NAME

psql - PostgreSQL interactive terminal

SYNOPSIS

psql [*option*...] [*dbname* [*username*]]

DESCRIPTION

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file or from command line arguments. In addition, psql provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

OPTIONS

-a
--echo-all

Print all nonempty input lines to standard output as they are read. (This does not apply to lines read interactively.) This is equivalent to setting the variable *ECHO* to all.

-A
--no-align

Switches to unaligned output mode. (The default output mode is aligned.) This is equivalent to **\pset format unaligned**.

-b
--echo-errors

Print failed SQL commands to standard error output. This is equivalent to setting the variable *ECHO* to errors.

-c *command*
--command=*command*

Specifies that psql is to execute the given command string, *command*. This option can be repeated and combined in any order with the **-f** option. When either **-c** or **-f** is specified, psql does not read commands from standard input; instead it terminates after processing all the **-c** and **-f** options in sequence.

command must be either a command string that is completely parsable by the server (i.e., it contains no psql-specific features), or a single backslash command. Thus you cannot mix SQL and psql meta-commands within a **-c** option. To achieve that, you could use repeated **-c** options or pipe the string into psql, for example:

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

or

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(\\ is the separator meta-command.)

Each SQL command string passed to **-c** is sent to the server as a single request. Because of this, the server executes it as a single transaction even if the string contains multiple SQL commands, unless there are explicit **BEGIN**/**COMMIT** commands included in the string to divide it into multiple transactions. (See Section 55.2.2.1 for more details about how the server handles multi-query strings.)

If having several commands executed in one transaction is not desired, use repeated **-c** commands or feed multiple commands to psql's standard input, either using echo as illustrated above, or via a shell here-document, for example:

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

--csv
Switches to CSV (Comma-Separated Values) output mode. This is equivalent to **\pset format csv**.

-d *dbname*
--dbname=*dbname*
Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line. The *dbname* can be a connection string. If so, connection string parameters will override any conflicting command line options.

-e
--echo-queries
Copy all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable *ECHO* to queries.

-E
--echo-hidden
Echo the actual queries generated by **\d** and other backslash commands. You can use this to study

psql's internal operations. This is equivalent to setting the variable *ECHO_HIDDEN* to on.

-f *filename*
--file=*filename*
Read commands from the file *filename*, rather than standard input. This option can be repeated and combined in any order with the **-c** option. When either **-c** or **-f** is specified, psql does not read commands from standard input; instead it terminates after processing all the **-c** and **-f** options in sequence. Except for that, this option is largely equivalent to the meta-command **\i**.

If *filename* is - (hyphen), then standard input is read until an EOF indication or **\q** meta-command. This can be used to intersperse interactive input with input from files. Note however that Readline is not used in this case (much as if **-n** had been specified).

Using this option is subtly different from writing psql < *filename*. In general, both will do what you expect, but using -f enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output you would have received had you entered everything by hand.

-F *separator*
--field-separator=*separator*
Use *separator* as the field separator for unaligned output. This is equivalent to **\pset fieldsep** or **\f**.

-h *hostname*
--host=*hostname*
Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

-H
--html
Switches to HTML output mode. This is equivalent to **\pset format html** or the **\H** command.

-l
--list
List all available databases, then exit. Other non-connection options are ignored. This is similar to the meta-command **\list**.

When this option is used, psql will connect to the database postgres, unless a different database is named on the command line (option **-d** or non-option argument, possibly via a service entry, but not via an environment variable).

-L *filename*
--log-file=*filename*
Write all query output into file *filename*, in addition to the normal output destination.

-n
--no-readline
Do not use Readline for line editing and do not use the command history (see the section called "Command-Line Editing" below).

-o *filename*
--output=*filename*
Put all query output into file *filename*. This is equivalent to the command **\o**.

-p *port*
--port=*port*
Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the **PGPORT** environment variable or, if not set, to the port specified at compile time, usually 5432.

-P *assignment*
--pset=*assignment*
Specifies printing options, in the style of **\pset**. Note that here you have to separate name and value with an equal sign instead of a space. For example, to set the output format to LaTeX, you could write -P format=latex.

-q
--quiet
Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the **-c** option. This is equivalent to setting the variable *QUIET* to on.

-R *separator*
--record-separator=*separator*
Use *separator* as the record separator for unaligned output. This is equivalent to **\pset recordsep**.

-s
--single-step
Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

-S
--single-line
Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.

> **Note**
> This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

-t
--tuples-only
Turn off printing of column names and result row count footers, etc. This is equivalent to **\t** or **\pset tuples_only**.

-T *table_options*
--table-attr=*table_options*
Specifies options to be placed within the HTML table tag. See **\pset tableattr** for details.

-U *username*
--username=*username*
Connect to the database as the user *username* instead of the default. (You must have permission to do so, of course.)

-v *assignment*
--set=*assignment*
--variable=*assignment*
Perform a variable assignment, like the **\set** meta-command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To set a variable with an empty value, use the equal sign but leave off the value. These assignments are done during command line processing, so variables that reflect connection state will get overwritten later.

-V
--version
Print the psql version and exit.

-w
--no-password
Never issue a password prompt. If the server requires password authentication and a password is not available from other sources such as a .pgpass file, the connection attempt will fail. This

option can be useful in batch jobs and scripts where no user is present to enter a password.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command **\connect** as well as the initial connection attempt.

-W
--password
Force psql to prompt for a password before connecting to a database, even if the password will not be used.

If the server requires password authentication and a password is not available from other sources such as a .pgpass file, psql will prompt for a password in any case. However, psql will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing **-W** to avoid the extra connection attempt.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command **\connect** as well as the initial connection attempt.

-x
--expanded
Turn on the expanded table formatting mode. This is equivalent to **\x** or **\pset expanded**.

-X
--no-psqlrc
Do not read the start-up file (neither the system-wide psqlrc file nor the user's ~/.psqlrc file).

-z
--field-separator-zero
Set the field separator for unaligned output to a zero byte. This is equivalent to **\pset fieldsep_zero**.

-0
--record-separator-zero
Set the record separator for unaligned output to a zero byte. This is useful for interfacing, for example, with xargs -0. This is equivalent to **\pset recordsep_zero**.

-1
--single-transaction
This option can only be used in combination with one or more **-c** and/or **-f** options. It causes psql to issue a **BEGIN** command before the first such option and a **COMMIT** command after the last one, thereby wrapping all the commands into a single transaction. If any of the commands fails

and the variable *ON_ERROR_STOP* was set, a **ROLLBACK** command is sent instead. This ensures that either all the commands complete successfully, or no changes are applied.

If the commands themselves contain **BEGIN**, **COMMIT**, or **ROLLBACK**, this option will not have the desired effects. Also, if an individual command cannot be executed inside a transaction block, specifying this option will cause the whole transaction to fail.

-?

--help[=*topic*]

Show help about psql and exit. The optional *topic* parameter (defaulting to options) selects which part of psql is explained: commands describes psql's backslash commands; options describes the command-line options that can be passed to psql; and variables shows help about psql configuration variables.

EXIT STATUS

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own occurs (e.g., out of memory, file not found), 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable *ON_ERROR_STOP* was set.

USAGE

Connecting to a Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server, and what user name you want to connect as. psql can be told about those parameters via command line options, namely **-d**, **-h**, **-p**, and **-U** respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all of these options are required; there are useful defaults. If you omit the host name, psql will connect via a Unix-domain socket to a server on the local host, or via TCP/IP to localhost on machines that don't have Unix-domain sockets. The default port number is determined at compile time. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your operating-system user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults aren't quite right, you can save yourself some typing by setting the environment variables **PGDATABASE**, **PGHOST**, **PGPORT** and/or **PGUSER** to appropriate values. (For additional environment variables, see Section 34.15.) It is also convenient to have a ~/.pgpass file to avoid regularly having to type in passwords. See Section 34.16 for more information.

An alternative way to specify connection parameters is in a *conninfo* string or a URI, which is used

instead of a database name. This mechanism give you very wide control over the connection. For example:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

This way you can also use LDAP for connection parameter lookup as described in Section 34.18. See Section 34.1.2 for more information on all the available connection options.

If the connection could not be made for any reason (e.g., insufficient privileges, server is not running on the targeted host, etc.), psql will return an error and terminate.

If both standard input and standard output are a terminal, then psql sets the client encoding to "auto", which will detect the appropriate client encoding from the locale settings (**LC_CTYPE** environment variable on Unix systems). If this doesn't work out as expected, the client encoding can be overridden using the environment variable **PGCLIENTENCODING**.

Entering SQL Commands

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string =>. For example:

```
$ psql testdb
psql (15.8)
Type "help" for help.

testdb=>
```

At the prompt, the user can type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

If untrusted users have access to a database that has not adopted a secure schema usage pattern, begin your session by removing publicly-writable schemas from *search_path*. One can add options=-csearch_path= to the connection string or issue SELECT pg_catalog.set_config('search_path', '', false) before other SQL commands. This consideration is not specific to psql; it applies to every interface for executing arbitrary SQL commands.

Whenever a command is executed, psql also polls for asynchronous notification events generated by **LISTEN** and **NOTIFY**.

While C-style block comments are passed to the server for processing and removal, SQL-standard comments are removed by psql.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands make psql more useful for administration or scripting. Meta-commands are often called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace in an argument you can quote it with single quotes. To include a single quote in an argument, write two single quotes within single-quoted text. Anything contained in single quotes is furthermore subject to C-like substitutions for \n (new line), \t (tab), \b (backspace), \r (carriage return), \f (form feed), *digits* (octal), and \x*digits* (hexadecimal). A backslash preceding any other character within single-quoted text quotes that single character, whatever it is.

If an unquoted colon (:) followed by a psql variable name appears within an argument, it is replaced by the variable's value, as described in SQL Interpolation below. The forms :'*variable_name*' and :"*variable_name*" described there work as well. The :{?*variable_name*} syntax allows testing whether a variable is defined. It is substituted by TRUE or FALSE. Escaping the colon with a backslash protects it from substitution.

Within an argument, text that is enclosed in backquotes (`) is taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) replaces the backquoted text. Within the text enclosed in backquotes, no special quoting or other processing occurs, except that appearances of :*variable_name* where *variable_name* is a psql variable name are replaced by the variable's value. Also, appearances of :'*variable_name*' are replaced by the variable's value suitably quoted to become a single shell command argument. (The latter form is almost always preferable, unless you are very sure of what is in the variable.) Because carriage return and line feed characters cannot be safely quoted on all platforms, the :'*variable_name*' form prints an error message and does not substitute the variable value when such characters appear in the value.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, FOO"BAR"BAZ is interpreted as fooBARbaz, and "A weird"" name" becomes A weird" name.

Parsing for arguments stops at the end of the line, or when another unquoted backslash is found. An unquoted backslash is taken as the beginning of a new meta-command. The special sequence \\ (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

Many of the meta-commands act on the current query buffer. This is simply a buffer holding whatever SQL command text has been typed but not yet sent to the server for execution. This will include previous input lines as well as any text appearing before the meta-command on the same line.

The following meta-commands are defined:

\a

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See **\pset** for a more general solution.

\c or \connect [-reuse-previous=*on*|*off*] [*dbname* [*username*] [*host*] [*port*] | *conninfo*]

Establishes a new connection to a PostgreSQL server. The connection parameters to use can be specified either using a positional syntax (one or more of database name, user, host, and port), or using a *conninfo* connection string as detailed in Section 34.1.1. If no arguments are given, a new connection is made using the same parameters as before.

Specifying any of *dbname*, *username*, *host* or *port* as - is equivalent to omitting that parameter.

The new connection can re-use connection parameters from the previous connection; not only database name, user, host, and port, but other settings such as *sslmode*. By default, parameters are re-used in the positional syntax, but not when a *conninfo* string is given. Passing a first argument of -reuse-previous=on or -reuse-previous=off overrides that default. If parameters are re-used, then any parameter not explicitly specified as a positional parameter or in the *conninfo* string is taken from the existing connection's parameters. An exception is that if the *host* setting is changed from its previous value using the positional syntax, any *hostaddr* setting present in the existing connection's parameters is dropped. Also, any password used for the existing connection will be re-used only if the user, host, and port settings are not changed. When the command neither specifies nor reuses a particular parameter, the libpq default is used.

If the new connection is successfully made, the previous connection is closed. If the connection attempt fails (wrong user name, access denied, etc.), the previous connection will be kept if psql is in interactive mode. But when executing a non-interactive script, the old connection is closed and an error is reported. That may or may not terminate the script; if it does not, all database-accessing

commands will fail until another \connect command is successfully executed. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand. Note that whenever a \connect command attempts to re-use parameters, the values re-used are those of the last successful connection, not of any failed attempts made subsequently. However, in the case of a non-interactive \connect failure, no parameters are allowed to be re-used later, since the script would likely be expecting the values from the failed \connect to be re-used.

Examples:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- changes only sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

\C [*title*]

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to \pset title *title*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

\cd [*directory*]

Changes the current working directory to *directory*. Without argument, changes to the current user's home directory.

Tip

To print your current working directory, use \! pwd.

\conninfo

Outputs information about the current database connection.

\copy { *table* [(*column_list*)] } from { *'filename'* | program *'command'* | stdin | pstdin } [[with] (*option* [, ...])] [where *condition*]
\copy { *table* [(*column_list*)] | (*query*) } to { *'filename'* | program *'command'* | stdout | pstdout } [[with] (*option* [, ...])]

Performs a frontend (client) copy. This is an operation that runs an SQL **COPY** command, but instead of the server reading or writing the specified file, psql reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

When program is specified, *command* is executed by psql and the data passed from or to *command* is routed between the server and the client. Again, the execution privileges are those of the local user, not the server, and no SQL superuser privileges are required.

For \copy ... from stdin, data rows are read from the same source that issued the command, continuing until \. is read or the stream reaches EOF. This option is useful for populating tables in-line within an SQL script file. For \copy ... to stdout, output is sent to the same place as psql command output, and the COPY *count* command status is not printed (since it might be confused with a data row). To read/write psql's standard input or output regardless of the current command source or \o option, write from pstdin or to pstdout.

The syntax of this command is similar to that of the SQL **COPY** command. All options other than the data source/destination are as specified for **COPY**. Because of this, special parsing rules apply to the **\copy** meta-command. Unlike most other meta-commands, the entire remainder of the line is always taken to be the arguments of **\copy**, and neither variable interpolation nor backquote expansion are performed in the arguments.

Tip

Another way to obtain the same result as \copy ... to is to use the SQL COPY ... TO STDOUT command and terminate it with \g *filename* or \g |*program*. Unlike \copy, this method allows the command to span multiple lines; also, variable interpolation and backquote expansion can be used.

Tip

These operations are not as efficient as the SQL **COPY** command with a file or program data source or destination, because all data must pass through the client/server connection. For large amounts of data the SQL command might be preferable. Also, because of this pass-through method, \copy ... from in CSV mode will erroneously treat a \. data value alone on a line as an end-of-input marker.

\copyright

Shows the copyright and distribution terms of PostgreSQL.

\crosstabview [*colV* [*colH* [*colD* [*sortcolH*]]]]

Executes the current query buffer (like \g) and shows the results in a crosstab grid. The query must return at least three columns. The output column identified by *colV* becomes a vertical header and the output column identified by *colH* becomes a horizontal header. *colD* identifies the output column to display within the grid. *sortcolH* identifies an optional sort column for the horizontal header.

Each column specification can be a column number (starting at 1) or a column name. The usual SQL case folding and quoting rules apply to column names. If omitted, *colV* is taken as column 1 and *colH* as column 2. *colH* must differ from *colV*. If *colD* is not specified, then there must be exactly three columns in the query result, and the column that is neither *colV* nor *colH* is taken to be *colD*.

The vertical header, displayed as the leftmost column, contains the values found in column *colV*, in the same order as in the query results, but with duplicates removed.

The horizontal header, displayed as the first row, contains the values found in column *colH*, with duplicates removed. By default, these appear in the same order as in the query results. But if the optional *sortcolH* argument is given, it identifies a column whose values must be integer numbers, and the values from *colH* will appear in the horizontal header sorted according to the corresponding *sortcolH* values.

Inside the crosstab grid, for each distinct value x of *colH* and each distinct value y of *colV*, the cell located at the intersection (x,y) contains the value of the colD column in the query result row for which the value of *colH* is x and the value of *colV* is y. If there is no such row, the cell is empty. If there are multiple such rows, an error is reported.

\d[S+] [*pattern*]

For each relation (table, view, materialized view, index, sequence, or foreign table) or composite type matching the *pattern*, show all columns, their types, the tablespace (if not the default) and any special attributes such as NOT NULL or defaults. Associated indexes, constraints, rules, and triggers are also shown. For foreign tables, the associated foreign server is shown as well. ("Matching the pattern" is defined in Patterns below.)

For some types of relation, \d shows additional information for each column: column values for sequences, indexed expressions for indexes, and foreign data wrapper options for foreign tables.

The command form \d+ is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table, the view definition if the relation is a view, a non-default replica identity setting and the access method name if the relation has an access method.

By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

Note

If **\d** is used without a *pattern* argument, it is equivalent to **\dtvmsE** which will show a list of

all visible tables, views, materialized views, sequences and foreign tables. This is purely a convenience measure.

\da[S] [*pattern*]
Lists aggregate functions, together with their return type and the data types they operate on. If *pattern* is specified, only aggregates whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

\dA[+] [*pattern*]
Lists access methods. If *pattern* is specified, only access methods whose names match the pattern are shown. If + is appended to the command name, each access method is listed with its associated handler function and description.

\dAc[+] [*access-method-pattern* [*input-type-pattern*]]
Lists operator classes (see Section 38.16.1). If *access-method-pattern* is specified, only operator classes associated with access methods whose names match that pattern are listed. If *input-type-pattern* is specified, only operator classes associated with input types whose names match that pattern are listed. If + is appended to the command name, each operator class is listed with its associated operator family and owner.

\dAf[+] [*access-method-pattern* [*input-type-pattern*]]
Lists operator families (see Section 38.16.5). If *access-method-pattern* is specified, only operator families associated with access methods whose names match that pattern are listed. If *input-type-pattern* is specified, only operator families associated with input types whose names match that pattern are listed. If + is appended to the command name, each operator family is listed with its owner.

\dAo[+] [*access-method-pattern* [*operator-family-pattern*]]
Lists operators associated with operator families (see Section 38.16.2). If *access-method-pattern* is specified, only members of operator families associated with access methods whose names match that pattern are listed. If *operator-family-pattern* is specified, only members of operator families whose names match that pattern are listed. If + is appended to the command name, each operator is listed with its sort operator family (if it is an ordering operator).

\dAp[+] [*access-method-pattern* [*operator-family-pattern*]]
Lists support functions associated with operator families (see Section 38.16.3). If *access-method-pattern* is specified, only functions of operator families associated with access methods whose names match that pattern are listed. If *operator-family-pattern* is specified, only functions of operator families whose names match that pattern are listed. If + is appended to the command name, functions are displayed verbosely, with their actual parameter lists.

\db[+] [*pattern*]

Lists tablespaces. If *pattern* is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each tablespace is listed with its associated options, on-disk size, permissions and description.

\dc[S+] [*pattern*]

Lists conversions between character-set encodings. If *pattern* is specified, only conversions whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If + is appended to the command name, each object is listed with its associated description.

\dconfig[+] [*pattern*]

Lists server configuration parameters and their values. If *pattern* is specified, only parameters whose names match the pattern are listed. Without a *pattern*, only parameters that are set to non-default values are listed. (Use \dconfig * to see all parameters.) If + is appended to the command name, each parameter is listed with its data type, context in which the parameter can be set, and access privileges (if non-default access privileges have been granted).

\dC[+] [*pattern*]

Lists type casts. If *pattern* is specified, only casts whose source or target types match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

\dd[S] [*pattern*]

Shows the descriptions of objects of type constraint, operator class, operator family, rule, and trigger. All other comments may be viewed by the respective backslash commands for those object types.

\dd displays descriptions for objects matching the *pattern*, or of visible objects of the appropriate type if no argument is given. But in either case, only objects that have a description are listed. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

Descriptions for objects can be created with the **COMMENT** SQL command.

\dD[S+] [*pattern*]

Lists domains. If *pattern* is specified, only domains whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If + is appended to the command name, each object is listed with its associated permissions and description.

\ddp [*pattern*]

Lists default access privilege settings. An entry is shown for each role (and schema, if applicable) for which the default privilege settings have been changed from the built-in defaults. If *pattern* is specified, only entries whose role name or schema name matches the pattern are listed.

The **ALTER DEFAULT PRIVILEGES** command is used to set default access privileges. The meaning of the privilege display is explained in Section 5.7.

\dE[S+] [*pattern*]
\di[S+] [*pattern*]
\dm[S+] [*pattern*]
\ds[S+] [*pattern*]
\dt[S+] [*pattern*]
\dv[S+] [*pattern*]

In this group of commands, the letters E, i, m, s, t, and v stand for foreign table, index, materialized view, sequence, table, and view, respectively. You can specify any or all of these letters, in any order, to obtain a listing of objects of these types. For example, \dti lists tables and indexes. If + is appended to the command name, each object is listed with its persistence status (permanent, temporary, or unlogged), physical size on disk, and associated description if any. If *pattern* is specified, only objects whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

\des[+] [*pattern*]

Lists foreign servers (mnemonic: "external servers"). If *pattern* is specified, only those servers whose name matches the pattern are listed. If the form \des+ is used, a full description of each server is shown, including the server's access privileges, type, version, options, and description.

\det[+] [*pattern*]

Lists foreign tables (mnemonic: "external tables"). If *pattern* is specified, only entries whose table name or schema name matches the pattern are listed. If the form \det+ is used, generic options and the foreign table description are also displayed.

\deu[+] [*pattern*]

Lists user mappings (mnemonic: "external users"). If *pattern* is specified, only those mappings whose user names match the pattern are listed. If the form \deu+ is used, additional information about each mapping is shown.

Caution

\deu+ might also display the user name and password of the remote user, so care should be taken not to disclose them.

\dew[+] [*pattern*]

Lists foreign-data wrappers (mnemonic: "external wrappers"). If *pattern* is specified, only those foreign-data wrappers whose name matches the pattern are listed. If the form \dew+ is used, the access privileges, options, and description of the foreign-data wrapper are also shown.

\df[anptwS+] [*pattern* [*arg_pattern* ...]]

Lists functions, together with their result data types, argument data types, and function types, which are classified as "agg" (aggregate), "normal", "procedure", "trigger", or "window". To display only functions of specific type(s), add the corresponding letters a, n, p, t, or w to the command. If *pattern* is specified, only functions whose names match the pattern are shown. Any additional arguments are type-name patterns, which are matched to the type names of the first, second, and so on arguments of the function. (Matching functions can have more arguments than what you specify. To prevent that, write a dash - as the last *arg_pattern*.) By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If the form \df+ is used, additional information about each function is shown, including volatility, parallel safety, owner, security classification, access privileges, language, source code and description.

\dF[+] [*pattern*]

Lists text search configurations. If *pattern* is specified, only configurations whose names match the pattern are shown. If the form \dF+ is used, a full description of each configuration is shown, including the underlying text search parser and the dictionary list for each parser token type.

\dFd[+] [*pattern*]

Lists text search dictionaries. If *pattern* is specified, only dictionaries whose names match the pattern are shown. If the form \dFd+ is used, additional information is shown about each selected dictionary, including the underlying text search template and the option values.

\dFp[+] [*pattern*]

Lists text search parsers. If *pattern* is specified, only parsers whose names match the pattern are shown. If the form \dFp+ is used, a full description of each parser is shown, including the underlying functions and the list of recognized token types.

\dFt[+] [*pattern*]

Lists text search templates. If *pattern* is specified, only templates whose names match the pattern are shown. If the form \dFt+ is used, additional information is shown about each template, including the underlying function names.

\dg[S+] [*pattern*]

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles",

this command is now equivalent to \du.) By default, only user-created roles are shown; supply the S modifier to include system roles. If *pattern* is specified, only those roles whose names match the pattern are listed. If the form \dg+ is used, additional information is shown about each role; currently this adds the comment for each role.

\dl[+]

This is an alias for **\lo_list**, which shows a list of large objects. If + is appended to the command name, each large object is listed with its associated permissions, if any.

\dL[S+] [*pattern*]

Lists procedural languages. If *pattern* is specified, only languages whose names match the pattern are listed. By default, only user-created languages are shown; supply the S modifier to include system objects. If + is appended to the command name, each language is listed with its call handler, validator, access privileges, and whether it is a system object.

\dn[S+] [*pattern*]

Lists schemas (namespaces). If *pattern* is specified, only schemas whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If + is appended to the command name, each object is listed with its associated permissions and description, if any.

\do[S+] [*pattern* [*arg_pattern* [*arg_pattern*]]]

Lists operators with their operand and result types. If *pattern* is specified, only operators whose names match the pattern are listed. If one *arg_pattern* is specified, only prefix operators whose right argument's type name matches that pattern are listed. If two *arg_pattern*s are specified, only binary operators whose argument type names match those patterns are listed. (Alternatively, write - for the unused argument of a unary operator.) By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If + is appended to the command name, additional information about each operator is shown, currently just the name of the underlying function.

\dO[S+] [*pattern*]

Lists collations. If *pattern* is specified, only collations whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects. If + is appended to the command name, each collation is listed with its associated description, if any. Note that only collations usable with the current database's encoding are shown, so the results may vary in different databases of the same installation.

\dp [*pattern*]

Lists tables, views and sequences with their associated access privileges. If *pattern* is specified,

only tables, views and sequences whose names match the pattern are listed.

The **GRANT** and **REVOKE** commands are used to set access privileges. The meaning of the privilege display is explained in Section 5.7.

\dP[itn+] [*pattern*]

Lists partitioned relations. If *pattern* is specified, only entries whose name matches the pattern are listed. The modifiers t (tables) and i (indexes) can be appended to the command, filtering the kind of relations to list. By default, partitioned tables and indexes are listed.

If the modifier n ("nested") is used, or a pattern is specified, then non-root partitioned relations are included, and a column is shown displaying the parent of each partitioned relation.

If + is appended to the command name, the sum of the sizes of each relation's partitions is also displayed, along with the relation's description. If n is combined with +, two sizes are shown: one including the total size of directly-attached leaf partitions, and another showing the total size of all partitions, including indirectly attached sub-partitions.

\drds [*role-pattern* [*database-pattern*]]

Lists defined configuration settings. These settings can be role-specific, database-specific, or both. *role-pattern* and *database-pattern* are used to select specific roles and databases to list, respectively. If omitted, or if * is specified, all settings are listed, including those not role-specific or database-specific, respectively.

The **ALTER ROLE** and **ALTER DATABASE** commands are used to define per-role and per-database configuration settings.

\dRp[+] [*pattern*]

Lists replication publications. If *pattern* is specified, only those publications whose names match the pattern are listed. If + is appended to the command name, the tables and schemas associated with each publication are shown as well.

\dRs[+] [*pattern*]

Lists replication subscriptions. If *pattern* is specified, only those subscriptions whose names match the pattern are listed. If + is appended to the command name, additional properties of the subscriptions are shown.

\dT[S+] [*pattern*]

Lists data types. If *pattern* is specified, only types whose names match the pattern are listed. If + is appended to the command name, each type is listed with its internal name and size, its allowed

values if it is an enum type, and its associated permissions. By default, only user-created objects are shown; supply a pattern or the S modifier to include system objects.

\du[S+] [*pattern*]

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to \dg.) By default, only user-created roles are shown; supply the S modifier to include system roles. If *pattern* is specified, only those roles whose names match the pattern are listed. If the form \du+ is used, additional information is shown about each role; currently this adds the comment for each role.

\dx[+] [*pattern*]

Lists installed extensions. If *pattern* is specified, only those extensions whose names match the pattern are listed. If the form \dx+ is used, all the objects belonging to each matching extension are listed.

\dX [*pattern*]

Lists extended statistics. If *pattern* is specified, only those extended statistics whose names match the pattern are listed.

The status of each kind of extended statistics is shown in a column named after its statistic kind (e.g. Ndistinct). defined means that it was requested when creating the statistics, and NULL means it wasn't requested. You can use pg_stats_ext if you'd like to know whether **ANALYZE** was run and statistics are available to the planner.

\dy[+] [*pattern*]

Lists event triggers. If *pattern* is specified, only those event triggers whose names match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

\e or \edit [*filename*] [*line_number*]

If *filename* is specified, the file is edited; after the editor exits, the file's content is copied into the current query buffer. If no *filename* is given, the current query buffer is copied to a temporary file which is then edited in the same fashion. Or, if the current query buffer is empty, the most recently executed query is copied to a temporary file and edited in the same fashion.

If you edit a file or the previous query, and you quit the editor without modifying the file, the query buffer is cleared. Otherwise, the new contents of the query buffer are re-parsed according to the normal rules of psql, treating the whole buffer as a single line. Any complete queries are immediately executed; that is, if the query buffer contains or ends with a semicolon, everything up to that point is executed and removed from the query buffer. Whatever remains in the query buffer

is redisplayed. Type semicolon or \g to send it, or \r to cancel it by clearing the query buffer.

Treating the buffer as a single line primarily affects meta-commands: whatever is in the buffer after a meta-command will be taken as argument(s) to the meta-command, even if it spans multiple lines. (Thus you cannot make meta-command-using scripts this way. Use **\i** for that.)

If a line number is specified, psql will position the cursor on the specified line of the file or query buffer. Note that if a single all-digits argument is given, psql assumes it is a line number, not a file name.

Tip
See Environment, below, for how to configure and customize your editor.

\echo *text* [...]
Prints the evaluated arguments to standard output, separated by spaces and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

=> **\echo ‘date‘**
Tue Oct 26 21:40:57 CEST 1999

If the first argument is an unquoted -n the trailing newline is not written (nor is the first argument).

Tip
If you use the **\o** command to redirect your query output you might wish to use **\qecho** instead of this command. See also **\warn**.

\ef [*function_description* [*line_number*]]
This command fetches and edits the definition of the named function or procedure, in the form of a **CREATE OR REPLACE FUNCTION** or **CREATE OR REPLACE PROCEDURE** command. Editing is done in the same way as for \edit. If you quit the editor without saving, the statement is discarded. If you save and exit the editor, the updated command is executed immediately if you added a semicolon to it. Otherwise it is redisplayed; type semicolon or \g to send it, or \r to cancel.

The target function can be specified by name alone, or by name and arguments, for example foo(integer, text). The argument types must be given if there is more than one function of the same name.

If no function is specified, a blank **CREATE FUNCTION** template is presented for editing.

If a line number is specified, psql will position the cursor on the specified line of the function

body. (Note that the function body typically does not begin on the first line of the file.)

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\ef**, and neither variable interpolation nor backquote expansion are performed in the arguments.

Tip
See Environment, below, for how to configure and customize your editor.

\encoding [*encoding*]
Sets the client character set encoding. Without an argument, this command shows the current encoding.

\errverbose
Repeats the most recent server error message at maximum verbosity, as though *VERBOSITY* were set to verbose and *SHOW_CONTEXT* were set to always.

\ev [*view_name* [*line_number*]]
This command fetches and edits the definition of the named view, in the form of a **CREATE OR REPLACE VIEW** command. Editing is done in the same way as for \edit. If you quit the editor without saving, the statement is discarded. If you save and exit the editor, the updated command is executed immediately if you added a semicolon to it. Otherwise it is redisplayed; type semicolon or \g to send it, or \r to cancel.

If no view is specified, a blank **CREATE VIEW** template is presented for editing.

If a line number is specified, psql will position the cursor on the specified line of the view definition.

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\ev**, and neither variable interpolation nor backquote expansion are performed in the arguments.

\f [*string*]
Sets the field separator for unaligned query output. The default is the vertical bar (|). It is equivalent to **\pset fieldsep**.

\g [(*option*=*value* [...])] [*filename*]
\g [(*option*=*value* [...])] [|*command*]
Sends the current query buffer to the server for execution.

If parentheses appear after \g, they surround a space-separated list of *option=value* formatting-option clauses, which are interpreted in the same way as \pset *option value* commands, but take effect only for the duration of this query. In this list, spaces are not allowed around = signs, but are required between option clauses. If =*value* is omitted, the named *option* is changed in the same way as for \pset *option* with no explicit *value*.

If a *filename* or |*command* argument is given, the query's output is written to the named file or piped to the given shell command, instead of displaying it as usual. The file or command is written to only if the query successfully returns zero or more tuples, not if the query fails or is a non-data-returning SQL command.

If the current query buffer is empty, the most recently sent query is re-executed instead. Except for that behavior, \g without any arguments is essentially equivalent to a semicolon. With arguments, \g provides a "one-shot" alternative to the **\o** command, and additionally allows one-shot adjustments of the output formatting options normally set by \pset.

When the last argument begins with |, the entire remainder of the line is taken to be the *command* to execute, and neither variable interpolation nor backquote expansion are performed in it. The rest of the line is simply passed literally to the shell.

\gdesc

Shows the description (that is, the column names and data types) of the result of the current query buffer. The query is not actually executed; however, if it contains some type of syntax error, that error will be reported in the normal way.

If the current query buffer is empty, the most recently sent query is described instead.

\getenv *psql_var env_var*

Gets the value of the environment variable *env_var* and assigns it to the psql variable *psql_var*. If *env_var* is not defined in the psql process's environment, *psql_var* is not changed. Example:

```
=> \getenv home HOME
=> \echo :home
/home/postgres
```

\gexec

Sends the current query buffer to the server, then treats each column of each row of the query's output (if any) as an SQL statement to be executed. For example, to create an index on each column of my_table:

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

The generated queries are executed in the order in which the rows are returned, and left-to-right within each row if there is more than one column. NULL fields are ignored. The generated queries are sent literally to the server for processing, so they cannot be psql meta-commands nor contain psql variable references. If any individual query fails, execution of the remaining queries continues unless *ON_ERROR_STOP* is set. Execution of each query is subject to *ECHO* processing. (Setting *ECHO* to all or queries is often advisable when using **gexec**.) Query logging, single-step mode, timing, and other query execution features apply to each generated query as well.

If the current query buffer is empty, the most recently sent query is re-executed instead.

\gset [*prefix*]

Sends the current query buffer to the server and stores the query's output into psql variables (see Variables below). The query to be executed must return exactly one row. Each column of the row is stored into a separate variable, named the same as the column. For example:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

If you specify a *prefix*, that string is prepended to the query's column names to create the variable names to use:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

If a column result is NULL, the corresponding variable is unset rather than being set.

If the query fails or does not return one row, no variables are changed.

If the current query buffer is empty, the most recently sent query is re-executed instead.

\gx [(*option*=*value* [...])] [*filename*]
\gx [(*option*=*value* [...])] [|*command*]
\gx is equivalent to \g, except that it forces expanded output mode for this query, as if expanded=on were included in the list of \pset options. See also \x.

\h or \help [*command*]
Gives syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk (*), then syntax help on all SQL commands is shown.

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\help**, and neither variable interpolation nor backquote expansion are performed in the arguments.

Note
To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **\help alter table**.

\H or \html
Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see **\pset** about setting other output options.

\i or \include *filename*
Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

If *filename* is - (hyphen), then standard input is read until an EOF indication or **\q** meta-command. This can be used to intersperse interactive input with input from files. Note that Readline behavior will be used only if it is active at the outermost level.

Note
If you want to see the lines on the screen as they are read you must set the variable *ECHO* to all.

\if *expression*
\elif *expression*

\else
\endif

This group of commands implements nestable conditional blocks. A conditional block must begin with an **\if** and end with an **\endif**. In between there may be any number of **\elif** clauses, which may optionally be followed by a single **\else** clause. Ordinary queries and other types of backslash commands may (and usually do) appear between the commands forming a conditional block.

The **\if** and **\elif** commands read their argument(s) and evaluate them as a Boolean expression. If the expression yields true then processing continues normally; otherwise, lines are skipped until a matching **\elif**, **\else**, or **\endif** is reached. Once an **\if** or **\elif** test has succeeded, the arguments of later **\elif** commands in the same block are not evaluated but are treated as false. Lines following an **\else** are processed only if no earlier matching **\if** or **\elif** succeeded.

The *expression* argument of an **\if** or **\elif** command is subject to variable interpolation and backquote expansion, just like any other backslash command argument. After that it is evaluated like the value of an on/off option variable. So a valid value is any unambiguous case-insensitive match for one of: true, false, 1, 0, on, off, yes, no. For example, t, T, and tR will all be considered to be true.

Expressions that do not properly evaluate to true or false will generate a warning and be treated as false.

Lines being skipped are parsed normally to identify queries and backslash commands, but queries are not sent to the server, and backslash commands other than conditionals (**\if**, **\elif**, **\else**, **\endif**) are ignored. Conditional commands are checked only for valid nesting. Variable references in skipped lines are not expanded, and backquote expansion is not performed either.

All the backslash commands of a given conditional block must appear in the same source file. If EOF is reached on the main input file or an **\include**-ed file before all local **\if**-blocks have been closed, then psql will raise an error.

Here is an example:

```
-- check for the existence of two separate records in the database and store
-- the results in separate psql variables
SELECT
   EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
   EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
```

```
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

\ir or \include_relative *filename*

The \ir command is similar to \i, but resolves relative file names differently. When executing in interactive mode, the two commands behave identically. However, when invoked from a script, \ir interprets file names relative to the directory in which the script is located, rather than the current working directory.

\l[+] or \list[+] [*pattern*]

List the databases in the server and show their names, owners, character set encodings, and access privileges. If *pattern* is specified, only databases whose names match the pattern are listed. If + is appended to the command name, database sizes, default tablespaces, and descriptions are also displayed. (Size information is only available for databases that the current user can connect to.)

\lo_export *loid filename*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function **lo_export**, which acts with the permissions of the user that the database server runs as and on the server's file system.

Tip

Use **\lo_list** to find out the large object's OID.

\lo_import *filename* [*comment*]

Stores the file into a PostgreSQL large object. Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801, which can be used to

access the newly-created large object in the future. For the sake of readability, it is recommended to always associate a human-readable comment with every object. Both OIDs and comments can be viewed with the **\lo_list** command.

Note that this command is subtly different from the server-side **lo_import** because it acts as the local user on the local file system, rather than the server's user and file system.

\lo_list[+]

Shows a list of all PostgreSQL large objects currently stored in the database, along with any comments provided for them. If + is appended to the command name, each large object is listed with its associated permissions, if any.

\lo_unlink *loid*

Deletes the large object with OID *loid* from the database.

Tip

Use **\lo_list** to find out the large object's OID.

\o or \out [*filename*]
\o or \out [|*command*]

Arranges to save future query results to the file *filename* or pipe future results to the shell command *command*. If no argument is specified, the query output is reset to the standard output.

If the argument begins with |, then the entire remainder of the line is taken to be the *command* to execute, and neither variable interpolation nor backquote expansion are performed in it. The rest of the line is simply passed literally to the shell.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as **\d**); but not error messages.

Tip

To intersperse text output in between query results, use **\qecho**.

\p or \print

Print the current query buffer to the standard output. If the current query buffer is empty, the most recently executed query is printed instead.

\password [*username*]

Changes the password of the specified user (by default, the current user). This command prompts

for the new password, encrypts it, and sends it to the server as an **ALTER ROLE** command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

\prompt [*text*] *name*

Prompts the user to supply text, which is assigned to the variable *name*. An optional prompt string, *text*, can be specified. (For multiword prompts, surround the text with single quotes.)

By default, \prompt uses the terminal for input and output. However, if the **-f** command line switch was used, \prompt uses standard input and standard output.

\pset [*option* [*value*]]

This command sets options affecting the output of query result tables. *option* indicates which option is to be set. The semantics of *value* vary depending on the selected option. For some options, omitting *value* causes the option to be toggled or unset, as described under the particular option. If no such behavior is mentioned, then omitting *value* just results in the current setting being displayed.

pset without any arguments displays the current status of all printing options.

Adjustable printing options are:

border

The *value* must be a number. In general, the higher the number the more borders and lines the tables will have, but details depend on the particular format. In HTML format, this will translate directly into the border=... attribute. In most other formats only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense, and values above 2 will be treated the same as border = 2. The latex and latex-longtable formats additionally allow a value of 3 to add dividing lines between data rows.

columns

Sets the target width for the wrapped format, and also the width limit for determining whether output is wide enough to require the pager or switch to the vertical display in expanded auto mode. Zero (the default) causes the target width to be controlled by the environment variable **COLUMNS**, or the detected screen width if **COLUMNS** is not set. In addition, if columns is zero then the wrapped format only affects screen output. If columns is nonzero then file and pipe output is wrapped to that width as well.

csv_fieldsep

Specifies the field separator to be used in CSV output format. If the separator character

appears in a field’s value, that field is output within double quotes, following standard CSV rules. The default is a comma.

expanded (or x)
: If *value* is specified it must be either on or off, which will enable or disable expanded mode, or auto. If *value* is omitted the command toggles between the on and off settings. When expanded mode is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn’t fit on the screen in the normal "horizontal" mode. In the auto setting, the expanded mode is used whenever the query output has more than one column and is wider than the screen; otherwise, the regular mode is used. The auto setting is only effective in the aligned and wrapped formats. In other formats, it always behaves as if the expanded mode is off.

fieldsep
: Specifies the field separator to be used in unaligned output format. That way one can create, for example, tab-separated output, which other programs might prefer. To set a tab as field separator, type \pset fieldsep ’\t’. The default field separator is ’|’ (a vertical bar).

fieldsep_zero
: Sets the field separator to use in unaligned output format to a zero byte.

footer
: If *value* is specified it must be either on or off which will enable or disable display of the table footer (the (*n* rows) count). If *value* is omitted the command toggles footer display on or off.

format
: Sets the output format to one of aligned, asciidoc, csv, html, latex, latex-longtable, troff-ms, unaligned, or wrapped. Unique abbreviations are allowed.

  aligned format is the standard, human-readable, nicely formatted text output; this is the default.

  unaligned format writes all columns of a row on one line, separated by the currently active field separator. This is useful for creating output that might be intended to be read in by other programs, for example, tab-separated or comma-separated format. However, the field separator character is not treated specially if it appears in a column’s value; so CSV format may be better suited for such purposes.

  csv format

writes column values separated by commas, applying the quoting rules described in **RFC 4180**. This output is compatible with the CSV format of the server's **COPY** command. A header line with column names is generated unless the tuples_only parameter is on. Titles and footers are not printed. Each row is terminated by the system-dependent end-of-line character, which is typically a single newline (\n) for Unix-like systems or a carriage return and newline sequence (\r\n) for Microsoft Windows. Field separator characters other than comma can be selected with **\pset csv_fieldsep**.

wrapped format is like aligned but wraps wide data values across lines to make the output fit in the target column width. The target width is determined as described under the columns option. Note that psql will not attempt to wrap column header titles; therefore, wrapped format behaves the same as aligned if the total width needed for column headers exceeds the target.

The asciidoc, html, latex, latex-longtable, and troff-ms formats put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! This might not be necessary in HTML, but in LaTeX you must have a complete document wrapper. The latex format uses LaTeX's tabular environment. The latex-longtable format requires the LaTeX longtable and booktabs packages.

linestyle

Sets the border line drawing style to one of ascii, old-ascii, or unicode. Unique abbreviations are allowed. (That would mean one letter is enough.) The default setting is ascii. This option only affects the aligned and wrapped output formats.

ascii style uses plain ASCII characters. Newlines in data are shown using a + symbol in the right-hand margin. When the wrapped format wraps data from one line to the next without a newline character, a dot (.) is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

old-ascii style uses plain ASCII characters, using the formatting style used in PostgreSQL 8.4 and earlier. Newlines in data are shown using a : symbol in place of the left-hand column separator. When the data is wrapped from one line to the next without a newline character, a ; symbol is used in place of the left-hand column separator.

unicode style uses Unicode box-drawing characters. Newlines in data are shown using a carriage return symbol in the right-hand margin. When the data is wrapped from one line to the next without a newline character, an ellipsis symbol is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

When the border setting is greater than zero, the linestyle option also determines the characters with which the border lines are drawn. Plain ASCII characters work everywhere, but Unicode characters look nicer on displays that recognize them.

null
Sets the string to be printed in place of a null value. The default is to print nothing, which can easily be mistaken for an empty string. For example, one might prefer \pset null '(null)'.

numericlocale
If *value* is specified it must be either on or off which will enable or disable display of a locale-specific character to separate groups of digits to the left of the decimal marker. If *value* is omitted the command toggles between regular and locale-specific numeric output.

pager
Controls use of a pager program for query and psql help output. When the pager option is off, the pager program is not used. When the pager option is on, the pager is used when appropriate, i.e., when the output is to a terminal and will not fit on the screen. The pager option can also be set to always, which causes the pager to be used for all terminal output regardless of whether it fits on the screen. \pset pager without a *value* toggles pager use on and off.

If the environment variable **PSQL_PAGER** or **PAGER** is set, output to be paged is piped to the specified program. Otherwise a platform-dependent default program (such as more) is used.

When using the \watch command to execute a query repeatedly, the environment variable **PSQL_WATCH_PAGER** is used to find the pager program instead, on Unix systems. This is configured separately because it may confuse traditional pagers, but can be used to send output to tools that understand psql's output format (such as pspg --stream).

pager_min_lines
If pager_min_lines is set to a number greater than the page height, the pager program will not be called unless there are at least this many lines of output to show. The default setting is 0.

recordsep
Specifies the record (line) separator to use in unaligned output format. The default is a newline character.

recordsep_zero
Sets the record separator to use in unaligned output format to a zero byte.

tableattr (or T)
: In HTML format, this specifies attributes to be placed inside the table tag. This could for example be cellpadding or bgcolor. Note that you probably don't want to specify border here, as that is already taken care of by \pset border. If no *value* is given, the table attributes are unset.

  In latex-longtable format, this controls the proportional width of each column containing a left-aligned data type. It is specified as a whitespace-separated list of values, e.g., '0.2 0.2 0.6'. Unspecified output columns use the last specified value.

title (or C)
: Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no *value* is given, the title is unset.

tuples_only (or t)
: If *value* is specified it must be either on or off which will enable or disable tuples-only mode. If *value* is omitted the command toggles between regular and tuples-only output. Regular output includes extra information such as column headers, titles, and various footers. In tuples-only mode, only actual table data is shown.

unicode_border_linestyle
: Sets the border drawing style for the unicode line style to one of single or double.

unicode_column_linestyle
: Sets the column drawing style for the unicode line style to one of single or double.

unicode_header_linestyle
: Sets the header drawing style for the unicode line style to one of single or double.

Illustrations of how these different formats look can be seen in Examples, below.

Tip

There are various shortcut commands for **\pset**. See **\a**, **\C**, **\f**, **\H**, **\t**, **\T**, and **\x**.

\q or \quit
: Quits the psql program. In a script file, only execution of that script is terminated.

\qecho *text* [...]
: This command is identical to **\echo** except that the output will be written to the query output channel, as set by **\o**.

\r or \reset

Resets (clears) the query buffer.

\s [*filename*]

Print psql's command line history to *filename*. If *filename* is omitted, the history is written to the standard output (using the pager if appropriate). This command is not available if psql was built without Readline support.

\set [*name* [*value* [...]]]

Sets the psql variable *name* to *value*, or if more than one value is given, to the concatenation of all of them. If only one argument is given, the variable is set to an empty-string value. To unset a variable, use the **\unset** command.

\set without any arguments displays the names and values of all currently-set psql variables.

Valid variable names can contain letters, digits, and underscores. See Variables below for details. Variable names are case-sensitive.

Certain variables are special, in that they control psql's behavior or are automatically set to reflect connection state. These variables are documented in Variables, below.

Note

This command is unrelated to the SQL command **SET**.

\setenv *name* [*value*]

Sets the environment variable *name* to *value*, or if the *value* is not supplied, unsets the environment variable. Example:

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

\sf[+] *function_description*

This command fetches and shows the definition of the named function or procedure, in the form of a **CREATE OR REPLACE FUNCTION** or **CREATE OR REPLACE PROCEDURE** command. The definition is printed to the current query output channel, as set by **\o**.

The target function can be specified by name alone, or by name and arguments, for example foo(integer, text). The argument types must be given if there is more than one function of the same name.

If + is appended to the command name, then the output lines are numbered, with the first line of the function body being line 1.

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\sf**, and neither variable interpolation nor backquote expansion are performed in the arguments.

\sv[+] *view_name*

This command fetches and shows the definition of the named view, in the form of a **CREATE OR REPLACE VIEW** command. The definition is printed to the current query output channel, as set by **\o**.

If + is appended to the command name, then the output lines are numbered from 1.

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\sv**, and neither variable interpolation nor backquote expansion are performed in the arguments.

\t

Toggles the display of output column name headings and row count footer. This command is equivalent to \pset tuples_only and is provided for convenience.

\T *table_options*

Specifies attributes to be placed within the table tag in HTML output format. This command is equivalent to \pset tableattr *table_options*.

\timing [*on* | *off*]

With a parameter, turns displaying of how long each SQL statement takes on or off. Without a parameter, toggles the display between on and off. The display is in milliseconds; intervals longer than 1 second are also shown in minutes:seconds format, with hours and days fields added if needed.

\unset *name*

Unsets (deletes) the psql variable *name*.

Most variables that control psql's behavior cannot be unset; instead, an \unset command is interpreted as setting them to their default values. See Variables below.

\w or \write *filename*
\w or \write |*command*

Writes the current query buffer to the file *filename* or pipes it to the shell command *command*. If the current query buffer is empty, the most recently executed query is written instead.

If the argument begins with |, then the entire remainder of the line is taken to be the *command* to execute, and neither variable interpolation nor backquote expansion are performed in it. The rest of the line is simply passed literally to the shell.

\warn *text* [...]

This command is identical to **\echo** except that the output will be written to psql's standard error channel, rather than standard output.

\watch [*seconds*]

Repeatedly execute the current query buffer (as \g does) until interrupted or the query fails. Wait the specified number of seconds (default 2) between executions. Each query result is displayed with a header that includes the \pset title string (if any), the time as of query start, and the delay interval.

If the current query buffer is empty, the most recently sent query is re-executed instead.

\x [*on* | *off* | *auto*]

Sets or toggles expanded table formatting mode. As such it is equivalent to \pset expanded.

\z [*pattern*]

Lists tables, views and sequences with their associated access privileges. If a *pattern* is specified, only tables, views and sequences whose names match the pattern are listed.

This is an alias for **\dp** ("display privileges").

\! [*command*]

With no argument, escapes to a sub-shell; psql resumes when the sub-shell exits. With an argument, executes the shell command *command*.

Unlike most other meta-commands, the entire remainder of the line is always taken to be the argument(s) of **\!**, and neither variable interpolation nor backquote expansion are performed in the arguments. The rest of the line is simply passed literally to the shell.

\? [*topic*]

Shows help information. The optional *topic* parameter (defaulting to commands) selects which part of psql is explained: commands describes psql's backslash commands; options describes the command-line options that can be passed to psql; and variables shows help about psql

configuration variables.

\;

Backslash-semicolon is not a meta-command in the same way as the preceding commands; rather, it simply causes a semicolon to be added to the query buffer without any further processing.

Normally, psql will dispatch an SQL command to the server as soon as it reaches the command-ending semicolon, even if more input remains on the current line. Thus for example entering

```
select 1; select 2; select 3;
```

will result in the three SQL commands being individually sent to the server, with each one's results being displayed before continuing to the next command. However, a semicolon entered as \; will not trigger command processing, so that the command before it and the one after are effectively combined and sent to the server in one request. So for example

```
select 1\; select 2\; select 3;
```

results in sending the three SQL commands to the server in a single request, when the non-backslashed semicolon is reached. The server executes such a request as a single transaction, unless there are explicit **BEGIN**/**COMMIT** commands included in the string to divide it into multiple transactions. (See Section 55.2.2.1 for more details about how the server handles multi-query strings.)

Patterns

The various \d commands accept a *pattern* parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, \dt FOO will display the table named foo. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, \dt "FOO""BAR" will display the table named FOO"BAR (not foo"bar). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance \dt FOO"FOO"BAR will display the table named fooFOObar.

Whenever the *pattern* parameter is omitted completely, the \d commands display all objects that are visible in the current schema search path -- this is equivalent to using * as the pattern. (An object is said to be visible if its containing schema is in the search path and no object of the same

kind and name appears earlier in the search path. This is equivalent to the statement that the object can be referenced by name without explicit schema qualification.) To see all objects in the database regardless of visibility, use *.* as the pattern.

Within a pattern, * matches any sequence of characters (including no characters) and ? matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, \dt int* displays tables whose names begin with int. But within double quotes, * and ? lose these special meanings and are just matched literally.

A relation pattern that contains a dot (.) is interpreted as a schema name pattern followed by an object name pattern. For example, \dt foo*.*bar* displays all tables whose table name includes bar that are in schemas whose schema name starts with foo. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally. A relation pattern that contains two dots (.) is interpreted as a database name followed by a schema name pattern followed by an object name pattern. The database name portion will not be treated as a pattern and must match the name of the currently connected database, else an error will be raised.

A schema pattern that contains a dot (.) is interpreted as a database name followed by a schema name pattern. For example, \dn mydb.*foo* displays all schemas whose schema name includes foo. The database name portion will not be treated as a pattern and must match the name of the currently connected database, else an error will be raised.

Advanced users can use regular-expression notations such as character classes, for example [0-9] to match any digit. All regular expression special characters work as specified in Section 9.7.3, except for . which is taken as a separator as mentioned above, * which is translated to the regular-expression notation .*, ? which is translated to ., and $ which is matched literally. You can emulate these pattern characters at need by writing ? for ., (*R*+|) for *R**, or (*R*|) for *R*?. $ is not needed as a regular-expression character since the pattern must match the whole name, unlike the usual interpretation of regular expressions (in other words, $ is automatically appended to your pattern). Write * at the beginning and/or end if you don’t wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (i.e., the argument of \do).

Advanced Features

Variables

psql provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. The name must

consist of letters (including non-Latin letters), digits, and underscores.

To set a variable, use the psql meta-command **\set**. For example,

```
testdb=> \set foo bar
```

sets the variable foo to the value bar. To retrieve the content of the variable, precede the name with a colon, for example:

```
testdb=> \echo :foo
bar
```

This works in both regular SQL commands and meta-commands; there is more detail in SQL Interpolation, below.

If you call **\set** without a second argument, the variable is set to an empty-string value. To unset (i.e., delete) a variable, use the command **\unset**. To show the values of all variables, call **\set** without any argument.

Note

The arguments of **\set** are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as \set :foo 'something' and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, \set bar :foo is a perfectly valid way to copy a variable.

A number of these variables are treated specially by psql. They represent certain option settings that can be changed at run time by altering the value of the variable, or in some cases represent changeable state of psql. By convention, all specially treated variables' names consist of all upper-case ASCII letters (and possibly digits and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes.

Variables that control psql's behavior generally cannot be unset or set to invalid values. An \unset command is allowed but is interpreted as setting the variable to its default value. A \set command without a second argument is interpreted as setting the variable to on, for control variables that accept that value, and is rejected for others. Also, control variables that accept the values on and off will also accept other common spellings of Boolean values, such as true and false.

The specially treated variables are:

AUTOCOMMIT

When on (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a **BEGIN** or **START TRANSACTION** SQL command. When off or unset, SQL commands are not committed until you explicitly issue **COMMIT** or **END**. The autocommit-off mode works by issuing an implicit **BEGIN** for you, just before any command that is not already in a transaction block and is not itself a **BEGIN** or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as **VACUUM**).

Note

In autocommit-off mode, you must explicitly abandon any failed transaction by entering **ABORT** or **ROLLBACK**. Also keep in mind that if you exit the session without committing, your work will be lost.

Note

The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you might wish to set it in the system-wide psqlrc file or your ~/.psqlrc file.

COMP_KEYWORD_CASE

Determines which letter case to use when completing an SQL key word. If set to lower or upper, the completed word will be in lower or upper case, respectively. If set to preserve-lower or preserve-upper (the default), the completed word will be in the case of the word already entered, but words being completed without anything entered will be in lower or upper case, respectively.

DBNAME

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be changed or unset.

ECHO

If set to all, all nonempty input lines are printed to standard output as they are read. (This does not apply to lines read interactively.) To select this behavior on program start-up, use the switch **-a**. If set to queries, psql prints each query to standard output as it is sent to the server. The switch to select this behavior is **-e**. If set to errors, then only failed queries are displayed on standard error output. The switch for this behavior is **-b**. If set to none (the default), then no queries are displayed.

ECHO_HIDDEN

When this variable is set to on and a backslash command queries the database, the query is

first shown. This feature helps you to study PostgreSQL internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch **-E**.) If you set this variable to the value noexec, the queries are just shown but are not actually sent to the server and executed. The default value is off.

ENCODING
The current client character set encoding. This is set every time you connect to a database (including program start-up), and when you change the encoding with \encoding, but it can be changed or unset.

ERROR
true if the last SQL query failed, false if it succeeded. See also *SQLSTATE*.

FETCH_COUNT
If this variable is set to an integer value greater than zero, the results of **SELECT** queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query might fail after having already displayed some rows.

Tip
Although you can use any output format with this feature, the default aligned format tends to look bad because each group of *FETCH_COUNT* rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

HIDE_TABLEAM
If this variable is set to true, a table’s access method details are not displayed. This is mainly useful for regression tests.

HIDE_TOAST_COMPRESSION
If this variable is set to true, column compression method details are not displayed. This is mainly useful for regression tests.

HISTCONTROL
If this variable is set to ignorespace, lines which begin with a space are not entered into the history list. If set to a value of ignoredups, lines matching the previous history line are not entered. A value of ignoreboth combines the two options. If set to none (the default), all lines read in interactive mode are saved on the history list.

Note
This feature was shamelessly plagiarized from Bash.

HISTFILE
The file name that will be used to store the history list. If unset, the file name is taken from the **PSQL_HISTORY** environment variable. If that is not set either, the default is ~/.psql_history, or %APPDATA%\postgresql\psql_history on Windows. For example, putting:

```
\set HISTFILE ~/.psql_history-:DBNAME
```

in ~/.psqlrc will cause psql to maintain a separate history for each database.

Note
This feature was shamelessly plagiarized from Bash.

HISTSIZE
The maximum number of commands to store in the command history (default 500). If set to a negative value, no limit is applied.

Note
This feature was shamelessly plagiarized from Bash.

HOST
The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be changed or unset.

IGNOREEOF
If set to 1 or less, sending an EOF character (usually Control+D) to an interactive session of psql will terminate the application. If set to a larger numeric value, that many consecutive EOF characters must be typed to make an interactive session terminate. If the variable is set to a non-numeric value, it is interpreted as 10. The default is 0.

Note
This feature was shamelessly plagiarized from Bash.

LASTOID
The value of the last affected OID, as returned from an **INSERT** or **\lo_import** command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed. PostgreSQL servers since version 12 do not support OID system

columns anymore, thus LASTOID will always be 0 following **INSERT** when targeting such servers.

LAST_ERROR_MESSAGE
LAST_ERROR_SQLSTATE
The primary error message and associated SQLSTATE code for the most recent failed query in the current psql session, or an empty string and 00000 if no error has occurred in the current session.

ON_ERROR_ROLLBACK
When set to on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When set to interactive, such errors are only ignored in interactive sessions, and not when reading script files. When set to off (the default), a statement in a transaction block that generates an error aborts the entire transaction. The error rollback mode works by issuing an implicit **SAVEPOINT** for you, just before each command that is in a transaction block, and then rolling back to the savepoint if the command fails.

ON_ERROR_STOP
By default, command processing continues after an error. When this variable is set to on, processing will instead stop immediately. In interactive mode, psql will return to the command prompt; otherwise, psql will exit, returning error code 3 to distinguish this case from fatal error conditions, which are reported using error code 1. In either case, any currently running scripts (the top-level script, if any, and any other scripts which it may have in invoked) will be terminated immediately. If the top-level command string contained multiple SQL commands, processing will stop with the current command.

PORT
The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be changed or unset.

PROMPT1
PROMPT2
PROMPT3
These specify what the prompts psql issues should look like. See Prompting below.

QUIET
Setting this variable to on is equivalent to the command line option **-q**. It is probably not too useful in interactive mode.

ROW_COUNT

The number of rows returned or affected by the last SQL query, or 0 if the query failed or did not report a row count.

SERVER_VERSION_NAME
SERVER_VERSION_NUM
The server's version number as a string, for example 9.6.2, 10.1 or 11beta1, and in numeric form, for example 90602 or 100001. These are set every time you connect to a database (including program start-up), but can be changed or unset.

SHOW_ALL_RESULTS
When this variable is set to off, only the last result of a combined query (\;) is shown instead of all of them. The default is on. The off behavior is for compatibility with older versions of psql.

SHOW_CONTEXT
This variable can be set to the values never, errors, or always to control whether CONTEXT fields are displayed in messages from the server. The default is errors (meaning that context will be shown in error messages, but not in notice or warning messages). This setting has no effect when *VERBOSITY* is set to terse or sqlstate. (See also **\errverbose**, for use when you want a verbose version of the error you just got.)

SINGLELINE
Setting this variable to on is equivalent to the command line option **-S**.

SINGLESTEP
Setting this variable to on is equivalent to the command line option **-s**.

SQLSTATE
The error code (see Appendix A) associated with the last SQL query's failure, or 00000 if it succeeded.

USER
The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be changed or unset.

VERBOSITY
This variable can be set to the values default, verbose, terse, or sqlstate to control the verbosity of error reports. (See also **\errverbose**, for use when you want a verbose version of the error you just got.)

VERSION
VERSION_NAME
VERSION_NUM
These variables are set at program start-up to reflect psql's version, respectively as a verbose string, a short string (e.g., 9.6.2, 10.1, or 11beta1), and a number (e.g., 90602 or 100001). They can be changed or unset.

SQL Interpolation

A key feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements, as well as the arguments of meta-commands. Furthermore, psql provides facilities for ensuring that variable values used as SQL literals and identifiers are properly quoted. The syntax for interpolating a value without any quoting is to prepend the variable name with a colon (:). For example,

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would query the table my_table. Note that this may be unsafe: the value of the variable is copied literally, so it can contain unbalanced quotes, or even backslash commands. You must make sure that it makes sense where you put it.

When a value is to be used as an SQL literal or identifier, it is safest to arrange for it to be quoted. To quote the value of a variable as an SQL literal, write a colon followed by the variable name in single quotes. To quote the value as an SQL identifier, write a colon followed by the variable name in double quotes. These constructs deal correctly with quotes and other special characters embedded within the variable value. The previous example would be more safely written this way:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

Variable interpolation will not be performed within quoted SQL literals and identifiers. Therefore, a construction such as ':foo' doesn't work to produce a quoted literal from a variable's value (and it would be unsafe if it did work, since it wouldn't correctly handle quotes embedded in the value).

One example use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then interpolate the variable's value as a quoted string:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(Note that this still won't work if my_file.txt contains NUL bytes. psql does not support embedded NUL bytes in variable values.)

Since colons can legally appear in SQL commands, an apparent attempt at interpolation (that is, :name, :'name', or :"name") is not replaced unless the named variable is currently set. In any case, you can escape a colon with a backslash to protect it from substitution.

The :{?*name*} special syntax returns TRUE or FALSE depending on whether the variable exists or not, and is thus always substituted, unless the colon is backslash-escaped.

The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntaxes for array slices and type casts are PostgreSQL extensions, which can sometimes conflict with the standard usage. The colon-quote syntax for escaping a variable's value as an SQL literal or identifier is a psql extension.

Prompting

The prompts psql issues can be customized to your preference. The three variables *PROMPT1*, *PROMPT2*, and *PROMPT3* contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new command. Prompt 2 is issued when more input is expected during command entry, for example because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you are running an SQL **COPY FROM STDIN** command and you need to type in a row value on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (%) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

%M
: The full host name (with domain name) of the database server, or [local] if the connection is over a Unix domain socket, or [local:*/dir/name*], if the Unix domain socket is not at the compiled in default location.

%m
: The host name of the database server, truncated at the first dot, or [local] if the connection is over a Unix domain socket.

%>
: The port number at which the database server is listening.

%n

The database session user name. (The expansion of this value might change during a database session as the result of the command **SET SESSION AUTHORIZATION**.)

%/

The name of the current database.

%~

Like %/, but the output is ~ (tilde) if the database is your default database.

%#

If the session user is a database superuser, then a #, otherwise a >. (The expansion of this value might change during a database session as the result of the command **SET SESSION AUTHORIZATION**.)

%p

The process ID of the backend currently connected to.

%R

In prompt 1 normally =, but @ if the session is in an inactive branch of a conditional block, or ^ if in single-line mode, or ! if the session is disconnected from the database (which can happen if **\connect** fails). In prompt 2 %R is replaced by a character that depends on why psql expects more input: - if the command simply wasn't terminated yet, but * if there is an unfinished /* ... */ comment, a single quote if there is an unfinished quoted string, a double quote if there is an unfinished quoted identifier, a dollar sign if there is an unfinished dollar-quoted string, or (if there is an unmatched left parenthesis. In prompt 3 %R doesn't produce anything.

%x

Transaction status: an empty string when not in a transaction block, or * when in a transaction block, or ! when in a failed transaction block, or ? when the transaction state is indeterminate (for example, because there is no connection).

%l

The line number inside the current statement, starting from 1.

%*digits*

The character with the indicated octal code is substituted.

%:*name*:

The value of the psql variable *name*. See Variables, above, for details.

%‘*command*‘
The output of *command*, similar to ordinary "back-tick" substitution.

%[... %]
Prompts can contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with %[and %]. Multiple pairs of these can occur within the prompt. For example:

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

results in a boldfaced (1;) yellow-on-black (33;40) prompt on VT100-compatible, color-capable terminals.

%w
Whitespace of the same width as the most recent output of *PROMPT1*. This can be used as a *PROMPT2* setting, so that multi-line statements are aligned with the first line, but there is no visible secondary prompt.

To insert a percent sign into your prompt, write %%. The default prompts are '%/%R%x%# ' for prompts 1 and 2, and '>> ' for prompt 3.

Note

This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql uses the Readline or libedit library, if available, for convenient line editing and retrieval. The command history is automatically saved when psql exits and is reloaded when psql starts up. Type up-arrow or control-P to retrieve previous lines.

You can also use tab completion to fill in partially-typed keywords and SQL object names in many (by no means all) contexts. For example, at the start of a command, typing ins and pressing TAB will fill in insert into. Then, typing a few characters of a table or schema name and pressing TAB will fill in the unfinished name, or offer a menu of possible completions when there's more than one. (Depending on the library in use, you may need to press TAB more than once to get a menu.)

Tab completion for SQL object names requires sending queries to the server to find possible matches. In some contexts this can interfere with other operations. For example, after **BEGIN** it will be too late to issue **SET TRANSACTION ISOLATION LEVEL** if a tab-completion query is issued in between. If you do not want tab completion at all, you can turn it off permanently by putting this in a file named .inputrc in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a Readline feature. Read its documentation for further details.)

The **-n** (**--no-readline**) command line option can also be useful to disable use of Readline for a single run of psql. This prevents tab completion, use or recording of command line history, and editing of multi-line commands. It is particularly useful when you need to copy-and-paste text that contains TAB characters.

ENVIRONMENT

COLUMNS

If \pset columns is zero, controls the width for the wrapped format and width for determining if wide output requires the pager or should be switched to the vertical format in expanded auto mode.

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters (see Section 34.15).

PG_COLOR

Specifies whether to use color in diagnostic messages. Possible values are always, auto and never.

PSQL_EDITOR
EDITOR
VISUAL

Editor used by the **\e**, **\ef**, and **\ev** commands. These variables are examined in the order listed; the first that is set is used. If none of them is set, the default is to use vi on Unix systems or notepad.exe on Windows systems.

PSQL_EDITOR_LINENUMBER_ARG

When **\e**, **\ef**, or **\ev** is used with a line number argument, this variable specifies the command-line argument used to pass the starting line number to the user's editor. For editors such as Emacs or vi, this is a plus sign. Include a trailing space in the value of the variable if there needs to be space between the option name and the line number. Examples:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

The default is + on Unix systems (corresponding to the default editor vi, and useful for many other common editors); but there is no default on Windows systems.

PSQL_HISTORY
Alternative location for the command history file. Tilde (~) expansion is performed.

PSQL_PAGER
PAGER
If a query's results do not fit on the screen, they are piped through this command. Typical values are more or less. Use of the pager can be disabled by setting **PSQL_PAGER** or **PAGER** to an empty string, or by adjusting the pager-related options of the **\pset** command. These variables are examined in the order listed; the first that is set is used. If neither of them is set, the default is to use more on most platforms, but less on Cygwin.

PSQL_WATCH_PAGER
When a query is executed repeatedly with the **\watch** command, a pager is not used by default. This behavior can be changed by setting **PSQL_WATCH_PAGER** to a pager command, on Unix systems. The pspg pager (not part of PostgreSQL but available in many open source software distributions) can display the output of **\watch** if started with the option --stream.

PSQLRC
Alternative location of the user's .psqlrc file. Tilde (~) expansion is performed.

SHELL
Command executed by the **\!** command.

TMPDIR
Directory for storing temporary files. The default is /tmp.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 34.15).

FILES

psqlrc and ~/.psqlrc

Unless it is passed an **-X** option, psql attempts to read and execute commands from the system-wide startup file (psqlrc) and then the user's personal startup file (~/.psqlrc), after connecting to the database but before accepting normal commands. These files can be used to set up the client and/or the server to taste, typically with **\set** and **SET** commands.

The system-wide startup file is named psqlrc. By default it is sought in the installation's "system configuration" directory, which is most reliably identified by running pg_config --sysconfdir. Typically this directory will be ../etc/ relative to the directory containing the PostgreSQL executables. The directory to look in can be set explicitly via the **PGSYSCONFDIR** environment variable.

The user's personal startup file is named .psqlrc and is sought in the invoking user's home directory. On Windows the personal startup file is instead named %APPDATA%\postgresql\psqlrc.conf. In either case, this default file path can be overridden by setting the **PSQLRC** environment variable.

Both the system-wide startup file and the user's personal startup file can be made psql-version-specific by appending a dash and the PostgreSQL major or minor release identifier to the file name, for example ~/.psqlrc-15 or ~/.psqlrc-15.8. The most specific version-matching file will be read in preference to a non-version-specific file. These version suffixes are added after determining the file path as explained above.

.psql_history

The command-line history is stored in the file ~/.psql_history, or %APPDATA%\postgresql\psql_history on Windows.

The location of the history file can be set explicitly via the *HISTFILE* psql variable or the **PSQL_HISTORY** environment variable.

NOTES

works best with servers of the same or an older major version. Backslash commands are particularly likely to fail if the server is of a newer version than psql itself. However, backslash commands of the \d family should work with servers of versions back to 9.2, though not necessarily with servers newer than psql itself. The general functionality of running SQL commands and displaying query results should also work with servers of a newer major version, but this cannot be guaranteed in all cases.

If you want to use psql to connect to several servers of different major versions, it is recommended that

you use the newest version of psql. Alternatively, you can keep around a copy of psql from each major version and be sure to use the version that matches the respective server. But in practice, this additional complication should not be necessary.

- PostgreSQL 9.6, the **-c** option implied **-X** (**--no-psqlrc**); this is no longer the case.

- PostgreSQL 8.4, psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. Now, some whitespace is required.

NOTES FOR WINDOWS USERS

psql is built as a "console application". Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within psql. If psql detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

- the code page by entering **cmd.exe /c chcp 1252**. (1252 is a code page that is appropriate for German; replace it with your value.) If you are using Cygwin, you can put this command in /etc/profile.

- the console font to Lucida Console, because the raster font does not work with the ANSI code page.

EXAMPLES

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Now look at the table definition again:

```
testdb=> \d my_table
            Table "public.my_table"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
```

```
first  | integer |           | not null | 0
second | text    |           |          |
```

Now we change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can display tables in different ways by using the **\pset** command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
```

```
   2 two
   3 three
   4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format csv
Output format is csv.
peter@localhost testdb=> \pset tuples_only
Tuples only is on.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep '\t'
Field separator is "    ".
peter@localhost testdb=> SELECT second, first FROM my_table;
one     1
two     2
three   3
four    4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
```

```
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Also, these output format options can be set for just one query by using \g:

```
peter@localhost testdb=> SELECT * FROM my_table
peter@localhost testdb-> \g (format=aligned tuples_only=off expanded=on)
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Here is an example of using the **\df** command to find only functions with names matching int*pl and whose second argument is of type bigint:

```
testdb=> \df int*pl * bigint
                          List of functions
  Schema   |  Name   | Result data type | Argument data types | Type
------------+---------+------------------+---------------------+------
 pg_catalog | int28pl | bigint           | smallint, bigint    | func
 pg_catalog | int48pl | bigint           | integer, bigint     | func
 pg_catalog | int8pl  | bigint           | bigint, bigint      | func
(3 rows)
```

When suitable, query results can be shown in a crosstab representation with the **\crosstabview** command:

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
```

```
-------+--------+-----
    1 | one    | f
    2 | two    | f
    3 | three  | t
    4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
    1 | f   |     |       |
    2 |     | f   |       |
    3 |     |     | t     |
    4 |     |     |       | t
(4 rows)
```

This second example shows a multiplication table with rows sorted in reverse numerical order and columns with an independent, ascending numerical order.

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```