Managing Databases

- Overview
- Creating a Database
- Template Databases
- Database Configuration
- Destroying a Database
- Tablespaces

database

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.

Overview

schema

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 ???), managed at the connection level, and authorization control (see ???), managed via the grant system. Foreign data wrappers (see ???) allow for objects within one database to act as proxies for objects in other database or clusters. The older dblink module (see ???) 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 ???.

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 ???) 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 Creating a Database) and destroyed with the `DROP DATABASE` command (see Destroying a Database). To determine the set of existing databases, examine the pg_database system catalog, for example SELECT datname FROM pg_database; The ??? program's `\l` meta-command and `-l` command-line option are also useful for listing the existing databases.

> The SQL standard calls databases “catalogs”, but there is no difference in practice.

Creating a Database

CREATE DATABASE

In order to create a database, the PostgreSQL server must be up and running (see ???).

Databases are created with the SQL command ???: 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 ??? 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 ???.) 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 Template Databases.

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 ??? reference page contains the invocation details. Note that `createdb` without any arguments will create a database with the current user name.

> ??? 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).

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 databases 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`.

> `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.

Database Configuration

Recall from ??? 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`.

Destroying a Database

Databases are destroyed with the command ???: 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 <dbname> (Unlike `createdb`, it is not the default action to drop the database with the current user name.)

Tablespaces

tablespace

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.

> 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 ??? 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.

> 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 ??? 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 ??? 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 ??? command.

To determine the set of existing tablespaces, examine the pg_tablespace system catalog, for example SELECT spcname, spcowner::regrole, pg_tablespace_location(oid) FROM pg_tablespace; It

is possible to find which databases use which tablespaces; see ???. The ??? 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.

□□□ #2
□ □□□ □□□ 2026-10-31 14:06:49 UTC
□ □□□ □□□ 2026-10-31 14:06:50 UTC