Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov
Translated by Liudmila Mantrova
Cover photo by Oleg Bartunov (Phu monastery and Bhrikuti peak, Nepal)

Feedback
Please send your feedback, comments and suggestions to:
edu@postgrespro.ru

Agenda

Databases and templates

Schemas and search path

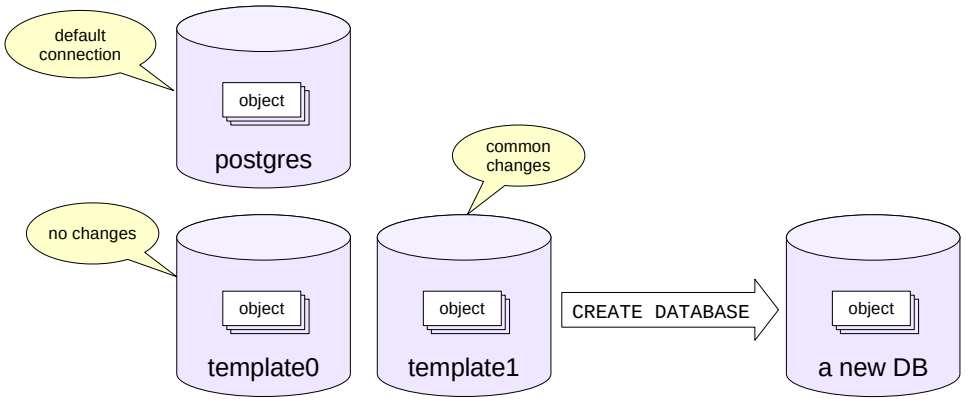
Special schemas

System catalog

Database Cluster

Cluster initialization creates three databases

A new database is always cloned from an existing one



3

A PostgreSQL instance manages several databases, which constitute a database cluster. During cluster initialization (which is performed either automatically during installation or manually using the `initdb` command), three identical databases are created. All other databases created by user are cloned from an already existing database.

By default, the `template1` DB is used as the source for creating new databases. You can extend it with additional objects and modules that will be copied into each new database.

The `template0` database must never be modified. This template is required at least in two situations. First, it is used to restore the database from a backup copy created by the pg_dump utility (as described in the “Backup Overview” lecture). Second, it is required when creating a new database with a non-default collation (we discuss it in more detail in the DBA2 course).

The `postgres` database is used to establish a connection on behalf of the postgres user by default. Keeping this database is not mandatory, but some utilities rely on its existence, so it’s not recommended to delete this database even if you do not need it.

https://postgrespro.com/docs/postgresql/15/manage-ag-templatedbs

Databases

You can view the list of all databases using the following psql command:

```
=> \l
```

```
                                                List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    | ICU Locale | Locale Provider |   Access privileges
-----------+----------+----------+-------------+-------------+------------+-----------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 student   | student  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
           |          |          |             |             |            |                 | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
           |          |          |             |             |            |                 | postgres=CTc/postgres
(4 rows)
```

It displays many fields that do not interest us right now.

When we create a new database, it is cloned from template1 by default.

```
=> CREATE DATABASE data_logical;
```

```
CREATE DATABASE
```

```
=> \c data_logical
```

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

```
=> \l
```

```
                                                List of databases
     Name     |  Owner   | Encoding |   Collate   |    Ctype    | ICU Locale | Locale Provider |   Access privileges
--------------+----------+----------+-------------+-------------+------------+-----------------+-----------------------
 data_logical | student  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 postgres     | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 student      | student  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 template0    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
              |          |          |             |             |            |                 | postgres=CTc/postgres
 template1    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
              |          |          |             |             |            |                 | postgres=CTc/postgres
(5 rows)
```

Schemas

Namespaces for objects

- classify objects into logical groups
- prevent name conflicts between applications

Schemas and users are different entities

Special schemas

- `public` — all objects are created here by default
- `pg_catalog` — system catalog
- `information_schema` — an alternative view of the system catalog
- `pg_temp` — a storage for temporary tables
- etc.

Schemas are virtually namespaces for database objects. They enable classifying objects into logical groups for easier management as well as prevent name conflicts when serving several users or applications.

In PostgreSQL, *schemas* and *users* are different entities (even though the default settings are well-suited for using the schema with the same name as the current user).

There are several special schemas that are usually present in each database.

By default, database objects are kept in the `public` schema unless another location is specified.

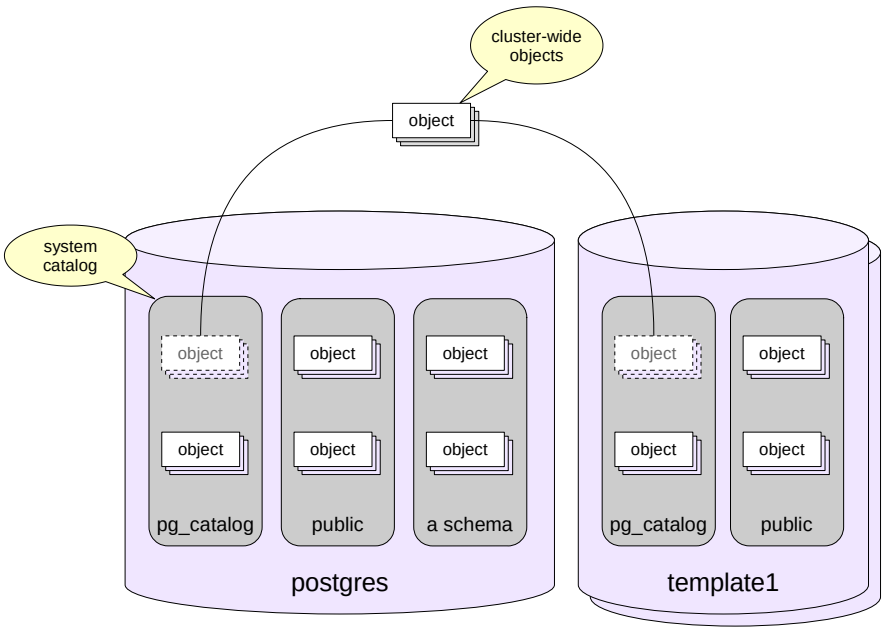
The `pg_catalog` schema stores objects of the *system catalog*. The system catalog comprises metadata of cluster objects, which is stored in tables within the cluster itself. An alternative view of the system catalog (defined in the SQL standard) is provided by `information_schema`.

The `pg_temp` schema stores temporary tables. (In fact, temporary tables are created in schemas called `pg_temp_1`, `pg_temp_2`, and so on: each user has its own schema, but they are all referred to as `pg_temp`.)

There are also some other schemas, but their purpose is merely technical.

https://postgrespro.com/docs/postgresql/15/ddl-schemas

Databases and Schemas



Schemas belong to databases, and all DB objects belong to this or that schema.

However, several system catalog tables store the objects that are common to the whole cluster. They contain the list of databases, the list of users, and some other information. These tables do not belong to any database, but they are equally visible in all databases.

Thus, a client connected to a database can see the descriptions of the objects that belong to this database as well as cluster-wide objects. Object descriptions for other databases are only available if you connect to these databases.

Schemas

There is a special psql command that displays the list of schemas (\dn = describe namespace):

```
=> \dn
```

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

This command does not show service schemas. To display them, you have to add the S modifier (it works in a similar way for many other commands):

```
=> \dnS
```

```
            List of schemas
        Name        |       Owner
--------------------+-------------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | pg_database_owner
(4 rows)
```

We have already touched upon some of these schemas (public, pg_catalog, information_schema); the rest will be covered in the subsequent lectures.

Another useful modifier is the “plus” sign, which displays additional information:

```
=> \dn+
```

```
                                    List of schemas
  Name  |       Owner       |           Access privileges            |      Description
--------+-------------------+----------------------------------------+------------------------
 public | pg_database_owner | pg_database_owner=UC/pg_database_owner+| standard public schema
        |                   | =U/pg_database_owner                   |
(1 row)
```

Let’s create a new schema:

```
=> CREATE SCHEMA special;
```

```
CREATE SCHEMA
```

```
=> \dn
```

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

Create a table:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

By default, the table will be created in the public schema. To view the list of tables for this schema, you can specify a template for schema and table names when running the \dt command:

```
=> \dt public.*
```

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

Tables (and other objects) can be moved between schemas. Since we are talking about the logical structure, only the system catalog is changed; the physical location of the data remains the same.

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

ALTER TABLE
```

What will remain in the public schema?

```
=> \dt public.*

Did not find any relation named "public.*".
```

Nothing. And what about the special schema?

```
=> \dt special.*

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

The table was moved. You can now access it by explicitly specifying its schema:

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

 n
---
(0 rows)
```

But if you omit the schema name, this table won't be found:

```
=> SELECT * FROM t;

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

Search Path

Postgres PROFESSIONAL

Determining the object's schema

a qualified name (*schema.name*) explicitly defines the schema

an unqualified name is looked up in schemas listed in search path

Search path

is defined by the *search_path* parameter,
the actual path is displayed by the `current_schemas` function

excludes non-existent and inaccessible schemas

`pg_temp` and `pg_catalog` schemas are implicitly included first
unless they are already specified in the *search_path* parameter

the first explicitly specified schema is used for object creation

Different schemas can contain objects with the same name, so when specifying an object, it's necessary to identify its schema.

If the object has a qualified name, it's easy: the explicitly specified schema is used. Otherwise, PostgreSQL tries to find the object name in one of the schemas listed in the search path, which is defined in the *search_path* configuration parameter.

The actual search path can differ from the *search_path* parameter value. It excludes non-existent schemas listed in *search_path* as well as those schemas that the current user cannot access (we will cover access control in one of the next lectures of this course). Besides, the following schemas are implicitly added to the beginning of the search path:

- the `pg_catalog` schema to ensure that the system catalog is always accessible
- the `pg_temp` schema if the user has created temporary objects

You can view the actual search path, including the implicitly added schemas, by calling the `current_schemas(true)` function. Schemas are looked up as they follow in the search path, from left to right. If the object with the specified name is not found in the schema, the search continues in the next one.

If you create an object with an unqualified name, it will get into the first schema that is explicitly specified in the search path.

We can say that *search_path* is somewhat analogous to the PATH variable in operating systems.

https://postgrespro.com/docs/postgresql/15/runtime-config-client#GUC-SEARCH-PATH

Search Path

By default, the search path looks as follows:

```
=> SHOW search_path;

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

The “$user” placeholder represents the schema with the same name as the current user (student in our case). Since there is no such schema, it is simply ignored.

To take the guesswork out of learning whether a schema exists or not, and if any schemas are used implicitly, you can use the following function:

```
=> SELECT current_schemas(true);

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

We can see that a non-existent schema has been removed, while the system catalog schema has been implicitly added.

Let’s set the search path like this:

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

SET
```

Now the table will be found.

```
=> SELECT * FROM t;

 n
---
(0 rows)
```

Here we have modified the parameter at the session level (its value won’t be saved for the next connections). Setting this parameter at the cluster level is not a good idea either: this path is probably required only by some of the clients, and not at all times.

You can also modify configuration for a particular database only:

```
=> ALTER DATABASE data_logical SET search_path = public, special;

ALTER DATABASE
```

Now this parameter will be set for all new connections to the data_logical database. Let’s check:

```
=> \c data_logical

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

=> SHOW search_path;

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

System Catalog

Description of all cluster objects

- the list of tables in each database (the `pg_catalog` schema) and several cluster-wide objects
- several views provided for convenience

Access

- SQL queries, special psql commands

Conventions

- table names start with `pg_`
- column names contain a three-letter prefix
- the `oid` column of type `oid` is used as the primary key
- object names are lowercase

The system catalog stores metadata of cluster objects. In each database, you can find a separate set of tables that describe the objects of this particular database, as well as several tables common to the whole cluster. There are also a number of views provided for convenience.

https://postgrespro.com/docs/postgresql/15/catalogs

You can access the system catalog using regular SQL queries. The psql client also offers a whole range of commands that provide convenient ways to view it. Catalog tables should not be modified directly; they get updated automatically as you run DDL commands.

https://postgrespro.com/docs/postgresql/15/app-psql

The names of all system catalog tables start with `pg_`, for example, `pg_database`. Column names usually start with a prefix that corresponds to the table name, for example, `datname`. Object names are lowercase, for example, `'postgres'`.

In most system catalog tables the `oid` column serves as a primary key. This column is of a special oid type, which means object identifier (a 32-bit integer).

https://postgrespro.com/docs/postgresql/15/datatype-oid

System Catalog

To display information about any objects, psql (just like any graphical environment) accesses the system catalog tables.

For example, to get the list of databases in the cluster, the \l command reads the following table:

```
=> SELECT datname FROM pg_database;
```

```
   datname
-------------
 postgres
 student
 template1
 template0
 data_logical
(5 rows)
```

We can always view the queries executed by a command:

```
=> \set ECHO_HIDDEN on
```

```
=> \l
```

```
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       d.daticulocale as "ICU Locale",
       CASE d.datlocprovider WHEN 'c' THEN 'libc' WHEN 'i' THEN 'icu' END AS "Locale Provider",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges"
FROM pg_catalog.pg_database d
ORDER BY 1;
**************************
```

```
                                                  List of databases
     Name     |  Owner   | Encoding |   Collate   |    Ctype    | ICU Locale | Locale Provider |   Access privileges
--------------+----------+----------+-------------+-------------+------------+-----------------+-----------------------
 data_logical | student  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 postgres     | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 student      | student  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            |
 template0    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
              |          |          |             |             |            |                 | postgres=CTc/postgres
 template1    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |            | libc            | =c/postgres          +
              |          |          |             |             |            |                 | postgres=CTc/postgres
(5 rows)
```

This is how you can explore the system catalog.

Let's turn off the command display.

```
=> \set ECHO_HIDDEN off
```

The list of schemas is stored in the following table:

```
=> SELECT nspname FROM pg_namespace;
```

```
      nspname
--------------------
 pg_toast
 pg_catalog
 public
 information_schema
 special
(5 rows)
```

Other objects like tables and indexes can be viewed as follows:

```
=> SELECT relname, relkind, relnamespace
FROM pg_class WHERE relname = 't';
```

```
 relname | relkind | relnamespace
---------+---------+--------------
 t       | r       |        16424
(1 row)
```

All column names start with rel (relation) here.

- relkind — object type (r — table, i — index, etc.);
- relnamespace — schema.

The relnamespace field is of the oid type; and here is the corresponding row of the pg_namespace table:

```
=> SELECT oid, nspname FROM pg_namespace WHERE nspname = 'special';
```

```
  oid  | nspname
-------+---------
 16424 | special
(1 row)
```

To simplify the transformation between the text and oid types, you can cast these fields to the regnamespace type:

```
=> SELECT relname, relkind, relnamespace::regnamespace::text
FROM pg_class WHERE relname = 't';
```

```
 relname | relkind | relnamespace
---------+---------+--------------
 t       | r       | special
(1 row)
```

And here is how you can get the list of objects that belong to a schema, e.g., to the pg_catalog schema:

```
=> SELECT relname, relkind FROM pg_class
WHERE relnamespace = 'pg_catalog'::regnamespace LIMIT 5;
```

```
            relname             | relkind
--------------------------------+---------
 pg_statistic                   | r
 pg_type                        | r
 pg_foreign_table               | r
 pg_proc_oid_index              | i
 pg_proc_proname_args_nsp_index | i
(5 rows)
```

Similar reg-types are defined for some other system catalog tables as well. It helps to shorten queries and do without explicit table joins.

Deleting Objects

Is it possible to drop the special schema?

```
=> DROP SCHEMA special;
```

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

You cannot drop a schema if it contains any objects. They have to be either moved or deleted.

But you can drop the schema together with all its objects at once:

```
=> DROP SCHEMA special CASCADE;
```

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

A database can be dropped if it has no active connections.

```
=> \conninfo
```

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

```
=> \c postgres
```

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

```
=> DROP DATABASE data_logical;
```

```
DROP DATABASE
```

Takeaways

At the logical level

- a cluster contains databases
- databases contain schemas
- schemas contain actual objects (tables, indexes, etc.)

New databases are created by cloning existing ones

Object schemas are determined by the search path

The full description of the database cluster contents is stored in the system catalog

Practice

1. In an empty database, create a schema that has the same name as the current user. Create the app schema.
   Create several tables in both schemas.
2. In psql, display the description of the created schemas and the list of tables that belong to them.
3. Set the search path to enable access to the tables in both schemas by an unqualified name; the “user” schema must have the priority.
   Check the result.

Task 1. Databases, Schemas, Tables

Create a database:

```
=> CREATE DATABASE data_logical;
```

```
CREATE DATABASE
```

```
=> \c data_logical
```

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

Create schemas:

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

```
CREATE SCHEMA
```

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

```
CREATE SCHEMA
```

Create some tables in the student schema:

```
=> 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
```

Create some tables in the app schema:

```
=> 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
```

Task 2. Schemas and Tables

View the list of schemas:

```
=> \dn
```

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

View the list of tables:

```
=> \dt student.*
```

```
        List of relations
 Schema  | Name | Type  |  Owner
---------+------+-------+---------
 student | a    | table | student
 student | b    | table | student
(2 rows)
```

```
=> \dt app.*
```

```
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+---------
 app    | a    | table | student
 app    | c    | table | student
(2 rows)
```

3. Search Path

With the current search path settings, we can see only those tables that belong to the student schema:

```
=> 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;
                      ^
```

Let's change the search path:

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

```
ALTER DATABASE
```

```
=> \c
```

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

```
=> SHOW search_path;
```

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

Now we can see the tables of both schemas, but the student schema has the priority:

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

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

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

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

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

```
  s
-----
 app
(1 row)
```