

Basic tools Using psql



Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

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.

Launching psql and connecting to the database

Getting help

Working with psql

Configuration

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/13/app-psql.html>

Launch

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

New connection in psql

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

Information about the current connection

```
=> \conninfo
```

When starting psql, you need to specify the connection parameters.

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* — local connection,
- *port* — usually 5432.

To make a new connection without leaving psql, run the `\connect` command.

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

Additional information about connection configuration options:

<https://postgrespro.com/docs/postgresql/13/libpq-envvars.html>

<https://postgrespro.com/docs/postgresql/13/libpq-pgservice.html>

<https://postgrespro.com/docs/postgresql/13/libpq-pgpass.html>

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 <i>command</i>	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` key 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.

Inside psql, you can get a list and a brief description of all psql commands. All psql commands start with a backslash.

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`, used to exit psql. Alternatively, you can also 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've connected to the student database as the student user. You'll learn more about databases and users in later modules.

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

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

- aligned,
- non-aligned,
- extended.

SQL commands, unlike psql ones, may span multiple rows. To send an SQL command, 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)

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

psql commands to switch display modes:

- \a — switches between aligned and non-aligned,
- \t — switches the header and the total row on and off.

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 only 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 the parameters currently set:

```

=> \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

```

Interfacing with the OS

psql can run shell commands:

```

=> \! pwd

/home/student

```

It can set environment variables:

```

=> \setenv TEST Hello

```

```

=> \! echo $TEST

```

Hello

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

```

=> \o dbal_log

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

```

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

```

=> \! cat 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 set the output to back to the screen:

```

=> \o

```

Executing scripts

Another way to run a query is with the \g command. You can specify parameters for this query alone in the brackets. Note the file name to output into at the end of the command.

```

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

```

Here are the file contents:

```

=> \! cat dbal_log

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

```

You can reroute the output to the OS with \g | cmd

We can run the commands using \i[nclude]:

```

=> \i dbal_log

```

```
count
-----
402
(1 row)

count
-----
411
(1 row)

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

Other ways to run a command 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
-----
402
(1 row)

count
-----
411
(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 (integral to `psql`).

Set a variable:

```
=> \set TEST Hi!
```

Use a colon to get its value:

```
=> \echo :TEST
```

```
Hi!
```

Unset a value:

```
=> \unset TEST
```

```
=> \echo :TEST
```

```
:TEST
```

You can set a variable's value as a query output using the `'gset` command:

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

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

```
2024-03-07 13:48:29.961792+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'
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 = '13.11 (Ubuntu 13.11-1.pgdg22.04+1)'
SERVER_VERSION_NUM = '130011'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
SQLSTATE = '00000'
USER = 'student'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 13.11 (Ubuntu 13.11-1.pgdg22.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, 64-bit'
VERSION_NAME = '13.11 (Ubuntu 13.11-1.pgdg22.04+1)'
VERSION_NUM = '130011'
curr_time = '2024-03-07 13:48:29.961792+03'
```

Conditional operators can be used with command files.

For example, let's check if `working_dir` has a value, and if it doesn't, set it as the current directory name. 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}
-- the 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 collect 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, the common script `psqlrc`,
- next, the user configuration `.psqlrc`.

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:

- psql prompt;
- desired page-by-page output viewing application;
- variables to store 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_depend	1130496
pg_proc	1048576
pg_rewrite	704512
pg_attribute	671744
pg_description	581632

(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 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

Practice



1. Run `psql` and check the current connection information.
2. Output all rows of the `pg_tables` table.
3. Set the parameter `less -XS` to display output page by page, then display `pg_tables` contents again.
4. The default prompt shows the name of the database. Configure the prompt to display additional information about the user:
`role@base=#`
5. Configure `psql` to display the execution time for all commands. Make sure that this setting is saved when you restart.

8

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

3. You can set the `PSQL_PAGER` environment variable in the `.psqlrc` file. Use the `\set env` command to set it. This will make the `less -XS` parameter apply to only `psql` output. For all other OS commands, the OS settings will be used (for example, from the `.profile` file).

4. Prompt customization is described in the documentation:

<https://postgrespro.com/docs/postgresql/13/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. pg_tables table

Limit the output to 5 rows. Note that if the records are too wide to fit on the screen, they will be wrapped to new lines. This makes them harder to read.

```
=> SELECT * FROM pg_tables LIMIT 5;
```

schemaname	tablename	tableowner	tablespace	hasindexes	hasrules	hastriggers	rowsecurity
pg_catalog	pg_statistic	postgres		t	f	f	f
pg_catalog	pg_type	postgres		t	f	f	f
pg_catalog	pg_foreign_table	postgres		t	f	f	f
pg_catalog	pg_authid	postgres	pg_global	t	f	f	f
pg_catalog	pg_statistic_ext_data	postgres		t	f	f	f

(5 rows)

3. Configuring page view in .psqlrc

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

When displaying output with the less command, you can use the up, down, left and right keys to scroll. The h command display the help file. The q command exits the display mode.

4. Configuring the prompt

To add role information to the prompt, add %n@ in front of the variables PROMPT1 and PROMPT2.

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

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

The PROMPT1 variable declares the prompt for the first row of the query. If the query spans several rows, every row after the first will start with the prompt defined by PROMPT2. Both variables have the same value by default, but you can configure them differently. The PROMPT3 variable is used only for the COPY command.

5. Displaying SQL command execution times

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

The full .psqlrc file will look 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, you need to relog to psql.

```
=> \q
```

```
student$ psql
```

After relogging, check:

- the prompt (should include the role name),
- how the query for pg_tables is displayed,
- if command execution times are displayed.

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/13/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.

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;
        ^
```

2. 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 checkpoint created. In case of an error, it will roll back to the last checkpoint. 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.