update background worker state.
BtreeVacuum	Waiting to read or update vacuum-related information for a B-tree index.
BufferContent	Waiting to access a data page in memory.
BufferMapping	Waiting to associate a data block with a buffer in the buffer pool.
CheckpointerComm	Waiting to manage fsync requests.
CommitTs	Waiting to read or update the last value set for a transaction commit timestamp.

LWLock Wait Event	**Description**
CommitTsBuffer	Waiting for I/O on a commit timestamp SLRU buffer.
CommitTsSLRU	Waiting to access the commit timestamp SLRU cache.
ControlFile	Waiting to read or update the pg_control file or create a new WAL file.
DSMRegistry	Waiting to read or update the dynamic shared memory registry.
DSMRegistryDSA	Waiting to access dynamic shared memory registry's dynamic shared memory allocator.
DSMRegistryHash	Waiting to access dynamic shared memory registry's shared hash table.
DynamicSharedMemoryControl	Waiting to read or update dynamic shared memory allocation information.
InjectionPoint	Waiting to read or update information related to injection points.
LockFastPath	Waiting to read or update a process' fast-path lock information.
LockManager	Waiting to read or update information about “heavyweight” locks.
LogicalRepLauncherDSA	Waiting to access logical replication launcher's dynamic shared memory allocator.
LogicalRepLauncherHash	Waiting to access logical replication launcher's shared hash table.
LogicalRepWorker	Waiting to read or update the state of logical replication workers.
MultiXactGen	Waiting to read or update shared multixact state.
MultiXactMemberBuffer	Waiting for I/O on a multixact member SLRU buffer.
MultiXactMemberSLRU	Waiting to access the multixact member SLRU cache.
MultiXactOffsetBuffer	Waiting for I/O on a multixact offset SLRU buffer.
MultiXactOffsetSLRU	Waiting to access the multixact offset SLRU cache.
MultiXactTruncation	Waiting to read or truncate multixact information.
NotifyBuffer	Waiting for I/O on a NOTIFY message SLRU buffer.
NotifyQueue	Waiting to read or update NOTIFY messages.
NotifyQueueTail	Waiting to update limit on NOTIFY message storage.
NotifySLRU	Waiting to access the NOTIFY message SLRU cache.
OidGen	Waiting to allocate a new OID.
ParallelAppend	Waiting to choose the next subplan during Parallel Append plan execution.

LWLock Wait Event	Description
ParallelHashJoin	Waiting to synchronize workers during Parallel Hash Join plan execution.
ParallelQueryDSA	Waiting for parallel query dynamic shared memory allocation.
ParallelVacuumDSA	Waiting for parallel vacuum dynamic shared memory allocation.
PerSessionDSA	Waiting for parallel query dynamic shared memory allocation.
PerSessionRecordType	Waiting to access a parallel query's information about composite types.
PerSessionRecordTypmod	Waiting to access a parallel query's information about type modifiers that identify anonymous record types.
PerXactPredicateList	Waiting to access the list of predicate locks held by the current serializable transaction during a parallel query.
PgStatsData	Waiting for shared memory stats data access.
PgStatsDSA	Waiting for stats dynamic shared memory allocator access.
PgStatsHash	Waiting for stats shared memory hash table access.
PredicateLockManager	Waiting to access predicate lock information used by serializable transactions.
ProcArray	Waiting to access the shared per-process data structures (typically, to get a snapshot or report a session's transaction ID).
RelationMapping	Waiting to read or update a `pg_filenode.map` file (used to track the filenode assignments of certain system catalogs).
RelCacheInit	Waiting to read or update a `pg_internal.init` relation cache initialization file.
ReplicationOrigin	Waiting to create, drop or use a replication origin.
ReplicationOriginState	Waiting to read or update the progress of one replication origin.
ReplicationSlotAllocation	Waiting to allocate or free a replication slot.
ReplicationSlotControl	Waiting to read or update replication slot state.
ReplicationSlotIO	Waiting for I/O on a replication slot.
SerialBuffer	Waiting for I/O on a serializable transaction conflict SLRU buffer.
SerialControl	Waiting to read or update shared `pg_serial` state.
SerializableFinishedList	Waiting to access the list of finished serializable transactions.
SerializablePredicateList	Waiting to access the list of predicate locks held by serializable transactions.

LWLock Wait Event	Description
SerializableXactHash	Waiting to read or update information about serializable transactions.
SerialSLRU	Waiting to access the serializable transaction conflict SLRU cache.
SharedTidBitmap	Waiting to access a shared TID bitmap during a parallel bitmap index scan.
SharedTupleStore	Waiting to access a shared tuple store during parallel query.
ShmemIndex	Waiting to find or allocate space in shared memory.
SInvalRead	Waiting to retrieve messages from the shared catalog invalidation queue.
SInvalWrite	Waiting to add a message to the shared catalog invalidation queue.
SubtransBuffer	Waiting for I/O on a sub-transaction SLRU buffer.
SubtransSLRU	Waiting to access the sub-transaction SLRU cache.
SyncRep	Waiting to read or update information about the state of synchronous replication.
SyncScan	Waiting to select the starting location of a synchronized table scan.
TablespaceCreate	Waiting to create or drop a tablespace.
TwoPhaseState	Waiting to read or update the state of prepared transactions.
WaitEventCustom	Waiting to read or update custom wait events information.
WALBufMapping	Waiting to replace a page in WAL buffers.
WALInsert	Waiting to insert WAL data into a memory buffer.
WALSummarizer	Waiting to read or update WAL summarization state.
WALWrite	Waiting for WAL buffers to be written to disk.
WrapLimitsVacuum	Waiting to update limits on transaction id and multixact consumption.
XactBuffer	Waiting for I/O on a transaction status SLRU buffer.
XactSLRU	Waiting to access the transaction status SLRU cache.
XactTruncation	Waiting to execute pg_xact_status or update the oldest transaction ID available to it.
XidGen	Waiting to allocate a new transaction ID.

Table 27.13. Wait Events of Type Timeout

Timeout Wait Event	Description
BaseBackupThrottle	Waiting during base backup when throttling activity.

Timeout Wait Event	Description
CheckpointWriteDelay	Waiting between writes while performing a checkpoint.
PgSleep	Waiting due to a call to pg_sleep or a sibling function.
RecoveryApplyDelay	Waiting to apply WAL during recovery because of a delay setting.
RecoveryRetrieveRetryInterval	Waiting during recovery when WAL data is not available from any source (pg_wal, archive or stream).
RegisterSyncRequest	Waiting while sending synchronization requests to the checkpointer, because the request queue is full.
SpinDelay	Waiting while acquiring a contended spinlock.
VacuumDelay	Waiting in a cost-based vacuum delay point.
VacuumTruncate	Waiting to acquire an exclusive lock to truncate off any empty pages at the end of a table vacuumed.
WalSummarizerError	Waiting after a WAL summarizer error.

Here are examples of how wait events can be viewed:

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE
 wait_event is NOT NULL;
 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

```
SELECT a.pid, a.wait_event, w.description
  FROM pg_stat_activity a JOIN
       pg_wait_events w ON (a.wait_event_type = w.type AND
                            a.wait_event = w.name)
  WHERE a.wait_event is NOT NULL and a.state = 'active';
-[ RECORD 1 ]----------------------------------------------------
------------
pid         | 686674
wait_event  | WALInitSync
description | Waiting for a newly initialized WAL file to reach
 durable storage
```

> **Note**
>
> Extensions can add Extension, InjectionPoint. and LWLock events to the lists shown in Table 27.8 and Table 27.12. In some cases, the name of an LWLock assigned by an extension will not be available in all server processes. It might be reported as just "extension" rather than the extension-assigned name.

27.2.4. pg_stat_replication

The pg_stat_replication view will contain one row per WAL sender process, showing statistics about replication to that sender's connected standby server. Only directly connected standbys are listed; no information is available about downstream standby servers.

Table 27.14. pg_stat_replication View

Column Type Description
pid integer Process ID of a WAL sender process
usesysid oid OID of the user logged into this WAL sender process
usename name Name of the user logged into this WAL sender process
application_name text Name of the application that is connected to this WAL sender
client_addr inet IP address of the client connected to this WAL sender. If this field is null, it indicates that the client is connected via a Unix socket on the server machine.
client_hostname text Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when log_hostname is enabled.
client_port integer TCP port number that the client is using for communication with this WAL sender, or -1 if a Unix socket is used
backend_start timestamp with time zone Time when this process was started, i.e., when the client connected to this WAL sender
backend_xmin xid This standby's xmin horizon reported by hot_standby_feedback.
state text Current WAL sender state. Possible values are: • startup: This WAL sender is starting up. • catchup: This WAL sender's connected standby is catching up with the primary. • streaming: This WAL sender is streaming changes after its connected standby server has caught up with the primary. • backup: This WAL sender is sending a backup. • stopping: This WAL sender is stopping.
sent_lsn pg_lsn Last write-ahead log location sent on this connection
write_lsn pg_lsn Last write-ahead log location written to disk by this standby server
flush_lsn pg_lsn Last write-ahead log location flushed to disk by this standby server
replay_lsn pg_lsn Last write-ahead log location replayed into the database on this standby server
write_lag interval

Column Type Description
Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written it (but not yet flushed it or applied it). This can be used to gauge the delay that `synchronous_commit` level `remote_write` incurred while committing if this server was configured as a synchronous standby.
`flush_lag` `interval` Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written and flushed it (but not yet applied it). This can be used to gauge the delay that `synchronous_commit` level `on` incurred while committing if this server was configured as a synchronous standby.
`replay_lag` `interval` Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written, flushed and applied it. This can be used to gauge the delay that `synchronous_commit` level `remote_apply` incurred while committing if this server was configured as a synchronous standby.
`sync_priority` `integer` Priority of this standby server for being chosen as the synchronous standby in a priority-based synchronous replication. This has no effect in a quorum-based synchronous replication.
`sync_state` `text` Synchronous state of this standby server. Possible values are: • `async`: This standby server is asynchronous. • `potential`: This standby server is now asynchronous, but can potentially become synchronous if one of current synchronous ones fails. • `sync`: This standby server is synchronous. • `quorum`: This standby server is considered as a candidate for quorum standbys.
`reply_time` `timestamp with time zone` Send time of last reply message received from standby server

The lag times reported in the `pg_stat_replication` view are measurements of the time taken for recent WAL to be written, flushed and replayed and for the sender to know about it. These times represent the commit delay that was (or would have been) introduced by each synchronous commit level, if the remote server was configured as a synchronous standby. For an asynchronous standby, the `replay_lag` column approximates the delay before recent transactions became visible to queries. If the standby server has entirely caught up with the sending server and there is no more WAL activity, the most recently measured lag times will continue to be displayed for a short time and then show NULL.

Lag times work automatically for physical replication. Logical decoding plugins may optionally emit tracking messages; if they do not, the tracking mechanism will simply display NULL lag.

> **Note**
>
> The reported lag times are not predictions of how long it will take for the standby to catch up with the sending server assuming the current rate of replay. Such a system would show similar times while new WAL is being generated, but would differ when the sender becomes idle. In particular, when the standby has caught up completely, `pg_stat_replication` shows the time taken to write, flush and replay the most recent reported WAL location rather than zero as some users might expect. This is consistent with the goal of measuring synchronous commit and transaction visibility delays for recent write transactions. To reduce confusion for users expecting a different model of lag, the lag columns revert to NULL after a short time on

a fully replayed idle system. Monitoring systems should choose whether to represent this as missing data, zero or continue to display the last known value.

27.2.5. `pg_stat_replication_slots`

The `pg_stat_replication_slots` view will contain one row per logical replication slot, showing statistics about its usage.

Table 27.15. `pg_stat_replication_slots` View

Column Type Description
`slot_name` `text` A unique, cluster-wide identifier for the replication slot
`spill_txns` `bigint` Number of transactions spilled to disk once the memory used by logical decoding to decode changes from WAL has exceeded `logical_decoding_work_mem`. The counter gets incremented for both top-level transactions and subtransactions.
`spill_count` `bigint` Number of times transactions were spilled to disk while decoding changes from WAL for this slot. This counter is incremented each time a transaction is spilled, and the same transaction may be spilled multiple times.
`spill_bytes` `bigint` Amount of decoded transaction data spilled to disk while performing decoding of changes from WAL for this slot. This and other spill counters can be used to gauge the I/O which occurred during logical decoding and allow tuning `logical_decoding_work_mem`.
`stream_txns` `bigint` Number of in-progress transactions streamed to the decoding output plugin after the memory used by logical decoding to decode changes from WAL for this slot has exceeded `logical_decoding_work_mem`. Streaming only works with top-level transactions (subtransactions can't be streamed independently), so the counter is not incremented for subtransactions.
`stream_countbigint` Number of times in-progress transactions were streamed to the decoding output plugin while decoding changes from WAL for this slot. This counter is incremented each time a transaction is streamed, and the same transaction may be streamed multiple times.
`stream_bytesbigint` Amount of transaction data decoded for streaming in-progress transactions to the decoding output plugin while decoding changes from WAL for this slot. This and other streaming counters for this slot can be used to tune `logical_decoding_work_mem`.
`total_txns` `bigint` Number of decoded transactions sent to the decoding output plugin for this slot. This counts top-level transactions only, and is not incremented for subtransactions. Note that this includes the transactions that are streamed and/or spilled.
`total_bytesbigint` Amount of transaction data decoded for sending transactions to the decoding output plugin while decoding changes from WAL for this slot. Note that this includes data that is streamed and/or spilled.
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.6. pg_stat_wal_receiver

The pg_stat_wal_receiver view will contain only one row, showing statistics about the WAL receiver from that receiver's connected server.

Table 27.16. pg_stat_wal_receiver View

Column Type Description
pid integer Process ID of the WAL receiver process
status text Activity status of the WAL receiver process
receive_start_lsn pg_lsn First write-ahead log location used when WAL receiver is started
receive_start_tli integer First timeline number used when WAL receiver is started
written_lsn pg_lsn Last write-ahead log location already received and written to disk, but not flushed. This should not be used for data integrity checks.
flushed_lsn pg_lsn Last write-ahead log location already received and flushed to disk, the initial value of this field being the first log location used when WAL receiver is started
received_tli integer Timeline number of last write-ahead log location received and flushed to disk, the initial value of this field being the timeline number of the first log location used when WAL receiver is started
last_msg_send_time timestamp with time zone Send time of last message received from origin WAL sender
last_msg_receipt_time timestamp with time zone Receipt time of last message received from origin WAL sender
latest_end_lsn pg_lsn Last write-ahead log location reported to origin WAL sender
latest_end_time timestamp with time zone Time of last write-ahead log location reported to origin WAL sender
slot_name text Replication slot name used by this WAL receiver
sender_host text Host of the PostgreSQL instance this WAL receiver is connected to. This can be a host name, an IP address, or a directory path if the connection is via Unix socket. (The path case can be distinguished because it will always be an absolute path, beginning with /.)
sender_port integer Port number of the PostgreSQL instance this WAL receiver is connected to.
conninfo text Connection string used by this WAL receiver, with security-sensitive fields obfuscated.

27.2.7. pg_stat_recovery_prefetch

The pg_stat_recovery_prefetch view will contain only one row. The columns wal_distance, block_distance and io_depth show current values, and the other columns show cumulative counters that can be reset with the pg_stat_reset_shared function.

Table 27.17. `pg_stat_recovery_prefetch` View

Column Type Description
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset
`prefetch` `bigint` Number of blocks prefetched because they were not in the buffer pool
`hit` `bigint` Number of blocks not prefetched because they were already in the buffer pool
`skip_init` `bigint` Number of blocks not prefetched because they would be zero-initialized
`skip_new` `bigint` Number of blocks not prefetched because they didn't exist yet
`skip_fpw` `bigint` Number of blocks not prefetched because a full page image was included in the WAL
`skip_rep` `bigint` Number of blocks not prefetched because they were already recently prefetched
`wal_distance` `int` How many bytes ahead the prefetcher is looking
`block_distance` `int` How many blocks ahead the prefetcher is looking
`io_depth` `int` How many prefetches have been initiated but are not yet known to have completed

27.2.8. `pg_stat_subscription`

Table 27.18. `pg_stat_subscription` View

Column Type Description
`subid` `oid` OID of the subscription
`subname` `name` Name of the subscription
`worker_type` `text` Type of the subscription worker process. Possible types are `apply`, `parallel apply`, and `table synchronization`.
`pid` `integer` Process ID of the subscription worker process
`leader_pid` `integer` Process ID of the leader apply worker if this process is a parallel apply worker; NULL if this process is a leader apply worker or a table synchronization worker
`relid` `oid` OID of the relation that the worker is synchronizing; NULL for the leader apply worker and parallel apply workers
`received_lsn` `pg_lsn` Last write-ahead log location received, the initial value of this field being 0; NULL for parallel apply workers
`last_msg_send_time` `timestamp with time zone`

Column Type Description
Send time of last message received from origin WAL sender; NULL for parallel apply workers
`last_msg_receipt_time` `timestamp with time zone` Receipt time of last message received from origin WAL sender; NULL for parallel apply workers
`latest_end_lsn` `pg_lsn` Last write-ahead log location reported to origin WAL sender; NULL for parallel apply workers
`latest_end_time` `timestamp with time zone` Time of last write-ahead log location reported to origin WAL sender; NULL for parallel apply workers

27.2.9. `pg_stat_subscription_stats`

The `pg_stat_subscription_stats` view will contain one row per subscription.

Table 27.19. `pg_stat_subscription_stats` View

Column Type Description
`subid` `oid` OID of the subscription
`subname` `name` Name of the subscription
`apply_error_count` `bigint` Number of times an error occurred while applying changes
`sync_error_count` `bigint` Number of times an error occurred during the initial table synchronization
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.10. `pg_stat_ssl`

The `pg_stat_ssl` view will contain one row per backend or WAL sender process, showing statistics about SSL usage on this connection. It can be joined to `pg_stat_activity` or `pg_stat_replication` on the `pid` column to get more details about the connection.

Table 27.20. `pg_stat_ssl` View

Column Type Description
`pid` `integer` Process ID of a backend or WAL sender process
`ssl` `boolean` True if SSL is used on this connection
`version` `text` Version of SSL in use, or NULL if SSL is not in use on this connection
`cipher` `text` Name of SSL cipher in use, or NULL if SSL is not in use on this connection
`bits` `integer`

Column Type Description
Number of bits in the encryption algorithm used, or NULL if SSL is not used on this connection
`client_dn` `text` Distinguished Name (DN) field from the client certificate used, or NULL if no client certificate was supplied or if SSL is not in use on this connection. This field is truncated if the DN field is longer than `NAMEDATALEN` (64 characters in a standard build).
`client_serial` `numeric` Serial number of the client certificate, or NULL if no client certificate was supplied or if SSL is not in use on this connection. The combination of certificate serial number and certificate issuer uniquely identifies a certificate (unless the issuer erroneously reuses serial numbers).
`issuer_dn` `text` DN of the issuer of the client certificate, or NULL if no client certificate was supplied or if SSL is not in use on this connection. This field is truncated like `client_dn`.

27.2.11. `pg_stat_gssapi`

The `pg_stat_gssapi` view will contain one row per backend, showing information about GSSAPI usage on this connection. It can be joined to `pg_stat_activity` or `pg_stat_replication` on the `pid` column to get more details about the connection.

Table 27.21. `pg_stat_gssapi` View

Column Type Description
`pid` `integer` Process ID of a backend
`gss_authenticated` `boolean` True if GSSAPI authentication was used for this connection
`principal` `text` Principal used to authenticate this connection, or NULL if GSSAPI was not used to authenticate this connection. This field is truncated if the principal is longer than `NAMEDATALEN` (64 characters in a standard build).
`encrypted` `boolean` True if GSSAPI encryption is in use on this connection
`credentials_delegated` `boolean` True if GSSAPI credentials were delegated on this connection.

27.2.12. `pg_stat_archiver`

The `pg_stat_archiver` view will always have a single row, containing data about the archiver process of the cluster.

Table 27.22. `pg_stat_archiver` View

Column Type Description
`archived_count` `bigint` Number of WAL files that have been successfully archived
`last_archived_wal` `text` Name of the WAL file most recently successfully archived

Column Type Description
`last_archived_time` `timestamp with time zone` Time of the most recent successful archive operation
`failed_count` `bigint` Number of failed attempts for archiving WAL files
`last_failed_wal` `text` Name of the WAL file of the most recent failed archival operation
`last_failed_time` `timestamp with time zone` Time of the most recent failed archival operation
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

Normally, WAL files are archived in order, oldest to newest, but that is not guaranteed, and does not hold under special circumstances like when promoting a standby or after crash recovery. Therefore it is not safe to assume that all files older than `last_archived_wal` have also been successfully archived.

27.2.13. `pg_stat_io`

The `pg_stat_io` view will contain one row for each combination of backend type, target I/O object, and I/O context, showing cluster-wide I/O statistics. Combinations which do not make sense are omitted.

Currently, I/O on relations (e.g. tables, indexes) is tracked. However, relation I/O which bypasses shared buffers (e.g. when moving a table from one tablespace to another) is currently not tracked.

Table 27.23. `pg_stat_io` View

Column Type Description
`backend_type` `text` Type of backend (e.g. background worker, autovacuum worker). See `pg_stat_activity` for more information on `backend_type`s. Some `backend_type`s do not accumulate I/O operation statistics and will not be included in the view.
`object` `text` Target object of an I/O operation. Possible values are: • `relation`: Permanent relations. • `temp relation`: Temporary relations.
`context` `text` The context of an I/O operation. Possible values are: • `normal`: The default or standard `context` for a type of I/O operation. For example, by default, relation data is read into and written out from shared buffers. Thus, reads and writes of relation data to and from shared buffers are tracked in `context` `normal`. • `vacuum`: I/O operations performed outside of shared buffers while vacuuming and analyzing permanent relations. Temporary table vacuums use the same local buffer pool as other temporary table I/O operations and are tracked in `context` `normal`. • `bulkread`: Certain large read I/O operations done outside of shared buffers, for example, a sequential scan of a large table. • `bulkwrite`: Certain large write I/O operations done outside of shared buffers, such as `COPY`.

Column Type Description
`reads` `bigint` Number of read operations, each of the size specified in `op_bytes`.
`read_time` `double precision` Time spent in read operations in milliseconds (if track_io_timing is enabled, otherwise zero)
`writes` `bigint` Number of write operations, each of the size specified in `op_bytes`.
`write_time` `double precision` Time spent in write operations in milliseconds (if track_io_timing is enabled, otherwise zero)
`writebacks` `bigint` Number of units of size `op_bytes` which the process requested the kernel write out to permanent storage.
`writeback_time` `double precision` Time spent in writeback operations in milliseconds (if track_io_timing is enabled, otherwise zero). This includes the time spent queueing write-out requests and, potentially, the time spent to write out the dirty data.
`extends` `bigint` Number of relation extend operations, each of the size specified in `op_bytes`.
`extend_time` `double precision` Time spent in extend operations in milliseconds (if track_io_timing is enabled, otherwise zero)
`op_bytes` `bigint` The number of bytes per unit of I/O read, written, or extended. Relation data reads, writes, and extends are done in `block_size` units, derived from the build-time parameter `BLCKSZ`, which is `8192` by default.
`hits` `bigint` The number of times a desired block was found in a shared buffer.
`evictions` `bigint` Number of times a block has been written out from a shared or local buffer in order to make it available for another use. In `context` `normal`, this counts the number of times a block was evicted from a buffer and replaced with another block. In `context`s `bulkwrite`, `bulkread`, and `vacuum`, this counts the number of times a block was evicted from shared buffers in order to add the shared buffer to a separate, size-limited ring buffer for use in a bulk I/O operation.
`reuses` `bigint` The number of times an existing buffer in a size-limited ring buffer outside of shared buffers was reused as part of an I/O operation in the `bulkread`, `bulkwrite`, or `vacuum` `context`s.
`fsyncs` `bigint` Number of `fsync` calls. These are only tracked in `context` `normal`.
`fsync_time` `double precision` Time spent in fsync operations in milliseconds (if track_io_timing is enabled, otherwise zero)
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset.

Some backend types never perform I/O operations on some I/O objects and/or in some I/O contexts. These rows are omitted from the view. For example, the checkpointer does not checkpoint temporary

tables, so there will be no rows for `backend_type checkpointer` and `object temp relation`.

In addition, some I/O operations will never be performed either by certain backend types or on certain I/O objects and/or in certain I/O contexts. These cells will be NULL. For example, temporary tables are not `fsync`ed, so `fsyncs` will be NULL for `object temp relation`. Also, the background writer does not perform reads, so `reads` will be NULL in rows for `backend_type background writer`.

`pg_stat_io` can be used to inform database tuning. For example:

- A high `evictions` count can indicate that shared buffers should be increased.
- Client backends rely on the checkpointer to ensure data is persisted to permanent storage. Large numbers of `fsyncs` by `client backend`s could indicate a misconfiguration of shared buffers or of the checkpointer. More information on configuring the checkpointer can be found in Section 28.5.
- Normally, client backends should be able to rely on auxiliary processes like the checkpointer and the background writer to write out dirty data as much as possible. Large numbers of writes by client backends could indicate a misconfiguration of shared buffers or of the checkpointer. More information on configuring the checkpointer can be found in Section 28.5.

> **Note**
>
> Columns tracking I/O time will only be non-zero when track_io_timing is enabled. The user should be careful when referencing these columns in combination with their corresponding I/O operations in case `track_io_timing` was not enabled for the entire time since the last stats reset.

27.2.14. `pg_stat_bgwriter`

The `pg_stat_bgwriter` view will always have a single row, containing data about the background writer of the cluster.

Table 27.24. `pg_stat_bgwriter` View

Column Type Description
`buffers_clean` `bigint` Number of buffers written by the background writer
`maxwritten_clean` `bigint` Number of times the background writer stopped a cleaning scan because it had written too many buffers
`buffers_alloc` `bigint` Number of buffers allocated
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.15. `pg_stat_checkpointer`

The `pg_stat_checkpointer` view will always have a single row, containing data about the checkpointer process of the cluster.

Table 27.25. `pg_stat_checkpointer` View

Column Type Description
`num_timed` `bigint` Number of scheduled checkpoints due to timeout. Note that checkpoints may be skipped if the server has been idle since the last one, and this value counts both completed and skipped checkpoints
`num_requested` `bigint` Number of requested checkpoints that have been performed
`restartpoints_timed` `bigint` Number of scheduled restartpoints due to timeout or after a failed attempt to perform it
`restartpoints_req` `bigint` Number of requested restartpoints
`restartpoints_done` `bigint` Number of restartpoints that have been performed
`write_time` `double precision` Total amount of time that has been spent in the portion of processing checkpoints and restartpoints where files are written to disk, in milliseconds
`sync_time` `double precision` Total amount of time that has been spent in the portion of processing checkpoints and restartpoints where files are synchronized to disk, in milliseconds
`buffers_written` `bigint` Number of buffers written during checkpoints and restartpoints
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.16. `pg_stat_wal`

The `pg_stat_wal` view will always have a single row, containing data about WAL activity of the cluster.

Table 27.26. `pg_stat_wal` View

Column Type Description
`wal_records` `bigint` Total number of WAL records generated
`wal_fpi` `bigint` Total number of WAL full page images generated
`wal_bytes` `numeric` Total amount of WAL generated in bytes
`wal_buffers_full` `bigint` Number of times WAL data was written to disk because WAL buffers became full
`wal_write` `bigint` Number of times WAL buffers were written out to disk via `XLogWrite` request. See Section 28.5 for more information about the internal WAL function `XLogWrite`.
`wal_sync` `bigint` Number of times WAL files were synced to disk via `issue_xlog_fsync` request (if fsync is `on` and wal_sync_method is either `fdatasync`, `fsync` or `fsync_writethrough`, otherwise zero). See Section 28.5 for more information about the internal WAL function `issue_xlog_fsync`.

Column Type Description
`wal_write_time` `double precision` Total amount of time spent writing WAL buffers to disk via `XLogWrite` request, in milliseconds (if track_wal_io_timing is enabled, otherwise zero). This includes the sync time when `wal_sync_method` is either `open_datasync` or `open_sync`.
`wal_sync_time` `double precision` Total amount of time spent syncing WAL files to disk via `issue_xlog_fsync` request, in milliseconds (if `track_wal_io_timing` is enabled, `fsync` is `on`, and `wal_sync_method` is either `fdatasync`, `fsync` or `fsync_writethrough`, otherwise zero).
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.17. `pg_stat_database`

The `pg_stat_database` view will contain one row for each database in the cluster, plus one for shared objects, showing database-wide statistics.

Table 27.27. `pg_stat_database` View

Column Type Description
`datid` `oid` OID of this database, or 0 for objects belonging to a shared relation
`datname` `name` Name of this database, or `NULL` for shared objects.
`numbackends` `integer` Number of backends currently connected to this database, or `NULL` for shared objects. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
`xact_commit` `bigint` Number of transactions in this database that have been committed
`xact_rollback` `bigint` Number of transactions in this database that have been rolled back
`blks_read` `bigint` Number of disk blocks read in this database
`blks_hit` `bigint` Number of times disk blocks were found already in the buffer cache, so that a read was not necessary (this only includes hits in the PostgreSQL buffer cache, not the operating system's file system cache)
`tup_returned` `bigint` Number of live rows fetched by sequential scans and index entries returned by index scans in this database
`tup_fetched` `bigint` Number of live rows fetched by index scans in this database
`tup_inserted` `bigint` Number of rows inserted by queries in this database
`tup_updated` `bigint` Number of rows updated by queries in this database
`tup_deleted` `bigint` Number of rows deleted by queries in this database

Column Type Description
`conflicts` `bigint` Number of queries canceled due to conflicts with recovery in this database. (Conflicts occur only on standby servers; see `pg_stat_database_conflicts` for details.)
`temp_files` `bigint` Number of temporary files created by queries in this database. All temporary files are counted, regardless of why the temporary file was created (e.g., sorting or hashing), and regardless of the log_temp_files setting.
`temp_bytes` `bigint` Total amount of data written to temporary files by queries in this database. All temporary files are counted, regardless of why the temporary file was created, and regardless of the log_temp_files setting.
`deadlocks` `bigint` Number of deadlocks detected in this database
`checksum_failures` `bigint` Number of data page checksum failures detected in this database (or on a shared object), or NULL if data checksums are not enabled.
`checksum_last_failure` `timestamp with time zone` Time at which the last data page checksum failure was detected in this database (or on a shared object), or NULL if data checksums are not enabled.
`blk_read_time` `double precision` Time spent reading data file blocks by backends in this database, in milliseconds (if track_io_timing is enabled, otherwise zero)
`blk_write_time` `double precision` Time spent writing data file blocks by backends in this database, in milliseconds (if track_io_timing is enabled, otherwise zero)
`session_time` `double precision` Time spent by database sessions in this database, in milliseconds (note that statistics are only updated when the state of a session changes, so if sessions have been idle for a long time, this idle time won't be included)
`active_time` `double precision` Time spent executing SQL statements in this database, in milliseconds (this corresponds to the states `active` and `fastpath function call` in `pg_stat_activity`)
`idle_in_transaction_time` `double precision` Time spent idling while in a transaction in this database, in milliseconds (this corresponds to the states `idle in transaction` and `idle in transaction (aborted)` in `pg_stat_activity`)
`sessions` `bigint` Total number of sessions established to this database
`sessions_abandoned` `bigint` Number of database sessions to this database that were terminated because connection to the client was lost
`sessions_fatal` `bigint` Number of database sessions to this database that were terminated by fatal errors
`sessions_killed` `bigint` Number of database sessions to this database that were terminated by operator intervention
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.18. pg_stat_database_conflicts

The pg_stat_database_conflicts view will contain one row per database, showing database-wide statistics about query cancels occurring due to conflicts with recovery on standby servers. This view will only contain information on standby servers, since conflicts do not occur on primary servers.

Table 27.28. pg_stat_database_conflicts View

Column Type Description
datid oid OID of a database
datname name Name of this database
confl_tablespace bigint Number of queries in this database that have been canceled due to dropped tablespaces
confl_lock bigint Number of queries in this database that have been canceled due to lock timeouts
confl_snapshot bigint Number of queries in this database that have been canceled due to old snapshots
confl_bufferpin bigint Number of queries in this database that have been canceled due to pinned buffers
confl_deadlock bigint Number of queries in this database that have been canceled due to deadlocks
confl_active_logicalslot bigint Number of uses of logical slots in this database that have been canceled due to old snapshots or too low a wal_level on the primary

27.2.19. pg_stat_all_tables

The pg_stat_all_tables view will contain one row for each table in the current database (including TOAST tables), showing statistics about accesses to that specific table. The pg_stat_user_tables and pg_stat_sys_tables views contain the same information, but filtered to only show user and system tables respectively.

Table 27.29. pg_stat_all_tables View

Column Type Description
relid oid OID of a table
schemaname name Name of the schema that this table is in
relname name Name of this table
seq_scan bigint Number of sequential scans initiated on this table
last_seq_scan timestamp with time zone The time of the last sequential scan on this table, based on the most recent transaction stop time
seq_tup_read bigint

Column Type Description
Number of live rows fetched by sequential scans
`idx_scan` `bigint` Number of index scans initiated on this table
`last_idx_scan` `timestamp with time zone` The time of the last index scan on this table, based on the most recent transaction stop time
`idx_tup_fetch` `bigint` Number of live rows fetched by index scans
`n_tup_ins` `bigint` Total number of rows inserted
`n_tup_upd` `bigint` Total number of rows updated. (This includes row updates counted in `n_tup_hot_upd` and `n_tup_newpage_upd`, and remaining non-HOT updates.)
`n_tup_del` `bigint` Total number of rows deleted
`n_tup_hot_upd` `bigint` Number of rows HOT updated. These are updates where no successor versions are required in indexes.
`n_tup_newpage_upd` `bigint` Number of rows updated where the successor version goes onto a *new* heap page, leaving behind an original version with a `t_ctid` field that points to a different heap page. These are always non-HOT updates.
`n_live_tup` `bigint` Estimated number of live rows
`n_dead_tup` `bigint` Estimated number of dead rows
`n_mod_since_analyze` `bigint` Estimated number of rows modified since this table was last analyzed
`n_ins_since_vacuum` `bigint` Estimated number of rows inserted since this table was last vacuumed
`last_vacuum` `timestamp with time zone` Last time at which this table was manually vacuumed (not counting `VACUUM FULL`)
`last_autovacuum` `timestamp with time zone` Last time at which this table was vacuumed by the autovacuum daemon
`last_analyze` `timestamp with time zone` Last time at which this table was manually analyzed
`last_autoanalyze` `timestamp with time zone` Last time at which this table was analyzed by the autovacuum daemon
`vacuum_count` `bigint` Number of times this table has been manually vacuumed (not counting `VACUUM FULL`)
`autovacuum_count` `bigint` Number of times this table has been vacuumed by the autovacuum daemon
`analyze_count` `bigint` Number of times this table has been manually analyzed
`autoanalyze_count` `bigint` Number of times this table has been analyzed by the autovacuum daemon

27.2.20. pg_stat_all_indexes

The pg_stat_all_indexes view will contain one row for each index in the current database, showing statistics about accesses to that specific index. The pg_stat_user_indexes and pg_stat_sys_indexes views contain the same information, but filtered to only show user and system indexes respectively.

Table 27.30. pg_stat_all_indexes View

Column Type Description
relid oid OID of the table for this index
indexrelid oid OID of this index
schemaname name Name of the schema this index is in
relname name Name of the table for this index
indexrelname name Name of this index
idx_scan bigint Number of index scans initiated on this index
last_idx_scan timestamp with time zone The time of the last scan on this index, based on the most recent transaction stop time
idx_tup_read bigint Number of index entries returned by scans on this index
idx_tup_fetch bigint Number of live table rows fetched by simple index scans using this index

Indexes can be used by simple index scans, “bitmap” index scans, and the optimizer. In a bitmap scan the output of several indexes can be combined via AND or OR rules, so it is difficult to associate individual heap row fetches with specific indexes when a bitmap scan is used. Therefore, a bitmap scan increments the pg_stat_all_indexes.idx_tup_read count(s) for the index(es) it uses, and it increments the pg_stat_all_tables.idx_tup_fetch count for the table, but it does not affect pg_stat_all_indexes.idx_tup_fetch. The optimizer also accesses indexes to check for supplied constants whose values are outside the recorded range of the optimizer statistics because the optimizer statistics might be stale.

> **Note**
>
> The idx_tup_read and idx_tup_fetch counts can be different even without any use of bitmap scans, because idx_tup_read counts index entries retrieved from the index while idx_tup_fetch counts live rows fetched from the table. The latter will be less if any dead or not-yet-committed rows are fetched using the index, or if any heap fetches are avoided by means of an index-only scan.

> **Note**
>
> Queries that use certain SQL constructs to search for rows matching any value out of a list or array of multiple scalar values (see Section 9.25) perform multiple “primitive” index scans

(up to one primitive scan per scalar value) during query execution. Each internal primitive index scan increments `pg_stat_all_indexes.idx_scan`, so it's possible for the count of index scans to significantly exceed the total number of index scan executor node executions.

27.2.21. `pg_statio_all_tables`

The `pg_statio_all_tables` view will contain one row for each table in the current database (including TOAST tables), showing statistics about I/O on that specific table. The `pg_statio_user_tables` and `pg_statio_sys_tables` views contain the same information, but filtered to only show user and system tables respectively.

Table 27.31. `pg_statio_all_tables` View

Column Type Description
`relid` `oid` OID of a table
`schemaname` `name` Name of the schema that this table is in
`relname` `name` Name of this table
`heap_blks_read` `bigint` Number of disk blocks read from this table
`heap_blks_hit` `bigint` Number of buffer hits in this table
`idx_blks_read` `bigint` Number of disk blocks read from all indexes on this table
`idx_blks_hit` `bigint` Number of buffer hits in all indexes on this table
`toast_blks_read` `bigint` Number of disk blocks read from this table's TOAST table (if any)
`toast_blks_hit` `bigint` Number of buffer hits in this table's TOAST table (if any)
`tidx_blks_read` `bigint` Number of disk blocks read from this table's TOAST table indexes (if any)
`tidx_blks_hit` `bigint` Number of buffer hits in this table's TOAST table indexes (if any)

27.2.22. `pg_statio_all_indexes`

The `pg_statio_all_indexes` view will contain one row for each index in the current database, showing statistics about I/O on that specific index. The `pg_statio_user_indexes` and `pg_statio_sys_indexes` views contain the same information, but filtered to only show user and system indexes respectively.

Table 27.32. `pg_statio_all_indexes` View

Column Type Description
`relid` `oid` OID of the table for this index

Column Type Description
`indexrelid` `oid` OID of this index
`schemaname` `name` Name of the schema this index is in
`relname` `name` Name of the table for this index
`indexrelname` `name` Name of this index
`idx_blks_read` `bigint` Number of disk blocks read from this index
`idx_blks_hit` `bigint` Number of buffer hits in this index

27.2.23. `pg_statio_all_sequences`

The `pg_statio_all_sequences` view will contain one row for each sequence in the current database, showing statistics about I/O on that specific sequence.

Table 27.33. `pg_statio_all_sequences` View

Column Type Description
`relid` `oid` OID of a sequence
`schemaname` `name` Name of the schema this sequence is in
`relname` `name` Name of this sequence
`blks_read` `bigint` Number of disk blocks read from this sequence
`blks_hit` `bigint` Number of buffer hits in this sequence

27.2.24. `pg_stat_user_functions`

The `pg_stat_user_functions` view will contain one row for each tracked function, showing statistics about executions of that function. The track_functions parameter controls exactly which functions are tracked.

Table 27.34. `pg_stat_user_functions` View

Column Type Description
`funcid` `oid` OID of a function
`schemaname` `name` Name of the schema this function is in
`funcname` `name` Name of this function

Column Type Description
`calls` `bigint` Number of times this function has been called
`total_time` `double precision` Total time spent in this function and all other functions called by it, in milliseconds
`self_time` `double precision` Total time spent in this function itself, not including other functions called by it, in milliseconds

27.2.25. `pg_stat_slru`

PostgreSQL accesses certain on-disk information via `SLRU` (*simple least-recently-used*) caches. The `pg_stat_slru` view will contain one row for each tracked SLRU cache, showing statistics about access to cached pages.

For each `SLRU` cache that's part of the core server, there is a configuration parameter that controls its size, with the suffix `_buffers` appended.

Table 27.35. `pg_stat_slru` View

Column Type Description
`name` `text` Name of the SLRU
`blks_zeroed` `bigint` Number of blocks zeroed during initializations
`blks_hit` `bigint` Number of times disk blocks were found already in the SLRU, so that a read was not necessary (this only includes hits in the SLRU, not the operating system's file system cache)
`blks_read` `bigint` Number of disk blocks read for this SLRU
`blks_written` `bigint` Number of disk blocks written for this SLRU
`blks_exists` `bigint` Number of blocks checked for existence for this SLRU
`flushes` `bigint` Number of flushes of dirty data for this SLRU
`truncates` `bigint` Number of truncates for this SLRU
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset

27.2.26. Statistics Functions

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions used by the standard views shown above. For details such as the functions' names, consult the definitions of the standard views. (For example, in psql you could issue `\d+ pg_stat_activity`.) The access functions for per-database statistics take a database OID as an argument to identify which database to report on. The per-table and per-index functions take a table or index OID. The functions for per-function statistics take a function OID. Note that only tables, indexes, and functions in the current database can be seen with these functions.

Additional functions related to the cumulative statistics system are listed in Table 27.36.

Table 27.36. Additional Statistics Functions

Function Description
`pg_backend_pid` () → `integer` Returns the process ID of the server process attached to the current session.
`pg_stat_get_activity` (`integer`) → `setof record` Returns a record of information about the backend with the specified process ID, or one record for each active backend in the system if `NULL` is specified. The fields returned are a subset of those in the `pg_stat_activity` view.
`pg_stat_get_snapshot_timestamp` () → `timestamp with time zone` Returns the timestamp of the current statistics snapshot, or NULL if no statistics snapshot has been taken. A snapshot is taken the first time cumulative statistics are accessed in a transaction if `stats_fetch_consistency` is set to `snapshot`
`pg_stat_get_xact_blocks_fetched` (`oid`) → `bigint` Returns the number of block read requests for table or index, in the current transaction. This number minus `pg_stat_get_xact_blocks_hit` gives the number of kernel `read()` calls; the number of actual physical reads is usually lower due to kernel-level buffering.
`pg_stat_get_xact_blocks_hit` (`oid`) → `bigint` Returns the number of block read requests for table or index, in the current transaction, found in cache (not triggering kernel `read()` calls).
`pg_stat_clear_snapshot` () → `void` Discards the current statistics snapshot or cached information.
`pg_stat_reset` () → `void` Resets all statistics counters for the current database to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_shared` ([*`target`* `text` `DEFAULT` `NULL`]) → `void` Resets some cluster-wide statistics counters to zero, depending on the argument. *`target`* can be: • `archiver`: Reset all the counters shown in the `pg_stat_archiver` view. • `bgwriter`: Reset all the counters shown in the `pg_stat_bgwriter` view. • `checkpointer`: Reset all the counters shown in the `pg_stat_checkpointer` view. • `io`: Reset all the counters shown in the `pg_stat_io` view. • `recovery_prefetch`: Reset all the counters shown in the `pg_stat_recovery_prefetch` view. • `slru`: Reset all the counters shown in the `pg_stat_slru` view. • `wal`: Reset all the counters shown in the `pg_stat_wal` view. • `NULL` or not specified: All the counters from the views listed above are reset. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

Function Description
`pg_stat_reset_single_table_counters ( oid ) → void` Resets statistics for a single table or index in the current database or shared across all databases in the cluster to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_single_function_counters ( oid ) → void` Resets statistics for a single function in the current database to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_slru ( [ target text DEFAULT NULL ] ) → void` Resets statistics to zero for a single SLRU cache, or for all SLRUs in the cluster. If *target* is `NULL` or is not specified, all the counters shown in the `pg_stat_slru` view for all SLRU caches are reset. The argument can be one of `commit_timestamp`, `multixact_member`, `multixact_offset`, `notify`, `serializable`, `subtransaction`, or `transaction` to reset the counters for only that entry. If the argument is `other` (or indeed, any unrecognized name), then the counters for all other SLRU caches, such as extension-defined caches, are reset. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_replication_slot ( text ) → void` Resets statistics of the replication slot defined by the argument. If the argument is `NULL`, resets statistics for all the replication slots. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_subscription_stats ( oid ) → void` Resets statistics for a single subscription shown in the `pg_stat_subscription_stats` view to zero. If the argument is `NULL`, reset statistics for all subscriptions. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

Warning

Using `pg_stat_reset()` also resets counters that autovacuum uses to determine when to trigger a vacuum or an analyze. Resetting these counters can cause autovacuum to not perform necessary work, which can cause problems such as table bloat or out-dated table statistics. A database-wide `ANALYZE` is recommended after the statistics have been reset.

`pg_stat_get_activity`, the underlying function of the `pg_stat_activity` view, returns a set of records containing all the available information about each backend process. Sometimes it may be more convenient to obtain just a subset of this information. In such cases, another set of per-backend statistics access functions can be used; these are shown in Table 27.37. These access functions use the session's backend ID number, which is a small integer (>= 0) that is distinct from the backend ID of any concurrent session, although a session's ID can be recycled as soon as it exits. The backend ID is used, among other things, to identify the session's temporary schema if it has one. The function `pg_stat_get_backend_idset` provides a convenient way to list all the active backends' ID numbers for invoking these functions. For example, to show the PIDs and current queries of all backends:

```
SELECT pg_stat_get_backend_pid(backendid) AS pid,
       pg_stat_get_backend_activity(backendid) AS query
```

```
FROM pg_stat_get_backend_idset() AS backendid;
```

Table 27.37. Per-Backend Statistics Functions

Function Description
`pg_stat_get_backend_activity ( integer ) → text` Returns the text of this backend's most recent query.
`pg_stat_get_backend_activity_start ( integer ) → timestamp with time zone` Returns the time when the backend's most recent query was started.
`pg_stat_get_backend_client_addr ( integer ) → inet` Returns the IP address of the client connected to this backend.
`pg_stat_get_backend_client_port ( integer ) → integer` Returns the TCP port number that the client is using for communication.
`pg_stat_get_backend_dbid ( integer ) → oid` Returns the OID of the database this backend is connected to.
`pg_stat_get_backend_idset () → setof integer` Returns the set of currently active backend ID numbers.
`pg_stat_get_backend_pid ( integer ) → integer` Returns the process ID of this backend.
`pg_stat_get_backend_start ( integer ) → timestamp with time zone` Returns the time when this process was started.
`pg_stat_get_backend_subxact ( integer ) → record` Returns a record of information about the subtransactions of the backend with the specified ID. The fields returned are *`subxact_count`*, which is the number of subtransactions in the backend's subtransaction cache, and *`subxact_overflow`*, which indicates whether the backend's subtransaction cache is overflowed or not.
`pg_stat_get_backend_userid ( integer ) → oid` Returns the OID of the user logged into this backend.
`pg_stat_get_backend_wait_event ( integer ) → text` Returns the wait event name if this backend is currently waiting, otherwise NULL. See Table 27.5 through Table 27.13.
`pg_stat_get_backend_wait_event_type ( integer ) → text` Returns the wait event type name if this backend is currently waiting, otherwise NULL. See Table 27.4 for details.
`pg_stat_get_backend_xact_start ( integer ) → timestamp with time zone` Returns the time when the backend's current transaction was started.

27.3. Viewing Locks

Another useful tool for monitoring database activity is the `pg_locks` system table. It allows the database administrator to view information about the outstanding locks in the lock manager. For example, this capability can be used to:

- View all the locks currently outstanding, all the locks on relations in a particular database, all the locks on a particular relation, or all the locks held by a particular PostgreSQL session.

- Determine the relation in the current database with the most ungranted locks (which might be a source of contention among database clients).
- Determine the effect of lock contention on overall database performance, as well as the extent to which contention varies with overall database traffic.

Details of the `pg_locks` view appear in Section 52.12. For more information on locking and managing concurrency with PostgreSQL, refer to Chapter 13.

27.4. Progress Reporting

PostgreSQL has the ability to report the progress of certain commands during command execution. Currently, the only commands which support progress reporting are `ANALYZE`, `CLUSTER`, `CREATE INDEX`, `VACUUM`, `COPY`, and BASE_BACKUP (i.e., replication command that pg_basebackup issues to take a base backup). This may be expanded in the future.

27.4.1. ANALYZE Progress Reporting

Whenever `ANALYZE` is running, the `pg_stat_progress_analyze` view will contain a row for each backend that is currently running that command. The tables below describe the information that will be reported and provide information about how to interpret it.

Table 27.38. `pg_stat_progress_analyze` View

Column Type Description
`pid` `integer` Process ID of backend.
`datid` `oid` OID of the database to which this backend is connected.
`datname` `name` Name of the database to which this backend is connected.
`relid` `oid` OID of the table being analyzed.
`phase` `text` Current processing phase. See Table 27.39.
`sample_blks_total` `bigint` Total number of heap blocks that will be sampled.
`sample_blks_scanned` `bigint` Number of heap blocks scanned.
`ext_stats_total` `bigint` Number of extended statistics.
`ext_stats_computed` `bigint` Number of extended statistics computed. This counter only advances when the phase is `computing extended statistics`.
`child_tables_total` `bigint` Number of child tables.
`child_tables_done` `bigint` Number of child tables scanned. This counter only advances when the phase is `acquiring inherited sample rows`.
`current_child_table_relid` `oid` OID of the child table currently being scanned. This field is only valid when the phase is `acquiring inherited sample rows`.

Table 27.39. ANALYZE Phases

Phase	Description
`initializing`	The command is preparing to begin scanning the heap. This phase is expected to be very brief.
`acquiring sample rows`	The command is currently scanning the table given by `relid` to obtain sample rows.
`acquiring inherited sample rows`	The command is currently scanning child tables to obtain sample rows. Columns `child_tables_total`, `child_tables_done`, and `current_child_table_relid` contain the progress information for this phase.
`computing statistics`	The command is computing statistics from the sample rows obtained during the table scan.
`computing extended statistics`	The command is computing extended statistics from the sample rows obtained during the table scan.
`finalizing analyze`	The command is updating `pg_class`. When this phase is completed, `ANALYZE` will end.

> **Note**
>
> Note that when `ANALYZE` is run on a partitioned table, all of its partitions are also recursively analyzed. In that case, `ANALYZE` progress is reported first for the parent table, whereby its inheritance statistics are collected, followed by that for each partition.

27.4.2. CLUSTER Progress Reporting

Whenever `CLUSTER` or `VACUUM FULL` is running, the `pg_stat_progress_cluster` view will contain a row for each backend that is currently running either command. The tables below describe the information that will be reported and provide information about how to interpret it.

Table 27.40. `pg_stat_progress_cluster` View

Column Type Description
`pid` `integer` Process ID of backend.
`datid` `oid` OID of the database to which this backend is connected.
`datname` `name` Name of the database to which this backend is connected.
`relid` `oid` OID of the table being clustered.
`command` `text` The command that is running. Either `CLUSTER` or `VACUUM FULL`.
`phase` `text` Current processing phase. See Table 27.41.
`cluster_index_relid` `oid` If the table is being scanned using an index, this is the OID of the index being used; otherwise, it is zero.
`heap_tuples_scanned` `bigint`

Column Type Description
Number of heap tuples scanned. This counter only advances when the phase is `seq scanning heap`, `index scanning heap` or `writing new heap`.
`heap_tuples_written` `bigint` Number of heap tuples written. This counter only advances when the phase is `seq scanning heap`, `index scanning heap` or `writing new heap`.
`heap_blks_total` `bigint` Total number of heap blocks in the table. This number is reported as of the beginning of `seq scanning heap`.
`heap_blks_scanned` `bigint` Number of heap blocks scanned. This counter only advances when the phase is `seq scanning heap`.
`index_rebuild_count` `bigint` Number of indexes rebuilt. This counter only advances when the phase is `rebuilding index`.

Table 27.41. CLUSTER and VACUUM FULL Phases

Phase	Description
`initializing`	The command is preparing to begin scanning the heap. This phase is expected to be very brief.
`seq scanning heap`	The command is currently scanning the table using a sequential scan.
`index scanning heap`	`CLUSTER` is currently scanning the table using an index scan.
`sorting tuples`	`CLUSTER` is currently sorting tuples.
`writing new heap`	`CLUSTER` is currently writing the new heap.
`swapping relation files`	The command is currently swapping newly-built files into place.
`rebuilding index`	The command is currently rebuilding an index.
`performing final cleanup`	The command is performing final cleanup. When this phase is completed, `CLUSTER` or `VACUUM FULL` will end.

27.4.3. COPY Progress Reporting

Whenever `COPY` is running, the `pg_stat_progress_copy` view will contain one row for each backend that is currently running a `COPY` command. The table below describes the information that will be reported and provides information about how to interpret it.

Table 27.42. `pg_stat_progress_copy` View

Column Type Description
`pid` `integer` Process ID of backend.
`datid` `oid` OID of the database to which this backend is connected.
`datname` `name` Name of the database to which this backend is connected.
`relid` `oid`

Column Type Description
OID of the table on which the COPY command is executed. It is set to 0 if copying from a SELECT query.
command text The command that is running: COPY FROM, or COPY TO.
type text The I/O type that the data is read from or written to: FILE, PROGRAM, PIPE (for COPY FROM STDIN and COPY TO STDOUT), or CALLBACK (used for example during the initial table synchronization in logical replication).
bytes_processed bigint Number of bytes already processed by COPY command.
bytes_total bigint Size of source file for COPY FROM command in bytes. It is set to 0 if not available.
tuples_processed bigint Number of tuples already processed by COPY command.
tuples_excluded bigint Number of tuples not processed because they were excluded by the WHERE clause of the COPY command.
tuples_skipped bigint Number of tuples skipped because they contain malformed data. This counter only advances when a value other than stop is specified to the ON_ERROR option.

27.4.4. CREATE INDEX Progress Reporting

Whenever CREATE INDEX or REINDEX is running, the pg_stat_progress_create_index view will contain one row for each backend that is currently creating indexes. The tables below describe the information that will be reported and provide information about how to interpret it.

Table 27.43. pg_stat_progress_create_index View

Column Type Description
pid integer Process ID of the backend creating indexes.
datid oid OID of the database to which this backend is connected.
datname name Name of the database to which this backend is connected.
relid oid OID of the table on which the index is being created.
index_relid oid OID of the index being created or reindexed. During a non-concurrent CREATE INDEX, this is 0.
command text Specific command type: CREATE INDEX, CREATE INDEX CONCURRENTLY, REINDEX, or REINDEX CONCURRENTLY.
phase text Current processing phase of index creation. See Table 27.44.
lockers_total bigint Total number of lockers to wait for, when applicable.

Column Type Description
`lockers_done` `bigint` Number of lockers already waited for.
`current_locker_pid` `bigint` Process ID of the locker currently being waited for.
`blocks_total` `bigint` Total number of blocks to be processed in the current phase.
`blocks_done` `bigint` Number of blocks already processed in the current phase.
`tuples_total` `bigint` Total number of tuples to be processed in the current phase.
`tuples_done` `bigint` Number of tuples already processed in the current phase.
`partitions_total` `bigint` Total number of partitions on which the index is to be created or attached, including both direct and indirect partitions. `0` during a `REINDEX`, or when the index is not partitioned.
`partitions_done` `bigint` Number of partitions on which the index has already been created or attached, including both direct and indirect partitions. `0` during a `REINDEX`, or when the index is not partitioned.

Table 27.44. CREATE INDEX Phases

Phase	Description
`initializing`	`CREATE INDEX` or `REINDEX` is preparing to create the index. This phase is expected to be very brief.
`waiting for writers before build`	`CREATE INDEX CONCURRENTLY` or `REINDEX CONCURRENTLY` is waiting for transactions with write locks that can potentially see the table to finish. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.
`building index`	The index is being built by the access method-specific code. In this phase, access methods that support progress reporting fill in their own progress data, and the subphase is indicated in this column. Typically, `blocks_total` and `blocks_done` will contain progress data, as well as potentially `tuples_total` and `tuples_done`.
`waiting for writers before validation`	`CREATE INDEX CONCURRENTLY` or `REINDEX CONCURRENTLY` is waiting for transactions with write locks that can potentially write into the table to finish. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.
`index validation: scanning index`	`CREATE INDEX CONCURRENTLY` is scanning the index searching for tuples that need to be validated. This phase is skipped when not in concurrent mode. Columns `blocks_total` (set to the total size of the index) and `blocks_done` contain the progress information for this phase.
`index validation: sorting tuples`	`CREATE INDEX CONCURRENTLY` is sorting the output of the index scanning phase.

Phase	Description
index validation: scanning table	CREATE INDEX CONCURRENTLY is scanning the table to validate the index tuples collected in the previous two phases. This phase is skipped when not in concurrent mode. Columns blocks_total (set to the total size of the table) and blocks_done contain the progress information for this phase.
waiting for old snapshots	CREATE INDEX CONCURRENTLY or REINDEX CONCURRENTLY is waiting for transactions that can potentially see the table to release their snapshots. This phase is skipped when not in concurrent mode. Columns lockers_total, lockers_done and current_locker_pid contain the progress information for this phase.
waiting for readers before marking dead	REINDEX CONCURRENTLY is waiting for transactions with read locks on the table to finish, before marking the old index dead. This phase is skipped when not in concurrent mode. Columns lockers_total, lockers_done and current_locker_pid contain the progress information for this phase.
waiting for readers before dropping	REINDEX CONCURRENTLY is waiting for transactions with read locks on the table to finish, before dropping the old index. This phase is skipped when not in concurrent mode. Columns lockers_total, lockers_done and current_locker_pid contain the progress information for this phase.

27.4.5. VACUUM Progress Reporting

Whenever VACUUM is running, the pg_stat_progress_vacuum view will contain one row for each backend (including autovacuum worker processes) that is currently vacuuming. The tables below describe the information that will be reported and provide information about how to interpret it. Progress for VACUUM FULL commands is reported via pg_stat_progress_cluster because both VACUUM FULL and CLUSTER rewrite the table, while regular VACUUM only modifies it in place. See Section 27.4.2.

Table 27.45. pg_stat_progress_vacuum View

Column Type Description
pid integer Process ID of backend.
datid oid OID of the database to which this backend is connected.
datname name Name of the database to which this backend is connected.
relid oid OID of the table being vacuumed.
phase text Current processing phase of vacuum. See Table 27.46.
heap_blks_total bigint Total number of heap blocks in the table. This number is reported as of the beginning of the scan; blocks added later will not be (and need not be) visited by this VACUUM.
heap_blks_scanned bigint Number of heap blocks scanned. Because the visibility map is used to optimize scans, some blocks will be skipped without inspection; skipped blocks are included in this total,

Column Type Description
so that this number will eventually become equal to `heap_blks_total` when the vacuum is complete. This counter only advances when the phase is `scanning heap`.
`heap_blks_vacuumed` `bigint` Number of heap blocks vacuumed. Unless the table has no indexes, this counter only advances when the phase is `vacuuming heap`. Blocks that contain no dead tuples are skipped, so the counter may sometimes skip forward in large increments.
`index_vacuum_count` `bigint` Number of completed index vacuum cycles.
`max_dead_tuple_bytes` `bigint` Amount of dead tuple data that we can store before needing to perform an index vacuum cycle, based on maintenance_work_mem.
`dead_tuple_bytes` `bigint` Amount of dead tuple data collected since the last index vacuum cycle.
`num_dead_item_ids` `bigint` Number of dead item identifiers collected since the last index vacuum cycle.
`indexes_total` `bigint` Total number of indexes that will be vacuumed or cleaned up. This number is reported at the beginning of the `vacuuming indexes` phase or the `cleaning up indexes` phase.
`indexes_processed` `bigint` Number of indexes processed. This counter only advances when the phase is `vacuuming indexes` or `cleaning up indexes`.

Table 27.46. VACUUM Phases

Phase	Description
`initializing`	`VACUUM` is preparing to begin scanning the heap. This phase is expected to be very brief.
`scanning heap`	`VACUUM` is currently scanning the heap. It will prune and defragment each page if required, and possibly perform freezing activity. The `heap_blks_scanned` column can be used to monitor the progress of the scan.
`vacuuming indexes`	`VACUUM` is currently vacuuming the indexes. If a table has any indexes, this will happen at least once per vacuum, after the heap has been completely scanned. It may happen multiple times per vacuum if maintenance_work_mem (or, in the case of autovacuum, autovacuum_work_mem if set) is insufficient to store the number of dead tuples found.
`vacuuming heap`	`VACUUM` is currently vacuuming the heap. Vacuuming the heap is distinct from scanning the heap, and occurs after each instance of vacuuming indexes. If `heap_blks_scanned` is less than `heap_blks_total`, the system will return to scanning the heap after this phase is completed; otherwise, it will begin cleaning up indexes after this phase is completed.
`cleaning up indexes`	`VACUUM` is currently cleaning up indexes. This occurs after the heap has been completely scanned and all vacuuming of the indexes and the heap has been completed.
`truncating heap`	`VACUUM` is currently truncating the heap so as to return empty pages at the end of the relation to the operating system. This occurs after cleaning up indexes.

Phase	Description
`performing final cleanup`	`VACUUM` is performing final cleanup. During this phase, `VACUUM` will vacuum the free space map, update statistics in `pg_class`, and report statistics to the cumulative statistics system. When this phase is completed, `VACUUM` will end.

27.4.6. Base Backup Progress Reporting

Whenever an application like pg_basebackup is taking a base backup, the `pg_stat_progress_basebackup` view will contain a row for each WAL sender process that is currently running the `BASE_BACKUP` replication command and streaming the backup. The tables below describe the information that will be reported and provide information about how to interpret it.

Table 27.47. `pg_stat_progress_basebackup` View

Column Type Description
`pid` `integer` Process ID of a WAL sender process.
`phase` `text` Current processing phase. See Table 27.48.
`backup_total` `bigint` Total amount of data that will be streamed. This is estimated and reported as of the beginning of `streaming database files` phase. Note that this is only an approximation since the database may change during `streaming database files` phase and WAL log may be included in the backup later. This is always the same value as `backup_streamed` once the amount of data streamed exceeds the estimated total size. If the estimation is disabled in pg_basebackup (i.e., `--no-estimate-size` option is specified), this is `NULL`.
`backup_streamed` `bigint` Amount of data streamed. This counter only advances when the phase is `streaming database files` or `transferring wal files`.
`tablespaces_total` `bigint` Total number of tablespaces that will be streamed.
`tablespaces_streamed` `bigint` Number of tablespaces streamed. This counter only advances when the phase is `streaming database files`.

Table 27.48. Base Backup Phases

Phase	Description
`initializing`	The WAL sender process is preparing to begin the backup. This phase is expected to be very brief.
`waiting for checkpoint to finish`	The WAL sender process is currently performing `pg_backup_start` to prepare to take a base backup, and waiting for the start-of-backup checkpoint to finish.
`estimating backup size`	The WAL sender process is currently estimating the total amount of database files that will be streamed as a base backup.
`streaming database files`	The WAL sender process is currently streaming database files as a base backup.
`waiting for wal archiving to finish`	The WAL sender process is currently performing `pg_backup_stop` to finish the backup, and waiting for all the WAL files required for the base backup to be successfully archived. If either

Phase	Description
	`--wal-method=none` or `--wal-method=stream` is specified in pg_basebackup, the backup will end when this phase is completed.
`transferring wal files`	The WAL sender process is currently transferring all WAL logs generated during the backup. This phase occurs after `waiting for wal archiving to finish` phase if `--wal-method=fetch` is specified in pg_basebackup. The backup will end when this phase is completed.

27.5. Dynamic Tracing

PostgreSQL provides facilities to support dynamic tracing of the database server. This allows an external utility to be called at specific points in the code and thereby trace execution.

A number of probes or trace points are already inserted into the source code. These probes are intended to be used by database developers and administrators. By default the probes are not compiled into PostgreSQL; the user needs to explicitly tell the configure script to make the probes available.

Currently, the DTrace[1] utility is supported, which, at the time of this writing, is available on Solaris, macOS, FreeBSD, NetBSD, and Oracle Linux. The SystemTap[2] project for Linux provides a DTrace equivalent and can also be used. Supporting other dynamic tracing utilities is theoretically possible by changing the definitions for the macros in `src/include/utils/probes.h`.

27.5.1. Compiling for Dynamic Tracing

By default, probes are not available, so you will need to explicitly tell the configure script to make the probes available in PostgreSQL. To include DTrace support specify `--enable-dtrace` to configure. See Section 17.3.3.6 for further information.

27.5.2. Built-in Probes

A number of standard probes are provided in the source code, as shown in Table 27.49; Table 27.50 shows the types used in the probes. More probes can certainly be added to enhance PostgreSQL's observability.

Table 27.49. Built-in DTrace Probes

Name	Parameters	Description
`transaction-start`	`(LocalTransactionId)`	Probe that fires at the start of a new transaction. arg0 is the transaction ID.
`transaction-commit`	`(LocalTransactionId)`	Probe that fires when a transaction completes successfully. arg0 is the transaction ID.
`transaction-abort`	`(LocalTransactionId)`	Probe that fires when a transaction completes unsuccessfully. arg0 is the transaction ID.
`query-start`	`(const char *)`	Probe that fires when the processing of a query is started. arg0 is the query string.

[1] https://en.wikipedia.org/wiki/DTrace
[2] https://sourceware.org/systemtap/

Name	Parameters	Description
query-done	(const char *)	Probe that fires when the processing of a query is complete. arg0 is the query string.
query-parse-start	(const char *)	Probe that fires when the parsing of a query is started. arg0 is the query string.
query-parse-done	(const char *)	Probe that fires when the parsing of a query is complete. arg0 is the query string.
query-rewrite-start	(const char *)	Probe that fires when the rewriting of a query is started. arg0 is the query string.
query-rewrite-done	(const char *)	Probe that fires when the rewriting of a query is complete. arg0 is the query string.
query-plan-start	()	Probe that fires when the planning of a query is started.
query-plan-done	()	Probe that fires when the planning of a query is complete.
query-execute-start	()	Probe that fires when the execution of a query is started.
query-execute-done	()	Probe that fires when the execution of a query is complete.
statement-status	(const char *)	Probe that fires anytime the server process updates its pg_stat_activity.status. arg0 is the new status string.
checkpoint-start	(int)	Probe that fires when a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
checkpoint-done	(int, int, int, int, int)	Probe that fires when a checkpoint is complete. (The probes listed next fire in sequence during checkpoint processing.) arg0 is the number of buffers written. arg1 is the total number of buffers. arg2, arg3 and arg4 contain the number of WAL files added, removed and recycled respectively.
clog-checkpoint-start	(bool)	Probe that fires when the CLOG portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
clog-checkpoint-done	(bool)	Probe that fires when the CLOG portion of a checkpoint is complete. arg0 has the same meaning as for clog-checkpoint-start.
subtrans-checkpoint-start	(bool)	Probe that fires when the SUBTRANS portion of a checkpoint

Name	Parameters	Description
		is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
`subtrans-checkpoint-done`	`(bool)`	Probe that fires when the SUBTRANS portion of a checkpoint is complete. arg0 has the same meaning as for `subtrans-checkpoint-start`.
`multixact-checkpoint-start`	`(bool)`	Probe that fires when the MultiXact portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
`multixact-checkpoint-done`	`(bool)`	Probe that fires when the MultiXact portion of a checkpoint is complete. arg0 has the same meaning as for `multixact-checkpoint-start`.
`buffer-checkpoint-start`	`(int)`	Probe that fires when the buffer-writing portion of a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
`buffer-sync-start`	`(int, int)`	Probe that fires when we begin to write dirty buffers during checkpoint (after identifying which buffers must be written). arg0 is the total number of buffers. arg1 is the number that are currently dirty and need to be written.
`buffer-sync-written`	`(int)`	Probe that fires after each buffer is written during checkpoint. arg0 is the ID number of the buffer.
`buffer-sync-done`	`(int, int, int)`	Probe that fires when all dirty buffers have been written. arg0 is the total number of buffers. arg1 is the number of buffers actually written by the checkpoint process. arg2 is the number that were expected to be written (arg1 of `buffer-sync-start`); any difference reflects other processes flushing buffers during the checkpoint.
`buffer-checkpoint-sync-start`	`()`	Probe that fires after dirty buffers have been written to the kernel, and before starting to issue fsync requests.
`buffer-checkpoint-done`	`()`	Probe that fires when syncing of buffers to disk is complete.
`twophase-checkpoint-start`	`()`	Probe that fires when the two-phase portion of a checkpoint is started.

Name	Parameters	Description
twophase-checkpoint-done	()	Probe that fires when the two-phase portion of a checkpoint is complete.
buffer-extend-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, unsigned int)	Probe that fires when a relation extension starts. arg0 contains the fork to be extended. arg1, arg2, and arg3 contain the tablespace, database, and relation OIDs identifying the relation. arg4 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer. arg5 is the number of blocks the caller would like to extend by.
buffer-extend-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, unsigned int, BlockNumber)	Probe that fires when a relation extension is complete. arg0 contains the fork to be extended. arg1, arg2, and arg3 contain the tablespace, database, and relation OIDs identifying the relation. arg4 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer. arg5 is the number of blocks the relation was extended by, this can be less than the number in the buffer-extend-start due to resource constraints. arg6 contains the BlockNumber of the first new block.
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when a buffer read is started. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer.
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	Probe that fires when a buffer read is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer. arg6 is true if the buffer was found in the pool, false if not.

Name	Parameters	Description
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires before issuing any write request for a shared buffer. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation.
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a write request is complete. (Note that this just reflects the time to pass the data to the kernel; it's typically not actually been written to disk yet.) The arguments are the same as for buffer-flush-start.
wal-buffer-write-dirty-start	()	Probe that fires when a server process begins to write a dirty WAL buffer because no more WAL buffer space is available. (If this happens often, it implies that wal_buffers is too small.)
wal-buffer-write-dirty-done	()	Probe that fires when a dirty WAL buffer write is complete.
wal-insert	(unsigned char, unsigned char)	Probe that fires when a WAL record is inserted. arg0 is the resource manager (rmid) for the record. arg1 contains the info flags.
wal-switch	()	Probe that fires when a WAL segment switch is requested.
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to read a block from a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer.
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block read is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer. arg6 is the number of bytes actually read, while arg7 is the number requested (if these are different it indicates a short read).

Name	Parameters	Description
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to write a block to a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer.
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block write is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or INVALID_PROC_NUMBER (-1) for a shared buffer. arg6 is the number of bytes actually written, while arg7 is the number requested (if these are different it indicates a short write).
sort-start	(int, bool, int, int, bool, int)	Probe that fires when a sort operation is started. arg0 indicates heap, index or datum sort. arg1 is true for unique-value enforcement. arg2 is the number of key columns. arg3 is the number of kilobytes of work memory allowed. arg4 is true if random access to the sort result is required. arg5 indicates serial when 0, parallel worker when 1, or parallel leader when 2.
sort-done	(bool, long)	Probe that fires when a sort is complete. arg0 is true for external sort, false for internal sort. arg1 is the number of disk blocks used for an external sort, or kilobytes of memory used for an internal sort.
lwlock-acquire	(char *, LWLockMode)	Probe that fires when an LWLock has been acquired. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
lwlock-release	(char *)	Probe that fires when an LWLock has been released (but note that any released waiters have not yet been awakened). arg0 is the LWLock's tranche.
lwlock-wait-start	(char *, LWLockMode)	Probe that fires when an LWLock was not immediately available and a

Name	Parameters	Description
		server process has begun to wait for the lock to become available. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
`lwlock-wait-done`	`(char *, LWLockMode)`	Probe that fires when a server process has been released from its wait for an LWLock (it does not actually have the lock yet). arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
`lwlock-condacquire`	`(char *, LWLockMode)`	Probe that fires when an LWLock was successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
`lwlock-condacquire-fail`	`(char *, LWLockMode)`	Probe that fires when an LWLock was not successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
`lock-wait-start`	`(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)`	Probe that fires when a request for a heavyweight lock (lmgr lock) has begun to wait because the lock is not available. arg0 through arg3 are the tag fields identifying the object being locked. arg4 indicates the type of object being locked. arg5 indicates the lock type being requested.
`lock-wait-done`	`(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)`	Probe that fires when a request for a heavyweight lock (lmgr lock) has finished waiting (i.e., has acquired the lock). The arguments are the same as for `lock-wait-start`.
`deadlock-found`	`()`	Probe that fires when a deadlock is found by the deadlock detector.

Table 27.50. Defined Types Used in Probe Parameters

Type	Definition
`LocalTransactionId`	`unsigned int`
`LWLockMode`	`int`
`LOCKMODE`	`int`
`BlockNumber`	`unsigned int`
`Oid`	`unsigned int`
`ForkNumber`	`int`
`bool`	`unsigned char`

27.5.3. Using Probes

The example below shows a DTrace script for analyzing transaction counts in the system, as an alternative to snapshotting `pg_stat_database` before and after a performance test:

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

When executed, the example D script gives output such as:

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **Note**
>
> SystemTap uses a different notation for trace scripts than DTrace does, even though the underlying trace points are compatible. One point worth noting is that at this writing, SystemTap scripts must reference probe names using double underscores in place of hyphens. This is expected to be fixed in future SystemTap releases.

You should remember that DTrace scripts need to be carefully written and debugged, otherwise the trace information collected might be meaningless. In most cases where problems are found it is the instrumentation that is at fault, not the underlying system. When discussing information found using dynamic tracing, be sure to enclose the script used to allow that too to be checked and discussed.

27.5.4. Defining New Probes

New probes can be defined within the code wherever the developer desires, though this will require a recompilation. Below are the steps for inserting new probes:

1. Decide on probe names and data to be made available through the probes
2. Add the probe definitions to `src/backend/utils/probes.d`

3. Include `pg_trace.h` if it is not already present in the module(s) containing the probe points, and insert `TRACE_POSTGRESQL` probe macros at the desired locations in the source code

4. Recompile and verify that the new probes are available

Example: Here is an example of how you would add a probe to trace all new transactions by transaction ID.

1. Decide that the probe will be named `transaction-start` and requires a parameter of type `LocalTransactionId`

2. Add the probe definition to `src/backend/utils/probes.d`:

   ```
   probe transaction__start(LocalTransactionId);
   ```

   Note the use of the double underline in the probe name. In a DTrace script using the probe, the double underline needs to be replaced with a hyphen, so `transaction-start` is the name to document for users.

3. At compile time, `transaction__start` is converted to a macro called `TRACE_POSTGRESQL_TRANSACTION_START` (notice the underscores are single here), which is available by including `pg_trace.h`. Add the macro call to the appropriate location in the source code. In this case, it looks like the following:

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```

4. After recompiling and running the new binary, check that your newly added probe is available by executing the following DTrace command. You should see similar output:

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand
    transaction-start
   18755 postgresql49877     postgres     StartTransactionCommand
    transaction-start
   18805 postgresql49876     postgres     StartTransactionCommand
    transaction-start
   18855 postgresql49875     postgres     StartTransactionCommand
    transaction-start
   18986 postgresql49873     postgres     StartTransactionCommand
    transaction-start
   ```

There are a few things to be careful about when adding trace macros to the C code:

- You should take care that the data types specified for a probe's parameters match the data types of the variables used in the macro. Otherwise, you will get compilation errors.

- On most platforms, if PostgreSQL is built with `--enable-dtrace`, the arguments to a trace macro will be evaluated whenever control passes through the macro, *even if no tracing is being done*. This is usually not worth worrying about if you are just reporting the values of a few local variables. But beware of putting expensive function calls into the arguments. If you need to do that, consider protecting the macro with a check to see if the trace is actually enabled:

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  Each trace macro has a corresponding `ENABLED` macro.

27.6. Monitoring Disk Usage

This section discusses how to monitor the disk usage of a PostgreSQL database system.

27.6.1. Determining Disk Usage

Each table has a primary heap disk file where most of the data is stored. If the table has any columns with potentially-wide values, there also might be a TOAST file associated with the table, which is used to store values too wide to fit comfortably in the main table (see Section 65.2). There will be one valid index on the TOAST table, if present. There also might be indexes associated with the base table. Each table and index is stored in a separate disk file — possibly more than one file, if the file would exceed one gigabyte. Naming conventions for these files are described in Section 65.1.

You can monitor disk space in three ways: using the SQL functions listed in Table 9.100, using the oid2name module, or using manual inspection of the system catalogs. The SQL functions are the easiest to use and are generally recommended. The remainder of this section shows how to do it by inspection of the system catalogs.

Using psql on a recently vacuumed or analyzed database, you can issue queries to see the disk usage of any table:

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE
 relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Each page is typically 8 kilobytes. (Remember, `relpages` is only updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`.) The file path name is of interest if you want to examine the table's disk file directly.

To show the space used by TOAST tables, use a query like the following:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
```

```
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

      relname      | relpages
-------------------+----------
 customer_id_index |       26
```

It is easy to find your largest tables and indexes using this information:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

27.6.2. Disk Full Failure

The most important disk monitoring task of a database administrator is to make sure the disk doesn't become full. A filled data disk will not result in data corruption, but it might prevent useful activity from occurring. If the disk holding the WAL files grows full, database server panic and consequent shutdown might occur.

If you cannot free up additional space on the disk by deleting other things, you can move some of the database files to other file systems by making use of tablespaces. See Section 22.6 for more information about that.

> **Tip**
>
> Some file systems perform badly when they are almost full, so do not wait until the disk is completely full to take action.

If your system supports per-user disk quotas, then the database will naturally be subject to whatever quota is placed on the user the server runs as. Exceeding the quota will have the same bad effects as running out of disk space entirely.

Chapter 28. Reliability and the Write-Ahead Log

This chapter explains how to control the reliability of PostgreSQL, including details about the Write-Ahead Log.

28.1. Reliability

Reliability is an important property of any serious database system, and PostgreSQL does everything possible to guarantee reliable operation. One aspect of reliable operation is that all data recorded by a committed transaction should be stored in a nonvolatile area that is safe from power loss, operating system failure, and hardware failure (except failure of the nonvolatile area itself, of course). Successfully writing the data to the computer's permanent storage (disk drive or equivalent) ordinarily meets this requirement. In fact, even if a computer is fatally damaged, if the disk drives survive they can be moved to another computer with similar hardware and all committed transactions will remain intact.

While forcing data to the disk platters periodically might seem like a simple operation, it is not. Because disk drives are dramatically slower than main memory and CPUs, several layers of caching exist between the computer's main memory and the disk platters. First, there is the operating system's buffer cache, which caches frequently requested disk blocks and combines disk writes. Fortunately, all operating systems give applications a way to force writes from the buffer cache to disk, and PostgreSQL uses those features. (See the wal_sync_method parameter to adjust how this is done.)

Next, there might be a cache in the disk drive controller; this is particularly common on RAID controller cards. Some of these caches are *write-through*, meaning writes are sent to the drive as soon as they arrive. Others are *write-back*, meaning data is sent to the drive at some later time. Such caches can be a reliability hazard because the memory in the disk controller cache is volatile, and will lose its contents in a power failure. Better controller cards have *battery-backup units* (BBUs), meaning the card has a battery that maintains power to the cache in case of system power loss. After power is restored the data will be written to the disk drives.

And finally, most disk drives have caches. Some are write-through while some are write-back, and the same concerns about data loss exist for write-back drive caches as for disk controller caches. Consumer-grade IDE and SATA drives are particularly likely to have write-back caches that will not survive a power failure. Many solid-state drives (SSD) also have volatile write-back caches.

These caches can typically be disabled; however, the method for doing this varies by operating system and drive type:

- On Linux, IDE and SATA drives can be queried using `hdparm -I`; write caching is enabled if there is a `*` next to `Write cache`. `hdparm -W 0` can be used to turn off write caching. SCSI drives can be queried using sdparm[1]. Use `sdparm --get=WCE` to check whether the write cache is enabled and `sdparm --clear=WCE` to disable it.

- On FreeBSD, IDE drives can be queried using `camcontrol identify` and write caching turned off using `hw.ata.wc=0` in `/boot/loader.conf`; SCSI drives can be queried using `camcontrol identify`, and the write cache both queried and changed using `sdparm` when available.

- On Solaris, the disk write cache is controlled by `format -e`. (The Solaris ZFS file system is safe with disk write-cache enabled because it issues its own disk cache flush commands.)

- On Windows, if `wal_sync_method` is `open_datasync` (the default), write caching can be disabled by unchecking `My Computer\Open\`*`disk drive`*`\Properties\Hard-`

[1] http://sg.danny.cz/sg/sdparm.html

ware\Properties\Policies\Enable write caching on the disk. Alternatively, set wal_sync_method to fdatasync (NTFS only) or fsync, which prevent write caching.

- On macOS, write caching can be prevented by setting wal_sync_method to fsync_writethrough.

Recent SATA drives (those following ATAPI-6 or later) offer a drive cache flush command (FLUSH CACHE EXT), while SCSI drives have long supported a similar command SYNCHRONIZE CACHE. These commands are not directly accessible to PostgreSQL, but some file systems (e.g., ZFS, ext4) can use them to flush data to the platters on write-back-enabled drives. Unfortunately, such file systems behave suboptimally when combined with battery-backup unit (BBU) disk controllers. In such setups, the synchronize command forces all data from the controller cache to the disks, eliminating much of the benefit of the BBU. You can run the pg_test_fsync program to see if you are affected. If you are affected, the performance benefits of the BBU can be regained by turning off write barriers in the file system or reconfiguring the disk controller, if that is an option. If write barriers are turned off, make sure the battery remains functional; a faulty battery can potentially lead to data loss. Hopefully file system and disk controller designers will eventually address this suboptimal behavior.

When the operating system sends a write request to the storage hardware, there is little it can do to make sure the data has arrived at a truly non-volatile storage area. Rather, it is the administrator's responsibility to make certain that all storage components ensure integrity for both data and file-system metadata. Avoid disk controllers that have non-battery-backed write caches. At the drive level, disable write-back caching if the drive cannot guarantee the data will be written before shutdown. If you use SSDs, be aware that many of these do not honor cache flush commands by default. You can test for reliable I/O subsystem behavior using diskchecker.pl[2].

Another risk of data loss is posed by the disk platter write operations themselves. Disk platters are divided into sectors, commonly 512 bytes each. Every physical read or write operation processes a whole sector. When a write request arrives at the drive, it might be for some multiple of 512 bytes (PostgreSQL typically writes 8192 bytes, or 16 sectors, at a time), and the process of writing could fail due to power loss at any time, meaning some of the 512-byte sectors were written while others were not. To guard against such failures, PostgreSQL periodically writes full page images to permanent WAL storage *before* modifying the actual page on disk. By doing this, during crash recovery PostgreSQL can restore partially-written pages from WAL. If you have file-system software that prevents partial page writes (e.g., ZFS), you can turn off this page imaging by turning off the full_page_writes parameter. Battery-Backed Unit (BBU) disk controllers do not prevent partial page writes unless they guarantee that data is written to the BBU as full (8kB) pages.

PostgreSQL also protects against some kinds of data corruption on storage devices that may occur because of hardware errors or media failure over time, such as reading/writing garbage data.

- Each individual record in a WAL file is protected by a CRC-32C (32-bit) check that allows us to tell if record contents are correct. The CRC value is set when we write each WAL record and checked during crash recovery, archive recovery and replication.

- Data pages are not currently checksummed by default, though full page images recorded in WAL records will be protected; see initdb for details about enabling data checksums.

- Internal data structures such as pg_xact, pg_subtrans, pg_multixact, pg_serial, pg_notify, pg_stat, pg_snapshots are not directly checksummed, nor are pages protected by full page writes. However, where such data structures are persistent, WAL records are written that allow recent changes to be accurately rebuilt at crash recovery and those WAL records are protected as discussed above.

- Individual state files in pg_twophase are protected by CRC-32C.

- Temporary data files used in larger SQL queries for sorts, materializations and intermediate results are not currently checksummed, nor will WAL records be written for changes to those files.

[2] https://brad.livejournal.com/2116715.html

PostgreSQL does not protect against correctable memory errors and it is assumed you will operate using RAM that uses industry standard Error Correcting Codes (ECC) or better protection.

28.2. Data Checksums

By default, data pages are not protected by checksums, but this can optionally be enabled for a cluster. When enabled, each data page includes a checksum that is updated when the page is written and verified each time the page is read. Only data pages are protected by checksums; internal data structures and temporary files are not.

Checksums are normally enabled when the cluster is initialized using initdb. They can also be enabled or disabled at a later time as an offline operation. Data checksums are enabled or disabled at the full cluster level, and cannot be specified individually for databases or tables.

The current state of checksums in the cluster can be verified by viewing the value of the read-only configuration variable data_checksums by issuing the command `SHOW data_checksums`.

When attempting to recover from page corruptions, it may be necessary to bypass the checksum protection. To do this, temporarily set the configuration parameter ignore_checksum_failure.

28.2.1. Off-line Enabling of Checksums

The pg_checksums application can be used to enable or disable data checksums, as well as verify checksums, on an offline cluster.

28.3. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is a standard method for ensuring data integrity. A detailed description can be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after WAL records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the WAL records. (This is roll-forward recovery, also known as REDO.)

> **Tip**
>
> Because WAL restores database file contents after a crash, journaled file systems are not necessary for reliable storage of the data files or WAL files. In fact, journaling overhead can reduce performance, especially if journaling causes file system *data* to be flushed to disk. Fortunately, data flushing during journaling can often be disabled with a file system mount option, e.g., `data=writeback` on a Linux ext3 file system. Journaled file systems do improve boot speed after a crash.

Using WAL results in a significantly reduced number of disk writes, because only the WAL file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction. The WAL file is written sequentially, and so the cost of syncing the WAL is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store. Furthermore, when the server is processing many small concurrent transactions, one `fsync` of the WAL file may suffice to commit many transactions.

WAL also makes it possible to support on-line backup and point-in-time recovery, as described in Section 25.3. By archiving the WAL data we can support reverting to any time instant covered by the available WAL data: we simply install a prior physical backup of the database, and replay the WAL just as far as the desired time. What's more, the physical backup doesn't have to be an instantaneous

snapshot of the database state — if it is made over some period of time, then replaying the WAL for that period will fix any internal inconsistencies.

28.4. Asynchronous Commit

Asynchronous commit is an option that allows transactions to complete more quickly, at the cost that the most recent transactions may be lost if the database should crash. In many applications this is an acceptable trade-off.

As described in the previous section, transaction commit is normally *synchronous*: the server waits for the transaction's WAL records to be flushed to permanent storage before returning a success indication to the client. The client is therefore guaranteed that a transaction reported to be committed will be preserved, even in the event of a server crash immediately after. However, for short transactions this delay is a major component of the total transaction time. Selecting asynchronous commit mode means that the server returns success as soon as the transaction is logically completed, before the WAL records it generated have actually made their way to disk. This can provide a significant boost in throughput for small transactions.

Asynchronous commit introduces the risk of data loss. There is a short time window between the report of transaction completion to the client and the time that the transaction is truly committed (that is, it is guaranteed not to be lost if the server crashes). Thus asynchronous commit should not be used if the client will take external actions relying on the assumption that the transaction will be remembered. As an example, a bank would certainly not use asynchronous commit for a transaction recording an ATM's dispensing of cash. But in many scenarios, such as event logging, there is no need for a strong guarantee of this kind.

The risk that is taken by using asynchronous commit is of data loss, not data corruption. If the database should crash, it will recover by replaying WAL up to the last record that was flushed. The database will therefore be restored to a self-consistent state, but any transactions that were not yet flushed to disk will not be reflected in that state. The net effect is therefore loss of the last few transactions. Because the transactions are replayed in commit order, no inconsistency can be introduced — for example, if transaction B made changes relying on the effects of a previous transaction A, it is not possible for A's effects to be lost while B's effects are preserved.

The user can select the commit mode of each transaction, so that it is possible to have both synchronous and asynchronous commit transactions running concurrently. This allows flexible trade-offs between performance and certainty of transaction durability. The commit mode is controlled by the user-settable parameter synchronous_commit, which can be changed in any of the ways that a configuration parameter can be set. The mode used for any one transaction depends on the value of `synchronous_commit` when transaction commit begins.

Certain utility commands, for instance `DROP TABLE`, are forced to commit synchronously regardless of the setting of `synchronous_commit`. This is to ensure consistency between the server's file system and the logical state of the database. The commands supporting two-phase commit, such as `PREPARE TRANSACTION`, are also always synchronous.

If the database crashes during the risk window between an asynchronous commit and the writing of the transaction's WAL records, then changes made during that transaction *will* be lost. The duration of the risk window is limited because a background process (the “WAL writer”) flushes unwritten WAL records to disk every wal_writer_delay milliseconds. The actual maximum duration of the risk window is three times `wal_writer_delay` because the WAL writer is designed to favor writing whole pages at a time during busy periods.

> **Caution**
>
> An immediate-mode shutdown is equivalent to a server crash, and will therefore cause loss of any unflushed asynchronous commits.

Asynchronous commit provides behavior different from setting fsync = off. `fsync` is a server-wide setting that will alter the behavior of all transactions. It disables all logic within PostgreSQL that attempts to synchronize writes to different portions of the database, and therefore a system crash (that is, a hardware or operating system crash, not a failure of PostgreSQL itself) could result in arbitrarily bad corruption of the database state. In many scenarios, asynchronous commit provides most of the performance improvement that could be obtained by turning off `fsync`, but without the risk of data corruption.

commit_delay also sounds very similar to asynchronous commit, but it is actually a synchronous commit method (in fact, `commit_delay` is ignored during an asynchronous commit). `commit_delay` causes a delay just before a transaction flushes WAL to disk, in the hope that a single flush executed by one such transaction can also serve other transactions committing at about the same time. The setting can be thought of as a way of increasing the time window in which transactions can join a group about to participate in a single flush, to amortize the cost of the flush among multiple transactions.

28.5. WAL Configuration

There are several WAL-related configuration parameters that affect database performance. This section explains their use. Consult Chapter 19 for general information about setting server configuration parameters.

Checkpoints are points in the sequence of transactions at which it is guaranteed that the heap and index data files have been updated with all information written before that checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the WAL file. (The change records were previously flushed to the WAL files.) In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the WAL (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are guaranteed to be already on disk. Hence, after a checkpoint, WAL segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the WAL segments must be archived before being recycled or removed.)

The checkpoint requirement of flushing all dirty data pages to disk can cause a significant I/O load. For this reason, checkpoint activity is throttled so that I/O begins at checkpoint start and completes before the next checkpoint is due to start; this minimizes performance degradation during checkpoints.

The server's checkpointer process automatically performs a checkpoint every so often. A checkpoint is begun every checkpoint_timeout seconds, or if max_wal_size is about to be exceeded, whichever comes first. The default settings are 5 minutes and 1 GB, respectively. If no WAL has been written since the previous checkpoint, new checkpoints will be skipped even if `checkpoint_timeout` has passed. (If WAL archiving is being used and you want to put a lower limit on how often files are archived in order to bound potential data loss, you should adjust the archive_timeout parameter rather than the checkpoint parameters.) It is also possible to force a checkpoint by using the SQL command `CHECKPOINT`.

Reducing `checkpoint_timeout` and/or `max_wal_size` causes checkpoints to occur more often. This allows faster after-crash recovery, since less work will need to be redone. However, one must balance this against the increased cost of flushing dirty data pages more often. If full_page_writes is set (as is the default), there is another factor to consider. To ensure data page consistency, the first modification of a data page after each checkpoint results in logging the entire page content. In that case, a smaller checkpoint interval increases the volume of output to the WAL, partially negating the goal of using a smaller interval, and in any case causing more disk I/O.

Checkpoints are fairly expensive, first because they require writing out all currently dirty buffers, and second because they result in extra subsequent WAL traffic as discussed above. It is therefore wise to set the checkpointing parameters high enough so that checkpoints don't happen too often. As a simple sanity check on your checkpointing parameters, you can set the checkpoint_warning parameter. If checkpoints happen closer together than `checkpoint_warning` seconds, a message will be output to the server log recommending increasing `max_wal_size`. Occasional appearance of such

a message is not cause for alarm, but if it appears often then the checkpoint control parameters should be increased. Bulk operations such as large `COPY` transfers might cause a number of such warnings to appear if you have not set `max_wal_size` high enough.

To avoid flooding the I/O system with a burst of page writes, writing dirty buffers during a checkpoint is spread over a period of time. That period is controlled by checkpoint_completion_target, which is given as a fraction of the checkpoint interval (configured by using `checkpoint_timeout`). The I/O rate is adjusted so that the checkpoint finishes when the given fraction of `checkpoint_timeout` seconds have elapsed, or before `max_wal_size` is exceeded, whichever is sooner. With the default value of 0.9, PostgreSQL can be expected to complete each checkpoint a bit before the next scheduled checkpoint (at around 90% of the last checkpoint's duration). This spreads out the I/O as much as possible so that the checkpoint I/O load is consistent throughout the checkpoint interval. The disadvantage of this is that prolonging checkpoints affects recovery time, because more WAL segments will need to be kept around for possible use in recovery. A user concerned about the amount of time required to recover might wish to reduce `checkpoint_timeout` so that checkpoints occur more frequently but still spread the I/O across the checkpoint interval. Alternatively, `checkpoint_completion_target` could be reduced, but this would result in times of more intense I/O (during the checkpoint) and times of less I/O (after the checkpoint completed but before the next scheduled checkpoint) and therefore is not recommended. Although `checkpoint_completion_target` could be set as high as 1.0, it is typically recommended to set it to no higher than 0.9 (the default) since checkpoints include some other activities besides writing dirty buffers. A setting of 1.0 is quite likely to result in checkpoints not being completed on time, which would result in performance loss due to unexpected variation in the number of WAL segments needed.

On Linux and POSIX platforms checkpoint_flush_after allows you to force OS pages written by the checkpoint to be flushed to disk after a configurable number of bytes. Otherwise, these pages may be kept in the OS's page cache, inducing a stall when `fsync` is issued at the end of a checkpoint. This setting will often help to reduce transaction latency, but it also can have an adverse effect on performance; particularly for workloads that are bigger than shared_buffers, but smaller than the OS's page cache.

The number of WAL segment files in `pg_wal` directory depends on `min_wal_size`, `max_wal_size` and the amount of WAL generated in previous checkpoint cycles. When old WAL segment files are no longer needed, they are removed or recycled (that is, renamed to become future segments in the numbered sequence). If, due to a short-term peak of WAL output rate, `max_wal_size` is exceeded, the unneeded segment files will be removed until the system gets back under this limit. Below that limit, the system recycles enough WAL files to cover the estimated need until the next checkpoint, and removes the rest. The estimate is based on a moving average of the number of WAL files used in previous checkpoint cycles. The moving average is increased immediately if the actual usage exceeds the estimate, so it accommodates peak usage rather than average usage to some extent. `min_wal_size` puts a minimum on the amount of WAL files recycled for future usage; that much WAL is always recycled for future use, even if the system is idle and the WAL usage estimate suggests that little WAL is needed.

Independently of `max_wal_size`, the most recent wal_keep_size megabytes of WAL files plus one additional WAL file are kept at all times. Also, if WAL archiving is used, old segments cannot be removed or recycled until they are archived. If WAL archiving cannot keep up with the pace that WAL is generated, or if `archive_command` or `archive_library` fails repeatedly, old WAL files will accumulate in `pg_wal` until the situation is resolved. A slow or failed standby server that uses a replication slot will have the same effect (see Section 26.2.6). Similarly, if WAL summarization is enabled, old segments are kept until they are summarized.

In archive recovery or standby mode, the server periodically performs *restartpoints*, which are similar to checkpoints in normal operation: the server forces all its state to disk, updates the `pg_control` file to indicate that the already-processed WAL data need not be scanned again, and then recycles any old WAL segment files in the `pg_wal` directory. Restartpoints can't be performed more frequently than checkpoints on the primary because restartpoints can only be performed at checkpoint records. A restartpoint can be demanded by a schedule or by an external request. The `restartpoints_timed` counter in the `pg_stat_checkpointer` view counts the first ones while the

`restartpoints_req` the second. A restartpoint is triggered by schedule when a checkpoint record is reached if at least checkpoint_timeout seconds have passed since the last performed restartpoint or when the previous attempt to perform the restartpoint has failed. In the last case, the next restartpoint will be scheduled in 15 seconds. A restartpoint is triggered by request due to similar reasons like checkpoint but mostly if WAL size is about to exceed max_wal_size However, because of limitations on when a restartpoint can be performed, `max_wal_size` is often exceeded during recovery, by up to one checkpoint cycle's worth of WAL. (`max_wal_size` is never a hard limit anyway, so you should always leave plenty of headroom to avoid running out of disk space.) The `restartpoints_done` counter in the `pg_stat_checkpointer` view counts the restartpoints that have really been performed.

In some cases, when the WAL size on the primary increases quickly, for instance during massive INSERT, the `restartpoints_req` counter on the standby may demonstrate a peak growth. This occurs because requests to create a new restartpoint due to increased XLOG consumption cannot be performed because the safe checkpoint record since the last restartpoint has not yet been replayed on the standby. This behavior is normal and does not lead to an increase in system resource consumption. Only the `restartpoints_done` counter among the restartpoint-related ones indicates that noticeable system resources have been spent.

There are two commonly used internal WAL functions: `XLogInsertRecord` and `XLogFlush`. `XLogInsertRecord` is used to place a new record into the WAL buffers in shared memory. If there is no space for the new record, `XLogInsertRecord` will have to write (move to kernel cache) a few filled WAL buffers. This is undesirable because `XLogInsertRecord` is used on every database low level modification (for example, row insertion) at a time when an exclusive lock is held on affected data pages, so the operation needs to be as fast as possible. What is worse, writing WAL buffers might also force the creation of a new WAL segment, which takes even more time. Normally, WAL buffers should be written and flushed by an `XLogFlush` request, which is made, for the most part, at transaction commit time to ensure that transaction records are flushed to permanent storage. On systems with high WAL output, `XLogFlush` requests might not occur often enough to prevent `XLogInsertRecord` from having to do writes. On such systems one should increase the number of WAL buffers by modifying the wal_buffers parameter. When full_page_writes is set and the system is very busy, setting `wal_buffers` higher will help smooth response times during the period immediately following each checkpoint.

The commit_delay parameter defines for how many microseconds a group commit leader process will sleep after acquiring a lock within `XLogFlush`, while group commit followers queue up behind the leader. This delay allows other server processes to add their commit records to the WAL buffers so that all of them will be flushed by the leader's eventual sync operation. No sleep will occur if fsync is not enabled, or if fewer than commit_siblings other sessions are currently in active transactions; this avoids sleeping when it's unlikely that any other session will commit soon. Note that on some platforms, the resolution of a sleep request is ten milliseconds, so that any nonzero `commit_delay` setting between 1 and 10000 microseconds would have the same effect. Note also that on some platforms, sleep operations may take slightly longer than requested by the parameter.

Since the purpose of `commit_delay` is to allow the cost of each flush operation to be amortized across concurrently committing transactions (potentially at the expense of transaction latency), it is necessary to quantify that cost before the setting can be chosen intelligently. The higher that cost is, the more effective `commit_delay` is expected to be in increasing transaction throughput, up to a point. The pg_test_fsync program can be used to measure the average time in microseconds that a single WAL flush operation takes. A value of half of the average time the program reports it takes to flush after a single 8kB write operation is often the most effective setting for `commit_delay`, so this value is recommended as the starting point to use when optimizing for a particular workload. While tuning `commit_delay` is particularly useful when the WAL is stored on high-latency rotating disks, benefits can be significant even on storage media with very fast sync times, such as solid-state drives or RAID arrays with a battery-backed write cache; but this should definitely be tested against a representative workload. Higher values of `commit_siblings` should be used in such cases, whereas smaller `commit_siblings` values are often helpful on higher latency media. Note that it is quite possible that a setting of `commit_delay` that is too high can increase transaction latency by so much that total transaction throughput suffers.

When commit_delay is set to zero (the default), it is still possible for a form of group commit to occur, but each group will consist only of sessions that reach the point where they need to flush their commit records during the window in which the previous flush operation (if any) is occurring. At higher client counts a "gangway effect" tends to occur, so that the effects of group commit become significant even when commit_delay is zero, and thus explicitly setting commit_delay tends to help less. Setting commit_delay can only help when (1) there are some concurrently committing transactions, and (2) throughput is limited to some degree by commit rate; but with high rotational latency this setting can be effective in increasing transaction throughput with as few as two clients (that is, a single committing client with one sibling transaction).

The wal_sync_method parameter determines how PostgreSQL will ask the kernel to force WAL updates out to disk. All the options should be the same in terms of reliability, with the exception of fsync_writethrough, which can sometimes force a flush of the disk cache even when other options do not do so. However, it's quite platform-specific which one will be the fastest. You can test the speeds of different options using the pg_test_fsync program. Note that this parameter is irrelevant if fsync has been turned off.

Enabling the wal_debug configuration parameter (provided that PostgreSQL has been compiled with support for it) will result in each XLogInsertRecord and XLogFlush WAL call being logged to the server log. This option might be replaced by a more general mechanism in the future.

There are two internal functions to write WAL data to disk: XLogWrite and issue_xlog_fsync. When track_wal_io_timing is enabled, the total amounts of time XLogWrite writes and issue_xlog_fsync syncs WAL data to disk are counted as wal_write_time and wal_sync_time in pg_stat_wal, respectively. XLogWrite is normally called by XLogInsertRecord (when there is no space for the new record in WAL buffers), XLogFlush and the WAL writer, to write WAL buffers to disk and call issue_xlog_fsync. issue_xlog_fsync is normally called by XLogWrite to sync WAL files to disk. If wal_sync_method is either open_datasync or open_sync, a write operation in XLogWrite guarantees to sync written WAL data to disk and issue_xlog_fsync does nothing. If wal_sync_method is either fdatasync, fsync, or fsync_writethrough, the write operation moves WAL buffers to kernel cache and issue_xlog_fsync syncs them to disk. Regardless of the setting of track_wal_io_timing, the number of times XLogWrite writes and issue_xlog_fsync syncs WAL data to disk are also counted as wal_write and wal_sync in pg_stat_wal, respectively.

The recovery_prefetch parameter can be used to reduce I/O wait times during recovery by instructing the kernel to initiate reads of disk blocks that will soon be needed but are not currently in PostgreSQL's buffer pool. The maintenance_io_concurrency and wal_decode_buffer_size settings limit prefetching concurrency and distance, respectively. By default, it is set to try, which enables the feature on systems where posix_fadvise is available.

28.6. WAL Internals

WAL is automatically enabled; no action is required from the administrator except ensuring that the disk-space requirements for the WAL files are met, and that any necessary tuning is done (see Section 28.5).

WAL records are appended to the WAL files as each new record is written. The insert position is described by a Log Sequence Number (LSN) that is a byte offset into the WAL, increasing monotonically with each new record. LSN values are returned as the datatype pg_lsn. Values can be compared to calculate the volume of WAL data that separates them, so they are used to measure the progress of replication and recovery.

WAL files are stored in the directory pg_wal under the data directory, as a set of segment files, normally each 16 MB in size (but the size can be changed by altering the --wal-segsize initdb option). Each segment is divided into pages, normally 8 kB each (this size can be changed via the --with-wal-blocksize configure option). The WAL record headers are described in access/xlogrecord.h; the record content is dependent on the type of event