

NAME

COPY – copy data between a file and a table

SYNOPSIS

```
COPY table_name [ ( column_name [, ...] ) ]
  FROM { 'filename' | PROGRAM 'command' | STDIN }
  [ [ WITH ] ( option [, ...] ) ]
  [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
  TO { 'filename' | PROGRAM 'command' | STDOUT }
  [ [ WITH ] ( option [, ...] ) ]
```

where *option* can be one of:

```
FORMAT format_name
FREEZE [ boolean ]
DELIMITER 'delimiter_character'
NULL 'null_string'
DEFAULT 'default_string'
HEADER [ boolean | MATCH ]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { ( column_name [, ...] ) | * }
FORCE_NOT_NULL { ( column_name [, ...] ) | * }
FORCE_NULL { ( column_name [, ...] ) | * }
ON_ERROR error_action
REJECT_LIMIT maxerror
ENCODING 'encoding_name'
LOG_VERBOSITY verbosity
```

DESCRIPTION

COPY moves data between PostgreSQL tables and standard file-system files. **COPY TO** copies the contents of a table *to* a file, while **COPY FROM** copies data *from* a file to a table (appending the data to whatever is in the table already). **COPY TO** can also copy the results of a **SELECT** query.

If a column list is specified, **COPY TO** copies only the data in the specified columns to the file. For **COPY FROM**, each field in the file is inserted, in order, into the specified column. Table columns not specified in the **COPY FROM** column list will receive their default values.

COPY with a file name instructs the PostgreSQL server to directly read from or write to a file. The file must be accessible by the PostgreSQL user (the user ID the server runs as) and the name must be specified from the viewpoint of the server. When PROGRAM is specified, the server executes the given command and reads from the standard output of the program, or writes to the standard input of the program. The command must be specified from the viewpoint of the server, and be executable by the PostgreSQL user. When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the server.

Each backend running **COPY** will report its progress in the pg_stat_progress_copy view. See Section 27.4.3 for details.

By default, **COPY** will fail if it encounters an error during processing. For use cases where a best-effort attempt at loading the entire file is desired, the ON_ERROR clause can be used to specify some other behavior.

PARAMETERS

table_name
The name (optionally schema-qualified) of an existing table.

column_name

An optional list of columns to be copied. If no column list is specified, all columns of the table except generated columns will be copied.

query

A **SELECT**, **VALUES**, **INSERT**, **UPDATE**, **DELETE**, or **MERGE** command whose results are to be copied. Note that parentheses are required around the query.

For **INSERT**, **UPDATE**, **DELETE**, and **MERGE** queries a **RETURNING** clause must be provided, and the target relation must not have a conditional rule, nor an **ALSO** rule, nor an **INSTEAD** rule that expands to multiple statements.

filename

The path name of the input or output file. An input file name can be an absolute or relative path, but an output file name must be an absolute path. Windows users might need to use an E" string and double any backslashes used in the path name.

PROGRAM

A command to execute. In **COPY FROM**, the input is read from standard output of the command, and in **COPY TO**, the output is written to the standard input of the command.

Note that the command is invoked by the shell, so if you need to pass any arguments that come from an untrusted source, you must be careful to strip or escape any special characters that might have a special meaning for the shell. For security reasons, it is best to use a fixed command string, or at least avoid including any user input in it.

STDIN

Specifies that input comes from the client application.

STDOUT

Specifies that output goes to the client application.

boolean

Specifies whether the selected option should be turned on or off. You can write **TRUE**, **ON**, or **1** to enable the option, and **FALSE**, **OFF**, or **0** to disable it. The *boolean* value can also be omitted, in which case **TRUE** is assumed.

FORMAT

Selects the data format to be read or written: text, csv (Comma Separated Values), or binary. The default is text. See File Formats below for details.

FREEZE

Requests copying the data with rows already frozen, just as they would be after running the **VACUUM FREEZE** command. This is intended as a performance option for initial data loading. Rows will be frozen only if the table being loaded has been created or truncated in the current subtransaction, there are no cursors open and there are no older snapshots held by this transaction. It is currently not possible to perform a **COPY FREEZE** on a partitioned table or foreign table. This option is only allowed in **COPY FROM**.

Note that all other sessions will immediately be able to see the data once it has been successfully loaded. This violates the normal rules of MVCC visibility and users should be aware of the potential problems this might cause.

DELIMITER

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in CSV format. This must be a single one-byte character. This option is not allowed when using binary format.

NULL

Specifies the string that represents a null value. The default is \N (backslash-N) in text format, and an

unquoted empty string in CSV format. You might prefer an empty string even in text format for cases where you don't want to distinguish nulls from empty strings. This option is not allowed when using binary format.

Note

When using **COPY FROM**, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with **COPY TO**.

DEFAULT

Specifies the string that represents a default value. Each time the string is found in the input file, the default value of the corresponding column will be used. This option is allowed only in **COPY FROM**, and only when not using binary format.

HEADER

Specifies that the file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table. On input, the first line is discarded when this option is set to true (or equivalent Boolean value). If this option is set to **MATCH**, the number and names of the columns in the header line must match the actual column names of the table, in order; otherwise an error is raised. This option is not allowed when using binary format. The **MATCH** option is only valid for **COPY FROM** commands.

QUOTE

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using CSV format.

ESCAPE

Specifies the character that should appear before a data character that matches the **QUOTE** value. The default is the same as the **QUOTE** value (so that the quoting character is doubled if it appears in the data). This must be a single one-byte character. This option is allowed only when using CSV format.

FORCE_QUOTE

Forces quoting to be used for all non-NULL values in each specified column. NULL output is never quoted. If ***** is specified, non-NULL values will be quoted in all columns. This option is allowed only in **COPY TO**, and only when using CSV format.

FORCE_NOT_NULL

Do not match the specified columns' values against the null string. In the default case where the null string is empty, this means that empty values will be read as zero-length strings rather than nulls, even when they are not quoted. If ***** is specified, the option will be applied to all columns. This option is allowed only in **COPY FROM**, and only when using CSV format.

FORCE_NULL

Match the specified columns' values against the null string, even if it has been quoted, and if a match is found set the value to NULL. In the default case where the null string is empty, this converts a quoted empty string into NULL. If ***** is specified, the option will be applied to all columns. This option is allowed only in **COPY FROM**, and only when using CSV format.

ON_ERROR

Specifies how to behave when encountering an error converting a column's input value into its data type. An *error_action* value of **stop** means fail the command, while **ignore** means discard the input row and continue with the next one. The default is **stop**.

The **ignore** option is applicable only for **COPY FROM** when the **FORMAT** is **text** or **csv**.

A **NOTICE** message containing the ignored row count is emitted at the end of the **COPY FROM** if at least one row was discarded. When **LOG_VERBOSITY** option is set to **verbose**, a **NOTICE** message containing the line of the input file and the column name whose input conversion has failed is emitted for each discarded row. When it is set to **silent**, no message is emitted regarding ignored rows.

REJECT_LIMIT

Specifies the maximum number of errors tolerated while converting a column's input value to its data type, when `ON_ERROR` is set to ignore. If the input causes more errors than the specified value, the **COPY** command fails, even with `ON_ERROR` set to ignore. This clause must be used with `ON_ERROR=ignore` and *maxerror* must be positive bigint. If not specified, `ON_ERROR=ignore` allows an unlimited number of errors, meaning **COPY** will skip all erroneous data.

ENCODING

Specifies that the file is encoded in the *encoding_name*. If this option is omitted, the current client encoding is used. See the Notes below for more details.

LOG_VERBOSITY

Specifies the amount of messages emitted by a **COPY** command: default, verbose, or silent. If verbose is specified, additional messages are emitted during processing. silent suppresses both verbose and default messages.

This is currently used in **COPY FROM** command when `ON_ERROR` option is set to ignore.

WHERE

The optional WHERE clause has the general form

WHERE *condition*

where *condition* is any expression that evaluates to a result of type boolean. Any row that does not satisfy this condition will not be inserted to the table. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

Currently, subqueries are not allowed in WHERE expressions, and the evaluation does not see any changes made by the **COPY** itself (this matters when the expression contains calls to VOLATILE functions).

OUTPUTS

On successful completion, a **COPY** command returns a command tag of the form

COPY *count*

The *count* is the number of rows copied.

Note

psql will print this command tag only if the command was not **COPY ... TO STDOUT**, or the equivalent psql meta-command `\copy ... to stdout`. This is to prevent confusing the command tag with the data that was just printed.

NOTES

COPY TO can be used with plain tables and populated materialized views. For example, **COPY table TO** copies the same rows as **SELECT * FROM ONLY table**. However it doesn't directly support other relation types, such as partitioned tables, inheritance child tables, or views. To copy all rows from such relations, use **COPY (SELECT * FROM table) TO**.

COPY FROM can be used with plain, foreign, or partitioned tables or with views that have **INSTEAD OF INSERT** triggers.

You must have select privilege on the table whose values are read by **COPY TO**, and insert privilege on the table into which values are inserted by **COPY FROM**. It is sufficient to have column privileges on the column(s) listed in the command.

If row-level security is enabled for the table, the relevant **SELECT** policies will apply to **COPY table TO** statements. Currently, **COPY FROM** is not supported for tables with row-level security. Use equivalent **INSERT** statements instead.

Files named in a **COPY** command are read or written directly by the server, not by the client application.

Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. Similarly, the command specified with PROGRAM is executed directly by the server, not by the client application, must be executable by the PostgreSQL user. **COPY** naming a file or command is only allowed to database superusers or users who are granted one of the roles `pg_read_server_files`, `pg_write_server_files`, or `pg_execute_server_program`, since it allows reading or writing any file or running a program that the server has privileges to access.

Do not confuse **COPY** with the psql instruction `\copy`. `\copy` invokes **COPY FROM STDIN** or **COPY TO STDOUT**, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the server when `\copy` is used.

It is recommended that the file name used in **COPY** always be specified as an absolute path. This is enforced by the server in the case of **COPY TO**, but for **COPY FROM** you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the working directory of the server process (normally the cluster's data directory), not the client's working directory.

Executing a command with PROGRAM might be restricted by the operating system's access control mechanisms, such as SELinux.

COPY FROM will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

For identity columns, the **COPY FROM** command will always write the column values provided in the input data, like the **INSERT** option `OVERRIDING SYSTEM VALUE`.

COPY input and output is affected by *DateStyle*. To ensure portability to other PostgreSQL installations that might use non-default *DateStyle* settings, *DateStyle* should be set to ISO before using **COPY TO**. It is also a good idea to avoid dumping data with *IntervalStyle* set to `sql_standard`, because negative interval values might be misinterpreted by a server that has a different setting for *IntervalStyle*.

Input data is interpreted according to `ENCODING` option or the current client encoding, and output data is encoded in `ENCODING` or the current client encoding, even if the data does not pass through the client but is read from or written to a file directly by the server.

The **COPY FROM** command physically inserts input rows into the table as it progresses. If the command fails, these rows are left in a deleted state; these rows will not be visible, but still occupy disk space. This might amount to considerable wasted disk space if the failure happened well into a large copy operation. **VACUUM** should be used to recover the wasted space.

`FORCE_NULL` and `FORCE_NOT_NULL` can be used simultaneously on the same column. This results in converting quoted null strings to null values and unquoted null strings to empty strings.

FILE FORMATS

Text Format

When the text format is used, the data read or written is a text file with one line per table row. Columns in a row are separated by the delimiter character. The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. **COPY FROM** will raise an error if any line of the input file contains more or fewer columns than are expected.

End of data can be represented by a line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a file, since the end of file serves perfectly well; in that context this provision exists only for backward compatibility. However, psql uses `\.` to terminate a **COPY FROM STDIN** operation (that is, reading in-line **COPY** data in an SQL script). In that context the rule is needed to be able to end the operation before the end of the script.

Backslash characters (`\`) can be used in the **COPY** data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters *must* be preceded by a backslash if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character.

The specified null string is sent by **COPY TO** without adding any backslashes; conversely, **COPY FROM** matches the input against the null string before removing backslashes. Therefore, a null string such as \N cannot be confused with the actual data value \N (which would be represented as \\N).

The following special backslash sequences are recognized by **COPY FROM**:

Sequence	Represents
\b	Backspace (ASCII 8)
\f	Form feed (ASCII 12)
\n	Newline (ASCII 10)
\r	Carriage return (ASCII 13)
\t	Tab (ASCII 9)
\v	Vertical tab (ASCII 11)
\digits	Backslash followed by one to three octal digits specifies the byte with that numeric code
\xdigits	Backslash x followed by one or two hex digits specifies the byte with that numeric code

Presently, **COPY TO** will never emit an octal or hex-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Any other backslashed character that is not mentioned in the above table will be taken to represent itself. However, beware of adding backslashes unnecessarily, since that might accidentally produce a string matching the end-of-data marker (\.) or the null string (\N by default). These strings will be recognized before any other backslash processing is done.

It is strongly recommended that applications generating **COPY** data convert data newlines and carriage returns to the \n and \r sequences respectively. At present it is possible to represent a data carriage return by a backslash and carriage return, and to represent a data newline by a backslash and newline. However, these representations might not be accepted in future releases. They are also highly vulnerable to corruption if the **COPY** file is transferred across different machines (for example, from Unix to Windows or vice versa).

All backslash sequences are interpreted after encoding conversion. The bytes specified with the octal and hex-digit backslash sequences must form valid characters in the database encoding.

COPY TO will terminate each row with a Unix-style newline (“\n”). Servers running on Microsoft Windows instead output carriage return/newline (“\r\n”), but only for **COPY** to a server file; for consistency across platforms, **COPY TO STDOUT** always sends “\n” regardless of server platform. **COPY FROM** can handle lines ending with newlines, carriage returns, or carriage return/newlines. To reduce the risk of error due to un-backslashed newlines or carriage returns that were meant as data, **COPY FROM** will complain if the line endings in the input are not all alike.

CSV Format

This format option is used for importing and exporting the Comma-Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping rules used by PostgreSQL's standard text format, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the DELIMITER character. If the value contains the delimiter character, the QUOTE character, the NULL string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the QUOTE character, and any occurrence within the value of a QUOTE character or the ESCAPE character is preceded by the escape character. You can also use FORCE_QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. PostgreSQL's **COPY** handles this by quoting. A NULL is output as the NULL parameter string and is not quoted, while a non-NULL value matching the NULL parameter string is quoted. For example, with the default settings, a NULL is written as an unquoted empty string, while an empty string data value is written with double

quotes (""), Reading values follows similar rules. You can use `FORCE_NOT_NULL` to prevent NULL input comparisons for specific columns. You can also use `FORCE_NULL` to convert quoted null string data values to NULL.

Because backslash is not a special character in the CSV format, the end-of-data marker used in text mode (`\.`) is not normally treated as special when reading CSV data. An exception is that `psql` will terminate a `COPY FROM STDIN` operation (that is, reading in-line **COPY** data in an SQL script) at a line containing only `\.`, whether it is text or CSV mode.

Note

PostgreSQL versions before v18 always recognized unquoted `\.` as an end-of-data marker, even when reading from a separate file. For compatibility with older versions, **COPY TO** will quote `\.` when it's alone on a line, even though this is no longer necessary.

Note

In CSV format, all characters are significant. A quoted value surrounded by white space, or any characters other than `DELIMITER`, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into PostgreSQL.

Note

CSV format will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-format files.

Note

Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and **COPY** might produce files that other programs cannot process.

Binary Format

The binary format option causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the text and CSV formats, but a binary-format file is less portable across machine architectures and PostgreSQL versions. Also, the binary format is very data type specific; for example it will not work to output binary data from a `smallint` column and read it into an `integer` column, even though that would work fine in text format.

The binary file format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

Note

PostgreSQL releases before 7.4 used a different binary file format.

File Header

The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

11-byte sequence `PGCOPY\377\0` — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)

Flags field

32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16–31 are reserved to denote critical

file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0–15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

If 1, OIDs are included in the data; if 0, not. Oid system columns are not supported in PostgreSQL anymore, but the format still contains the indicator.

Header extension area length

32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, –1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a binary-format file are assumed to be in binary format (format code one). It is anticipated that a future extension might add a header field that allows per-column format codes to be specified.

To determine the appropriate binary format for the actual tuple data you should consult the PostgreSQL source, in particular the ***send** and ***recv** functions for each column's data type (typically these functions are found in the `src/backend/utils/adts/` directory of the source distribution).

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. Note that oid system columns are not supported in current versions of PostgreSQL.

File Trailer

The file trailer consists of a 16-bit integer word containing –1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither –1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

EXAMPLES

The following example copies a table to the client using the vertical bar (|) as the field delimiter:

```
COPY country TO STDOUT (DELIMITER '|');
```

To copy data from a file into the country table:

```
COPY country FROM 'usr1/proj/bray/sql/country_data';
```

To copy into a file just the countries whose names start with 'A':

`COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_countries.copy';`
 To copy into a compressed file, you can pipe the output through an external compression program:

`COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';`

Here is a sample of data suitable for copying into a table from STDIN:

```
AF  AFGHANISTAN
AL  ALBANIA
DZ  ALGERIA
ZM  ZAMBIA
ZW  ZIMBABWE
```

Note that the white space on each line is actually a tab character.

The following is the same data, output in binary format. The data is shown after filtering through the Unix utility `od -c`. The table has three columns; the first has type `char(2)`, the second has type `text`, and the third has type `integer`. All the rows have a null value in the third column.

```
0000000 P G C O P Y \n 377 \r \n \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 003 \0 \0 \0 002 A F \0 \0 \0 013 A
0000040 F G H A N I S T A N 377 377 377 377 \0 003
0000060 \0 \0 \0 002 A L \0 \0 \0 007 A L B A N I
0000100 A 377 377 377 377 \0 003 \0 \0 \0 002 D Z \0 \0 \0
0000120 007 A L G E R I A 377 377 377 377 \0 003 \0 \0
0000140 \0 002 Z M \0 \0 \0 006 Z A M B I A 377 377
0000160 377 377 \0 003 \0 \0 \0 002 Z W \0 \0 \0 \b Z I
0000200 M B A B W E 377 377 377 377 377 377
```

COMPATIBILITY

There is no **COPY** statement in the SQL standard.

The following syntax was used before PostgreSQL version 9.0 and is still supported:

```
COPY table_name [ ( column_name [, ...] ) ]
FROM { 'filename' | STDIN }
[ [ WITH ]
  [ BINARY ]
  [ DELIMITER [ AS ] 'delimiter_character' ]
  [ NULL [ AS ] 'null_string' ]
  [ CSV [ HEADER ]
    [ QUOTE [ AS ] 'quote_character' ]
    [ ESCAPE [ AS ] 'escape_character' ]
    [ FORCE NOT NULL column_name [, ...] ] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
TO { 'filename' | STDOUT }
[ [ WITH ]
  [ BINARY ]
  [ DELIMITER [ AS ] 'delimiter_character' ]
  [ NULL [ AS ] 'null_string' ]
  [ CSV [ HEADER ]
    [ QUOTE [ AS ] 'quote_character' ]
    [ ESCAPE [ AS ] 'escape_character' ]
    [ FORCE QUOTE { column_name [, ...] | * } ] ] ] ]
```

Note that in this syntax, BINARY and CSV are treated as independent keywords, not as arguments of a FORMAT option.

The following syntax was used before PostgreSQL version 7.3 and is still supported:

```
COPY [ BINARY ] table_name
  FROM { 'filename' | STDIN }
  [ [USING] DELIMITERS 'delimiter_character' ]
  [ WITH NULL AS 'null_string' ]
```

```
COPY [ BINARY ] table_name
  TO { 'filename' | STDOUT }
  [ [USING] DELIMITERS 'delimiter_character' ]
  [ WITH NULL AS 'null_string' ]
```

SEE ALSO

Section 27.4.3