

# Basic Tools Using psql



## 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](mailto: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.

Launching psql and Connecting to the Database

Getting Help

Working with psql

Configuring psql

# Purpose of psql



Terminal client for working with PostgreSQL

Comes with the DBMS

Used by administrators and developers for interactive work and script execution

3

There are other third-party tools available, but they are not considered in the scope of the course.

The psql terminal client will be used throughout the course. Those who are used to working with GUI tools may find it uncomfortable at first. Nevertheless, it is very powerful if you get used to it.

This is the only client supplied with the DBMS. The knowledge of psql will be useful to both developers and DB administrators, regardless of which tool they choose to work with at the end of the day.

<https://postgrespro.com/docs/postgresql/16/app-psql>

## Launch

```
$ psql -d database -U user -h host -p port
```

## New connection in psql

```
=> \c[onnect] database user node port
```

## Information about the current connection

```
=> \conninfo
```

The required connection parameters include: database name, user name, server name, port number. If these parameters are not specified, psql will try to connect using the default values:

- *database* — matches the user name;
- *user* — matches the OS user name;
- *node* — connection via Unix socket;
- *port* — usually 5432.

To make a new connection without leaving psql, run the `\connect` command. It uses the current connection's parameters as defaults.

The `\conninfo` command provides information about the current connection.

Additional information about connection configuration options:

<https://postgrespro.com/docs/postgresql/16/libpq-envvars>

<https://postgrespro.com/docs/postgresql/16/libpq-pgservice>

<https://postgrespro.com/docs/postgresql/16/libpq-pgpass>

# Getting Help



## In the OS command line

```
$ psql --help
$ man psql
```

## In psql

|                 |                           |
|-----------------|---------------------------|
| => \?           | list of psql commands     |
| => \? variables | psql variables            |
| => \h[elp]      | list of SQL commands      |
| => \h command   | syntax of the SQL command |
| => \q           | quit                      |

5

Reference information on psql can be obtained not only from the documentation, but also from within the system directly.

psql with the `--help` option displays a startup help message. If the documentation package is installed with the system, you can view the manual for psql using the `man psql` command.

psql can execute SQL commands as well as its own commands. All psql commands start with a backslash and, as a rule, can be abbreviated to their first letter.

Inside psql, you can get a list and a brief description of all psql commands: `\?`.

The `\help` command provides a list of SQL commands that the server supports, as well as the syntax of an SQL command (if specified).

Another command that is useful to know, although it has nothing to do with the help, is `\q` — exit psql. Alternatively, you can use the `exit` and `quit` commands to quit.

## Executing SQL Commands and Formatting the Output

Run psql:

```
student$ psql
```

Check the connection:

```
=> \conninfo
```

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

Using the default parameters, we have connected to the student database as the student user. You will learn more about databases and users in later modules.

The \connect command creates a new connection without leaving psql.

---

SQL commands, unlike psql ones, may span multiple rows. To send an SQL command for execution, end it with a semicolon:

```
=> SELECT schemaname, tablename, tableowner
FROM pg_tables
LIMIT 5;
```

| schemaname | tablename             | tableowner |
|------------|-----------------------|------------|
| pg_catalog | pg_statistic          | postgres   |
| pg_catalog | pg_type               | postgres   |
| pg_catalog | pg_foreign_table      | postgres   |
| pg_catalog | pg_authid             | postgres   |
| pg_catalog | pg_statistic_ext_data | postgres   |

(5 rows)

psql can give output in different formats. Here are some of them:

- Aligned format
- Unaligned format
- Extended format

The aligned format is the default. It sets each column's width based on its contents. There is also the header and the total row.

---

psql commands to switch display modes:

- \a — switches aligned format and unaligned format mode
- \t — switches the header and footer display

Let's switch to non-aligned, turn the header and the total row off, and use a whitespace as the separator:

```
=> \t \a
```

Tuples only is on.

Output format is unaligned.

```
=> \pset fieldsep ' '
```

Field separator is " ".

```
=> SELECT schemaname, tablename, tableowner FROM pg_tables LIMIT 5;
```

```
pg_catalog pg_statistic postgres
pg_catalog pg_type postgres
pg_catalog pg_foreign_table postgres
pg_catalog pg_authid postgres
pg_catalog pg_statistic_ext_data postgres
```

```
=> \t \a
```

Tuples only is off.

Output format is aligned.

The extended format is convenient for displaying multiple columns for one or several records:

```
=> \x
```

Expanded display is on.

```
=> SELECT * FROM pg_tables WHERE tablename = 'pg_class';
```

```
-[ RECORD 1 ]-----
schemaname | pg_catalog
tablename  | pg_class
tableowner | postgres
tablespace |
hasindexes | t
hasrules   | f
hastriggers | f
rowsecurity | f
```

```
=> \x
```

Expanded display is off.

The extended mode can be set for a single query. To do that, add \gx at the end instead of a semicolon:

```
=> SELECT * FROM pg_tables WHERE tablename = 'pg_proc' \gx
```

```
-[ RECORD 1 ]-----
schemaname | pg_catalog
tablename  | pg_proc
tableowner  | postgres
tablespace  |
hasindexes  | t
hasrules    | f
hastriggers | f
rowsecurity | f
```

You can see all the formatting options by using the \pset command. If used with no parameters, it will display current settings:

```
=> \pset

border          1
columns         0
csv_fieldsep    ','
expanded        off
fieldsep        ' '
fieldsep_zero   off
footer          on
format          aligned
linestyle       ascii
null            ''
numericlocale   off
pager           1
pager_min_lines 0
recordsep       '\n'
recordsep_zero  off
tableattr
title
tuples_only     off
unicode_border_linestyle single
unicode_column_linestyle single
unicode_header_linestyle single
xheader_width   full
```

## Interacting with the OS

psql can run shell commands:

```
=> \! pwd
```

```
/home/student
```

You can set OS environment variables:

```
=> \setenv HELLO Hello
```

```
=> \! echo $HELLO
```

```
Hello
```

It can write output into a file with the \o[ut] command:

```
=> \o tmp/dbal_log
```

```
=> SELECT schemaname, tablename, tableowner FROM pg_tables LIMIT 5;
```

There is nothing on the screen! Let's check the file:

```
=> \! cat tmp/dbal_log
```

```
schemaname |      tablename      | tableowner
-----+-----+-----
pg_catalog | pg_statistic         | postgres
pg_catalog | pg_type              | postgres
pg_catalog | pg_foreign_table     | postgres
pg_catalog | pg_authid            | postgres
pg_catalog | pg_statistic_ext_data | postgres
(5 rows)
```

Let's get the output back to the screen:

```
=> \o
```

## Executing Scripts

Another way to run a query is with the \g command. You can specify parameters for this particular query in the brackets.

Query output can be redirected to OS command by appending the command after a vertical bar. For example, display query output with line numbers:

```
=> SELECT format('SELECT count(*) FROM %I;', tablename)
FROM pg_tables
LIMIT 3
\g (tuples_only=on format=unaligned) | cat -n

    1  SELECT count(*) FROM pg_statistic;
    2  SELECT count(*) FROM pg_type;
    3  SELECT count(*) FROM pg_foreign_table;
```

The \g command can specify a filename to redirect output:

```
=> SELECT format('SELECT count(*) FROM %I;', tablename)
FROM pg_tables
LIMIT 3
\g (tuples_only=on format=unaligned) tmp/dbal_log
```

This is what we get:

```
=> \! cat tmp/dbal_log

SELECT count(*) FROM pg_statistic;
SELECT count(*) FROM pg_type;
SELECT count(*) FROM pg_foreign_table;
```

---

We can execute this file as a script using \i[include]:

```
=> \i tmp/dbal_log

count
-----
409
(1 row)

count
-----
613
(1 row)

count
-----
0
(1 row)
```

Other ways to run commands from a file:

- psql < filename
  - psql -f filename
- 

You can skip creating a file in the last example if you end the query with \gexec:

```
=> SELECT format('SELECT count(*) FROM %I;', tablename)
FROM pg_tables
LIMIT 3
\gexec

count
-----
409
(1 row)

count
-----
613
(1 row)

count
-----
0
(1 row)
```

gexec considers the contents of each column of each row an SQL-operator, and tries to execute them one by one.

---

## psql Variables and Control Structures

Not unlike shell, psql has its own variables, including some pre-defined ones (with special meaning for psql).

Store the value of the OS USER environment variable in a psql variable:

```
=> \getenv User USER
```

Set a variable Test:

```
=> \set Test Hi
```

To substitute a variable's value, use a colon prefix before the variable name:

```
=> \echo :Test :User!
```

Hi student!

To reset a variable, use:

```
=> \unset Test
```

```
=> \echo :Test
```

:Test

---

Query results can be saved to variables. For this purpose use the \gset command termination:

```
=> SELECT now() AS curr_time \gset
```

```
=> \echo :curr_time
```

2025-09-24 16:56:30.481473+03

The query must return only one record.

---

If used with no parameters, \set displays all currently set variables and their values:

```
=> \set
```



```
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'student'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
ERROR = 'false'
FETCH_COUNT = '0'
HIDE_TABLEAM = 'off'
HIDE_TOAST_COMPRESSION = 'off'
HISTCONTROL = 'none'
HISTFILE = 'hist'
HISTSIZE = '500'
HOST = '/var/run/postgresql'
IGNOREEOF = '0'
LAST_ERROR_MESSAGE = ''
LAST_ERROR_SQLSTATE = '00000'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5432'
PROMPT1 = '%/R%x%# '
PROMPT2 = '%/R%x%# '
PROMPT3 = '>> '
QUIET = 'off'
ROW_COUNT = '1'
SERVER_VERSION_NAME = '16.10 (Ubuntu 16.10-1.pgdg24.04+1)'
SERVER_VERSION_NUM = '160010'
SHELL_ERROR = 'false'
SHELL_EXIT_CODE = '0'
SHOW_ALL_RESULTS = 'on'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
SQLSTATE = '00000'
USER = 'student'
User = 'student'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 16.10 (Ubuntu 16.10-1.pgdg24.04+1) on x86_64-pc-linux-gnu, compiled
by gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0, 64-bit'
VERSION_NAME = '16.10 (Ubuntu 16.10-1.pgdg24.04+1)'
VERSION_NUM = '160010'
curr_time = '2025-09-24 16:56:30.481473+03'
```

In scripts, you can use conditional operators.

For example, let's check if `working_dir` has a value, and if it does not, set it as the current directory. The following command checks if the value is set and returns a Boolean value:

```
=> \echo :{?working_dir}
```

FALSE

This conditional `psql` operator checks if the variable exists and sets the default value if needed:

```
=> \if :{?working_dir}
-- variable is defined
\else
-- set the value as the output of the OS command
\set working_dir `pwd`
\endif
```

Now we can be sure that the `working_dir` variable is defined:

```
=> \echo :working_dir
```

/home/student

### System Catalog Commands

There is a set of commands (mostly starting with `\d`) used to quickly and conveniently get information about database objects.

Example:

```
=> \d pg_tables
```

| View "pg_catalog.pg_tables" |         |           |          |         |
|-----------------------------|---------|-----------|----------|---------|
| Column                      | Type    | Collation | Nullable | Default |
| schemaname                  | name    |           |          |         |
| tablename                   | name    |           |          |         |
| tableowner                  | name    |           |          |         |
| tablespace                  | name    |           |          |         |
| hasindexes                  | boolean |           |          |         |
| hasrules                    | boolean |           |          |         |
| hastriggers                 | boolean |           |          |         |
| rowsecurity                 | boolean |           |          |         |

We will see more of these commands later.

### Configuring psql

On startup, `psql` runs two scripts (if they exist):

- First a common system `psqlrc` script
- Then a custom `.psqlrc` file

The user configuration file must be in the home directory. The system script's location can be discovered with the following command:

```
student$ pg_config --sysconfdir
```

```
/etc/postgresql-common
```

Neither file exists by default.

---

You can use the files to configure your session parameters, for example:

- `psql` prompt
- Program for page-by-page viewing of query results
- Variables for storing the text of frequently used commands

For example, let's store a query that returns 5 largest tables in a variable `top5`:

```
=> \set top5 'SELECT tablename, pg_total_relation_size(schemaname||'.'||tablename) AS bytes FROM pg_tables ORDER BY bytes DESC LIMIT 5;'
```

Now we can execute the query by just typing:

```
=> :top5
```

| tablename      | bytes   |
|----------------|---------|
| pg_proc        | 1245184 |
| pg_rewrite     | 745472  |
| pg_attribute   | 720896  |
| pg_description | 630784  |
| pg_statistic   | 294912  |

(5 rows)

If you write the `\set` command into the `~/.psqlrc` file, the `top5` variable will be available immediately after `psql` startup.

---

Thanks to readline support, `psql` can autocomplete keywords and object names, and it also stores the command history. The name and the size of the history file are set by `HISTFILE` and `HISTSIZE` variables.

# Takeaways



psql is a terminal client for working with PostgreSQL

Connection parameters are required at startup

Executes SQL and psql commands

Includes tools for interactive work, as well as for preparing and executing scripts

1. Run `psql` and check the current connection information.
2. Display a detailed list of databases.
3. By default, `psql` uses `less` command for page-by-page output. Replace it with `less -XS` and display the detailed database list again.
4. The default prompt shows only the name of the database. Configure the prompt to display in the following format: `user@database=#`.
5. Configure `psql` to display the execution time for all commands. Make sure that this setting is saved when you restart.

1. When starting `psql`, if you omit the connection parameters, the default values will apply.

2. Use the `\l+` command.

3. The pager program is configured using the `PSQL_PAGER` environment variable. The setting can be configured in the `.psqlrc` file using the `\setenv` command. This will set the value to `'less -XS'` specifically for `psql` sessions, while maintaining the OS default settings in all other cases. By default, `less` wraps long lines during viewing and clears its output upon exit. The `-XS` parameter disables this default behavior.

4. Prompt customization is described in the documentation:

<https://postgrespro.com/docs/postgresql/16/app-psql#APP-PSQL-PROMPTING>

5. The `psql` command to output the duration of a query execution can be found in the PostgreSQL documentation or within `psql` itself with the `\?` command.

## 1. Running psql and Displaying Connection Information

```
student$ psql
```

```
=> \conninfo
```

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

---

## 2. Paging Results Using less

When query output exceeds terminal dimensions, psql sends it to the less pager. You can navigate through query results using standard navigation keys. The h command displays the help file. The q command exits the display mode.

Note that by default less will wrap long lines, which can make the results difficult to read. Besides, the output will be cleared after quitting less.

For example, \l+ output becomes unreadable due to wrapping.

```
=> \l+
```

List of databases

| Name                | Owner                 | Encoding          | Locale Provider | Collate     | Ctype                       | ICU |
|---------------------|-----------------------|-------------------|-----------------|-------------|-----------------------------|-----|
| Locale              | ICU Rules             | Access privileges | Size            | Tablespace  |                             |     |
| Description         |                       |                   |                 |             |                             |     |
| postgres            | postgres              | UTF8              | libc            | en_US.UTF-8 | en_US.UTF-8                 |     |
| connection database |                       |                   | 7361 kB         | pg_default  | default administrative      |     |
| 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           |                   | 7361 kB         | pg_default  | unmodifiable empty database |     |
| template1           | postgres              | UTF8              | libc            | en_US.UTF-8 | en_US.UTF-8                 |     |
|                     | =c/postgres           |                   | 7361 kB         | pg_default  | default template for new    |     |
| databases           |                       |                   |                 |             |                             |     |
|                     | postgres=CtC/postgres |                   |                 |             |                             |     |

(4 rows)

---

## 3. Configuring Page View in .psqlrc

When using the less pager with -XS flags, long lines will not wrap and output remains visible after exiting the less pager. For such configuration, simply set the PSQL\_PAGER environment variable using the \setenv command. We will save this setting in the ~/.psqlrc script:

```
student$ echo "\setenv PSQL_PAGER 'less -XS'" > ~/.psqlrc
```

## 4. Customizing Prompts

To include role information in your prompts, prepend %n@ to both PROMPT1 and PROMPT2 variables.

```
student$ echo "\set PROMPT1 '%n@%/%R%x%# '" >> ~/.psqlrc
```

```
student$ echo "\set PROMPT2 '%n@%/%R%x%# '" >> ~/.psqlrc
```

The PROMPT1 variable controls the primary prompt displayed for the first line of a user's query input. For multi-line queries, PROMPT2 controls the prompt display from the second line onward. While both variables share identical default values, you can configure distinct prompts for initial and following lines. Note that PROMPT3 serves only for COPY command operations.

## 5. Output of SQL Execution Timing

```
student$ echo '\timing on' >> ~/.psqlrc
```

The complete contents of your .psqlrc file will be like this:

```
student$ cat ~/.psqlrc
```

```
\setenv PSQL_PAGER 'less -XS'  
\set PROMPT1 '%n@%/%R%x%# '  
\set PROMPT2 '%n@%/%R%x%# '  
\timing on
```

For the changes to take effect, restart your psql session.

```
=> \q
```

```
student$ psql
```

After restarting, verify the new configuration:

- Prompt (must include role name)
- Displaying detailed information about databases
- Output of command execution time

1. Open a transaction and execute a command that ends with any error. Make sure that no other commands can be executed inside this transaction.
2. Set the `ON_ERROR_ROLLBACK` parameter to `on` and make sure that after the error, you can continue executing commands inside the transaction.

1. To open a transaction, run the command

`BEGIN;`

2. Setting the `ON_ERROR_ROLLBACK` parameter to `ON` causes `psql` to create a `SAVEPOINT` before each SQL command inside an open transaction and, in case of an error, roll back to this savepoint.

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

## 1. psql and In-Transaction Errors

```
student$ psql
```

The psql tool autocommits transactions by default, so each SQL command is executed within a separate transaction. So each SQL command is executed within a separate transaction.

To start a transaction explicitly, the BEGIN command is used:

```
student@student=# BEGIN;
```

```
BEGIN
```

Note that the psql prompt has changed. The asterisk character shows that the transaction is currently open.

```
student@student=*>* CREATE TABLE t (id int);
```

```
CREATE TABLE
```

Consider that we have made a typo in the following command:

```
student@student=*>* INSERTINTO t VALUES(1);
```

```
ERROR:  syntax error at or near "INSERTINTO"
LINE 1: INSERTINTO t VALUES(1);
        ^
```

The asterisk will change to an exclamation mark, indicating an error. Now, rewrite the command:

```
student@student=!*# INSERT INTO t VALUES(1);
```

```
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

But PostgreSQL cannot roll back just a single command, so it terminates and rolls back the whole transaction. To continue, we must send a command that says that the transaction is complete. It can be either COMMIT or ROLLBACK, since the transaction is already cancelled.

```
student@student=!*# COMMIT;
```

```
ROLLBACK
```

Creating the table was cancelled, so there is no such table in the database:

```
student@student=# SELECT * FROM t;
```

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

### The ON\_ERROR\_ROLLBACK Variable

We can change how psql behaves here.

```
student@student=# \set ON_ERROR_ROLLBACK on
```

Now, before every transaction command, there will be a savepoint created. In case of an error, it will roll back to the last savepoint. This way, transaction commands can continue executing.

```
student@student=# BEGIN;
```

```
BEGIN
```

```
student@student=*>* CREATE TABLE t (id int);
```

```
CREATE TABLE
```

```
student@student=*>* INSERTINTO t VALUES(1);
```

```
ERROR:  syntax error at or near "INSERTINTO"
LINE 1: INSERTINTO t VALUES(1);
        ^
```

```
student@student=*>* INSERT INTO t VALUES(1);
```

```
INSERT 0 1
```

```
student@student=*>* COMMIT;
```

```
COMMIT
```

```
student@student=# SELECT * FROM t;
```



```
id
----
1
(1 row)
```

The `ON_ERROR_ROLLBACK` variable can be set to interactive. This will make such behavior work only in the interactive mode, but not when executing scripts.

```
student@student=# DROP TABLE t;
```

```
DROP TABLE
```