

Data Organization

Databases and Schemas



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Alexey Beresnev

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Databases and Templates

Schemas and Search Path

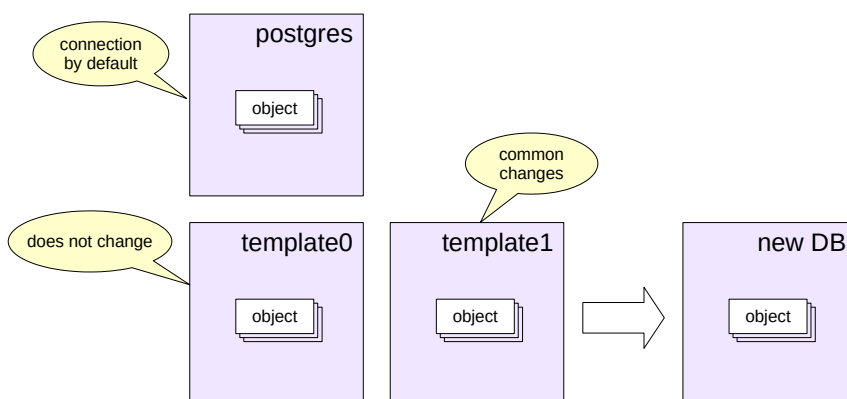
Special Schemas, Temporary Objects

Managing Databases, Schemas, and Objects

Database Cluster

Cluster initialization creates three databases

A new database is always cloned from an existing one



3

A PostgreSQL instance manages a database cluster which comprises multiple databases. When a cluster is initialized, three identical databases are created in a specific way.

<https://postgrespro.com/docs/postgresql/16/bki>

All other databases created by users are cloned from an existing one.

The template1 database is used by default when creating new databases. Any objects and extensions added to the template will be copied into any database created from it.

The template0 must never be modified. It is required in at least two scenarios. Firstly, it is necessary when restoring a database from a backup created via `pg_dump` (since the copy will include not only the database objects, but also any objects from template1). Secondly, it is used when creating a new database with an encoding different from the one specified during cluster initialization.

The postgres database is used to connect to by default by the postgres user. It is not a hard requirement to have it, but some utilities expect it to be there, so removing it is not a good idea, even if you never use the database directly.

<https://postgrespro.com/docs/postgresql/16/manage-ag-templatedbs>

Databases

The `\l` command shows a list of all databases:

```
=> \l
```

List of databases							
Name	Owner	Encoding	Locale Provider	Collate	Ctype	ICU	
Locale	ICU Rules	Access privileges					
postgres	postgres	UTF8	libc	en_US.UTF-8	en_US.UTF-8		
student	student	UTF8	libc	en_US.UTF-8	en_US.UTF-8		
template0	postgres	UTF8	libc	en_US.UTF-8	en_US.UTF-8		
	=c/postgres		+				
	postgres=CtC/postgres						
template1	postgres	UTF8	libc	en_US.UTF-8	en_US.UTF-8		
	=c/postgres		+				
	postgres=CtC/postgres						

(4 rows)

The list includes a fourth database, `student`, which is not created during cluster initialization. This database was specifically added for course purposes — it allows users to omit the database name when launching `psql`, as it defaults to a database matching the OS username.

You can also view the list of databases in the database itself:

```
=> SELECT datname, datistemplate, dataallowconn, datconnlimit FROM pg_database;
```

datname	datistemplate	dataallowconn	datconnlimit
postgres	f	t	-1
student	f	t	-1
template1	t	t	-1
template0	t	f	-1

(4 rows)

- `datistemplate` — whether the database is a template
- `dataallowconn` — whether database connections are allowed
- `datconnlimit` — maximum number of connections (-1 = unlimited)

Creating a Database from a Template

Connect to the `template1` database:

```
=> \c template1
```

You are now connected to database "template1" as user "student".

Check if we can use the `digest` function, which calculates the hash code of a text string:

```
=> SELECT digest('Hello, world!', 'md5');
```

```
ERROR:  function digest(unknown, unknown) does not exist
LINE 1: SELECT digest('Hello, world!', 'md5');
                ^
```

HINT: No function matches the given name and argument types. You might need to add explicit type casts.

There is no such function.

In fact, the `digest` function is defined in the `pgcrypto` extension. Install it:

```
=> CREATE EXTENSION pgcrypto;
```

```
CREATE EXTENSION
```

Now, we can use the functions provided by the `pgcrypto` extension. For example, the MD5 digest calculation function:

```
=> SELECT digest('Hello, world!', 'md5');
```

```
          digest
-----
 \x6cd3556deb0da54bca060b4c39479839
(1 row)
```

In order to be used to create a database from, the template must have no active connections. So, disconnect from template1 first.

```
=> \c student
```

You are now connected to database "student" as user "student".

New databases are created with the CREATE DATABASE command:

```
=> CREATE DATABASE db;
```

```
CREATE DATABASE
```

```
=> \c db
```

You are now connected to database "db" as user "student".

A new database can also be created from the OS by using the createdb tool.

```
=> SELECT datname, datistemplate, dataallowconn, datconndefaults FROM pg_database;
```

datname	datistemplate	dataallowconn	datconndefaults
postgres	f	t	-1
student	f	t	-1
template1	t	t	-1
template0	t	f	-1
db	f	t	-1

(5 rows)

Since template1 is the template used by default, the newly created database will have the pgcrypto extension already installed:

```
=> SELECT digest('Hello, world!', 'md5');
```

```
          digest
-----
 \x6cd3556deb0da54bca060b4c39479839
(1 row)
```

Database Management

The created database can be renamed (must have no active connections):

```
=> \c student
```

You are now connected to database "student" as user "student".

```
=> ALTER DATABASE db RENAME TO appdb;
```

```
ALTER DATABASE
```

```
=> SELECT datname, datistemplate, dataallowconn, datconndefaults FROM pg_database;
```

datname	datistemplate	dataallowconn	datconndefaults
postgres	f	t	-1
student	f	t	-1
template1	t	t	-1
template0	t	f	-1
appdb	f	t	-1

(5 rows)

Other parameters can also be changed, such as the maximum number of concurrent connections:

```
=> ALTER DATABASE appdb CONNECTION LIMIT 10;
```

```
ALTER DATABASE
```

```
=> SELECT datname, datistemplate, dataallowconn, datconndefaults FROM pg_database;
```

datname	datistemplate	dataallowconn	datconnlimit
postgres	f	t	-1
student	f	t	-1
template1	t	t	-1
template0	t	f	-1
appdb	f	t	10

(5 rows)

Database Size

A database size is shown by the following function:

```
=> SELECT pg_database_size('appdb');
```

```
pg_database_size
-----
          7729635
(1 row)
```

The output can be displayed in a more readable form:

```
=> SELECT pg_size_pretty(pg_database_size('appdb'));
```

```
pg_size_pretty
-----
          7548 kB
(1 row)
```

The database has no user objects so far (except for pgcrypto), so this is the size of an “empty” database.

Database object namespace

each object belongs to a schema

Purposes

dividing objects into logical groups

prevent name conflicts between applications

Schemas and users are different entities

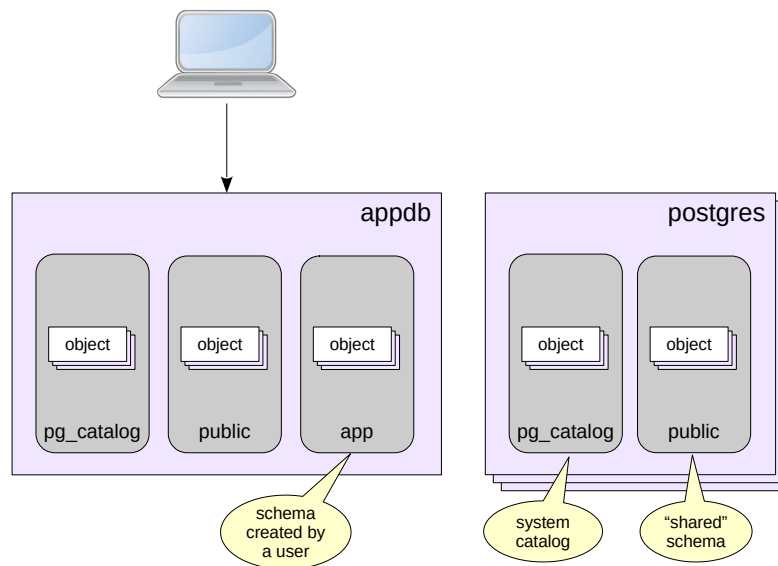
Schemas are namespaces for database objects. They separate objects into groups for easier management and serve to prevent name conflicts when multiple users or applications access the same database.

Every object that exists in the database belongs to a schema.

In PostgreSQL, schema and user are different entities (although the default settings allow users to conveniently operate schemas of the same name).

<https://postgrespro.com/docs/postgresql/16/ddl-schemas>

Databases and Schemas



A cluster comprises multiple databases. Each database contains various schemas across which database objects are distributed.

There are several standard schemas that exist in any database. More schemas can be added by users.

Clients may connect to only one database at a time, but within the database, the client can work with objects in any schema.

Schemas

```
=> \c appdb
```

You are now connected to database "appdb" as user "student".

The psql command (dn = describe namespace) shows a list of schemas in a database:

```
=> \dn
```

```
      List of schemas
Name | Owner
-----+-----
public | pg_database_owner
(1 row)
```

Create a new schema:

```
=> CREATE SCHEMA app;
```

CREATE SCHEMA

```
=> \dn
```

```
      List of schemas
Name | Owner
-----+-----
app | student
public | pg_database_owner
(2 rows)
```

Now create a table (it will get into the public schema by default):

```
=> CREATE TABLE t(s text);
```

CREATE TABLE

```
=> INSERT INTO t VALUES ('I am table t');
```

INSERT 0 1

You can get a list of tables with the \dt command:

```
=> \dt
```

```
      List of relations
Schema | Name | Type | Owner
-----+-----+-----+-----
public | t    | table | student
(1 row)
```

Objects can be moved between schemas. Since this is the logical level, the moving takes place only in the system catalog; physically, the data remains in place.

```
=> ALTER TABLE t SET SCHEMA app;
```

ALTER TABLE

Now, the table t can be accessed using its schema name:

```
=> SELECT * FROM app.t;
```

```
 s
-----
I am table t
(1 row)
```

Without the schema name, the table is not found.

```
=> SELECT * FROM t;
```

```
ERROR: relation "t" does not exist
LINE 1: SELECT * FROM t;
                        ^
```

So, how do you access objects located in various schemas?

Determining an object's schema

a schema is explicitly defined by a qualified name (*schema.name*)

a name without a qualifier is looked up in the schemas specified in the search path

Search path

defined by the *search_path* parameter

non-existent schemas and schemas with restricted access are excluded;
implicit schemas are included

the actual value is shown by the *current_schemas* function

the first explicitly specified schema in the path is where objects are created

When specifying an object, the schema it belongs to must be determined, since different schemas may contain objects of the same name.

When the object name is qualified (with the schema name), the object is looked up in the given schema, as shown on the previous slide. If the name is used without a qualifier, PostgreSQL tries to look the name up in one of the schemas listed in the search path, which is determined by the *search_path* configuration parameter.

The actual search path may differ from the *search_path* parameter value: any non-existent schemas are excluded, as well as schemas to which the user does not have access (see the Access Control module for details). In addition, some special schemas are implicitly added to the beginning of the search path.

The actual search path, including implicit schemas, is returned by a call to the *current_schemas(true)* function. The schemas are searched in the order specified in the search path, from left to right. If the desired object name is not found in the first schema, the next one is searched, and so on.

When an object is created with an unqualified name, it is placed into the first explicitly defined schema in the path.

The concept of *search_path* is similar to the *PATH* variable in operating systems.

<https://postgrespro.com/docs/postgresql/16/ddl-schemas#DDL-SCHEMAS-PATH>

<https://postgrespro.com/docs/postgresql/16/runtime-config-client#GUC-SEARCH-PATH>

Public schema

- included in the search path by default
- all objects will belong to this schema, unless configured otherwise

Schema named after the user

- included in the search path by default but is not created automatically
- if created, the user objects will belong to this schema

pg_catalog schema

- a schema for system catalog objects
- if not explicitly included in the path, implicitly included as the first one

There are several special schemas usually present in any database.

The public schema is used by default for storing objects, unless intentionally configured otherwise.

The pg_catalog schema contains *system catalog objects*. The system catalog is a collection of tables containing metadata about objects belonging to the cluster. information_schema is another schema with an alternative representation of the system catalog (as defined in the SQL standard).

If pg_catalog is not specified in the search path, the schema will be implicitly placed in the front of the path so that system objects remain visible.

Search Path

First, let's find out why the table was created in the public schema. To do that, look at the search path:

```
=> SHOW search_path;
```

```
search_path
-----
"$user", public
(1 row)
```

Construction "\$user" defines the schema with the same name as the current user (in this case, student). Since there is no such schema, it is ignored, and the table is created in public.

To avoid the headache of tracking which schemas exist, which do not, and which are not explicitly declared, use the following function:

```
=> SELECT current_schemas(true);
```

```
current_schemas
-----
{pg_catalog,public}
(1 row)
```

When the function is called with a true argument, the result includes system schemas (e.g., pg_catalog).

We can define the search path, for example:

```
=> SET search_path = public, app;
```

SET

Now, the table t will be found:

```
=> SELECT * FROM t;
```

```
s
-----
I am table t
(1 row)
```

We have just set a configuration parameter at the session level. If we restart the session, it will have reverted to default. Setting it on the cluster level would not be a good solution either, since this path may not be needed often and not by all users.

Thankfully, you can set such parameters for a specific database:

```
=> ALTER DATABASE appdb SET search_path = public, app;
```

ALTER DATABASE

Now, it will be applied for all new connections to appdb. Let's try:

```
=> \c appdb
```

You are now connected to database "appdb" as user "student".

```
=> SHOW search_path;
```

```
search_path
-----
public, app
(1 row)
```

```
=> SELECT current_schemas(true);
```

```
current_schemas
-----
{pg_catalog,public,app}
(1 row)
```

Another function current_schema() returns the first non-system schema name. This identifies the regular schema where new objects will be created by default.

```
=> SELECT current_schema();
```

current_schema

public

(1 row)

Temporary tables

- exist for the duration of the session or transaction
- not logged (no recovery after a crash)
- do not utilize the shared buffer cache

pg_temp_N schema

- created automatically for temporary tables
- pg_temp: a link to a specific temporary schema for the session
- if not explicitly included in the path, implicitly included as the *very* first one
- at the end of the session, all objects of the temporary schema are dropped;
- the schema itself remains and is reused for other sessions

PostgreSQL can work with temporary tables. Temporary tables store data that should only be available to the current session and only for as long as the session is active (or even for the duration of a single transaction within the session).

When recovering after a crash, previous sessions do not exist, so temporary tables contents will be lost. Therefore, such tables are unlogged. Additionally, temporary table pages are never stored in the shared buffer cache, residing in the internal memory of their backend process instead. Thanks to that, temporary tables can be accessed a bit more quickly than regular ones.

Temporary tables are organized using schemas. When a session is started, a temporary schema named pg_temp_N (pg_temp_1, pg_temp_2, etc.) is created for it. It can be accessed by the name pg_temp (without a number, the name will always refer to the temporary schema specific for the session).

If pg_temp is not specified in the path, it is searched before all others. Otherwise, you can specify its position in the path explicitly, like with pg_catalog.

After the session ends, all objects belonging to the temporary schema are dropped, and the schema itself remains to be reused later on.

There are other special schemas, more technical in nature.

Temporary Tables and pg_temp

Create a temporary table:

```
=> CREATE TEMP TABLE t(s text);
```

CREATE TABLE

```
=> \dt
```

```
          List of relations
 Schema | Name | Type  | Owner
-----+-----+-----+-----
 pg_temp_3 | t    | table | student
(1 row)
```

The table is created in a special schema. Each session has its own temporary schema, so that sessions cannot access each other's temporary schemas.

But, where did the regular table t disappear?

The answer lies in the extended search path. Now, the search path includes the temporary schema, and the object in it "overshadows" the object in the app schema by the same name.

```
=> SELECT current_schemas(true);
```

```
          current_schemas
-----
 {pg_temp_3,pg_catalog,public,app}
(1 row)
```

```
=> INSERT INTO t VALUES ('I am temporary table');
```

INSERT 0 1

Still, you can access objects in either schema by providing the schema name explicitly. For the temporary table, use the pseudo-schema pg_temp: it will automatically transform to the current session's pg_temp_N:

```
=> SELECT * FROM app.t;
```

```
          s
-----
 I am table t
(1 row)
```

```
=> SELECT * FROM pg_temp.t;
```

```
          s
-----
 I am temporary table
(1 row)
```

Temporary schemas are not limited to just tables.

```
=> CREATE VIEW v AS SELECT * FROM pg_temp.t;
```

NOTICE: view "v" will be a temporary view
CREATE VIEW

Temporary schemas and data in them may have different scopes of life (depending on the clauses ON COMMIT DELETE, PRESERVE, and DROP). In any case, all temporary objects are deleted upon reconnecting to a database:

```
=> \c appdb
```

You are now connected to database "appdb" as user "student".

```
=> SELECT current_schemas(true);
```

```
          current_schemas
-----
 {pg_catalog,public,app}
(1 row)
```

```
=> SELECT * FROM pg_temp.v;
```



```
ERROR:  relation "pg_temp.v" does not exist
LINE 1: SELECT * FROM pg_temp.v;
                        ^
```

```
=> SELECT * FROM pg_temp.t;
```

```
ERROR:  relation "pg_temp.t" does not exist
LINE 1: SELECT * FROM pg_temp.t;
                        ^
```

Deleting Objects

A schema cannot be deleted as long as any objects remain in it:

```
=> DROP SCHEMA app;
```

```
ERROR:  cannot drop schema app because other objects depend on it
DETAIL:  table t depends on schema app
HINT:   Use DROP ... CASCADE to drop the dependent objects too.
```

But you can delete a schema together with all its objects:

```
=> DROP SCHEMA app CASCADE;
```

```
NOTICE:  drop cascades to table t
DROP SCHEMA
```

If the database is no longer needed, it can be deleted as well.

```
=> \conninfo
```

```
You are connected to database "appdb" as user "student" via socket in
"/var/run/postgresql" at port "5432".
```

```
=> \c student
```

```
You are now connected to database "student" as user "student".
```

```
=> DROP DATABASE appdb;
```

```
DROP DATABASE
```

On the logical level

- cluster contains databases,
- database contains schemas,
- schema contains specific objects (tables, indexes etc.)

Databases are created by cloning existing ones

A schema can be specified explicitly or determined using the search path

Some schemas have specific purposes

1. Create a new database and connect to it.
2. Check the size of the created database.
3. Create two schemas. Name one app, and the other after your user name. Create several tables in both schemas and populate them with data.
4. Check how much the database size has increased.
5. Modify the search path variable value so that when connecting to the database, tables from both schemas are accessible by an unqualified name. The “username” schema should have priority.

1. Database

```
=> CREATE DATABASE data_databases;
```

```
CREATE DATABASE
```

```
=> \c data_databases
```

You are now connected to database "data_databases" as user "student".

2. Database Size

```
=> SELECT pg_size_pretty(pg_database_size('data_databases'));
```

```
pg_size_pretty
-----
7516 kB
(1 row)
```

Save the value in a psql variable:

```
=> SELECT pg_database_size('data_databases') AS oldsize \gset
```

3. Schemas and Tables

```
=> CREATE SCHEMA app;
```

```
CREATE SCHEMA
```

```
=> CREATE SCHEMA student;
```

```
CREATE SCHEMA
```

Which schema receives tables created without explicit schema qualification?

```
=> SELECT current_schema();
```

```
current_schema
-----
student
(1 row)
```

student schema tables:

```
=> CREATE TABLE a(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO a VALUES ('student');
```

```
INSERT 0 1
```

```
=> CREATE TABLE b(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO b VALUES ('student');
```

```
INSERT 0 1
```

app schema tables:

```
=> CREATE TABLE app.a(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO app.a VALUES ('app');
```

```
INSERT 0 1
```

```
=> CREATE TABLE app.c(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO app.c VALUES ('app');
```

```
INSERT 0 1
```

4. Database Size Change

```
=> SELECT pg_size_pretty(pg_database_size('data_databases'));
```

```
pg_size_pretty
-----
7612 kB
(1 row)
```

```
=> SELECT pg_database_size('data_databases') AS newsize \gset
```

The size has changed to:

```
=> SELECT pg_size_pretty(:newsize::bigint - :oldsize::bigint);
```

```
pg_size_pretty
-----
96 kB
(1 row)
```

5. Search Path

With the current search path configuration, only student schema tables are visible:

```
=> SELECT * FROM a;
```

```
      s
-----
student
(1 row)
```

```
=> SELECT * FROM b;
```

```
      s
-----
student
(1 row)
```

```
=> SELECT * FROM c;
```

```
ERROR:  relation "c" does not exist
LINE 1: SELECT * FROM c;
                        ^
```

Modify the search path:

```
=> ALTER DATABASE data_databases SET search_path = "$user",app,public;
```

```
ALTER DATABASE
```

```
=> \c
```

You are now connected to database "data_databases" as user "student".

```
=> SHOW search_path;
```

```
search_path
-----
"$user", app, public
(1 row)
```

Now, tables from both schemas are visible, but student takes priority:

```
=> SELECT * FROM a;
```

```
      s
-----
student
(1 row)
```

```
=> SELECT * FROM b;
```

```
      s
-----
student
(1 row)
```

```
=> SELECT * FROM c;
```

```
s
```

```
-----
```

```
app
```

```
(1 row)
```

1. Create a database. For all sessions connecting to this database, set the *temp_buffers* parameter to four times the default value.

1. Use the command ALTER DATABASE ... SET:

<https://postgrespro.com/docs/postgresql/16/sql-alterdatabase>

More about the *temp_buffers* parameter:

<https://postgrespro.com/docs/postgresql/16/runtime-config-resource#GUC-TEMP-BUFFERS>

1. Setting the temp_buffers Parameter

```
=> CREATE DATABASE data_databases;
```

```
CREATE DATABASE
```

```
=> \c data_databases
```

You are now connected to database "data_databases" as user "student".

The temp_buffers parameter determines the amount of memory allocated for the local cache for temporary tables during any session. If the temporary tables data does not fit into temp_buffers, the pages are evicted, like with the regular buffer cache. Too low of a value may affect the server performance if temporary tables are used frequently.

The default temp_buffers value is 8MB:

```
=> SELECT name, setting, unit, boot_val, reset_val
FROM pg_settings
WHERE name = 'temp_buffers' \gx
```

```
-[ RECORD 1 ]-----
name       | temp_buffers
setting    | 1024
unit       | 8kB
boot_val   | 1024
reset_val  | 1024
```

Set the value to 32MB for all new sessions:

```
=> ALTER DATABASE data_databases SET temp_buffers = '32MB';
```

```
ALTER DATABASE
```

```
=> \c
```

You are now connected to database "data_databases" as user "student".

```
=> SHOW temp_buffers;
```

```
temp_buffers
-----
32MB
(1 row)
```

The parameter changes made by ALTER DATABASE are stored in the pg_db_role_setting table. You can view them in psql with the following command:

```
=> \drds
```

```
                List of settings
Role | Database | Settings
-----+-----+-----
      | data_databases | temp_buffers=32MB
(1 row)
```

Naturally, you do not have to set the temp_buffers parameter just at the database level. For example, you can set them for the whole cluster if you define it in postgresql.conf.