这一部分是快速浏览一下PostgreSQL的SQL语法。详细的还要看Part 4。

This part describes the use of the SQL language in PostgreSQL. We start with describing the general syntax of SQL, then explain how to create the structures to hold data, how to populate the database, and how to query it. The middle part lists the available data types and functions for use in SQL commands. The rest treats several aspects that are important for tuning a database for optimal performance.

The information in this part is arranged so that a novice user can follow it start to end to gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should see [**Part VI**](https://www.postgresql.org/docs/10/reference.html).

Readers of this part should know how to connect to a PostgreSQL database and issue SQL commands. Readers that are unfamiliar with these issues are encouraged to read [**Part I**](https://www.postgresql.org/docs/10/tutorial.html) first. SQLcommands are typically entered using the PostgreSQL interactive terminal psql, but other programs that have similar functionality can be used as well.

**4.1. Lexical Structure 词法结构**

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon (“;”). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a key word, an identifier, a quoted identifier, a literal (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

For example, the following is (syntactically) valid SQL input:

SELECT \* FROM MY\_TABLE;

UPDATE MY\_TABLE SET A = 5;

INSERT INTO MY\_TABLE VALUES (3, 'hi there');

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

Additionally, comments can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a “SELECT”, an “UPDATE”, and an “INSERT” command. But for instance the UPDATE command always requires a SET token to appear in a certain position, and this particular variation of INSERT also requires a VALUES in order to be complete. The precise syntax rules for each command are described in [**Part VI**](https://www.postgresql.org/docs/10/reference.html).

### 4.1.1. Identifiers and Key Words 标识符和关键词

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of key words, that is, words that have a fixed meaning in the SQL language. The tokens MY\_TABLE and A are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called “names”. Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in [**Appendix C**](https://www.postgresql.org/docs/10/sql-keywords-appendix.html).

SQL identifiers and key words must begin with a letter (a-z, but also letters with diacritical marks and non-Latin letters) or an underscore (\_). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), or dollar signs ($). Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard, so their use might render applications less portable. The SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than NAMEDATALEN-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63 bytes. If this limit is problematic, it can be raised by changing the NAMEDATALEN constant in src/include/pg\_config\_manual.h.

Key words and unquoted identifiers are case insensitive. Therefore:

UPDATE MY\_TABLE SET A = 5;

can equivalently be written as:

uPDaTE my\_TabLE SeT a = 5;

A convention often used is to write key words in upper case and names in lower case, e.g.:

UPDATE my\_table SET a = 5;

有第二种标识符，限定标识符和引用标识符

There is a second kind of identifier: the delimited identifier or quoted identifier. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named “select”, whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

UPDATE "my\_table" SET "a" = 5;

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

A variant of quoted identifiers allows including escaped Unicode characters identified by their code points. This variant starts with U& (upper or lower case U followed by ampersand) immediately before the opening double quote, without any spaces in between, for example U&"foo". (Note that this creates an ambiguity with the operator &. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the identifier "data" could be written as

U&"d\0061t\+000061"

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

U&"\0441\043B\043E\043D"

If a different escape character than backslash is desired, it can be specified using the UESCAPE clause after the string, for example:

U&"d!0061t!+000061" UESCAPE '!'

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character. Note that the escape character is written in single quotes, not double quotes.

To include the escape character in the identifier literally, write it twice.

The Unicode escape syntax works only when the server encoding is UTF8. When other server encodings are used, only code points in the ASCII range (up to \007F) can be specified. Both the 4-digit and the 6-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but combined into a single code point that is then encoded in UTF-8.)

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by PostgreSQL, but "Foo" and "FOO" are different from these three and each other. (The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, foo should be equivalent to "FOO" not "foo" according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

### 4.1.2. Constants

There are three kinds of implicitly-typed constants in PostgreSQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

#### 4.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g., 'Dianne''s horse'. Note that this is not the same as a double-quote character (").

Two string constants that are only separated by whitespace with at least one newline are concatenated and effectively treated as if the string had been written as one constant. For example:

SELECT 'foo'

'bar';

is equivalent to:

SELECT 'foobar';

but:

SELECT 'foo' 'bar';

is not valid syntax. (This slightly bizarre behavior is specified by SQL; PostgreSQL is following the standard.)

#### 4.1.2.2. String Constants With C-Style Escapes

PostgreSQL also accepts “escape” string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter E (upper or lower case) just before the opening single quote, e.g., E'foo'. (When continuing an escape string constant across lines, write E only before the first opening quote.) Within an escape string, a backslash character (\) begins a C-like backslash escape sequence, in which the combination of backslash and following character(s) represent a special byte value, as shown in [**Table 4.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-BACKSLASH-TABLE).

**Table 4.1. Backslash Escape Sequences**

| **Backslash Escape Sequence** | **Interpretation** |
| --- | --- |
| \b | backspace |
| \f | form feed |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \***o***, \***oo***, \***ooo*** (***o*** = 0 - 7) | octal byte value |
| \x***h***, \x***hh*** (***h*** = 0 - 9, A - F) | hexadecimal byte value |
| \u***xxxx***, \U***xxxxxxxx*** (***x*** = 0 - 9, A - F) | 16 or 32-bit hexadecimal Unicode character value |

Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (\\). Also, a single quote can be included in an escape string by writing \', in addition to the normal way of ''.

It is your responsibility that the byte sequences you create, especially when using the octal or hexadecimal escapes, compose valid characters in the server character set encoding. When the server encoding is UTF-8, then the Unicode escapes or the alternative Unicode escape syntax, explained in [**Section 4.1.2.3**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS-UESCAPE), should be used instead. (The alternative would be doing the UTF-8 encoding by hand and writing out the bytes, which would be very cumbersome.)

The Unicode escape syntax works fully only when the server encoding is UTF8. When other server encodings are used, only code points in the ASCII range (up to \u007F) can be specified. Both the 4-digit and the 8-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 8-digit form technically makes this unnecessary. (When surrogate pairs are used when the server encoding is UTF8, they are first combined into a single code point that is then encoded in UTF-8.)

**Caution**

If the configuration parameter [**standard\_conforming\_strings**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-STANDARD-CONFORMING-STRINGS) is off, then PostgreSQLrecognizes backslash escapes in both regular and escape string constants. However, as of PostgreSQL 9.1, the default is on, meaning that backslash escapes are recognized only in escape string constants. This behavior is more standards-compliant, but might break applications which rely on the historical behavior, where backslash escapes were always recognized. As a workaround, you can set this parameter to off, but it is better to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the string constant with an E.

In addition to standard\_conforming\_strings, the configuration parameters [**escape\_string\_warning**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-ESCAPE-STRING-WARNING) and [**backslash\_quote**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-BACKSLASH-QUOTE) govern treatment of backslashes in string constants.

The character with the code zero cannot be in a string constant.

#### 4.1.2.3. String Constants With Unicode Escapes

PostgreSQL also supports another type of escape syntax for strings that allows specifying arbitrary Unicode characters by code point. A Unicode escape string constant starts with U& (upper or lower case letter U followed by ampersand) immediately before the opening quote, without any spaces in between, for example U&'foo'. (Note that this creates an ambiguity with the operator &. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the string 'data' could be written as

U&'d\0061t\+000061'

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

U&'\0441\043B\043E\043D'

If a different escape character than backslash is desired, it can be specified using the UESCAPE clause after the string, for example:

U&'d!0061t!+000061' UESCAPE '!'

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character.

The Unicode escape syntax works only when the server encoding is UTF8. When other server encodings are used, only code points in the ASCII range (up to \007F) can be specified. Both the 4-digit and the 6-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (When surrogate pairs are used when the server encoding is UTF8, they are first combined into a single code point that is then encoded in UTF-8.)

Also, the Unicode escape syntax for string constants only works when the configuration parameter [**standard\_conforming\_strings**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-STANDARD-CONFORMING-STRINGS) is turned on. This is because otherwise this syntax could confuse clients that parse the SQL statements to the point that it could lead to SQL injections and similar security issues. If the parameter is set to off, this syntax will be rejected with an error message.

To include the escape character in the string literally, write it twice.

#### 4.1.2.4. Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those must be doubled. To allow more readable queries in such situations, PostgreSQL provides another way, called “dollar quoting”, to write string constants. A dollar-quoted string constant consists of a dollar sign ($), an optional “tag” of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. For example, here are two different ways to specify the string “Dianne's horse” using dollar quoting:

$$Dianne's horse$$

$SomeTag$Dianne's horse$SomeTag$

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped. Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always written literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level. This is most commonly used in writing function definitions. For example:

$function$

BEGIN

RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);

END;

$function$

Here, the sequence $q$[\t\r\n\v\\]$q$ represents a dollar-quoted literal string [\t\r\n\v\\], which will be recognized when the function body is executed by PostgreSQL. But since the sequence does not match the outer dollar quoting delimiter $function$, it is just some more characters within the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that it cannot contain a dollar sign. Tags are case sensitive, so $tag$String content$tag$ is correct, but $TAG$String content$tag$ is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants, as is often needed in procedural function definitions. With single-quote syntax, each backslash in the above example would have to be written as four backslashes, which would be reduced to two backslashes in parsing the original string constant, and then to one when the inner string constant is re-parsed during function execution.

#### 4.1.2.5. Bit-String Constants

Bit-string constants look like regular string constants with a B (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., B'1001'. The only characters allowed within bit-string constants are 0 and 1.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading X (upper or lower case), e.g., X'1FF'. This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants. Dollar quoting cannot be used in a bit-string constant.

#### 4.1.2.6. Numeric Constants

Numeric constants are accepted in these general forms:

***digits***

***digits***.[***digits***][e[+-]***digits***]

[***digits***].***digits***[e[+-]***digits***]

***digits***e[+-]***digits***

where ***digits*** is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (e), if one is present. There cannot be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42  
3.5  
4.  
.001  
5e2  
1.925e-3

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type integer if its value fits in type integer (32 bits); otherwise it is presumed to be type bigint if its value fits in type bigint (64 bits); otherwise it is taken to be type numeric. Constants that contain decimal points and/or exponents are always initially presumed to be type numeric.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type real(float4) by writing:

REAL '1.23' -- string style

1.23::REAL -- PostgreSQL (historical) style

These are actually just special cases of the general casting notations discussed next.

#### 4.1.2.7. Constants Of Other Types

A constant of an arbitrary type can be entered using any one of the following notations:

***type*** '***string***'

'***string***'::***type***

CAST ( '***string***' AS ***type*** )

The string constant's text is passed to the input conversion routine for the type called ***type***. The result is a constant of the indicated type. The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.

It is also possible to specify a type coercion using a function-like syntax:

***typename*** ( '***string***' )

but not all type names can be used in this way; see [**Section 4.2.9**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-TYPE-CASTS) for details.

The ::, CAST(), and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions, as discussed in [**Section 4.2.9**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-TYPE-CASTS). To avoid syntactic ambiguity, the ***type*** '***string***'syntax can only be used to specify the type of a simple literal constant. Another restriction on the ***type*** '***string***' syntax is that it does not work for array types; use :: or CAST() to specify the type of an array constant.

The CAST() syntax conforms to SQL. The ***type*** '***string***' syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but PostgreSQL allows it for all types. The syntax with :: is historical PostgreSQL usage, as is the function-call syntax.

### 4.1.3. Operators

An operator name is a sequence of up to NAMEDATALEN-1 (63 by default) characters from the following list:

+ - \* / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on operator names, however:

* -- and /\* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
* A multiple-character operator name cannot end in + or -, unless the name also contains at least one of these characters:

~ ! @ # % ^ & | ` ?

For example, @- is an allowed operator name, but \*- is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named @, you cannot write X\*@Y; you must write X\* @Y to ensure that PostgreSQL reads it as two operator names not one.

### 4.1.4. Special Characters 特殊字符

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

* A dollar sign ($) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign can be part of an identifier or a dollar-quoted string constant.
* Parentheses (()) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
* Brackets ([]) are used to select the elements of an array. See [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html) for more information on arrays.
* Commas (,) are used in some syntactical constructs to separate the elements of a list.
* The semicolon (;) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
* The colon (:) is used to select “slices” from arrays. (See [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html).) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
* The asterisk (\*) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
* The period (.) is used in numeric constants, and to separate schema, table, and column names.

### 4.1.5. Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

-- This is a standard SQL comment

Alternatively, C-style block comments can be used:

/\* multiline comment

\* with nesting: /\* nested block comment \*/

\*/

where the comment begins with /\* and extends to the matching occurrence of \*/. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that might contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

### 4.1.6. Operator Precedence 操作优先级

**[Table 4.2](https://www.postgresql.org/docs/10/sql-syntax-lexical.html" \l "SQL-PRECEDENCE-TABLE" \o "Table 4.2. Operator Precedence (highest to lowest))** shows the precedence and associativity of the operators in PostgreSQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser.

You will sometimes need to add parentheses when using combinations of binary and unary operators. For instance:

SELECT 5 ! - 6;

will be parsed as:

SELECT 5 ! (- 6);

because the parser has no idea — until it is too late — that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write:

SELECT (5 !) - 6;

This is the price one pays for extensibility.

**Table 4.2. Operator Precedence (highest to lowest)**

| **Operator/Element** | **Associativity** | **Description** |
| --- | --- | --- |
| . | left | table/column name separator |
| :: | left | PostgreSQL-style typecast |
| [ ] | left | array element selection |
| + - | right | unary plus, unary minus |
| ^ | left | exponentiation |
| \* / % | left | multiplication, division, modulo |
| + - | left | addition, subtraction |
| (any other operator) | left | all other native and user-defined operators |
| BETWEEN IN LIKE ILIKE SIMILAR |  | range containment, set membership, string matching |
| < > = <= >= <> |  | comparison operators |
| IS ISNULL NOTNULL |  | IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc |
| NOT | right | logical negation |
| AND | left | logical conjunction |
| OR | left | logical disjunction |

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a “+” operator for some custom data type it will have the same precedence as the built-in “+” operator, no matter what yours does.

When a schema-qualified operator name is used in the OPERATOR syntax, as for example in:

SELECT 3 OPERATOR(pg\_catalog.+) 4;

the OPERATOR construct is taken to have the default precedence shown in [**Table 4.2**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-PRECEDENCE-TABLE) for “any other operator”. This is true no matter which specific operator appears inside OPERATOR().

Note

PostgreSQL versions before 9.5 used slightly different operator precedence rules. In particular, <= >= and <> used to be treated as generic operators; IS tests used to have higher priority; and NOT BETWEEN and related constructs acted inconsistently, being taken in some cases as having the precedence of NOT rather than BETWEEN. These rules were changed for better compliance with the SQL standard and to reduce confusion from inconsistent treatment of logically equivalent constructs. In most cases, these changes will result in no behavioral change, or perhaps in “no such operator” failures which can be resolved by adding parentheses. However there are corner cases in which a query might change behavior without any parsing error being reported. If you are concerned about whether these changes have silently broken something, you can test your application with the configuration parameter [**operator\_precedence\_warning**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-OPERATOR-PRECEDENCE-WARNING) turned on to see if any warnings are logged.

## 4.2. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a scalar, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called scalar expressions (or even simply expressions). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

值表达式的类型

A value expression is one of the following:

* A constant or literal value 常量或者字符值
* A column reference
* A positional parameter reference, in the body of a function definition or prepared statement
* A subscripted expression
* A field selection expression
* An operator invocation
* A function call
* An aggregate expression
* A window function call
* A type cast
* A collation expression
* A scalar subquery
* An array constructor
* A row constructor
* Another value expression in parentheses (used to group subexpressions and override precedence)

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html). An example is the IS NULL clause.

We have already discussed constants in [**Section 4.1.2**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS). The following sections discuss the remaining options.

### 4.2.1. Column References

A column can be referenced in the form:

***correlation***.***columnname***

***correlation*** is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a FROM clause. The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query. (See also [**Chapter 7**](https://www.postgresql.org/docs/10/queries.html).)

### 4.2.2. Positional Parameters 位置的参数

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

$***number***

For example, consider the definition of a function, dept, as:

CREATE FUNCTION dept(text) RETURNS dept

AS $$ SELECT \* FROM dept WHERE name = $1 $$

LANGUAGE SQL;

Here the $1 references the value of the first function argument whenever the function is invoked.

### 4.2.3. Subscripts 下标

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing

***expression***[***subscript***]

or multiple adjacent elements (an “array slice”) can be extracted by writing

***expression***[***lower\_subscript***:***upper\_subscript***]

(Here, the brackets [ ] are meant to appear literally.) Each ***subscript*** is itself an expression, which must yield an integer value.

In general the array ***expression*** must be parenthesized, but the parentheses can be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

mytable.arraycolumn[4]

mytable.two\_d\_column[17][34]

$1[10:42]

(arrayfunction(a,b))[42]

The parentheses in the last example are required. See [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html) for more about arrays.

### 4.2.4. Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing

***expression***.***fieldname***

In general the row ***expression*** must be parenthesized, but the parentheses can be omitted when the expression to be selected from is just a table reference or positional parameter. For example:

mytable.mycolumn

$1.somecolumn

(rowfunction(a,b)).col3

(Thus, a qualified column reference is actually just a special case of the field selection syntax.) An important special case is extracting a field from a table column that is of a composite type:

(compositecol).somefield

(mytable.compositecol).somefield

The parentheses are required here to show that compositecol is a column name not a table name, or that mytable is a table name not a schema name in the second case.

You can ask for all fields of a composite value by writing .\*:

(compositecol).\*

This notation behaves differently depending on context; see [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE) for details.

### 4.2.5. Operator Invocations ？？？

There are three possible syntaxes for an operator invocation:

|  |
| --- |
| ***expression*** ***operator*** ***expression*** (binary infix operator) |
| ***operator*** ***expression*** (unary prefix operator) |
| ***expression*** ***operator*** (unary postfix operator) |

where the ***operator*** token follows the syntax rules of [**Section 4.1.3**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-OPERATORS), or is one of the key words AND, OR, and NOT, or is a qualified operator name in the form:

OPERATOR(***schema***.***operatorname***)

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html) describes the built-in operators.

### 4.2.6. Function Calls ？？？

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

***function\_name*** ([***expression*** [, ***expression*** ... ]] )

For example, the following computes the square root of 2:

sqrt(2)

The list of built-in functions is in [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html). Other functions can be added by the user.

When issuing queries in a database where some users mistrust other users, observe security precautions from [**Section 10.3**](https://www.postgresql.org/docs/10/typeconv-func.html) when writing function calls.

The arguments can optionally have names attached. See [**Section 4.3**](https://www.postgresql.org/docs/10/sql-syntax-calling-funcs.html) for details.

**Note**

A function that takes a single argument of composite type can optionally be called using field-selection syntax, and conversely field selection can be written in functional style. That is, the notations col(table) and table.col are interchangeable. This behavior is not SQL-standard but is provided in PostgreSQL because it allows use of functions to emulate “computed fields”. For more information see [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE).

### 4.2.7. Aggregate Expressions ？？？

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

***aggregate\_name*** (***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** (ALL ***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** (DISTINCT ***expression*** [ , ... ] [ ***order\_by\_clause*** ] ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ]

***aggregate\_name*** ( [ ***expression*** [ , ... ] ] ) WITHIN GROUP ( ***order\_by\_clause*** ) [ FILTER ( WHERE ***filter\_clause*** ) ]

where ***aggregate\_name*** is a previously defined aggregate (possibly qualified with a schema name) and ***expression*** is any value expression that does not itself contain an aggregate expression or a window function call. The optional ***order\_by\_clause*** and ***filter\_clause*** are described below.

The first form of aggregate expression invokes the aggregate once for each input row. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate once for each distinct value of the expression (or distinct set of values, for multiple expressions) found in the input rows. The fourth form invokes the aggregate once for each input row; since no particular input value is specified, it is generally only useful for the count(\*) aggregate function. The last form is used with ordered-set aggregate functions, which are described below.

Most aggregate functions ignore null inputs, so that rows in which one or more of the expression(s) yield null are discarded. This can be assumed to be true, unless otherwise specified, for all built-in aggregates.

For example, count(\*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null, since count ignores nulls; and count(distinct f1) yields the number of distinct non-null values of f1.

Ordinarily, the input rows are fed to the aggregate function in an unspecified order. In many cases this does not matter; for example, min produces the same result no matter what order it receives the inputs in. However, some aggregate functions (such as array\_agg and string\_agg) produce results that depend on the ordering of the input rows. When using such an aggregate, the optional ***order\_by\_clause*** can be used to specify the desired ordering. The ***order\_by\_clause*** has the same syntax as for a query-level ORDER BY clause, as described in [**Section 7.5**](https://www.postgresql.org/docs/10/queries-order.html), except that its expressions are always just expressions and cannot be output-column names or numbers. For example:

SELECT array\_agg(a ORDER BY b DESC) FROM table;

When dealing with multiple-argument aggregate functions, note that the ORDER BY clause goes after all the aggregate arguments. For example, write this:

SELECT string\_agg(a, ',' ORDER BY a) FROM table;

not this:

SELECT string\_agg(a ORDER BY a, ',') FROM table; -- incorrect

The latter is syntactically valid, but it represents a call of a single-argument aggregate function with two ORDER BY keys (the second one being rather useless since it's a constant).

If DISTINCT is specified in addition to an ***order\_by\_clause***, then all the ORDER BY expressions must match regular arguments of the aggregate; that is, you cannot sort on an expression that is not included in the DISTINCT list.

Note

The ability to specify both DISTINCT and ORDER BY in an aggregate function is a PostgreSQLextension.

Placing ORDER BY within the aggregate's regular argument list, as described so far, is used when ordering the input rows for general-purpose and statistical aggregates, for which ordering is optional. There is a subclass of aggregate functions called ordered-set aggregates for which an ***order\_by\_clause*** is required, usually because the aggregate's computation is only sensible in terms of a specific ordering of its input rows. Typical examples of ordered-set aggregates include rank and percentile calculations. For an ordered-set aggregate, the ***order\_by\_clause*** is written inside WITHIN GROUP (...), as shown in the final syntax alternative above. The expressions in the ***order\_by\_clause*** are evaluated once per input row just like regular aggregate arguments, sorted as per the ***order\_by\_clause***'s requirements, and fed to the aggregate function as input arguments. (This is unlike the case for a non-WITHIN GROUP ***order\_by\_clause***, which is not treated as argument(s) to the aggregate function.) The argument expressions preceding WITHIN GROUP, if any, are called direct arguments to distinguish them from the aggregated arguments listed in the ***order\_by\_clause***. Unlike regular aggregate arguments, direct arguments are evaluated only once per aggregate call, not once per input row. This means that they can contain variables only if those variables are grouped by GROUP BY; this restriction is the same as if the direct arguments were not inside an aggregate expression at all. Direct arguments are typically used for things like percentile fractions, which only make sense as a single value per aggregation calculation. The direct argument list can be empty; in this case, write just () not (\*). (PostgreSQL will actually accept either spelling, but only the first way conforms to the SQL standard.)

An example of an ordered-set aggregate call is:

SELECT percentile\_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;

percentile\_cont

-----------------

50489

which obtains the 50th percentile, or median, value of the income column from table households. Here, 0.5 is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

If FILTER is specified, then only the input rows for which the ***filter\_clause*** evaluates to true are fed to the aggregate function; other rows are discarded. For example:

SELECT

count(\*) AS unfiltered,

count(\*) FILTER (WHERE i < 5) AS filtered

FROM generate\_series(1,10) AS s(i);

unfiltered | filtered

------------+----------

10 | 4

(1 row)

The predefined aggregate functions are described in [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html). Other aggregate functions can be added by the user.

An aggregate expression can only appear in the result list or HAVING clause of a SELECT command. It is forbidden in other clauses, such as WHERE, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see [**Section 4.2.11**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-SCALAR-SUBQUERIES) and [**Section 9.22**](https://www.postgresql.org/docs/10/functions-subquery.html)), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments (and ***filter\_clause*** if any) contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies with respect to the query level that the aggregate belongs to.

### 4.2.8. Window Function Calls ？？？

A window function call represents the application of an aggregate-like function over some portion of the rows selected by a query. Unlike non-window aggregate calls, this is not tied to grouping of the selected rows into a single output row — each row remains separate in the query output. However the window function has access to all the rows that would be part of the current row's group according to the grouping specification (PARTITION BY list) of the window function call. The syntax of a window function call is one of the following:

***function\_name*** ([***expression*** [, ***expression*** ... ]]) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ***window\_name***

***function\_name*** ([***expression*** [, ***expression*** ... ]]) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ( ***window\_definition*** )

***function\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ***window\_name***

***function\_name*** ( \* ) [ FILTER ( WHERE ***filter\_clause*** ) ] OVER ( ***window\_definition*** )

where ***window\_definition*** has the syntax

[ ***existing\_window\_name*** ]

[ PARTITION BY ***expression*** [, ...] ]

[ ORDER BY ***expression*** [ ASC | DESC | USING ***operator*** ] [ NULLS { FIRST | LAST } ] [, ...] ]

[ ***frame\_clause*** ]

and the optional ***frame\_clause*** can be one of

{ RANGE | ROWS } ***frame\_start***

{ RANGE | ROWS } BETWEEN ***frame\_start*** AND ***frame\_end***

where ***frame\_start*** and ***frame\_end*** can be one of

UNBOUNDED PRECEDING

***value*** PRECEDING

CURRENT ROW

***value*** FOLLOWING

UNBOUNDED FOLLOWING

Here, ***expression*** represents any value expression that does not itself contain window function calls.

***window\_name*** is a reference to a named window specification defined in the query's WINDOW clause. Alternatively, a full ***window\_definition*** can be given within parentheses, using the same syntax as for defining a named window in the WINDOW clause; see the [**SELECT**](https://www.postgresql.org/docs/10/sql-select.html) reference page for details. It's worth pointing out that OVER wname is not exactly equivalent to OVER (wname ...); the latter implies copying and modifying the window definition, and will be rejected if the referenced window specification includes a frame clause.

The PARTITION BY clause groups the rows of the query into partitions, which are processed separately by the window function. PARTITION BY works similarly to a query-level GROUP BY clause, except that its expressions are always just expressions and cannot be output-column names or numbers. Without PARTITION BY, all rows produced by the query are treated as a single partition. The ORDER BYclause determines the order in which the rows of a partition are processed by the window function. It works similarly to a query-level ORDER BY clause, but likewise cannot use output-column names or numbers. Without ORDER BY, rows are processed in an unspecified order.

The ***frame\_clause*** specifies the set of rows constituting the window frame, which is a subset of the current partition, for those window functions that act on the frame instead of the whole partition. The frame can be specified in either RANGE or ROWS mode; in either case, it runs from the ***frame\_start*** to the ***frame\_end***. If ***frame\_end*** is omitted, it defaults to CURRENT ROW.

A ***frame\_start*** of UNBOUNDED PRECEDING means that the frame starts with the first row of the partition, and similarly a ***frame\_end*** of UNBOUNDED FOLLOWING means that the frame ends with the last row of the partition.

In RANGE mode, a ***frame\_start*** of CURRENT ROW means the frame starts with the current row's first peer row (a row that ORDER BY considers equivalent to the current row), while a ***frame\_end*** of CURRENT ROW means the frame ends with the last equivalent ORDER BY peer. In ROWS mode, CURRENT ROW simply means the current row.

The ***value*** PRECEDING and ***value*** FOLLOWING cases are currently only allowed in ROWS mode. They indicate that the frame starts or ends the specified number of rows before or after the current row. ***value*** must be an integer expression not containing any variables, aggregate functions, or window functions. The value must not be null or negative; but it can be zero, which just selects the current row.

The default framing option is RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. With ORDER BY, this sets the frame to be all rows from the partition start up through the current row's last ORDER BY peer. Without ORDER BY, all rows of the partition are included in the window frame, since all rows become peers of the current row.

Restrictions are that ***frame\_start*** cannot be UNBOUNDED FOLLOWING, ***frame\_end*** cannot be UNBOUNDED PRECEDING, and the ***frame\_end*** choice cannot appear earlier in the above list than the ***frame\_start***choice — for example RANGE BETWEEN CURRENT ROW AND ***value*** PRECEDING is not allowed.

If FILTER is specified, then only the input rows for which the ***filter\_clause*** evaluates to true are fed to the window function; other rows are discarded. Only window functions that are aggregates accept a FILTER clause.

The built-in window functions are described in [**Table 9.57**](https://www.postgresql.org/docs/10/functions-window.html#FUNCTIONS-WINDOW-TABLE). Other window functions can be added by the user. Also, any built-in or user-defined general-purpose or statistical aggregate can be used as a window function. (Ordered-set and hypothetical-set aggregates cannot presently be used as window functions.)

The syntaxes using \* are used for calling parameter-less aggregate functions as window functions, for example count(\*) OVER (PARTITION BY x ORDER BY y). The asterisk (\*) is customarily not used for window-specific functions. Window-specific functions do not allow DISTINCT or ORDER BY to be used within the function argument list.

Window function calls are permitted only in the SELECT list and the ORDER BY clause of the query.

More information about window functions can be found in [**Section 3.5**](https://www.postgresql.org/docs/10/tutorial-window.html), [**Section 9.21**](https://www.postgresql.org/docs/10/functions-window.html), and [**Section 7.2.5**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-WINDOW).

### 4.2.9. Type Casts ？？？

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

CAST ( ***expression*** AS ***type*** )

***expression***::***type***

The CAST syntax conforms to SQL; the syntax with :: is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants, as shown in [**Section 4.1.2.7**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC). A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast can usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked “OK to apply implicitly” in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:

***typename*** ( ***expression*** )

However, this only works for types whose names are also valid as function names. For example, double precision cannot be used this way, but the equivalent float8 can. Also, the names interval, time, and timestamp can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided.

Note

The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform the conversion. By convention, these conversion functions have the same name as their output type, and thus the “function-like syntax” is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on. For further details see [**CREATE CAST**](https://www.postgresql.org/docs/10/sql-createcast.html).

### 4.2.10. Collation Expressions 核对表达式 ？？

The COLLATE clause overrides the collation of an expression. It is appended to the expression it applies to:

***expr*** COLLATE ***collation***

where ***collation*** is a possibly schema-qualified identifier. The COLLATE clause binds tighter than operators; parentheses can be used when necessary.

If no collation is explicitly specified, the database system either derives a collation from the columns involved in the expression, or it defaults to the default collation of the database if no column is involved in the expression.

The two common uses of the COLLATE clause are overriding the sort order in an ORDER BY clause, for example:

SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";

and overriding the collation of a function or operator call that has locale-sensitive results, for example:

SELECT \* FROM tbl WHERE a > 'foo' COLLATE "C";

Note that in the latter case the COLLATE clause is attached to an input argument of the operator we wish to affect. It doesn't matter which argument of the operator or function call the COLLATE clause is attached to, because the collation that is applied by the operator or function is derived by considering all arguments, and an explicit COLLATE clause will override the collations of all other arguments. (Attaching non-matching COLLATE clauses to more than one argument, however, is an error. For more details see [**Section 23.2**](https://www.postgresql.org/docs/10/collation.html).) Thus, this gives the same result as the previous example:

SELECT \* FROM tbl WHERE a COLLATE "C" > 'foo';

But this is an error:

SELECT \* FROM tbl WHERE (a > 'foo') COLLATE "C";

because it attempts to apply a collation to the result of the > operator, which is of the non-collatable data type boolean.

### 4.2.11. Scalar Subqueries 子查询的结果集

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. (See [**Chapter 7**](https://www.postgresql.org/docs/10/queries.html) for information about writing queries.) The SELECT query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also [**Section 9.22**](https://www.postgresql.org/docs/10/functions-subquery.html) for other expressions involving subqueries.

For example, the following finds the largest city population in each state:

SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)

FROM states;

### 4.2.12. Array Constructors 构造一个数组

An array constructor is an expression that builds an array value using values for its member elements. A simple array constructor consists of the key word ARRAY, a left square bracket [, a list of expressions (separated by commas) for the array element values, and finally a right square bracket ]. For example:

SELECT ARRAY[1,2,3+4];

array

---------

{1,2,7}

(1 row)

By default, the array element type is the common type of the member expressions, determined using the same rules as for UNION or CASE constructs (see [**Section 10.5**](https://www.postgresql.org/docs/10/typeconv-union-case.html)). You can override this by explicitly casting the array constructor to the desired type, for example:

SELECT ARRAY[1,2,22.7]::integer[];

array

----------

{1,2,23}

(1 row)

This has the same effect as casting each expression to the array element type individually. For more on casting, see [**Section 4.2.9**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-TYPE-CASTS).

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word ARRAY can be omitted. For example, these produce the same result:

SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];

array

---------------

{{1,2},{3,4}}

(1 row)

SELECT ARRAY[[1,2],[3,4]];

array

---------------

{{1,2},{3,4}}

(1 row)

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions. Any cast applied to the outer ARRAY constructor propagates automatically to all the inner constructors.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;

array

------------------------------------------------

{{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}

(1 row)

You can construct an empty array, but since it's impossible to have an array with no type, you must explicitly cast your empty array to the desired type. For example:

SELECT ARRAY[]::integer[];

array

-------

{}

(1 row)

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word ARRAY followed by a parenthesized (not bracketed) subquery. For example:

SELECT ARRAY(SELECT oid FROM pg\_proc WHERE proname LIKE 'bytea%');

array

-----------------------------------------------------------------------

{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}

(1 row)

SELECT ARRAY(SELECT ARRAY[i, i\*2] FROM generate\_series(1,5) AS a(i));

array

----------------------------------

{{1,2},{2,4},{3,6},{4,8},{5,10}}

(1 row)

The subquery must return a single column. If the subquery's output column is of a non-array type, the resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery's output column. If the subquery's output column is of an array type, the result will be an array of the same type but one higher dimension; in this case all the subquery rows must yield arrays of identical dimensionality, else the result would not be rectangular.

The subscripts of an array value built with ARRAY always begin with one. For more information about arrays, see [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html).

### 4.2.13. Row Constructors 构造一行数据

A row constructor is an expression that builds a row value (also called a composite value) using values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example:

SELECT ROW(1,2.5,'this is a test');

The key word ROW is optional when there is more than one expression in the list.

A row constructor can include the syntax ***rowvalue***.\*, which will be expanded to a list of the elements of the row value, just as occurs when the .\* syntax is used at the top level of a SELECT list (see [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE)). For example, if table t has columns f1 and f2, these are the same:

SELECT ROW(t.\*, 42) FROM t;

SELECT ROW(t.f1, t.f2, 42) FROM t;

Note

Before PostgreSQL 8.2, the .\* syntax was not expanded in row constructors, so that writing ROW(t.\*, 42) created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without .\*, for instance ROW(t, 42).

By default, the value created by a ROW expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with CREATE TYPE AS. An explicit cast might be needed to avoid ambiguity. For example:

CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- No cast needed since only one getf1() exists

SELECT getf1(ROW(1,2.5,'this is a test'));

getf1

-------

1

(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:

SELECT getf1(ROW(1,2.5,'this is a test'));

ERROR: function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);

getf1

-------

1

(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));

getf1

-------

11

(1 row)

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to compare two row values or test a row with IS NULL or IS NOT NULL, for example:

SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.\*) IS NULL FROM table; -- detect all-null rows

For more detail see [**Section 9.23**](https://www.postgresql.org/docs/10/functions-comparisons.html). Row constructors can also be used in connection with subqueries, as discussed in [**Section 9.22**](https://www.postgresql.org/docs/10/functions-subquery.html).

### 4.2.14. Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

SELECT true OR somefunc();

then somefunc() would (probably) not be called at all. The same would be the case if one wrote:

SELECT somefunc() OR true;

Note that this is not the same as the left-to-right “short-circuiting” of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in WHERE and HAVING clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (AND/OR/NOT combinations) in those clauses can be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a CASE construct (see [**Section 9.17**](https://www.postgresql.org/docs/10/functions-conditional.html)) can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a WHERE clause:

SELECT ... WHERE x > 0 AND y/x > 1.5;

But this is safe:

SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;

A CASE construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would be better to sidestep the problem by writing y > 1.5\*x instead.)

CASE is not a cure-all for such issues, however. One limitation of the technique illustrated above is that it does not prevent early evaluation of constant subexpressions. As described in [**Section 37.6**](https://www.postgresql.org/docs/10/xfunc-volatility.html), functions and operators marked IMMUTABLE can be evaluated when the query is planned rather than when it is executed. Thus for example

SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;

is likely to result in a division-by-zero failure due to the planner trying to simplify the constant subexpression, even if every row in the table has x > 0 so that the ELSE arm would never be entered at run time.

While that particular example might seem silly, related cases that don't obviously involve constants can occur in queries executed within functions, since the values of function arguments and local variables can be inserted into queries as constants for planning purposes. Within PL/pgSQL functions, for example, using an IF-THEN-ELSE statement to protect a risky computation is much safer than just nesting it in a CASE expression.

Another limitation of the same kind is that a CASE cannot prevent evaluation of an aggregate expression contained within it, because aggregate expressions are computed before other expressions in a SELECT list or HAVING clause are considered. For example, the following query can cause a division-by-zero error despite seemingly having protected against it:

SELECT CASE WHEN min(employees) > 0

THEN avg(expenses / employees)

END

FROM departments;

The min() and avg() aggregates are computed concurrently over all the input rows, so if any row has employees equal to zero, the division-by-zero error will occur before there is any opportunity to test the result of min(). Instead, use a WHERE or FILTER clause to prevent problematic input rows from reaching an aggregate function in the first place.

## 4.3. Calling Functions 函数调用

PostgreSQL allows functions that have named parameters to be called using either positional or named notation. Named notation is especially useful for functions that have a large number of parameters, since it makes the associations between parameters and actual arguments more explicit and reliable. In positional notation, a function call is written with its argument values in the same order as they are defined in the function declaration. In named notation, the arguments are matched to the function parameters by name and can be written in any order. For each notation, also consider the effect of function argument types, documented in [**Section 10.3**](https://www.postgresql.org/docs/10/typeconv-func.html).

In either notation, parameters that have default values given in the function declaration need not be written in the call at all. But this is particularly useful in named notation, since any combination of parameters can be omitted; while in positional notation parameters can only be omitted from right to left.

PostgreSQL also supports mixed notation, which combines positional and named notation. In this case, positional parameters are written first and named parameters appear after them.

The following examples will illustrate the usage of all three notations, using the following function definition:

CREATE FUNCTION concat\_lower\_or\_upper(a text, b text, uppercase boolean DEFAULT false)

RETURNS text

AS

$$

SELECT CASE

WHEN $3 THEN UPPER($1 || ' ' || $2)

ELSE LOWER($1 || ' ' || $2)

END;

$$

LANGUAGE SQL IMMUTABLE STRICT;

Function concat\_lower\_or\_upper has two mandatory parameters, a and b. Additionally there is one optional parameter uppercase which defaults to false. The a and b inputs will be concatenated, and forced to either upper or lower case depending on the uppercase parameter. The remaining details of this function definition are not important here (see [**Chapter 37**](https://www.postgresql.org/docs/10/extend.html) for more information).

### 4.3.1. Using Positional Notation 根据位置设变量

Positional notation is the traditional mechanism for passing arguments to functions in PostgreSQL. An example is:

SELECT concat\_lower\_or\_upper('Hello', 'World', true);

concat\_lower\_or\_upper

-----------------------

HELLO WORLD

(1 row)

All arguments are specified in order. The result is upper case since uppercase is specified as true. Another example is:

SELECT concat\_lower\_or\_upper('Hello', 'World');

concat\_lower\_or\_upper

-----------------------

hello world

(1 row)

Here, the uppercase parameter is omitted, so it receives its default value of false, resulting in lower case output. In positional notation, arguments can be omitted from right to left so long as they have defaults.

### 4.3.2. Using Named Notation 根据命名设变量

In named notation, each argument's name is specified using => to separate it from the argument expression. For example:

SELECT concat\_lower\_or\_upper(a => 'Hello', b => 'World');

concat\_lower\_or\_upper

-----------------------

hello world

(1 row)

Again, the argument uppercase was omitted so it is set to false implicitly. One advantage of using named notation is that the arguments may be specified in any order, for example:

SELECT concat\_lower\_or\_upper(a => 'Hello', b => 'World', uppercase => true);

concat\_lower\_or\_upper

-----------------------

HELLO WORLD

(1 row)

SELECT concat\_lower\_or\_upper(a => 'Hello', uppercase => true, b => 'World');

concat\_lower\_or\_upper

-----------------------

HELLO WORLD

(1 row)

An older syntax based on ":=" is supported for backward compatibility:

SELECT concat\_lower\_or\_upper(a := 'Hello', uppercase := true, b := 'World');

concat\_lower\_or\_upper

-----------------------

HELLO WORLD

(1 row)

### 4.3.3. Using Mixed Notation 混合

The mixed notation combines positional and named notation. However, as already mentioned, named arguments cannot precede positional arguments. For example:

SELECT concat\_lower\_or\_upper('Hello', 'World', uppercase => true);

concat\_lower\_or\_upper

-----------------------

HELLO WORLD

(1 row)

In the above query, the arguments a and b are specified positionally, while uppercase is specified by name. In this example, that adds little except documentation. With a more complex function having numerous parameters that have default values, named or mixed notation can save a great deal of writing and reduce chances for error.

Note

Named and mixed call notations currently cannot be used when calling an aggregate function (but they do work when an aggregate function is used as a window function).

## Chapter 5. Data Definition 数据类型

This chapter covers how one creates the database structures that will hold one's data. In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables. Subsequently, we discuss how tables can be organized into schemas, and how privileges can be assigned to tables. Finally, we will briefly look at other features that affect the data storage, such as inheritance, table partitioning, views, functions, and triggers.

## 5.1. Table Basics

A table in a relational database is much like a table on paper: It consists of rows and columns. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table. When a table is read, the rows will appear in an unspecified order, unless sorting is explicitly requested. This is covered in [**Chapter 7**](https://www.postgresql.org/docs/10/queries.html). Furthermore, SQL does not assign unique identifiers to rows, so it is possible to have several completely identical rows in a table. This is a consequence of the mathematical model that underlies SQL but is usually not desirable. Later in this chapter we will see how to deal with this issue.

Each column has a data type. The data type constrains the set of possible values that can be assigned to a column and assigns semantics to the data stored in the column so that it can be used for computations. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available.

PostgreSQL includes a sizable set of built-in data types that fit many applications. Users can also define their own data types. Most built-in data types have obvious names and semantics, so we defer a detailed explanation to [**Chapter 8**](https://www.postgresql.org/docs/10/datatype.html). Some of the frequently used data types are integer for whole numbers, numeric for possibly fractional numbers, text for character strings, date for dates, time for time-of-day values, and timestamp for values containing both date and time.

To create a table, you use the aptly named [**CREATE TABLE**](https://www.postgresql.org/docs/10/sql-createtable.html) command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column. For example:

CREATE TABLE my\_first\_table (

first\_column text,

second\_column integer

);

This creates a table named my\_first\_table with two columns. The first column is named first\_column and has a data type of text; the second column has the name second\_column and the type integer. The table and column names follow the identifier syntax explained in [**Section 4.1.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-IDENTIFIERS). The type names are usually also identifiers, but there are some exceptions. Note that the column list is comma-separated and surrounded by parentheses.

Of course, the previous example was heavily contrived. Normally, you would give names to your tables and columns that convey what kind of data they store. So let's look at a more realistic example:

CREATE TABLE products (

product\_no integer,

name text,

price numeric

);

(The numeric type can store fractional components, as would be typical of monetary amounts.)

Tip

When you create many interrelated tables it is wise to choose a consistent naming pattern for the tables and columns. For instance, there is a choice of using singular or plural nouns for table names, both of which are favored by some theorist or other.

There is a limit on how many columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.

If you no longer need a table, you can remove it using the [**DROP TABLE**](https://www.postgresql.org/docs/10/sql-droptable.html) command. For example:

DROP TABLE my\_first\_table;

DROP TABLE products;

Attempting to drop a table that does not exist is an error. Nevertheless, it is common in SQL script files to unconditionally try to drop each table before creating it, ignoring any error messages, so that the script works whether or not the table exists. (If you like, you can use the DROP TABLE IF EXISTS variant to avoid the error messages, but this is not standard SQL.)

If you need to modify a table that already exists, see [**Section 5.5**](https://www.postgresql.org/docs/10/ddl-alter.html) later in this chapter.

With the tools discussed so far you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now you can skip ahead to [**Chapter 6**](https://www.postgresql.org/docs/10/dml.html) and read the rest of this chapter later.

## 5.2. Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. (Details about data manipulation commands are in [**Chapter 6**](https://www.postgresql.org/docs/10/dml.html).)

If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

In a table definition, default values are listed after the column data type. For example:

CREATE TABLE products (

product\_no integer,

name text,

price numeric **DEFAULT 9.99**

);

The default value can be an expression, which will be evaluated whenever the default value is inserted (not when the table is created). A common example is for a timestamp column to have a default of CURRENT\_TIMESTAMP, so that it gets set to the time of row insertion. Another common example is generating a “serial number” for each row. In PostgreSQL this is typically done by something like:

CREATE TABLE products (

product\_no integer **DEFAULT nextval('products\_product\_no\_seq')**,

...

);

where the nextval() function supplies successive values from a sequence object (see [**Section 9.16**](https://www.postgresql.org/docs/10/functions-sequence.html)). This arrangement is sufficiently common that there's a special shorthand for it:

CREATE TABLE products (

product\_no **SERIAL**,

...

);

The SERIAL shorthand is discussed further in [**Section 8.1.4**](https://www.postgresql.org/docs/10/datatype-numeric.html#DATATYPE-SERIAL).

## 5.3. Constraints 约束 列内容的约束

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

### 5.3.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CHECK (price > 0)**

);

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word CHECK followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CONSTRAINT positive\_price** CHECK (price > 0)

);

So, to specify a named constraint, use the key word CONSTRAINT followed by an identifier followed by the constraint definition. (If you don't specify a constraint name in this way, the system chooses a name for you.)

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price:

CREATE TABLE products (

product\_no integer,

name text,

price numeric CHECK (price > 0),

discounted\_price numeric CHECK (discounted\_price > 0),

**CHECK (price > discounted\_price)**

);

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two constraints are column constraints, whereas the third one is a table constraint because it is written separately from any one column definition. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to. (PostgreSQL doesn't enforce that rule, but you should follow it if you want your table definitions to work with other database systems.) The above example could also be written as:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

CHECK (price > discounted\_price)

);

or even:

CREATE TABLE products (

product\_no integer,

name text,

price numeric CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0 AND price > discounted\_price)

);

It's a matter of taste.

Names can be assigned to table constraints in the same way as column constraints:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

**CONSTRAINT valid\_discount** CHECK (price > discounted\_price)

);

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if any operand is null, they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section can be used.

### 5.3.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

CREATE TABLE products (

product\_no integer **NOT NULL**,

name text **NOT NULL**,

price numeric

);

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint CHECK (***column\_name*** IS NOT NULL), but in PostgreSQLcreating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created this way.

Of course, a column can have more than one constraint. Just write the constraints one after another:

CREATE TABLE products (

product\_no integer NOT NULL,

name text NOT NULL,

price numeric NOT NULL CHECK (price > 0)

);

The order doesn't matter. It does not necessarily determine in which order the constraints are checked.

The NOT NULL constraint has an inverse: the NULL constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column might be null. The NULL constraint is not present in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with some other database systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with:

CREATE TABLE products (

product\_no integer NULL,

name text NULL,

price numeric NULL

);

and then insert the NOT key word where desired.

Tip

In most database designs the majority of columns should be marked not null.

### 5.3.3. Unique Constraints

Unique constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table. The syntax is:

CREATE TABLE products (

product\_no integer **UNIQUE**,

name text,

price numeric

);

when written as a column constraint, and:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

**UNIQUE (product\_no)**

);

when written as a table constraint.

To define a unique constraint for a group of columns, write it as a table constraint with the column names separated by commas:

CREATE TABLE example (

a integer,

b integer,

c integer,

**UNIQUE (a, c)**

);

This specifies that the combination of values in the indicated columns is unique across the whole table, though any one of the columns need not be (and ordinarily isn't) unique.

You can assign your own name for a unique constraint, in the usual way:

CREATE TABLE products (

product\_no integer **CONSTRAINT must\_be\_different** UNIQUE,

name text,

price numeric

);

Adding a unique constraint will automatically create a unique B-tree index on the column or group of columns listed in the constraint. A uniqueness restriction covering only some rows cannot be written as a unique constraint, but it is possible to enforce such a restriction by creating a unique [**partial index**](https://www.postgresql.org/docs/10/indexes-partial.html).

In general, a unique constraint is violated if there is more than one row in the table where the values of all of the columns included in the constraint are equal. However, two null values are never considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but we have heard that other SQL databases might not follow this rule. So be careful when developing applications that are intended to be portable.

### 5.3.4. Primary Keys

A primary key constraint indicates that a column, or group of columns, can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null. So, the following two table definitions accept the same data:

CREATE TABLE products (

product\_no integer UNIQUE NOT NULL,

name text,

price numeric

);

CREATE TABLE products (

product\_no integer **PRIMARY KEY**,

name text,

price numeric

);

Primary keys can span more than one column; the syntax is similar to unique constraints:

CREATE TABLE example (

a integer,

b integer,

c integer,

**PRIMARY KEY (a, c)**

);

Adding a primary key will automatically create a unique B-tree index on the column or group of columns listed in the primary key, and will force the column(s) to be marked NOT NULL.

A table can have at most one primary key. (There can be any number of unique and not-null constraints, which are functionally almost the same thing, but only one can be identified as the primary key.) Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

Primary keys are useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely. There are also various ways in which the database system makes use of a primary key if one has been declared; for example, the primary key defines the default target column(s) for foreign keys referencing its table.

### 5.3.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integritybetween two related tables.

Say you have the product table that we have used several times already:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

product\_no integer **REFERENCES products (product\_no)**,

quantity integer

);

Now it is impossible to create orders with non-NULL product\_no entries that do not appear in the products table.

We say that in this situation the orders table is the referencing table and the products table is the referenced table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to:

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

product\_no integer **REFERENCES products**,

quantity integer

);

because in absence of a column list the primary key of the referenced table is used as the referenced column(s).

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

CREATE TABLE t1 (

a integer PRIMARY KEY,

b integer,

c integer,

**FOREIGN KEY (b, c) REFERENCES other\_table (c1, c2)**

);

Of course, the number and type of the constrained columns need to match the number and type of the referenced columns.

You can assign your own name for a foreign key constraint, in the usual way.

A table can have more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products,

order\_id integer REFERENCES orders,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

Notice that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that do not relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to handle that as well. Intuitively, we have a few options:

* Disallow deleting a referenced product
* Delete the orders as well
* Something else?

To illustrate this, let's implement the following policy on the many-to-many relationship example above: when someone wants to remove a product that is still referenced by an order (via order\_items), we disallow it. If someone removes an order, the order items are removed as well:

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products **ON DELETE RESTRICT**,

order\_id integer REFERENCES orders **ON DELETE CASCADE**,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

Restricting and cascading deletes are the two most common options. RESTRICT prevents deletion of a referenced row. NO ACTION means that if any referencing rows still exist when the constraint is checked, an error is raised; this is the default behavior if you do not specify anything. (The essential difference between these two choices is that NO ACTION allows the check to be deferred until later in the transaction, whereas RESTRICT does not.) CASCADE specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well. There are two other options: SET NULL and SET DEFAULT. These cause the referencing column(s) in the referencing row(s) to be set to nulls or their default values, respectively, when the referenced row is deleted. Note that these do not excuse you from observing any constraints. For example, if an action specifies SET DEFAULT but the default value would not satisfy the foreign key constraint, the operation will fail.

Analogous to ON DELETE there is also ON UPDATE which is invoked when a referenced column is changed (updated). The possible actions are the same. In this case, CASCADE means that the updated values of the referenced column(s) should be copied into the referencing row(s).

Normally, a referencing row need not satisfy the foreign key constraint if any of its referencing columns are null. If MATCH FULL is added to the foreign key declaration, a referencing row escapes satisfying the constraint only if all its referencing columns are null (so a mix of null and non-null values is guaranteed to fail a MATCH FULL constraint). If you don't want referencing rows to be able to avoid satisfying the foreign key constraint, declare the referencing column(s) as NOT NULL.

A foreign key must reference columns that either are a primary key or form a unique constraint. This means that the referenced columns always have an index (the one underlying the primary key or unique constraint); so checks on whether a referencing row has a match will be efficient. Since a DELETE of a row from the referenced table or an UPDATE of a referenced column will require a scan of the referencing table for rows matching the old value, it is often a good idea to index the referencing columns too. Because this is not always needed, and there are many choices available on how to index, declaration of a foreign key constraint does not automatically create an index on the referencing columns.

More information about updating and deleting data is in [**Chapter 6**](https://www.postgresql.org/docs/10/dml.html). Also see the description of foreign key constraint syntax in the reference documentation for [**CREATE TABLE**](https://www.postgresql.org/docs/10/sql-createtable.html).

### 5.3.6. Exclusion Constraints 排斥约束

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null. The syntax is:

CREATE TABLE circles (

c circle,

EXCLUDE USING gist (c WITH &&)

);

See also [CREATE TABLE ... CONSTRAINT ... EXCLUDE](https://www.postgresql.org/docs/10/sql-createtable.html#SQL-CREATETABLE-EXCLUDE) for details.

Adding an exclusion constraint will automatically create an index of the type specified in the constraint declaration.

**5.4. System Columns 系统列**

Every table has several *system columns* that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns; just know they exist.

oid

The object identifier (object ID) of a row. This column is only present if the table was created using WITH OIDS, or if the [**default\_with\_oids**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-DEFAULT-WITH-OIDS) configuration variable was set at the time. This column is of type oid (same name as the column); see [**Section 8.18**](https://www.postgresql.org/docs/10/datatype-oid.html) for more information about the type.

tableoid

The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies (see [**Section 5.9**](https://www.postgresql.org/docs/10/ddl-inherit.html)), since without it, it's difficult to tell which individual table a row came from. The tableoid can be joined against the oid column of pg\_class to obtain the table name.

xmin

The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

cmin

The command identifier (starting at zero) within the inserting transaction.

xmax

The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

cmax

The command identifier within the deleting transaction, or zero.

ctid

The physical location of the row version within its table. Note that although the ctid can be used to locate the row version very quickly, a row's ctid will change if it is updated or moved by VACUUM FULL. Therefore ctid is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that this is the case. If you need to identify the rows in a table, using a sequence generator is strongly recommended. However, OIDs can be used as well, provided that a few additional precautions are taken:

* A unique constraint should be created on the OID column of each table for which the OID will be used to identify rows. When such a unique constraint (or unique index) exists, the system takes care not to generate an OID matching an already-existing row. (Of course, this is only possible if the table contains fewer than 232 (4 billion) rows, and in practice the table size had better be much less than that, or performance might suffer.)
* OIDs should never be assumed to be unique across tables; use the combination of tableoid and row OID if you need a database-wide identifier.
* Of course, the tables in question must be created WITH OIDS. As of PostgreSQL 8.1, WITHOUT OIDS is the default.

Transaction identifiers are also 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see [**Chapter 24**](https://www.postgresql.org/docs/10/maintenance.html) for details. It is unwise, however, to depend on the uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 232 (4 billion) SQL commands within a single transaction. In practice this limit is not a problem — note that the limit is on the number of SQL commands, not the number of rows processed. Also, only commands that actually modify the database contents will consume a command identifier.

## 5.5. Modifying Tables 表修改

When you create a table and you realize that you made a mistake, or the requirements of the application change, you can drop the table and create it again. But this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign key constraint). Therefore PostgreSQL provides a family of commands to make modifications to existing tables. Note that this is conceptually distinct from altering the data contained in the table: here we are interested in altering the definition, or structure, of the table.

You can:

* Add columns
* Remove columns
* Add constraints
* Remove constraints
* Change default values
* Change column data types
* Rename columns
* Rename tables

All these actions are performed using the [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html) command, whose reference page contains details beyond those given here.

### 5.5.1. Adding a Column

To add a column, use a command like:

ALTER TABLE products ADD COLUMN description text;

The new column is initially filled with whatever default value is given (null if you don't specify a DEFAULT clause).

You can also define constraints on the column at the same time, using the usual syntax:

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

In fact all the options that can be applied to a column description in CREATE TABLE can be used here. Keep in mind however that the default value must satisfy the given constraints, or the ADD will fail. Alternatively, you can add constraints later (see below) after you've filled in the new column correctly.

Tip

Adding a column with a default requires updating each row of the table (to store the new column value). However, if no default is specified, PostgreSQL is able to avoid the physical update. So if you intend to fill the column with mostly nondefault values, it's best to add the column with no default, insert the correct values using UPDATE, and then add any desired default as described below.

### 5.5.2. Removing a Column

To remove a column, use a command like:

ALTER TABLE products DROP COLUMN description;

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQLwill not silently drop that constraint. You can authorize dropping everything that depends on the column by adding CASCADE:

ALTER TABLE products DROP COLUMN description CASCADE;

See [**Section 5.13**](https://www.postgresql.org/docs/10/ddl-depend.html) for a description of the general mechanism behind this.

### 5.5.3. Adding a Constraint

To add a constraint, the table constraint syntax is used. For example:

ALTER TABLE products ADD CHECK (name <> '');

ALTER TABLE products ADD CONSTRAINT some\_name UNIQUE (product\_no);

ALTER TABLE products ADD FOREIGN KEY (product\_group\_id) REFERENCES product\_groups;

To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

ALTER TABLE products ALTER COLUMN product\_no SET NOT NULL;

The constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.

### 5.5.4. Removing a Constraint

To remove a constraint you need to know its name. If you gave it a name then that's easy. Otherwise the system assigned a generated name, which you need to find out. The psql command \d***tablename*** can be helpful here; other interfaces might also provide a way to inspect table details. Then the command is:

ALTER TABLE products DROP CONSTRAINT some\_name;

(If you are dealing with a generated constraint name like $2, don't forget that you'll need to double-quote it to make it a valid identifier.)

As with dropping a column, you need to add CASCADE if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced column(s).

This works the same for all constraint types except not-null constraints. To drop a not null constraint use:

ALTER TABLE products ALTER COLUMN product\_no DROP NOT NULL;

(Recall that not-null constraints do not have names.)

### 5.5.5. Changing a Column's Default Value

To set a new default for a column, use a command like:

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

Note that this doesn't affect any existing rows in the table, it just changes the default for future INSERT commands.

To remove any default value, use:

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn't been defined, because the default is implicitly the null value.

### 5.5.6. Changing a Column's Data Type

To convert a column to a different data type, use a command like:

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

This will succeed only if each existing entry in the column can be converted to the new type by an implicit cast. If a more complex conversion is needed, you can add a USING clause that specifies how to compute the new values from the old.

PostgreSQL will attempt to convert the column's default value (if any) to the new type, as well as any constraints that involve the column. But these conversions might fail, or might produce surprising results. It's often best to drop any constraints on the column before altering its type, and then add back suitably modified constraints afterwards.

### 5.5.7. Renaming a Column

To rename a column:

ALTER TABLE products RENAME COLUMN product\_no TO product\_number;

### 5.5.8. Renaming a Table

To rename a table:

ALTER TABLE products RENAME TO items;

## 5.6. Privileges 权限设置

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted.

There are different kinds of privileges: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, and USAGE. The privileges applicable to a particular object vary depending on the object's type (table, function, etc). For complete information on the different types of privileges supported by PostgreSQL, refer to the [**GRANT**](https://www.postgresql.org/docs/10/sql-grant.html) reference page. The following sections and chapters will also show you how those privileges are used.

The right to modify or destroy an object is always the privilege of the owner only.

An object can be assigned to a new owner with an ALTER command of the appropriate kind for the object, e.g. [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html). Superusers can always do this; ordinary roles can only do it if they are both the current owner of the object (or a member of the owning role) and a member of the new owning role.

To assign privileges, the GRANT command is used. For example, if joe is an existing role, and accounts is an existing table, the privilege to update the table can be granted with:

GRANT UPDATE ON accounts TO joe;

Writing ALL in place of a specific privilege grants all privileges that are relevant for the object type.

The special “role” name PUBLIC can be used to grant a privilege to every role on the system. Also, “group” roles can be set up to help manage privileges when there are many users of a database — for details see [**Chapter 21**](https://www.postgresql.org/docs/10/user-manag.html).

To revoke a privilege, use the fittingly named REVOKE command:

REVOKE ALL ON accounts FROM PUBLIC;

The special privileges of the object owner (i.e., the right to do DROP, GRANT, REVOKE, etc.) are always implicit in being the owner, and cannot be granted or revoked. But the object owner can choose to revoke their own ordinary privileges, for example to make a table read-only for themselves as well as others.

Ordinarily, only the object's owner (or a superuser) can grant or revoke privileges on an object. However, it is possible to grant a privilege “with grant option”, which gives the recipient the right to grant it in turn to others. If the grant option is subsequently revoked then all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege. For details see the [**GRANT**](https://www.postgresql.org/docs/10/sql-grant.html) and [**REVOKE**](https://www.postgresql.org/docs/10/sql-revoke.html) reference pages.

## 5.7. Row Security Policies 行安全策略 ！！！

In addition to the SQL-standard [**privilege system**](https://www.postgresql.org/docs/10/ddl-priv.html) available through [**GRANT**](https://www.postgresql.org/docs/10/sql-grant.html), tables can have row security policies that restrict, on a per-user basis, which rows can be returned by normal queries or inserted, updated, or deleted by data modification commands. This feature is also known as Row-Level Security. By default, tables do not have any policies, so that if a user has access privileges to a table according to the SQL privilege system, all rows within it are equally available for querying or updating.

When row security is enabled on a table (with [**ALTER TABLE ... ENABLE ROW LEVEL SECURITY**](https://www.postgresql.org/docs/10/sql-altertable.html)), all normal access to the table for selecting rows or modifying rows must be allowed by a row security policy. (However, the table's owner is typically not subject to row security policies.) If no policy exists for the table, a default-deny policy is used, meaning that no rows are visible or can be modified. Operations that apply to the whole table, such as TRUNCATE and REFERENCES, are not subject to row security.

Row security policies can be specific to commands, or to roles, or to both. A policy can be specified to apply to ALL commands, or to SELECT, INSERT, UPDATE, or DELETE. Multiple roles can be assigned to a given policy, and normal role membership and inheritance rules apply.

To specify which rows are visible or modifiable according to a policy, an expression is required that returns a Boolean result. This expression will be evaluated for each row prior to any conditions or functions coming from the user's query. (The only exceptions to this rule are leakproof functions, which are guaranteed to not leak information; the optimizer may choose to apply such functions ahead of the row-security check.) Rows for which the expression does not return true will not be processed. Separate expressions may be specified to provide independent control over the rows which are visible and the rows which are allowed to be modified. Policy expressions are run as part of the query and with the privileges of the user running the query, although security-definer functions can be used to access data not available to the calling user.

Superusers and roles with the BYPASSRLS attribute always bypass the row security system when accessing a table. Table owners normally bypass row security as well, though a table owner can choose to be subject to row security with [**ALTER TABLE ... FORCE ROW LEVEL SECURITY**](https://www.postgresql.org/docs/10/sql-altertable.html).

Enabling and disabling row security, as well as adding policies to a table, is always the privilege of the table owner only.

Policies are created using the [**CREATE POLICY**](https://www.postgresql.org/docs/10/sql-createpolicy.html) command, altered using the [**ALTER POLICY**](https://www.postgresql.org/docs/10/sql-alterpolicy.html) command, and dropped using the [**DROP POLICY**](https://www.postgresql.org/docs/10/sql-droppolicy.html) command. To enable and disable row security for a given table, use the [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html) command.

Each policy has a name and multiple policies can be defined for a table. As policies are table-specific, each policy for a table must have a unique name. Different tables may have policies with the same name.

When multiple policies apply to a given query, they are combined using either OR (for permissive policies, which are the default) or using AND (for restrictive policies). This is similar to the rule that a given role has the privileges of all roles that they are a member of. Permissive vs. restrictive policies are discussed further below.

As a simple example, here is how to create a policy on the account relation to allow only members of the managers role to access rows, and only rows of their accounts:

CREATE TABLE accounts (manager text, company text, contact\_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account\_managers ON accounts TO managers

USING (manager = current\_user);

The policy above implicitly provides a WITH CHECK clause identical to its USING clause, so that the constraint applies both to rows selected by a command (so a manager cannot SELECT, UPDATE, or DELETEexisting rows belonging to a different manager) and to rows modified by a command (so rows belonging to a different manager cannot be created via INSERT or UPDATE).

If no role is specified, or the special user name PUBLIC is used, then the policy applies to all users on the system. To allow all users to access only their own row in a users table, a simple policy can be used:

CREATE POLICY user\_policy ON users

USING (user\_name = current\_user);

This works similarly to the previous example.

To use a different policy for rows that are being added to the table compared to those rows that are visible, multiple policies can be combined. This pair of policies would allow all users to view all rows in the users table, but only modify their own:

CREATE POLICY user\_sel\_policy ON users

FOR SELECT

USING (true);

CREATE POLICY user\_mod\_policy ON users

USING (user\_name = current\_user);

In a SELECT command, these two policies are combined using OR, with the net effect being that all rows can be selected. In other command types, only the second policy applies, so that the effects are the same as before.

Row security can also be disabled with the ALTER TABLE command. Disabling row security does not remove any policies that are defined on the table; they are simply ignored. Then all rows in the table are visible and modifiable, subject to the standard SQL privileges system.

Below is a larger example of how this feature can be used in production environments. The table passwd emulates a Unix password file:

-- Simple passwd-file based example

CREATE TABLE passwd (

user\_name text UNIQUE NOT NULL,

pwhash text,

uid int PRIMARY KEY,

gid int NOT NULL,

real\_name text NOT NULL,

home\_phone text,

extra\_info text,

home\_dir text NOT NULL,

shell text NOT NULL

);

CREATE ROLE admin; -- Administrator

CREATE ROLE bob; -- Normal user

CREATE ROLE alice; -- Normal user

-- Populate the table

INSERT INTO passwd VALUES

('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');

INSERT INTO passwd VALUES

('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');

INSERT INTO passwd VALUES

('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- Be sure to enable row level security on the table

ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Create policies

-- Administrator can see all rows and add any rows

CREATE POLICY admin\_all ON passwd TO admin USING (true) WITH CHECK (true);

-- Normal users can view all rows

CREATE POLICY all\_view ON passwd FOR SELECT USING (true);

-- Normal users can update their own records, but

-- limit which shells a normal user is allowed to set

CREATE POLICY user\_mod ON passwd FOR UPDATE

USING (current\_user = user\_name)

WITH CHECK (

current\_user = user\_name AND

shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')

);

-- Allow admin all normal rights

GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;

-- Users only get select access on public columns

GRANT SELECT

(user\_name, uid, gid, real\_name, home\_phone, extra\_info, home\_dir, shell)

ON passwd TO public;

-- Allow users to update certain columns

GRANT UPDATE

(pwhash, real\_name, home\_phone, extra\_info, shell)

ON passwd TO public;

As with any security settings, it's important to test and ensure that the system is behaving as expected. Using the example above, this demonstrates that the permission system is working properly.

-- admin can view all rows and fields

postgres=> set role admin;

SET

postgres=> table passwd;

user\_name | pwhash | uid | gid | real\_name | home\_phone | extra\_info | home\_dir | shell

-----------+--------+-----+-----+-----------+--------------+------------+-------------+-----------

admin | xxx | 0 | 0 | Admin | 111-222-3333 | | /root | /bin/dash

bob | xxx | 1 | 1 | Bob | 123-456-7890 | | /home/bob | /bin/zsh

alice | xxx | 2 | 1 | Alice | 098-765-4321 | | /home/alice | /bin/zsh

(3 rows)

-- Test what Alice is able to do

postgres=> set role alice;

SET

postgres=> table passwd;

ERROR: permission denied for relation passwd

postgres=> select user\_name,real\_name,home\_phone,extra\_info,home\_dir,shell from passwd;

user\_name | real\_name | home\_phone | extra\_info | home\_dir | shell

-----------+-----------+--------------+------------+-------------+-----------

admin | Admin | 111-222-3333 | | /root | /bin/dash

bob | Bob | 123-456-7890 | | /home/bob | /bin/zsh

alice | Alice | 098-765-4321 | | /home/alice | /bin/zsh

(3 rows)

postgres=> update passwd set user\_name = 'joe';

ERROR: permission denied for relation passwd

-- Alice is allowed to change her own real\_name, but no others

postgres=> update passwd set real\_name = 'Alice Doe';

UPDATE 1

postgres=> update passwd set real\_name = 'John Doe' where user\_name = 'admin';

UPDATE 0

postgres=> update passwd set shell = '/bin/xx';

ERROR: new row violates WITH CHECK OPTION for "passwd"

postgres=> delete from passwd;

ERROR: permission denied for relation passwd

postgres=> insert into passwd (user\_name) values ('xxx');

ERROR: permission denied for relation passwd

-- Alice can change her own password; RLS silently prevents updating other rows

postgres=> update passwd set pwhash = 'abc';

UPDATE 1

All of the policies constructed thus far have been permissive policies, meaning that when multiple policies are applied they are combined using the “OR” Boolean operator. While permissive policies can be constructed to only allow access to rows in the intended cases, it can be simpler to combine permissive policies with restrictive policies (which the records must pass and which are combined using the “AND” Boolean operator). Building on the example above, we add a restrictive policy to require the administrator to be connected over a local Unix socket to access the records of the passwdtable:

CREATE POLICY admin\_local\_only ON passwd AS RESTRICTIVE TO admin

USING (pg\_catalog.inet\_client\_addr() IS NULL);

We can then see that an administrator connecting over a network will not see any records, due to the restrictive policy:

=> SELECT current\_user;

current\_user

--------------

admin

(1 row)

=> select inet\_client\_addr();

inet\_client\_addr

------------------

127.0.0.1

(1 row)

=> SELECT current\_user;

current\_user

--------------

admin

(1 row)

=> TABLE passwd;

user\_name | pwhash | uid | gid | real\_name | home\_phone | extra\_info | home\_dir | shell

-----------+--------+-----+-----+-----------+------------+------------+----------+-------

(0 rows)

=> UPDATE passwd set pwhash = NULL;

UPDATE 0

Referential integrity checks, such as unique or primary key constraints and foreign key references, always bypass row security to ensure that data integrity is maintained. Care must be taken when developing schemas and row level policies to avoid “covert channel” leaks of information through such referential integrity checks.

In some contexts it is important to be sure that row security is not being applied. For example, when taking a backup, it could be disastrous if row security silently caused some rows to be omitted from the backup. In such a situation, you can set the [**row\_security**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-ROW-SECURITY) configuration parameter to off. This does not in itself bypass row security; what it does is throw an error if any query's results would get filtered by a policy. The reason for the error can then be investigated and fixed.

In the examples above, the policy expressions consider only the current values in the row to be accessed or updated. This is the simplest and best-performing case; when possible, it's best to design row security applications to work this way. If it is necessary to consult other rows or other tables to make a policy decision, that can be accomplished using sub-SELECTs, or functions that contain SELECTs, in the policy expressions. Be aware however that such accesses can create race conditions that could allow information leakage if care is not taken. As an example, consider the following table design:

-- definition of privilege groups

CREATE TABLE groups (group\_id int PRIMARY KEY,

group\_name text NOT NULL);

INSERT INTO groups VALUES

(1, 'low'),

(2, 'medium'),

(5, 'high');

GRANT ALL ON groups TO alice; -- alice is the administrator

GRANT SELECT ON groups TO public;

-- definition of users' privilege levels

CREATE TABLE users (user\_name text PRIMARY KEY,

group\_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES

('alice', 5),

('bob', 2),

('mallory', 2);

GRANT ALL ON users TO alice;

GRANT SELECT ON users TO public;

-- table holding the information to be protected

CREATE TABLE information (info text,

group\_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES

('barely secret', 1),

('slightly secret', 2),

('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- a row should be visible to/updatable by users whose security group\_id is

-- greater than or equal to the row's group\_id

CREATE POLICY fp\_s ON information FOR SELECT

USING (group\_id <= (SELECT group\_id FROM users WHERE user\_name = current\_user));

CREATE POLICY fp\_u ON information FOR UPDATE

USING (group\_id <= (SELECT group\_id FROM users WHERE user\_name = current\_user));

-- we rely only on RLS to protect the information table

GRANT ALL ON information TO public;

Now suppose that alice wishes to change the “slightly secret” information, but decides that mallory should not be trusted with the new content of that row, so she does:

BEGIN;

UPDATE users SET group\_id = 1 WHERE user\_name = 'mallory';

UPDATE information SET info = 'secret from mallory' WHERE group\_id = 2;

COMMIT;

That looks safe; there is no window wherein mallory should be able to see the “secret from mallory” string. However, there is a race condition here. If mallory is concurrently doing, say,

SELECT \* FROM information WHERE group\_id = 2 FOR UPDATE;

and her transaction is in READ COMMITTED mode, it is possible for her to see “secret from mallory”. That happens if her transaction reaches the information row just after alice's does. It blocks waiting for alice's transaction to commit, then fetches the updated row contents thanks to the FOR UPDATE clause. However, it does not fetch an updated row for the implicit SELECT from users, because that sub-SELECT did not have FOR UPDATE; instead the users row is read with the snapshot taken at the start of the query. Therefore, the policy expression tests the old value of mallory's privilege level and allows her to see the updated row.

There are several ways around this problem. One simple answer is to use SELECT ... FOR SHARE in sub-SELECTs in row security policies. However, that requires granting UPDATE privilege on the referenced table (here users) to the affected users, which might be undesirable. (But another row security policy could be applied to prevent them from actually exercising that privilege; or the sub-SELECT could be embedded into a security definer function.) Also, heavy concurrent use of row share locks on the referenced table could pose a performance problem, especially if updates of it are frequent. Another solution, practical if updates of the referenced table are infrequent, is to take an exclusive lock on the referenced table when updating it, so that no concurrent transactions could be examining old row values. Or one could just wait for all concurrent transactions to end after committing an update of the referenced table and before making changes that rely on the new security situation.

For additional details see [**CREATE POLICY**](https://www.postgresql.org/docs/10/sql-createpolicy.html) and [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html).

## 5.8. Schemas

A PostgreSQL database cluster contains one or more named databases. Users and groups of users are shared across the entire cluster, but no other data is shared across databases. Any given client connection to the server can access only the data in a single database, the one specified in the connection request.

Note

Users of a cluster do not necessarily have the privilege to access every database in the cluster. Sharing of user names means that there cannot be different users named, say, joein two databases in the same cluster; but the system can be configured to allow joe access to only some of the databases.

A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both schema1 and myschema can contain tables named mytable. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database they are connected to, if they have privileges to do so.

There are several reasons why one might want to use schemas:

* To allow many users to use one database without interfering with each other.
* To organize database objects into logical groups to make them more manageable.
* Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

### 5.8.1. Creating a Schema

To create a schema, use the [**CREATE SCHEMA**](https://www.postgresql.org/docs/10/sql-createschema.html) command. Give the schema a name of your choice. For example:

CREATE SCHEMA myschema;

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a dot:

***schema***.***table***

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

***database***.***schema***.***table***

can be used too, but at present this is just for pro forma compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So to create a table in the new schema, use:

CREATE TABLE myschema.mytable (

...

);

To drop a schema if it's empty (all objects in it have been dropped), use:

DROP SCHEMA myschema;

To drop a schema including all contained objects, use:

DROP SCHEMA myschema CASCADE;

See [**Section 5.13**](https://www.postgresql.org/docs/10/ddl-depend.html) for a description of the general mechanism behind this.

Often you will want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

CREATE SCHEMA ***schema\_name*** AUTHORIZATION ***user\_name***;

You can even omit the schema name, in which case the schema name will be the same as the user name. See [**Section 5.8.6**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATTERNS) for how this can be useful.

Schema names beginning with pg\_ are reserved for system purposes and cannot be created by users.

### 5.8.2. The Public Schema

In the previous sections we created tables without specifying any schema names. By default such tables (and other objects) are automatically put into a schema named “public”. Every new database contains such a schema. Thus, the following are equivalent:

CREATE TABLE products ( ... );

and:

CREATE TABLE public.products ( ... );

### 5.8.3. The Schema Search Path

Qualified names are tedious to write, and it's often best not to wire a particular schema name into applications anyway. Therefore tables are often referred to by unqualified names, which consist of just the table name. The system determines which table is meant by following a search path, which is a list of schemas to look in. The first matching table in the search path is taken to be the one wanted. If there is no match in the search path, an error is reported, even if matching table names exist in other schemas in the database.

The ability to create like-named objects in different schemas complicates writing a query that references precisely the same objects every time. It also opens up the potential for users to change the behavior of other users' queries, maliciously or accidentally. Due to the prevalence of unqualified names in queries and their use in PostgreSQL internals, adding a schema to search\_path effectively trusts all users having CREATE privilege on that schema. When you run an ordinary query, a malicious user able to create objects in a schema of your search path can take control and execute arbitrary SQL functions as though you executed them.

The first schema named in the search path is called the current schema. Aside from being the first schema searched, it is also the schema in which new tables will be created if the CREATE TABLEcommand does not specify a schema name.

To show the current search path, use the following command:

SHOW search\_path;

In the default setup this returns:

search\_path

--------------

"$user", public

The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored. The second element refers to the public schema that we have seen already.

The first schema in the search path that exists is the default location for creating new objects. That is the reason that by default objects are created in the public schema. When objects are referenced in any other context without schema qualification (table modification, data modification, or query commands) the search path is traversed until a matching object is found. Therefore, in the default configuration, any unqualified access again can only refer to the public schema.

To put our new schema in the path, we use:

SET search\_path TO myschema,public;

(We omit the $user here because we have no immediate need for it.) And then we can access the table without schema qualification:

DROP TABLE mytable;

Also, since myschema is the first element in the path, new objects would by default be created in it.

We could also have written:

SET search\_path TO myschema;

Then we no longer have access to the public schema without explicit qualification. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

See also [**Section 9.25**](https://www.postgresql.org/docs/10/functions-info.html) for other ways to manipulate the schema search path.

The search path works in the same way for data type names, function names, and operator names as it does for table names. Data type and function names can be qualified in exactly the same way as table names. If you need to write a qualified operator name in an expression, there is a special provision: you must write

OPERATOR(***schema***.***operator***)

This is needed to avoid syntactic ambiguity. An example is:

SELECT 3 OPERATOR(pg\_catalog.+) 4;

In practice one usually relies on the search path for operators, so as not to have to write anything so ugly as that.

### 5.8.4. Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema must grant the USAGE privilege on the schema. To allow users to make use of the objects in the schema, additional privileges might need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else's schema. To allow that, the CREATE privilege on the schema needs to be granted. Note that by default, everyone has CREATE and USAGEprivileges on the schema public. This allows all users that are able to connect to a given database to create objects in its public schema. Some [**usage patterns**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATTERNS) call for revoking that privilege:

REVOKE CREATE ON SCHEMA public FROM PUBLIC;

(The first “public” is the schema, the second “public” means “every user”. In the first sense it is an identifier, in the second sense it is a key word, hence the different capitalization; recall the guidelines from [**Section 4.1.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-IDENTIFIERS).)

### 5.8.5. The System Catalog Schema

In addition to public and user-created schemas, each database contains a pg\_catalog schema, which contains the system tables and all the built-in data types, functions, and operators. pg\_catalog is always effectively part of the search path. If it is not named explicitly in the path then it is implicitly searched before searching the path's schemas. This ensures that built-in names will always be findable. However, you can explicitly place pg\_catalog at the end of your search path if you prefer to have user-defined names override built-in names.

Since system table names begin with pg\_, it is best to avoid such names to ensure that you won't suffer a conflict if some future version defines a system table named the same as your table. (With the default search path, an unqualified reference to your table name would then be resolved as the system table instead.) System tables will continue to follow the convention of having names beginning with pg\_, so that they will not conflict with unqualified user-table names so long as users avoid the pg\_ prefix.

### 5.8.6. Usage Patterns

Schemas can be used to organize your data in many ways. There are a few usage patterns easily supported by the default configuration, only one of which suffices when database users mistrust other database users:

* Constrain ordinary users to user-private schemas. To implement this, issue REVOKE CREATE ON SCHEMA public FROM PUBLIC, and create a schema for each user with the same name as that user. If affected users had logged in before this, consider auditing the public schema for objects named like objects in schema pg\_catalog. Recall that the default search path starts with $user, which resolves to the user name. Therefore, if each user has a separate schema, they access their own schemas by default.
* Remove the public schema from each user's default search path using ALTER ROLE ***user*** SET search\_path = "$user". Everyone retains the ability to create objects in the public schema, but only qualified names will choose those objects. While qualified table references are fine, calls to functions in the public schema [**will be unsafe or unreliable**](https://www.postgresql.org/docs/10/typeconv-func.html). Also, a user holding the CREATEROLE privilege can undo this setting and issue arbitrary queries under the identity of users relying on the setting. If you create functions or extensions in the public schema or grant CREATEROLE to users not warranting this almost-superuser ability, use the first pattern instead.
* Remove the public schema from search\_path in [postgresql.conf](https://www.postgresql.org/docs/10/config-setting.html#CONFIG-SETTING-CONFIGURATION-FILE). The ensuing user experience matches the previous pattern. In addition to that pattern's implications for functions and CREATEROLE, this trusts database owners like CREATEROLE. If you create functions or extensions in the public schema or assign the CREATEROLE privilege, CREATEDB privilege or individual database ownership to users not warranting almost-superuser access, use the first pattern instead.
* Keep the default. All users access the public schema implicitly. This simulates the situation where schemas are not available at all, giving a smooth transition from the non-schema-aware world. However, any user can issue arbitrary queries under the identity of any user not electing to protect itself individually. This pattern is acceptable only when the database has a single user or a few mutually-trusting users.

For any pattern, to install shared applications (tables to be used by everyone, additional functions provided by third parties, etc.), put them into separate schemas. Remember to grant appropriate privileges to allow the other users to access them. Users can then refer to these additional objects by qualifying the names with a schema name, or they can put the additional schemas into their search path, as they choose.

### 5.8.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does not exist. Moreover, some implementations do not allow you to create schemas that have a different name than their owner. In fact, the concepts of schema and user are nearly equivalent in a database system that implements only the basic schema support specified in the standard. Therefore, many users consider qualified names to really consist of ***user\_name***.***table\_name***. This is how PostgreSQL will effectively behave if you create a per-user schema for every user.

Also, there is no concept of a public schema in the SQL standard. For maximum conformance to the standard, you should not use the public schema.

Of course, some SQL database systems might not implement schemas at all, or provide namespace support by allowing (possibly limited) cross-database access. If you need to work with those systems, then maximum portability would be achieved by not using schemas at all.

## 5.9. Inheritance 表的继承性！！！

[**5.9.1. Caveats**](https://www.postgresql.org/docs/10/ddl-inherit.html#DDL-INHERIT-CAVEATS)

PostgreSQL implements table inheritance, which can be a useful tool for database designers. (SQL:1999 and later define a type inheritance feature, which differs in many respects from the features described here.)

Let's start with an example: suppose we are trying to build a data model for cities. Each state has many cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular state. This can be done by creating two tables, one for state capitals and one for cities that are not capitals. However, what happens when we want to ask for data about a city, regardless of whether it is a capital or not? The inheritance feature can help to resolve this problem. We define the capitals table so that it inherits from cities:

CREATE TABLE cities (

name text,

population float,

altitude int -- in feet

);

CREATE TABLE capitals (

state char(2)

) INHERITS (cities);

In this case, the capitals table inherits all the columns of its parent table, cities. State capitals also have an extra column, state, that shows their state.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendant tables. The latter behavior is the default. For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 feet:

SELECT name, altitude

FROM cities

WHERE altitude > 500;

Given the sample data from the PostgreSQL tutorial (see [**Section 2.1**](https://www.postgresql.org/docs/10/tutorial-sql-intro.html)), this returns:

name | altitude

-----------+----------

Las Vegas | 2174

Mariposa | 1953

Madison | 845

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude over 500 feet:

SELECT name, altitude

FROM ONLY cities

WHERE altitude > 500;

name | altitude

-----------+----------

Las Vegas | 2174

Mariposa | 1953

Here the ONLY keyword indicates that the query should apply only to cities, and not any tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed — SELECT, UPDATE and DELETE — support the ONLY keyword.

You can also write the table name with a trailing \* to explicitly specify that descendant tables are included:

SELECT name, altitude

FROM cities\*

WHERE altitude > 500;

Writing \* is not necessary, since this behavior is always the default. However, this syntax is still supported for compatibility with older releases where the default could be changed.

In some cases you might wish to know which table a particular row originated from. There is a system column called tableoid in each table which can tell you the originating table:

SELECT c.tableoid, c.name, c.altitude

FROM cities c

WHERE c.altitude > 500;

which returns:

tableoid | name | altitude

----------+-----------+----------

139793 | Las Vegas | 2174

139793 | Mariposa | 1953

139798 | Madison | 845

(If you try to reproduce this example, you will probably get different numeric OIDs.) By doing a join with pg\_class you can see the actual table names:

SELECT p.relname, c.name, c.altitude

FROM cities c, pg\_class p

WHERE c.altitude > 500 AND c.tableoid = p.oid;

which returns:

relname | name | altitude

----------+-----------+----------

cities | Las Vegas | 2174

cities | Mariposa | 1953

capitals | Madison | 845

Another way to get the same effect is to use the regclass alias type, which will print the table OID symbolically:

SELECT c.tableoid::regclass, c.name, c.altitude

FROM cities c

WHERE c.altitude > 500;

Inheritance does not automatically propagate data from INSERT or COPY commands to other tables in the inheritance hierarchy. In our example, the following INSERT statement will fail:

INSERT INTO cities (name, population, altitude, state)

VALUES ('Albany', NULL, NULL, 'NY');

We might hope that the data would somehow be routed to the capitals table, but this does not happen: INSERT always inserts into exactly the table specified. In some cases it is possible to redirect the insertion using a rule (see [**Chapter 40**](https://www.postgresql.org/docs/10/rules.html)). However that does not help for the above case because the cities table does not contain the column state, and so the command will be rejected before the rule can be applied.

All check constraints and not-null constraints on a parent table are automatically inherited by its children, unless explicitly specified otherwise with NO INHERIT clauses. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table's definition are added to these. If the same column name appears in multiple parent tables, or in both a parent table and the child's definition, then these columns are “merged” so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. Inheritable check constraints and not-null constraints are merged in a similar fashion. Thus, for example, a merged column will be marked not-null if any one of the column definitions it came from is marked not-null. Check constraints are merged if they have the same name, and the merge will fail if their conditions are different.

Table inheritance is typically established when the child table is created, using the INHERITS clause of the [**CREATE TABLE**](https://www.postgresql.org/docs/10/sql-createtable.html) statement. Alternatively, a table which is already defined in a compatible way can have a new parent relationship added, using the INHERIT variant of [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html). To do this the new child table must already include columns with the same names and types as the columns of the parent. It must also include check constraints with the same names and check expressions as those of the parent. Similarly an inheritance link can be removed from a child using the NO INHERITvariant of ALTER TABLE. Dynamically adding and removing inheritance links like this can be useful when the inheritance relationship is being used for table partitioning (see [**Section 5.10**](https://www.postgresql.org/docs/10/ddl-partitioning.html)).

One convenient way to create a compatible table that will later be made a new child is to use the LIKE clause in CREATE TABLE. This creates a new table with the same columns as the source table. If there are any CHECK constraints defined on the source table, the INCLUDING CONSTRAINTS option to LIKE should be specified, as the new child must have constraints matching the parent to be considered compatible.

A parent table cannot be dropped while any of its children remain. Neither can columns or check constraints of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the CASCADE option (see [**Section 5.13**](https://www.postgresql.org/docs/10/ddl-depend.html)).

[**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html) will propagate any changes in column data definitions and check constraints down the inheritance hierarchy. Again, dropping columns that are depended on by other tables is only possible when using the CASCADE option. ALTER TABLE follows the same rules for duplicate column merging and rejection that apply during CREATE TABLE.

Inherited queries perform access permission checks on the parent table only. Thus, for example, granting UPDATE permission on the cities table implies permission to update rows in the capitalstable as well, when they are accessed through cities. This preserves the appearance that the data is (also) in the parent table. But the capitals table could not be updated directly without an additional grant. In a similar way, the parent table's row security policies (see [**Section 5.7**](https://www.postgresql.org/docs/10/ddl-rowsecurity.html)) are applied to rows coming from child tables during an inherited query. A child table's policies, if any, are applied only when it is the table explicitly named in the query; and in that case, any policies attached to its parent(s) are ignored.

Foreign tables (see [**Section 5.11**](https://www.postgresql.org/docs/10/ddl-foreign-data.html)) can also be part of inheritance hierarchies, either as parent or child tables, just as regular tables can be. If a foreign table is part of an inheritance hierarchy then any operations not supported by the foreign table are not supported on the whole hierarchy either.

### 5.9.1. Caveats 警告

Note that not all SQL commands are able to work on inheritance hierarchies. Commands that are used for data querying, data modification, or schema modification (e.g., SELECT, UPDATE, DELETE, most variants of ALTER TABLE, but not INSERT or ALTER TABLE ... RENAME) typically default to including child tables and support the ONLY notation to exclude them. Commands that do database maintenance and tuning (e.g., REINDEX, VACUUM) typically only work on individual, physical tables and do not support recursing over inheritance hierarchies. The respective behavior of each individual command is documented in its reference page ([**SQL Commands**](https://www.postgresql.org/docs/10/sql-commands.html)).

A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Thus, in the terms of the above example:

* If we declared cities.name to be UNIQUE or a PRIMARY KEY, this would not stop the capitals table from having rows with names duplicating rows in cities. And those duplicate rows would by default show up in queries from cities. In fact, by default capitals would have no unique constraint at all, and so could contain multiple rows with the same name. You could add a unique constraint to capitals, but this would not prevent duplication compared to cities.
* Similarly, if we were to specify that cities.name REFERENCES some other table, this constraint would not automatically propagate to capitals. In this case you could work around it by manually adding the same REFERENCES constraint to capitals.
* Specifying that another table's column REFERENCES cities(name) would allow the other table to contain city names, but not capital names. There is no good workaround for this case.

Some functionality not implemented for inheritance hierarchies is implemented for declarative partitioning. Considerable care is needed in deciding whether partitioning with legacy inheritance is useful for your application.

## 5.10. Table Partitioning 表分区！！！

PostgreSQL supports basic table partitioning. This section describes why and how to implement partitioning as part of your database design.

### 5.10.1. Overview

把一个大表分成若干小的物理分片。

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

* Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. The partitioning substitutes for leading columns of indexes, reducing index size and making it more likely that the heavily-used parts of the indexes fit in memory.
* When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of using an index and random access reads scattered across the whole table.
* Bulk loads and deletes can be accomplished by adding or removing partitions, if that requirement is planned into the partitioning design. Doing ALTER TABLE DETACH PARTITION or dropping an individual partition using DROP TABLE is far faster than a bulk operation. These commands also entirely avoid the VACUUM overhead caused by a bulk DELETE.
* Seldom-used data can be migrated to cheaper and slower storage media.

The benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.

PostgreSQL offers built-in support for the following forms of partitioning:

Range Partitioning

The table is partitioned into “ranges” defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example, one might partition by date ranges, or by ranges of identifiers for particular business objects.

List Partitioning

The table is partitioned by explicitly listing which key values appear in each partition.

If your application needs to use other forms of partitioning not listed above, alternative methods such as inheritance and UNION ALL views can be used instead. Such methods offer flexibility but do not have some of the performance benefits of built-in declarative partitioning.

### 5.10.2. Declarative Partitioning 声明分区

PostgreSQL offers a way to specify how to divide a table into pieces called partitions. The table that is divided is referred to as a partitioned table. The specification consists of the partitioning methodand a list of columns or expressions to be used as the partition key.

All rows inserted into a partitioned table will be routed to one of the partitions based on the value of the partition key. Each partition has a subset of the data defined by its partition bounds. Currently supported partitioning methods include range and list, where each partition is assigned a range of keys and a list of keys, respectively.

Partitions may themselves be defined as partitioned tables, using what is called sub-partitioning. Partitions may have their own indexes, constraints and default values, distinct from those of other partitions. Indexes must be created separately for each partition. See [**CREATE TABLE**](https://www.postgresql.org/docs/10/sql-createtable.html) for more details on creating partitioned tables and partitions.

It is not possible to turn a regular table into a partitioned table or vice versa. However, it is possible to add a regular or partitioned table containing data as a partition of a partitioned table, or remove a partition from a partitioned table turning it into a standalone table; see [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html) to learn more about the ATTACH PARTITION and DETACH PARTITION sub-commands.

Individual partitions are linked to the partitioned table with inheritance behind-the-scenes; however, it is not possible to use some of the inheritance features discussed in the previous section with partitioned tables and partitions. For example, a partition cannot have any parents other than the partitioned table it is a partition of, nor can a regular table inherit from a partitioned table making the latter its parent. That means partitioned tables and partitions do not participate in inheritance with regular tables. Since a partition hierarchy consisting of the partitioned table and its partitions is still an inheritance hierarchy, all the normal rules of inheritance apply as described in [**Section 5.9**](https://www.postgresql.org/docs/10/ddl-inherit.html) with some exceptions, most notably:

* Both CHECK and NOT NULL constraints of a partitioned table are always inherited by all its partitions. CHECK constraints that are marked NO INHERIT are not allowed to be created on partitioned tables.
* Using ONLY to add or drop a constraint on only the partitioned table is supported when there are no partitions. Once partitions exist, using ONLY will result in an error as adding or dropping constraints on only the partitioned table, when partitions exist, is not supported. Instead, constraints can be added or dropped, when they are not present in the parent table, directly on the partitions. As a partitioned table does not have any data directly, attempts to use TRUNCATE ONLY on a partitioned table will always return an error.
* Partitions cannot have columns that are not present in the parent. It is neither possible to specify columns when creating partitions with CREATE TABLE nor is it possible to add columns to partitions after-the-fact using ALTER TABLE. Tables may be added as a partition with ALTER TABLE ... ATTACH PARTITION only if their columns exactly match the parent, including any oidcolumn.
* You cannot drop the NOT NULL constraint on a partition's column if the constraint is present in the parent table.

Partitions can also be foreign tables (see [**CREATE FOREIGN TABLE**](https://www.postgresql.org/docs/10/sql-createforeigntable.html)), although these have some limitations that normal tables do not. For example, data inserted into the partitioned table is not routed to foreign table partitions.

#### 5.10.2.1. Example

Suppose we are constructing a database for a large ice cream company. The company measures peak temperatures every day as well as ice cream sales in each region. Conceptually, we want a table like:

CREATE TABLE measurement (

city\_id int not null,

logdate date not null,

peaktemp int,

unitsales int

);

We know that most queries will access just the last week's, month's or quarter's data, since the main use of this table will be to prepare online reports for management. To reduce the amount of old data that needs to be stored, we decide to only keep the most recent 3 years worth of data. At the beginning of each month we will remove the oldest month's data. In this situation we can use partitioning to help us meet all of our different requirements for the measurements table.

To use declarative partitioning in this case, use the following steps:

1. Create measurement table as a partitioned table by specifying the PARTITION BY clause, which includes the partitioning method (RANGE in this case) and the list of column(s) to use as the partition key.

CREATE TABLE measurement (

city\_id int not null,

logdate date not null,

peaktemp int,

unitsales int

) PARTITION BY RANGE (logdate);

You may decide to use multiple columns in the partition key for range partitioning, if desired. Of course, this will often result in a larger number of partitions, each of which is individually smaller. On the other hand, using fewer columns may lead to a coarser-grained partitioning criteria with smaller number of partitions. A query accessing the partitioned table will have to scan fewer partitions if the conditions involve some or all of these columns. For example, consider a table range partitioned using columns lastname and firstname (in that order) as the partition key.

1. Create partitions. Each partition's definition must specify the bounds that correspond to the partitioning method and partition key of the parent. Note that specifying bounds such that the new partition's values will overlap with those in one or more existing partitions will cause an error. Inserting data into the parent table that does not map to one of the existing partitions will cause an error; an appropriate partition must be added manually.

Partitions thus created are in every way normal PostgreSQL tables (or, possibly, foreign tables). It is possible to specify a tablespace and storage parameters for each partition separately.

It is not necessary to create table constraints describing partition boundary condition for partitions. Instead, partition constraints are generated implicitly from the partition bound specification whenever there is need to refer to them.

CREATE TABLE measurement\_y2006m02 PARTITION OF measurement

FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement\_y2006m03 PARTITION OF measurement

FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

...

CREATE TABLE measurement\_y2007m11 PARTITION OF measurement

FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement\_y2007m12 PARTITION OF measurement

FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')

TABLESPACE fasttablespace;

CREATE TABLE measurement\_y2008m01 PARTITION OF measurement

FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')

WITH (parallel\_workers = 4)

TABLESPACE fasttablespace;

To implement sub-partitioning, specify the PARTITION BY clause in the commands used to create individual partitions, for example:

CREATE TABLE measurement\_y2006m02 PARTITION OF measurement

FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')

PARTITION BY RANGE (peaktemp);

After creating partitions of measurement\_y2006m02, any data inserted into measurement that is mapped to measurement\_y2006m02 (or data that is directly inserted into measurement\_y2006m02, provided it satisfies its partition constraint) will be further redirected to one of its partitions based on the peaktemp column. The partition key specified may overlap with the parent's partition key, although care should be taken when specifying the bounds of a sub-partition such that the set of data it accepts constitutes a subset of what the partition's own bounds allows; the system does not try to check whether that's really the case.

1. Create an index on the key column(s), as well as any other indexes you might want for every partition. (The key index is not strictly necessary, but in most scenarios it is helpful. If you intend the key values to be unique then you should always create a unique or primary-key constraint for each partition.)

CREATE INDEX ON measurement\_y2006m02 (logdate);

CREATE INDEX ON measurement\_y2006m03 (logdate);

...

CREATE INDEX ON measurement\_y2007m11 (logdate);

CREATE INDEX ON measurement\_y2007m12 (logdate);

CREATE INDEX ON measurement\_y2008m01 (logdate);

1. Ensure that the [**constraint\_exclusion**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION) configuration parameter is not disabled in postgresql.conf. If it is, queries will not be optimized as desired.

In the above example we would be creating a new partition each month, so it might be wise to write a script that generates the required DDL automatically.

#### 5.10.2.2. Partition Maintenance

Normally the set of partitions established when initially defining the table are not intended to remain static. It is common to want to remove old partitions of data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is to drop the partition that is no longer necessary:

DROP TABLE measurement\_y2006m02;

This can very quickly delete millions of records because it doesn't have to individually delete every record. Note however that the above command requires taking an ACCESS EXCLUSIVE lock on the parent table.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

ALTER TABLE measurement DETACH PARTITION measurement\_y2006m02;

This allows further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using COPY, pg\_dump, or similar tools. It might also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports.

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above:

CREATE TABLE measurement\_y2008m02 PARTITION OF measurement

FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')

TABLESPACE fasttablespace;

As an alternative, it is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later. This allows the data to be loaded, checked, and transformed prior to it appearing in the partitioned table:

CREATE TABLE measurement\_y2008m02

(LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)

TABLESPACE fasttablespace;

ALTER TABLE measurement\_y2008m02 ADD CONSTRAINT y2008m02

CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement\_y2008m02 from 'measurement\_y2008m02'

-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement\_y2008m02

FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );

Before running the ATTACH PARTITION command, it is recommended to create a CHECK constraint on the table to be attached describing the desired partition constraint. That way, the system will be able to skip the scan to validate the implicit partition constraint. Without such a constraint, the table will be scanned to validate the partition constraint while holding an ACCESS EXCLUSIVE lock on the parent table. One may then drop the constraint after ATTACH PARTITION is finished, because it is no longer necessary.

#### 5.10.2.3. Limitations

The following limitations apply to partitioned tables:

* There is no facility available to create the matching indexes on all partitions automatically. Indexes must be added to each partition with separate commands. This also means that there is no way to create a primary key, unique constraint, or exclusion constraint spanning all partitions; it is only possible to constrain each leaf partition individually.
* Since primary keys are not supported on partitioned tables, foreign keys referencing partitioned tables are not supported, nor are foreign key references from a partitioned table to some other table.
* Using the ON CONFLICT clause with partitioned tables will cause an error, because unique or exclusion constraints can only be created on individual partitions. There is no support for enforcing uniqueness (or an exclusion constraint) across an entire partitioning hierarchy.
* An UPDATE that causes a row to move from one partition to another fails, because the new value of the row fails to satisfy the implicit partition constraint of the original partition.
* Row triggers, if necessary, must be defined on individual partitions, not the partitioned table.
* Mixing temporary and permanent relations in the same partition tree is not allowed. Hence, if the partitioned table is permanent, so must be its partitions and likewise if the partitioned table is temporary. When using temporary relations, all members of the partition tree have to be from the same session.

### 5.10.3. Implementation Using Inheritance

While the built-in declarative partitioning is suitable for most common use cases, there are some circumstances where a more flexible approach may be useful. Partitioning can be implemented using table inheritance, which allows for several features which are not supported by declarative partitioning, such as:

* Partitioning enforces a rule that all partitions must have exactly the same set of columns as the parent, but table inheritance allows children to have extra columns not present in the parent.
* Table inheritance allows for multiple inheritance.
* Declarative partitioning only supports list and range partitioning, whereas table inheritance allows data to be divided in a manner of the user's choosing. (Note, however, that if constraint exclusion is unable to prune partitions effectively, query performance will be very poor.)
* Some operations require a stronger lock when using declarative partitioning than when using table inheritance. For example, adding or removing a partition to or from a partitioned table requires taking an ACCESS EXCLUSIVE lock on the parent table, whereas a SHARE UPDATE EXCLUSIVE lock is enough in the case of regular inheritance.

#### 5.10.3.1. Example

We use the same measurement table we used above. To implement it as a partitioned table using inheritance, use the following steps:

1. Create the “master” table, from which all of the partitions will inherit. This table will contain no data. Do not define any check constraints on this table, unless you intend them to be applied equally to all partitions. There is no point in defining any indexes or unique constraints on it, either. For our example, the master table is the measurement table as originally defined.
2. Create several “child” tables that each inherit from the master table. Normally, these tables will not add any columns to the set inherited from the master. Just as with declarative partitioning, these partitions are in every way normal PostgreSQL tables (or foreign tables).
3. CREATE TABLE measurement\_y2006m02 () INHERITS (measurement);
4. CREATE TABLE measurement\_y2006m03 () INHERITS (measurement);
5. ...
6. CREATE TABLE measurement\_y2007m11 () INHERITS (measurement);
7. CREATE TABLE measurement\_y2007m12 () INHERITS (measurement);

CREATE TABLE measurement\_y2008m01 () INHERITS (measurement);

1. Add non-overlapping table constraints to the partition tables to define the allowed key values in each partition.

Typical examples would be:

CHECK ( x = 1 )

CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))

CHECK ( outletID >= 100 AND outletID < 200 )

Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions. A common mistake is to set up range constraints like:

CHECK ( outletID BETWEEN 100 AND 200 )

CHECK ( outletID BETWEEN 200 AND 300 )

This is wrong since it is not clear which partition the key value 200 belongs in.

It would be better to instead create partitions as follows:

CREATE TABLE measurement\_y2006m02 (

CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )

) INHERITS (measurement);

CREATE TABLE measurement\_y2006m03 (

CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )

) INHERITS (measurement);

...

CREATE TABLE measurement\_y2007m11 (

CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )

) INHERITS (measurement);

CREATE TABLE measurement\_y2007m12 (

CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )

) INHERITS (measurement);

CREATE TABLE measurement\_y2008m01 (

CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )

) INHERITS (measurement);

1. For each partition, create an index on the key column(s), as well as any other indexes you might want.
2. CREATE INDEX measurement\_y2006m02\_logdate ON measurement\_y2006m02 (logdate);
3. CREATE INDEX measurement\_y2006m03\_logdate ON measurement\_y2006m03 (logdate);
4. CREATE INDEX measurement\_y2007m11\_logdate ON measurement\_y2007m11 (logdate);
5. CREATE INDEX measurement\_y2007m12\_logdate ON measurement\_y2007m12 (logdate);

CREATE INDEX measurement\_y2008m01\_logdate ON measurement\_y2008m01 (logdate);

1. We want our application to be able to say INSERT INTO measurement ... and have the data be redirected into the appropriate partition table. We can arrange that by attaching a suitable trigger function to the master table. If data will be added only to the latest partition, we can use a very simple trigger function:
2. CREATE OR REPLACE FUNCTION measurement\_insert\_trigger()
3. RETURNS TRIGGER AS $$
4. BEGIN
5. INSERT INTO measurement\_y2008m01 VALUES (NEW.\*);
6. RETURN NULL;
7. END;
8. $$

LANGUAGE plpgsql;

After creating the function, we create a trigger which calls the trigger function:

CREATE TRIGGER insert\_measurement\_trigger

BEFORE INSERT ON measurement

FOR EACH ROW EXECUTE PROCEDURE measurement\_insert\_trigger();

We must redefine the trigger function each month so that it always points to the current partition. The trigger definition does not need to be updated, however.

We might want to insert data and have the server automatically locate the partition into which the row should be added. We could do this with a more complex trigger function, for example:

CREATE OR REPLACE FUNCTION measurement\_insert\_trigger()

RETURNS TRIGGER AS $$

BEGIN

IF ( NEW.logdate >= DATE '2006-02-01' AND

NEW.logdate < DATE '2006-03-01' ) THEN

INSERT INTO measurement\_y2006m02 VALUES (NEW.\*);

ELSIF ( NEW.logdate >= DATE '2006-03-01' AND

NEW.logdate < DATE '2006-04-01' ) THEN

INSERT INTO measurement\_y2006m03 VALUES (NEW.\*);

...

ELSIF ( NEW.logdate >= DATE '2008-01-01' AND

NEW.logdate < DATE '2008-02-01' ) THEN

INSERT INTO measurement\_y2008m01 VALUES (NEW.\*);

ELSE

RAISE EXCEPTION 'Date out of range. Fix the measurement\_insert\_trigger() function!';

END IF;

RETURN NULL;

END;

$$

LANGUAGE plpgsql;

The trigger definition is the same as before. Note that each IF test must exactly match the CHECK constraint for its partition.

While this function is more complex than the single-month case, it doesn't need to be updated as often, since branches can be added in advance of being needed.

Note

In practice it might be best to check the newest partition first, if most inserts go into that partition. For simplicity we have shown the trigger's tests in the same order as in other parts of this example.

A different approach to redirecting inserts into the appropriate partition table is to set up rules, instead of a trigger, on the master table. For example:

CREATE RULE measurement\_insert\_y2006m02 AS

ON INSERT TO measurement WHERE

( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )

DO INSTEAD

INSERT INTO measurement\_y2006m02 VALUES (NEW.\*);

...

CREATE RULE measurement\_insert\_y2008m01 AS

ON INSERT TO measurement WHERE

( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )

DO INSTEAD

INSERT INTO measurement\_y2008m01 VALUES (NEW.\*);

A rule has significantly more overhead than a trigger, but the overhead is paid once per query rather than once per row, so this method might be advantageous for bulk-insert situations. In most cases, however, the trigger method will offer better performance.

Be aware that COPY ignores rules. If you want to use COPY to insert data, you'll need to copy into the correct partition table rather than into the master. COPY does fire triggers, so you can use it normally if you use the trigger approach.

Another disadvantage of the rule approach is that there is no simple way to force an error if the set of rules doesn't cover the insertion date; the data will silently go into the master table instead.

1. Ensure that the [**constraint\_exclusion**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION) configuration parameter is not disabled in postgresql.conf. If it is, queries will not be optimized as desired.

As we can see, a complex partitioning scheme could require a substantial amount of DDL. In the above example we would be creating a new partition each month, so it might be wise to write a script that generates the required DDL automatically.

#### 5.10.3.2. Partition Maintenance

To remove old data quickly, simply drop the partition that is no longer necessary:

DROP TABLE measurement\_y2006m02;

To remove the partition from the partitioned table but retain access to it as a table in its own right:

ALTER TABLE measurement\_y2006m02 NO INHERIT measurement;

To add a new partition to handle new data, create an empty partition just as the original partitions were created above:

CREATE TABLE measurement\_y2008m02 (

CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )

) INHERITS (measurement);

Alternatively, one may want to create the new table outside the partition structure, and make it a partition after the data is loaded, checked, and transformed.

CREATE TABLE measurement\_y2008m02

(LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);

ALTER TABLE measurement\_y2008m02 ADD CONSTRAINT y2008m02

CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement\_y2008m02 from 'measurement\_y2008m02'

-- possibly some other data preparation work

ALTER TABLE measurement\_y2008m02 INHERIT measurement;

#### 5.10.3.3. Caveats

The following caveats apply to partitioned tables implemented using inheritance:

* There is no automatic way to verify that all of the CHECK constraints are mutually exclusive. It is safer to create code that generates partitions and creates and/or modifies associated objects than to write each by hand.
* The schemes shown here assume that the partition key column(s) of a row never change, or at least do not change enough to require it to move to another partition. An UPDATE that attempts to do that will fail because of the CHECK constraints. If you need to handle such cases, you can put suitable update triggers on the partition tables, but it makes management of the structure much more complicated.
* If you are using manual VACUUM or ANALYZE commands, don't forget that you need to run them on each partition individually. A command like:

ANALYZE measurement;

will only process the master table.

* INSERT statements with ON CONFLICT clauses are unlikely to work as expected, as the ON CONFLICT action is only taken in case of unique violations on the specified target relation, not its child relations.
* Triggers or rules will be needed to route rows to the desired partition, unless the application is explicitly aware of the partitioning scheme. Triggers may be complicated to write, and will be much slower than the tuple routing performed internally by declarative partitioning.

### 5.10.4. Partitioning and Constraint Exclusion

Constraint exclusion is a query optimization technique that improves performance for partitioned tables defined in the fashion described above (both declaratively partitioned tables and those implemented using inheritance). As an example:

SET constraint\_exclusion = on;

SELECT count(\*) FROM measurement WHERE logdate >= DATE '2008-01-01';

Without constraint exclusion, the above query would scan each of the partitions of the measurement table. With constraint exclusion enabled, the planner will examine the constraints of each partition and try to prove that the partition need not be scanned because it could not contain any rows meeting the query's WHERE clause. When the planner can prove this, it excludes the partition from the query plan.

You can use the EXPLAIN command to show the difference between a plan with constraint\_exclusion on and a plan with it off. A typical unoptimized plan for this type of table setup is:

SET constraint\_exclusion = off;

EXPLAIN SELECT count(\*) FROM measurement WHERE logdate >= DATE '2008-01-01';

QUERY PLAN

-----------------------------------------------------------------------------------------------

Aggregate (cost=158.66..158.68 rows=1 width=0)

-> Append (cost=0.00..151.88 rows=2715 width=0)

-> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

-> Seq Scan on measurement\_y2006m02 measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

-> Seq Scan on measurement\_y2006m03 measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

...

-> Seq Scan on measurement\_y2007m12 measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

-> Seq Scan on measurement\_y2008m01 measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

Some or all of the partitions might use index scans instead of full-table sequential scans, but the point here is that there is no need to scan the older partitions at all to answer this query. When we enable constraint exclusion, we get a significantly cheaper plan that will deliver the same answer:

SET constraint\_exclusion = on;

EXPLAIN SELECT count(\*) FROM measurement WHERE logdate >= DATE '2008-01-01';

QUERY PLAN

-----------------------------------------------------------------------------------------------

Aggregate (cost=63.47..63.48 rows=1 width=0)

-> Append (cost=0.00..60.75 rows=1086 width=0)

-> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

-> Seq Scan on measurement\_y2008m01 measurement (cost=0.00..30.38 rows=543 width=0)

Filter: (logdate >= '2008-01-01'::date)

Note that constraint exclusion is driven only by CHECK constraints, not by the presence of indexes. Therefore it isn't necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

The default (and recommended) setting of [**constraint\_exclusion**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION) is actually neither on nor off, but an intermediate setting called partition, which causes the technique to be applied only to queries that are likely to be working on partitioned tables. The on setting causes the planner to examine CHECK constraints in all queries, even simple ones that are unlikely to benefit.

The following caveats apply to constraint exclusion, which is used by both inheritance and partitioned tables:

* Constraint exclusion only works when the query's WHERE clause contains constants (or externally supplied parameters). For example, a comparison against a non-immutable function such as CURRENT\_TIMESTAMP cannot be optimized, since the planner cannot know which partition the function value might fall into at run time.
* Keep the partitioning constraints simple, else the planner may not be able to prove that partitions don't need to be visited. Use simple equality conditions for list partitioning, or simple range tests for range partitioning, as illustrated in the preceding examples. A good rule of thumb is that partitioning constraints should contain only comparisons of the partitioning column(s) to constants using B-tree-indexable operators, which applies even to partitioned tables, because only B-tree-indexable column(s) are allowed in the partition key. (This is not a problem when using declarative partitioning, since the automatically generated constraints are simple enough to be understood by the planner.)
* All constraints on all partitions of the master table are examined during constraint exclusion, so large numbers of partitions are likely to increase query planning time considerably. Partitioning using these techniques will work well with up to perhaps a hundred partitions; don't try to use many thousands of partitions.

### 5.10.5. Declarative Partitioning Best Practices

The choice of how to partition a table should be made carefully as the performance of query planning and execution can be negatively affected by poor design.

One of the most critical design decisions will be the column or columns by which you partition your data. Often the best choice will be to partition by the column or set of columns which most commonly appear in WHERE clauses of queries being executed on the partitioned table. WHERE clause items that match and are compatible with the partition key can be used to prune unneeded partitions. Removal of unwanted data is also a factor to consider when planning your partitioning strategy. An entire partition can be detached fairly quickly, so it may be beneficial to design the partition strategy in such a way that all data to be removed at once is located in a single partition.

Choosing the target number of partitions that the table should be divided into is also a critical decision to make. Not having enough partitions may mean that indexes remain too large and that data locality remains poor which could result in low cache hit ratios. However, dividing the table into too many partitions can also cause issues. Too many partitions can mean longer query planning times and higher memory consumption during both query planning and execution. When choosing how to partition your table, it's also important to consider what changes may occur in the future. For example, if you choose to have one partition per customer and you currently have a small number of large customers, consider the implications if in several years you instead find yourself with a large number of small customers. In this case, it may be better to choose to partition by RANGE and choose a reasonable number of partitions, each containing a fixed number of customers, rather than trying to partition by LIST and hoping that the number of customers does not increase beyond what it is practical to partition the data by.

Sub-partitioning can be useful to further divide partitions that are expected to become larger than other partitions, although excessive sub-partitioning can easily lead to large numbers of partitions and can cause the same problems mentioned in the preceding paragraph.

It is also important to consider the overhead of partitioning during query planning and execution. The query planner is generally able to handle partition hierarchies with up to a few hundred partitions. Planning times become longer and memory consumption becomes higher as more partitions are added. This is particularly true for the UPDATE and DELETE commands. Another reason to be concerned about having a large number of partitions is that the server's memory consumption may grow significantly over a period of time, especially if many sessions touch large numbers of partitions. That's because each partition requires its metadata to be loaded into the local memory of each session that touches it.

With data warehouse type workloads, it can make sense to use a larger number of partitions than with an OLTP type workload. Generally, in data warehouses, query planning time is less of a concern as the majority of processing time is spent during query execution. With either of these two types of workload, it is important to make the right decisions early, as re-partitioning large quantities of data can be painfully slow. Simulations of the intended workload are often beneficial for optimizing the partitioning strategy. Never assume that more partitions are better than fewer partitions and vice-versa.

## 5.11. Foreign Data 外部数据 ！！！

PostgreSQL implements portions of the SQL/MED specification, allowing you to access data that resides outside PostgreSQL using regular SQL queries. Such data is referred to as foreign data. (Note that this usage is not to be confused with foreign keys, which are a type of constraint within the database.)

Foreign data is accessed with help from a foreign data wrapper. A foreign data wrapper is a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it. There are some foreign data wrappers available as contrib modules; see [**Appendix F**](https://www.postgresql.org/docs/10/contrib.html). Other kinds of foreign data wrappers might be found as third party products. If none of the existing foreign data wrappers suit your needs, you can write your own; see [**Chapter 56**](https://www.postgresql.org/docs/10/fdwhandler.html).

To access foreign data, you need to create a foreign server object, which defines how to connect to a particular external data source according to the set of options used by its supporting foreign data wrapper. Then you need to create one or more foreign tables, which define the structure of the remote data. A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the PostgreSQL server. Whenever it is used, PostgreSQL asks the foreign data wrapper to fetch data from the external source, or transmit data to the external source in the case of update commands.

Accessing remote data may require authenticating to the external data source. This information can be provided by a user mapping, which can provide additional data such as user names and passwords based on the current PostgreSQL role.

For additional information, see [**CREATE FOREIGN DATA WRAPPER**](https://www.postgresql.org/docs/10/sql-createforeigndatawrapper.html), [**CREATE SERVER**](https://www.postgresql.org/docs/10/sql-createserver.html), [**CREATE USER MAPPING**](https://www.postgresql.org/docs/10/sql-createusermapping.html), [**CREATE FOREIGN TABLE**](https://www.postgresql.org/docs/10/sql-createforeigntable.html), and [**IMPORT FOREIGN SCHEMA**](https://www.postgresql.org/docs/10/sql-importforeignschema.html).

## 5.12. Other Database Objects

Tables are the central objects in a relational database structure, because they hold your data. But they are not the only objects that exist in a database. Many other kinds of objects can be created to make the use and management of the data more efficient or convenient. They are not discussed in this chapter, but we give you a list here so that you are aware of what is possible:

* Views
* Functions and operators
* Data types and domains
* Triggers and rewrite rules

Detailed information on these topics appears in [**Part V**](https://www.postgresql.org/docs/10/server-programming.html).

## 5.13. Dependency Tracking 依赖跟踪

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc. you implicitly create a net of dependencies between the objects. For instance, a table with a foreign key constraint depends on the table it references.

To ensure the integrity of the entire database structure, PostgreSQL makes sure that you cannot drop objects that other objects still depend on. For example, attempting to drop the products table we considered in [**Section 5.3.5**](https://www.postgresql.org/docs/10/ddl-constraints.html#DDL-CONSTRAINTS-FK), with the orders table depending on it, would result in an error message like this:

DROP TABLE products;

ERROR: cannot drop table products because other objects depend on it

DETAIL: constraint orders\_product\_no\_fkey on table orders depends on table products

HINT: Use DROP ... CASCADE to drop the dependent objects too.

The error message contains a useful hint: if you do not want to bother deleting all the dependent objects individually, you can run:

DROP TABLE products CASCADE;

and all the dependent objects will be removed, as will any objects that depend on them, recursively. In this case, it doesn't remove the orders table, it only removes the foreign key constraint. It stops there because nothing depends on the foreign key constraint. (If you want to check what DROP ... CASCADE will do, run DROP without CASCADE and read the DETAIL output.)

Almost all DROP commands in PostgreSQL support specifying CASCADE. Of course, the nature of the possible dependencies varies with the type of the object. You can also write RESTRICT instead of CASCADE to get the default behavior, which is to prevent dropping objects that any other objects depend on.

Note

According to the SQL standard, specifying either RESTRICT or CASCADE is required in a DROPcommand. No database system actually enforces that rule, but whether the default behavior is RESTRICT or CASCADE varies across systems.

If a DROP command lists multiple objects, CASCADE is only required when there are dependencies outside the specified group. For example, when saying DROP TABLE tab1, tab2 the existence of a foreign key referencing tab1 from tab2 would not mean that CASCADE is needed to succeed.

For user-defined functions, PostgreSQL tracks dependencies associated with a function's externally-visible properties, such as its argument and result types, but not dependencies that could only be known by examining the function body. As an example, consider this situation:

CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',

'green', 'blue', 'purple');

CREATE TABLE my\_colors (color rainbow, note text);

CREATE FUNCTION get\_color\_note (rainbow) RETURNS text AS

'SELECT note FROM my\_colors WHERE color = $1'

LANGUAGE SQL;

(See [**Section 37.4**](https://www.postgresql.org/docs/10/xfunc-sql.html) for an explanation of SQL-language functions.) PostgreSQL will be aware that the get\_color\_note function depends on the rainbow type: dropping the type would force dropping the function, because its argument type would no longer be defined. But PostgreSQL will not consider get\_color\_note to depend on the my\_colors table, and so will not drop the function if the table is dropped. While there are disadvantages to this approach, there are also benefits. The function is still valid in some sense if the table is missing, though executing it would cause an error; creating a new table of the same name would allow the function to work again.

## Chapter 6. Data Manipulation 数据操作

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. The chapter after this will finally explain how to extract your long-lost data from the database.

## 6.1. Inserting Data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is conceptually inserted one row at a time. Of course you can also insert more than one row, but there is no way to insert less than one row. Even if you know only some column values, a complete row must be created.

To create a new row, use the [**INSERT**](https://www.postgresql.org/docs/10/sql-insert.html) command. The command requires the table name and column values. For example, consider the products table from [**Chapter 5**](https://www.postgresql.org/docs/10/ddl.html):

CREATE TABLE products (

product\_no integer,

name text,

price numeric

);

An example command to insert a row would be:

INSERT INTO products VALUES (1, 'Cheese', 9.99);

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

INSERT INTO products (product\_no, name, price) VALUES (1, 'Cheese', 9.99);

INSERT INTO products (name, price, product\_no) VALUES ('Cheese', 9.99, 1);

Many users consider it good practice to always list the column names.

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

INSERT INTO products (product\_no, name) VALUES (1, 'Cheese');

INSERT INTO products VALUES (1, 'Cheese');

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

INSERT INTO products (product\_no, name, price) VALUES (1, 'Cheese', DEFAULT);

INSERT INTO products DEFAULT VALUES;

You can insert multiple rows in a single command:

INSERT INTO products (product\_no, name, price) VALUES

(1, 'Cheese', 9.99),

(2, 'Bread', 1.99),

(3, 'Milk', 2.99);

It is also possible to insert the result of a query (which might be no rows, one row, or many rows):

INSERT INTO products (product\_no, name, price)

SELECT product\_no, name, price FROM new\_products

WHERE release\_date = 'today';

This provides the full power of the SQL query mechanism ([**Chapter 7**](https://www.postgresql.org/docs/10/queries.html)) for computing the rows to be inserted.

Tip

When inserting a lot of data at the same time, consider using the [**COPY**](https://www.postgresql.org/docs/10/sql-copy.html) command. It is not as flexible as the [**INSERT**](https://www.postgresql.org/docs/10/sql-insert.html) command, but is more efficient. Refer to [**Section 14.4**](https://www.postgresql.org/docs/10/populate.html) for more information on improving bulk loading performance.

## 6.2. Updating Data

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To update existing rows, use the [**UPDATE**](https://www.postgresql.org/docs/10/sql-update.html) command. This requires three pieces of information:

1. The name of the table and column to update
2. The new value of the column
3. Which row(s) to update

Recall from [**Chapter 5**](https://www.postgresql.org/docs/10/ddl.html) that SQL does not, in general, provide a unique identifier for rows. Therefore it is not always possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (independent of whether you declared it or not) can you reliably address individual rows by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

UPDATE products SET price = 10 WHERE price = 5;

This might cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let's look at that command in detail. First is the key word UPDATE followed by the table name. As usual, the table name can be schema-qualified, otherwise it is looked up in the path. Next is the key word SET followed by the column name, an equal sign, and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

UPDATE products SET price = price \* 1.10;

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the WHERE clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the WHERE condition are updated. Note that the equals sign in the SET clause is an assignment while the one in the WHERE clause is a comparison, but this does not create any ambiguity. Of course, the WHERE condition does not have to be an equality test. Many other operators are available (see [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html)). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an UPDATE command by listing more than one assignment in the SET clause. For example:

UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;

## 6.3. Deleting Data

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the [**DELETE**](https://www.postgresql.org/docs/10/sql-delete.html) command to remove rows; the syntax is very similar to the UPDATE command. For instance, to remove all rows from the products table that have a price of 10, use:

DELETE FROM products WHERE price = 10;

If you simply write:

DELETE FROM products;

then all rows in the table will be deleted! Caveat programmer.

## 6.4. Returning Data From Modified Rows 返回被修改行的数据 ！！！

Sometimes it is useful to obtain data from modified rows while they are being manipulated. The INSERT, UPDATE, and DELETE commands all have an optional RETURNING clause that supports this. Use of RETURNING avoids performing an extra database query to collect the data, and is especially valuable when it would otherwise be difficult to identify the modified rows reliably.

The allowed contents of a RETURNING clause are the same as a SELECT command's output list (see [**Section 7.3**](https://www.postgresql.org/docs/10/queries-select-lists.html)). It can contain column names of the command's target table, or value expressions using those columns. A common shorthand is RETURNING \*, which selects all columns of the target table in order.

In an INSERT, the data available to RETURNING is the row as it was inserted. This is not so useful in trivial inserts, since it would just repeat the data provided by the client. But it can be very handy when relying on computed default values. For example, when using a [serial](https://www.postgresql.org/docs/10/datatype-numeric.html#DATATYPE-SERIAL) column to provide unique identifiers, RETURNING can return the ID assigned to a new row:

CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;

The RETURNING clause is also very useful with INSERT ... SELECT.

In an UPDATE, the data available to RETURNING is the new content of the modified row. For example:

UPDATE products SET price = price \* 1.10

WHERE price <= 99.99

RETURNING name, price AS new\_price;

In a DELETE, the data available to RETURNING is the content of the deleted row. For example:

DELETE FROM products

WHERE obsoletion\_date = 'today'

RETURNING \*;

If there are triggers ([**Chapter 38**](https://www.postgresql.org/docs/10/triggers.html)) on the target table, the data available to RETURNING is the row as modified by the triggers. Thus, inspecting columns computed by triggers is another common use-case for RETURNING.

## Chapter 7. Queries

The previous chapters explained how to create tables, how to fill them with data, and how to manipulate that data. Now we finally discuss how to retrieve the data from the database.

## 7.1. Overview

The process of retrieving or the command to retrieve data from a database is called a query. In SQL the [**SELECT**](https://www.postgresql.org/docs/10/sql-select.html) command is used to specify queries. The general syntax of the SELECT command is

[WITH ***with\_queries***] SELECT ***select\_list*** FROM ***table\_expression*** [***sort\_specification***]

The following sections describe the details of the select list, the table expression, and the sort specification. WITH queries are treated last since they are an advanced feature.

A simple kind of query has the form:

SELECT \* FROM table1;

Assuming that there is a table called table1, this command would retrieve all rows and all user-defined columns from table1. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification \* means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if table1 has columns named a, b, and c (and perhaps others) you can make the following query:

SELECT a, b + c FROM table1;

(assuming that b and c are of a numerical data type). See [**Section 7.3**](https://www.postgresql.org/docs/10/queries-select-lists.html) for more details.

FROM table1 is a simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the SELECT command as a calculator:

SELECT 3 \* 4;

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

SELECT random();

## 7.2. Table Expressions

A table expression computes a table. The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. All these transformations produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

### 7.2.1. The FROM Clause

The [FROM**Clause**](https://www.postgresql.org/docs/10/sql-select.html#SQL-FROM) derives a table from one or more other tables given in a comma-separated table reference list.

FROM ***table\_reference*** [, ***table\_reference*** [, ...]]

A table reference can be a table name (possibly schema-qualified), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. If more than one table reference is listed in the FROM clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed; see below). The result of the FROM list is an intermediate virtual table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the parent of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its descendant tables, unless the key word ONLY precedes the table name. However, the reference produces only the columns that appear in the named table — any columns added in subtables are ignored.

Instead of writing ONLY before the table name, you can write \* after the table name to explicitly specify that descendant tables are included. There is no real reason to use this syntax any more, because searching descendant tables is now always the default behavior. However, it is supported for compatibility with older releases.

#### 7.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available. The general syntax of a joined table is

***T1*** ***join\_type*** ***T2*** [ ***join\_condition*** ]

Joins of all types can be chained together, or nested: either or both ***T1*** and ***T2*** can be joined tables. Parentheses can be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

**Join Types**

Cross join

***T1*** CROSS JOIN ***T2***

For every possible combination of rows from ***T1*** and ***T2*** (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in ***T1*** followed by all columns in ***T2***. If the tables have N and M rows respectively, the joined table will have N \* M rows.

FROM ***T1*** CROSS JOIN ***T2*** is equivalent to FROM ***T1*** INNER JOIN ***T2*** ON TRUE (see below). It is also equivalent to FROM ***T1***, ***T2***.

Note

This latter equivalence does not hold exactly when more than two tables appear, because JOIN binds more tightly than comma. For example FROM ***T1*** CROSS JOIN ***T2***INNER JOIN ***T3*** ON ***condition*** is not the same as FROM ***T1***, ***T2*** INNER JOIN ***T3*** ON***condition*** because the ***condition*** can reference ***T1*** in the first case but not the second.

Qualified joins

***T1*** { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN ***T2*** ON ***boolean\_expression***

***T1*** { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN ***T2*** USING ( ***join column list*** )

***T1*** NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN ***T2***

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join.

The join condition is specified in the ON or USING clause, or implicitly by the word NATURAL. The join condition determines which rows from the two source tables are considered to “match”, as explained in detail below.

The possible types of qualified join are:

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.

FULL OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from ***T1*** and ***T2*** match if the ONexpression evaluates to true.

The USING clause is a shorthand that allows you to take advantage of the specific situation where both sides of the join use the same name for the joining column(s). It takes a comma-separated list of the shared column names and forms a join condition that includes an equality comparison for each one. For example, joining ***T1*** and ***T2*** with USING (a, b) produces the join condition ON ***T1***.a = ***T2***.a AND ***T1***.b = ***T2***.b.

Furthermore, the output of JOIN USING suppresses redundant columns: there is no need to print both of the matched columns, since they must have equal values. While JOIN ON produces all columns from ***T1*** followed by all columns from ***T2***, JOIN USING produces one output column for each of the listed column pairs (in the listed order), followed by any remaining columns from ***T1***, followed by any remaining columns from ***T2***.

Finally, NATURAL is a shorthand form of USING: it forms a USING list consisting of all column names that appear in both input tables. As with USING, these columns appear only once in the output table. If there are no common column names, NATURAL JOIN behaves like JOIN ... ON TRUE, producing a cross-product join.

Note

USING is reasonably safe from column changes in the joined relations since only the listed columns are combined. NATURAL is considerably more risky since any schema changes to either relation that cause a new matching column name to be present will cause the join to combine that new column as well.

To put this together, assume we have tables t1:

num | name

-----+------

1 | a

2 | b

3 | c

and t2:

num | value

-----+-------

1 | xxx

3 | yyy

5 | zzz

then we get the following results for the various joins:

=> **SELECT \* FROM t1 CROSS JOIN t2;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

1 | a | 3 | yyy

1 | a | 5 | zzz

2 | b | 1 | xxx

2 | b | 3 | yyy

2 | b | 5 | zzz

3 | c | 1 | xxx

3 | c | 3 | yyy

3 | c | 5 | zzz

(9 rows)

=> **SELECT \* FROM t1 INNER JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

3 | c | 3 | yyy

(2 rows)

=> **SELECT \* FROM t1 INNER JOIN t2 USING (num);**

num | name | value

-----+------+-------

1 | a | xxx

3 | c | yyy

(2 rows)

=> **SELECT \* FROM t1 NATURAL INNER JOIN t2;**

num | name | value

-----+------+-------

1 | a | xxx

3 | c | yyy

(2 rows)

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | 3 | yyy

(3 rows)

=> **SELECT \* FROM t1 LEFT JOIN t2 USING (num);**

num | name | value

-----+------+-------

1 | a | xxx

2 | b |

3 | c | yyy

(3 rows)

=> **SELECT \* FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

3 | c | 3 | yyy

| | 5 | zzz

(3 rows)

=> **SELECT \* FROM t1 FULL JOIN t2 ON t1.num = t2.num;**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | 3 | yyy

| | 5 | zzz

(4 rows)

The join condition specified with ON can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

2 | b | |

3 | c | |

(3 rows)

Notice that placing the restriction in the WHERE clause produces a different result:

=> **SELECT \* FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';**

num | name | num | value

-----+------+-----+-------

1 | a | 1 | xxx

(1 row)

This is because a restriction placed in the ON clause is processed before the join, while a restriction placed in the WHERE clause is processed after the join. That does not matter with inner joins, but it matters a lot with outer joins.

#### 7.2.1.2. Table And Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a table alias.

To create a table alias, write

FROM ***table\_reference*** AS ***alias***

or

FROM ***table\_reference*** ***alias***

The AS key word is optional noise. ***alias*** can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

SELECT \* FROM some\_very\_long\_table\_name s JOIN another\_fairly\_long\_name a ON s.id = a.num;

The alias becomes the new name of the table reference so far as the current query is concerned — it is not allowed to refer to the table by the original name elsewhere in the query. Thus, this is not valid:

SELECT \* FROM my\_table AS m WHERE my\_table.a > 5; -- wrong

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.:

SELECT \* FROM people AS mother JOIN people AS child ON mother.id = child.mother\_id;

Additionally, an alias is required if the table reference is a subquery (see [**Section 7.2.1.3**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-SUBQUERIES)).

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias b to the second instance of my\_table, but the second statement assigns the alias to the result of the join:

SELECT \* FROM my\_table AS a CROSS JOIN my\_table AS b ...

SELECT \* FROM (my\_table AS a CROSS JOIN my\_table) AS b ...

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

FROM ***table\_reference*** [AS] ***alias*** ( ***column1*** [, ***column2*** [, ...]] )

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, the alias hides the original name(s) within the JOIN. For example:

SELECT a.\* FROM my\_table AS a JOIN your\_table AS b ON ...

is valid SQL, but:

SELECT a.\* FROM (my\_table AS a JOIN your\_table AS b ON ...) AS c

is not valid; the table alias a is not visible outside the alias c.

#### 7.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and must be assigned a table alias name (as in [**Section 7.2.1.2**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLE-ALIASES)). For example:

FROM (SELECT \* FROM table1) AS alias\_name

This example is equivalent to FROM table1 AS alias\_name. More interesting cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a VALUES list:

FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))

AS names(first, last)

Again, a table alias is required. Assigning alias names to the columns of the VALUES list is optional, but is good practice. For more information see [**Section 7.7**](https://www.postgresql.org/docs/10/queries-values.html).

#### 7.2.1.4. Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the FROMclause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as columns of a table, view, or subquery.

Table functions may also be combined using the ROWS FROM syntax, with the results returned in parallel columns; the number of result rows in this case is that of the largest function result, with smaller results padded with null values to match.

***function\_call*** [WITH ORDINALITY] [[AS] ***table\_alias*** [(***column\_alias*** [, ... ])]]

ROWS FROM( ***function\_call*** [, ... ] ) [WITH ORDINALITY] [[AS] ***table\_alias*** [(***column\_alias*** [, ... ])]]

If the WITH ORDINALITY clause is specified, an additional column of type bigint will be added to the function result columns. This column numbers the rows of the function result set, starting from 1. (This is a generalization of the SQL-standard syntax for UNNEST ... WITH ORDINALITY.) By default, the ordinal column is called ordinality, but a different column name can be assigned to it using an ASclause.

The special table function UNNEST may be called with any number of array parameters, and it returns a corresponding number of columns, as if UNNEST ([**Section 9.18**](https://www.postgresql.org/docs/10/functions-array.html)) had been called on each parameter separately and combined using the ROWS FROM construct.

UNNEST( ***array\_expression*** [, ... ] ) [WITH ORDINALITY] [[AS] ***table\_alias*** [(***column\_alias*** [, ... ])]]

If no ***table\_alias*** is specified, the function name is used as the table name; in the case of a ROWS FROM() construct, the first function's name is used.

If column aliases are not supplied, then for a function returning a base data type, the column name is also the same as the function name. For a function returning a composite type, the result columns get the names of the individual attributes of the type.

Some examples:

CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

SELECT \* FROM getfoo(1) AS t1;

SELECT \* FROM foo

WHERE foosubid IN (

SELECT foosubid

FROM getfoo(foo.fooid) z

WHERE z.fooid = foo.fooid

);

CREATE VIEW vw\_getfoo AS SELECT \* FROM getfoo(1);

SELECT \* FROM vw\_getfoo;

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudo-type record. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. This syntax looks like:

***function\_call*** [AS] ***alias*** (***column\_definition*** [, ... ])

***function\_call*** AS [***alias***] (***column\_definition*** [, ... ])

ROWS FROM( ... ***function\_call*** AS (***column\_definition*** [, ... ]) [, ... ] )

When not using the ROWS FROM() syntax, the ***column\_definition*** list replaces the column alias list that could otherwise be attached to the FROM item; the names in the column definitions serve as column aliases. When using the ROWS FROM() syntax, a ***column\_definition*** list can be attached to each member function separately; or if there is only one member function and no WITH ORDINALITYclause, a ***column\_definition*** list can be written in place of a column alias list following ROWS FROM().

Consider this example:

SELECT \*

FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg\_proc')

AS t1(proname name, prosrc text)

WHERE proname LIKE 'bytea%';

The [**dblink**](https://www.postgresql.org/docs/10/contrib-dblink-function.html) function (part of the [**dblink**](https://www.postgresql.org/docs/10/dblink.html) module) executes a remote query. It is declared to return record since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what \* should expand to.

#### 7.2.1.5. LATERAL Subqueries

Subqueries appearing in FROM can be preceded by the key word LATERAL. This allows them to reference columns provided by preceding FROM items. (Without LATERAL, each subquery is evaluated independently and so cannot cross-reference any other FROM item.)

Table functions appearing in FROM can also be preceded by the key word LATERAL, but for functions the key word is optional; the function's arguments can contain references to columns provided by preceding FROM items in any case.

A LATERAL item can appear at top level in the FROM list, or within a JOIN tree. In the latter case it can also refer to any items that are on the left-hand side of a JOIN that it is on the right-hand side of.

When a FROM item contains LATERAL cross-references, evaluation proceeds as follows: for each row of the FROM item providing the cross-referenced column(s), or set of rows of multiple FROM items providing the columns, the LATERAL item is evaluated using that row or row set's values of the columns. The resulting row(s) are joined as usual with the rows they were computed from. This is repeated for each row or set of rows from the column source table(s).

A trivial example of LATERAL is

SELECT \* FROM foo, LATERAL (SELECT \* FROM bar WHERE bar.id = foo.bar\_id) ss;

This is not especially useful since it has exactly the same result as the more conventional

SELECT \* FROM foo, bar WHERE bar.id = foo.bar\_id;

LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined. A common application is providing an argument value for a set-returning function. For example, supposing that vertices(polygon) returns the set of vertices of a polygon, we could identify close-together vertices of polygons stored in a table with:

SELECT p1.id, p2.id, v1, v2

FROM polygons p1, polygons p2,

LATERAL vertices(p1.poly) v1,

LATERAL vertices(p2.poly) v2

WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;

This query could also be written

SELECT p1.id, p2.id, v1, v2

FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,

polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2

WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;

or in several other equivalent formulations. (As already mentioned, the LATERAL key word is unnecessary in this example, but we use it for clarity.)

It is often particularly handy to LEFT JOIN to a LATERAL subquery, so that source rows will appear in the result even if the LATERAL subquery produces no rows for them. For example, if get\_product\_names() returns the names of products made by a manufacturer, but some manufacturers in our table currently produce no products, we could find out which ones those are like this:

SELECT m.name

FROM manufacturers m LEFT JOIN LATERAL get\_product\_names(m.id) pname ON true

WHERE pname IS NULL;

### 7.2.2. The WHERE Clause

The syntax of the [WHERE**Clause**](https://www.postgresql.org/docs/10/sql-select.html#SQL-WHERE) is

WHERE ***search\_condition***

where ***search\_condition*** is any value expression (see [**Section 4.2**](https://www.postgresql.org/docs/10/sql-expressions.html)) that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (i.e., if the result is false or null) it is discarded. The search condition typically references at least one column of the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

Note

The join condition of an inner join can be written either in the WHERE clause or in the JOINclause. For example, these table expressions are equivalent:

FROM a, b WHERE a.id = b.id AND b.val > 5

and:

FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5

or perhaps even:

FROM a NATURAL JOIN b WHERE b.val > 5

Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other SQL database management systems, even though it is in the SQL standard. For outer joins there is no choice: they must be done in the FROMclause. The ON or USING clause of an outer join is not equivalent to a WHERE condition, because it results in the addition of rows (for unmatched input rows) as well as the removal of rows in the final result.

Here are some examples of WHERE clauses:

SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

fdt is the table derived in the FROM clause. Rows that do not meet the search condition of the WHERE clause are eliminated from fdt. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how fdt is referenced in the subqueries. Qualifying c1 as fdt.c1 is only necessary if c1 is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries.

### 7.2.3. The GROUP BY and HAVING Clauses

After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

SELECT ***select\_list***

FROM ...

[WHERE ...]

GROUP BY ***grouping\_column\_reference*** [, ***grouping\_column\_reference***]...

The [GROUP BY**Clause**](https://www.postgresql.org/docs/10/sql-select.html#SQL-GROUPBY) is used to group together those rows in a table that have the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows having common values into one group row that represents all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

=> **SELECT \* FROM test1;**

x | y

---+---

a | 3

c | 2

b | 5

a | 1

(4 rows)

=> **SELECT x FROM test1 GROUP BY x;**

x

---

a

b

c

(3 rows)

In the second query, we could not have written SELECT \* FROM test1 GROUP BY x, because there is no single value for the column y that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not listed in GROUP BY cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

=> **SELECT x, sum(y) FROM test1 GROUP BY x;**

x | sum

---+-----

a | 4

b | 5

c | 2

(3 rows)

Here sum is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html).

Tip

Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the DISTINCT clause (see [**Section 7.3.3**](https://www.postgresql.org/docs/10/queries-select-lists.html#QUERIES-DISTINCT)).

Here is another example: it calculates the total sales for each product (rather than the total sales of all products):

SELECT product\_id, p.name, (sum(s.units) \* p.price) AS sales

FROM products p LEFT JOIN sales s USING (product\_id)

GROUP BY product\_id, p.name, p.price;

In this example, the columns product\_id, p.name, and p.price must be in the GROUP BY clause since they are referenced in the query select list (but see below). The column s.units does not have to be in the GROUP BY list since it is only used in an aggregate expression (sum(...)), which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

If the products table is set up so that, say, product\_id is the primary key, then it would be enough to group by product\_id in the above example, since name and price would be functionally dependenton the product ID, and so there would be no ambiguity about which name and price value to return for each product ID group.

In strict SQL, GROUP BY can only group by columns of the source table but PostgreSQL extends this to also allow GROUP BY to group by columns in the select list. Grouping by value expressions instead of simple column names is also allowed.

If a table has been grouped using GROUP BY, but only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from the result. The syntax is:

SELECT ***select\_list*** FROM ... [WHERE ...] GROUP BY ... HAVING ***boolean\_expression***

Expressions in the HAVING clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

=> **SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;**

x | sum

---+-----

a | 4

b | 5

(2 rows)

=> **SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';**

x | sum

---+-----

a | 4

b | 5

(2 rows)

Again, a more realistic example:

SELECT product\_id, p.name, (sum(s.units) \* (p.price - p.cost)) AS profit

FROM products p LEFT JOIN sales s USING (product\_id)

WHERE s.date > CURRENT\_DATE - INTERVAL '4 weeks'

GROUP BY product\_id, p.name, p.price, p.cost

HAVING sum(p.price \* s.units) > 5000;

In the example above, the WHERE clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the HAVING clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

If a query contains aggregate function calls, but no GROUP BY clause, grouping still occurs: the result is a single group row (or perhaps no rows at all, if the single row is then eliminated by HAVING). The same is true if it contains a HAVING clause, even without any aggregate function calls or GROUP BY clause.

### 7.2.4. GROUPING SETS, CUBE, and ROLLUP

More complex grouping operations than those described above are possible using the concept of grouping sets. The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned. For example:

=> **SELECT \* FROM items\_sold;**

brand | size | sales

-------+------+-------

Foo | L | 10

Foo | M | 20

Bar | M | 15

Bar | L | 5

(4 rows)

=> **SELECT brand, size, sum(sales) FROM items\_sold GROUP BY GROUPING SETS ((brand), (size), ());**

brand | size | sum

-------+------+-----

Foo | | 30

Bar | | 20

| L | 15

| M | 35

| | 50

(5 rows)

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear. To distinguish which grouping a particular output row resulted from, see [**Table 9.56**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-GROUPING-TABLE).

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form

ROLLUP ( ***e1***, ***e2***, ***e3***, ... )

represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

GROUPING SETS (

( ***e1***, ***e2***, ***e3***, ... ),

...

( ***e1***, ***e2*** ),

( ***e1*** ),

( )

)

This is commonly used for analysis over hierarchical data; e.g. total salary by department, division, and company-wide total.

A clause of the form

CUBE ( ***e1***, ***e2***, ... )

represents the given list and all of its possible subsets (i.e. the power set). Thus

CUBE ( a, b, c )

is equivalent to

GROUPING SETS (

( a, b, c ),

( a, b ),

( a, c ),

( a ),

( b, c ),

( b ),

( c ),

( )

)

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets. For example:

CUBE ( (a, b), (c, d) )

is equivalent to

GROUPING SETS (

( a, b, c, d ),

( a, b ),

( c, d ),

( )

)

and

ROLLUP ( a, (b, c), d )

is equivalent to

GROUPING SETS (

( a, b, c, d ),

( a, b, c ),

( a ),

( )

)

The CUBE and ROLLUP constructs can be used either directly in the GROUP BY clause, or nested inside a GROUPING SETS clause. If one GROUPING SETS clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

If multiple grouping items are specified in a single GROUP BY clause, then the final list of grouping sets is the cross product of the individual items. For example:

GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))

is equivalent to

GROUP BY GROUPING SETS (

(a, b, c, d), (a, b, c, e),

(a, b, d), (a, b, e),

(a, c, d), (a, c, e),

(a, d), (a, e)

)

Note

The construct (a, b) is normally recognized in expressions as a [**row constructor**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). Within the GROUP BY clause, this does not apply at the top levels of expressions, and (a, b) is parsed as a list of expressions as described above. If for some reason you need a row constructor in a grouping expression, use ROW(a, b).

### 7.2.5. Window Function Processing

If the query contains any window functions (see [**Section 3.5**](https://www.postgresql.org/docs/10/tutorial-window.html), [**Section 9.21**](https://www.postgresql.org/docs/10/functions-window.html) and [**Section 4.2.8**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS)), these functions are evaluated after any grouping, aggregation, and HAVING filtering is performed. That is, if the query uses any aggregates, GROUP BY, or HAVING, then the rows seen by the window functions are the group rows instead of the original table rows from FROM/WHERE.

When multiple window functions are used, all the window functions having syntactically equivalent PARTITION BY and ORDER BY clauses in their window definitions are guaranteed to be evaluated in a single pass over the data. Therefore they will see the same sort ordering, even if the ORDER BY does not uniquely determine an ordering. However, no guarantees are made about the evaluation of functions having different PARTITION BY or ORDER BY specifications. (In such cases a sort step is typically required between the passes of window function evaluations, and the sort is not guaranteed to preserve ordering of rows that its ORDER BY sees as equivalent.)

Currently, window functions always require presorted data, and so the query output will be ordered according to one or another of the window functions' PARTITION BY/ORDER BY clauses. It is not recommended to rely on this, however. Use an explicit top-level ORDER BY clause if you want to be sure the results are sorted in a particular way.

## 7.3. Select Lists

As shown in the previous section, the table expression in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the select list. The select list determines which columns of the intermediate table are actually output.

### 7.3.1. Select-List Items

The simplest kind of select list is \* which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in [**Section 4.2**](https://www.postgresql.org/docs/10/sql-expressions.html)). For instance, it could be a list of column names:

SELECT a, b, c FROM ...

The columns names a, b, and c are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in [**Section 7.2.1.2**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLE-ALIASES). The name space available in the select list is the same as in the WHERE clause, unless grouping is used, in which case it is the same as in the HAVING clause.

If more than one table has a column of the same name, the table name must also be given, as in:

SELECT tbl1.a, tbl2.a, tbl1.b FROM ...

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

SELECT tbl1.\*, tbl2.a FROM ...

See [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE) for more about the ***table\_name***.\* notation.

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they can be constant arithmetic expressions, for instance.

### 7.3.2. Column Labels

The entries in the select list can be assigned names for subsequent processing, such as for use in an ORDER BY clause or for display by the client application. For example:

SELECT a AS value, b + c AS sum FROM ...

If no output column name is specified using AS, the system assigns a default column name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

The AS keyword is optional, but only if the new column name does not match any PostgreSQL keyword (see [**Appendix C**](https://www.postgresql.org/docs/10/sql-keywords-appendix.html)). To avoid an accidental match to a keyword, you can double-quote the column name. For example, VALUE is a keyword, so this does not work:

SELECT a value, b + c AS sum FROM ...

but this does:

SELECT a "value", b + c AS sum FROM ...

For protection against possible future keyword additions, it is recommended that you always either write AS or double-quote the output column name.

Note

The naming of output columns here is different from that done in the FROM clause (see [**Section 7.2.1.2**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLE-ALIASES)). It is possible to rename the same column twice, but the name assigned in the select list is the one that will be passed on.

### 7.3.3. DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The DISTINCT key word is written directly after SELECT to specify this:

SELECT DISTINCT ***select\_list*** ...

(Instead of DISTINCT the key word ALL can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

SELECT DISTINCT ON (***expression*** [, ***expression*** ...]) ***select\_list*** ...

Here ***expression*** is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the “first row” of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ONprocessing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subqueries in FROM, this construct can be avoided, but it is often the most convenient alternative.

## 7.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

***query1*** UNION [ALL] ***query2***

***query1*** INTERSECT [ALL] ***query2***

***query1*** EXCEPT [ALL] ***query2***

***query1*** and ***query2*** are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

***query1*** UNION ***query2*** UNION ***query3***

which is executed as:

(***query1*** UNION ***query2***) UNION ***query3***

UNION effectively appends the result of ***query2*** to the result of ***query1*** (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

INTERSECT returns all rows that are both in the result of ***query1*** and in the result of ***query2***. Duplicate rows are eliminated unless INTERSECT ALL is used.

EXCEPT returns all rows that are in the result of ***query1*** but not in the result of ***query2***. (This is sometimes called the difference between two queries.) Again, duplicates are eliminated unless EXCEPT ALLis used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be “union compatible”, which means that they return the same number of columns and the corresponding columns have compatible data types, as described in [**Section 10.5**](https://www.postgresql.org/docs/10/typeconv-union-case.html).

**7.5. Sorting Rows**

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

SELECT ***select\_list***

FROM ***table\_expression***

ORDER BY ***sort\_expression1*** [ASC | DESC] [NULLS { FIRST | LAST }]

[, ***sort\_expression2*** [ASC | DESC] [NULLS { FIRST | LAST }] ...]

The sort expression(s) can be any expression that would be valid in the query's select list. An example is:

SELECT a, b FROM table1 ORDER BY a + b, c;

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional ASC or DESC keyword to set the sort direction to ascending or descending. ASC order is the default. Ascending order puts smaller values first, where “smaller” is defined in terms of the < operator. Similarly, descending order is determined with the > operator. [**[5]**](https://www.postgresql.org/docs/10/queries-order.html#ftn.id-1.5.6.9.5.10)

The NULLS FIRST and NULLS LAST options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, NULLS FIRST is the default for DESC order, and NULLS LAST otherwise.

Note that the ordering options are considered independently for each sort column. For example ORDER BY x, y DESC means ORDER BY x ASC, y DESC, which is not the same as ORDER BY x DESC, y DESC.

A ***sort\_expression*** can also be the column label or number of an output column, as in:

SELECT a + b AS sum, c FROM table1 ORDER BY sum;

SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;

both of which sort by the first output column. Note that an output column name has to stand alone, that is, it cannot be used in an expression — for example, this is *not* correct:

SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; -- wrong

This restriction is made to reduce ambiguity. There is still ambiguity if an ORDER BY item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use AS to rename an output column to match some other table column's name.

ORDER BY can be applied to the result of a UNION, INTERSECT, or EXCEPT combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

[**[5]**](https://www.postgresql.org/docs/10/queries-order.html#id-1.5.6.9.5.10) Actually, PostgreSQL uses the *default B-tree operator class* for the expression's data type to determine the sort ordering for ASC and DESC. Conventionally, data types will be set up so that the < and > operators correspond to this sort ordering, but a user-defined data type's designer could choose to do something different.

## 7.6. LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

SELECT ***select\_list***

FROM ***table\_expression***

[ ORDER BY ... ]

[ LIMIT { ***number*** | ALL } ] [ OFFSET ***number*** ]

If a limit count is given, no more than that many rows will be returned (but possibly fewer, if the query itself yields fewer rows). LIMIT ALL is the same as omitting the LIMIT clause, as is LIMIT with a NULL argument.

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause, as is OFFSET with a NULL argument.

If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating query plans, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

## 7.7. VALUES Lists

VALUES provides a way to generate a “constant table” that can be used in a query without having to actually create and populate a table on-disk. The syntax is

VALUES ( ***expression*** [, ...] ) [, ...]

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for UNION (see [**Section 10.5**](https://www.postgresql.org/docs/10/typeconv-union-case.html)).

As an example:

VALUES (1, 'one'), (2, 'two'), (3, 'three');

will return a table of two columns and three rows. It's effectively equivalent to:

SELECT 1 AS column1, 'one' AS column2

UNION ALL

SELECT 2, 'two'

UNION ALL

SELECT 3, 'three';

By default, PostgreSQL assigns the names column1, column2, etc. to the columns of a VALUES table. The column names are not specified by the SQL standard and different database systems do it differently, so it's usually better to override the default names with a table alias list, like this:

=> SELECT \* FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);

num | letter

-----+--------

1 | one

2 | two

3 | three

(3 rows)

Syntactically, VALUES followed by expression lists is treated as equivalent to:

SELECT ***select\_list*** FROM ***table\_expression***

and can appear anywhere a SELECT can. For example, you can use it as part of a UNION, or attach a ***sort\_specification*** (ORDER BY, LIMIT, and/or OFFSET) to it. VALUES is most commonly used as the data source in an INSERT command, and next most commonly as a subquery.

For more information see [**VALUES**](https://www.postgresql.org/docs/10/sql-values.html).

## 7.8. WITH Queries (Common Table Expressions)

[**7.8.1.**SELECT**in**WITH](https://www.postgresql.org/docs/10/queries-with.html#QUERIES-WITH-SELECT)

[**7.8.2. Data-Modifying Statements in**WITH](https://www.postgresql.org/docs/10/queries-with.html#QUERIES-WITH-MODIFYING)

WITH provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

### 7.8.1. SELECT in WITH

The basic value of SELECT in WITH is to break down complicated queries into simpler parts. An example is:

WITH regional\_sales AS (

SELECT region, SUM(amount) AS total\_sales

FROM orders

GROUP BY region

), top\_regions AS (

SELECT region

FROM regional\_sales

WHERE total\_sales > (SELECT SUM(total\_sales)/10 FROM regional\_sales)

)

SELECT region,

product,

SUM(quantity) AS product\_units,

SUM(amount) AS product\_sales

FROM orders

WHERE region IN (SELECT region FROM top\_regions)

GROUP BY region, product;

which displays per-product sales totals in only the top sales regions. The WITH clause defines two auxiliary statements named regional\_sales and top\_regions, where the output of regional\_sales is used in top\_regions and the output of top\_regions is used in the primary SELECT query. This example could have been written without WITH, but we'd have needed two levels of nested sub-SELECTs. It's a bit easier to follow this way.

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

WITH RECURSIVE t(n) AS (

VALUES (1)

UNION ALL

SELECT n+1 FROM t WHERE n < 100

)

SELECT sum(n) FROM t;

The general form of a recursive WITH query is always a non-recursive term, then UNION (or UNION ALL), then a recursive term, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

**Recursive Query Evaluation**

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.
2. So long as the working table is not empty, repeat these steps:
   1. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.
   2. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

Note

Strictly speaking, this process is iteration not recursion, but RECURSIVE is the terminology chosen by the SQL standards committee.

In the example above, the working table has just a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output because of the WHEREclause, and so the query terminates.

Recursive queries are typically used to deal with hierarchical or tree-structured data. A useful example is this query to find all the direct and indirect sub-parts of a product, given only a table that shows immediate inclusions:

WITH RECURSIVE included\_parts(sub\_part, part, quantity) AS (

SELECT sub\_part, part, quantity FROM parts WHERE part = 'our\_product'

UNION ALL

SELECT p.sub\_part, p.part, p.quantity

FROM included\_parts pr, parts p

WHERE p.part = pr.sub\_part

)

SELECT sub\_part, SUM(quantity) as total\_quantity

FROM included\_parts

GROUP BY sub\_part

When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using UNIONinstead of UNION ALL can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values. For example, consider the following query that searches a table graph using a link field:

WITH RECURSIVE search\_graph(id, link, data, depth) AS (

SELECT g.id, g.link, g.data, 1

FROM graph g

UNION ALL

SELECT g.id, g.link, g.data, sg.depth + 1

FROM graph g, search\_graph sg

WHERE g.id = sg.link

)

SELECT \* FROM search\_graph;

This query will loop if the link relationships contain cycles. Because we require a “depth” output, just changing UNION ALL to UNION would not eliminate the looping. Instead we need to recognize whether we have reached the same row again while following a particular path of links. We add two columns path and cycle to the loop-prone query:

WITH RECURSIVE search\_graph(id, link, data, depth, path, cycle) AS (

SELECT g.id, g.link, g.data, 1,

ARRAY[g.id],

false

FROM graph g

UNION ALL

SELECT g.id, g.link, g.data, sg.depth + 1,

path || g.id,

g.id = ANY(path)

FROM graph g, search\_graph sg

WHERE g.id = sg.link AND NOT cycle

)

SELECT \* FROM search\_graph;

Aside from preventing cycles, the array value is often useful in its own right as representing the “path” taken to reach any particular row.

In the general case where more than one field needs to be checked to recognize a cycle, use an array of rows. For example, if we needed to compare fields f1 and f2:

WITH RECURSIVE search\_graph(id, link, data, depth, path, cycle) AS (

SELECT g.id, g.link, g.data, 1,

ARRAY[ROW(g.f1, g.f2)],

false

FROM graph g

UNION ALL

SELECT g.id, g.link, g.data, sg.depth + 1,

path || ROW(g.f1, g.f2),

ROW(g.f1, g.f2) = ANY(path)

FROM graph g, search\_graph sg

WHERE g.id = sg.link AND NOT cycle

)

SELECT \* FROM search\_graph;

Tip

Omit the ROW() syntax in the common case where only one field needs to be checked to recognize a cycle. This allows a simple array rather than a composite-type array to be used, gaining efficiency.

Tip

The recursive query evaluation algorithm produces its output in breadth-first search order. You can display the results in depth-first search order by making the outer query ORDER BY a “path” column constructed in this way.

A helpful trick for testing queries when you are not certain if they might loop is to place a LIMIT in the parent query. For example, this query would loop forever without the LIMIT:

WITH RECURSIVE t(n) AS (

SELECT 1

UNION ALL

SELECT n+1 FROM t

)

SELECT n FROM t LIMIT 100;

This works because PostgreSQL's implementation evaluates only as many rows of a WITH query as are actually fetched by the parent query. Using this trick in production is not recommended, because other systems might work differently. Also, it usually won't work if you make the outer query sort the recursive query's results or join them to some other table, because in such cases the outer query will usually try to fetch all of the WITH query's output anyway.

A useful property of WITH queries is that they are evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to avoid redundant work. Another possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary subquery. The WITHquery will generally be evaluated as written, without suppression of rows that the parent query might discard afterwards. (But, as mentioned above, evaluation might stop early if the reference(s) to the query demand only a limited number of rows.)

The examples above only show WITH being used with SELECT, but it can be attached in the same way to INSERT, UPDATE, or DELETE. In each case it effectively provides temporary table(s) that can be referred to in the main command.

### 7.8.2. Data-Modifying Statements in WITH

You can use data-modifying statements (INSERT, UPDATE, or DELETE) in WITH. This allows you to perform several different operations in the same query. An example is:

WITH moved\_rows AS (

DELETE FROM products

WHERE

"date" >= '2010-10-01' AND

"date" < '2010-11-01'

RETURNING \*

)

INSERT INTO products\_log

SELECT \* FROM moved\_rows;

This query effectively moves rows from products to products\_log. The DELETE in WITH deletes the specified rows from products, returning their contents by means of its RETURNING clause; and then the primary query reads that output and inserts it into products\_log.

A fine point of the above example is that the WITH clause is attached to the INSERT, not the sub-SELECT within the INSERT. This is necessary because data-modifying statements are only allowed in WITHclauses that are attached to the top-level statement. However, normal WITH visibility rules apply, so it is possible to refer to the WITH statement's output from the sub-SELECT.

Data-modifying statements in WITH usually have RETURNING clauses (see [**Section 6.4**](https://www.postgresql.org/docs/10/dml-returning.html)), as shown in the example above. It is the output of the RETURNING clause, not the target table of the data-modifying statement, that forms the temporary table that can be referred to by the rest of the query. If a data-modifying statement in WITH lacks a RETURNING clause, then it forms no temporary table and cannot be referred to in the rest of the query. Such a statement will be executed nonetheless. A not-particularly-useful example is:

WITH t AS (

DELETE FROM foo

)

DELETE FROM bar;

This example would remove all rows from tables foo and bar. The number of affected rows reported to the client would only include rows removed from bar.

Recursive self-references in data-modifying statements are not allowed. In some cases it is possible to work around this limitation by referring to the output of a recursive WITH, for example:

WITH RECURSIVE included\_parts(sub\_part, part) AS (

SELECT sub\_part, part FROM parts WHERE part = 'our\_product'

UNION ALL

SELECT p.sub\_part, p.part

FROM included\_parts pr, parts p

WHERE p.part = pr.sub\_part

)

DELETE FROM parts

WHERE part IN (SELECT part FROM included\_parts);

This query would remove all direct and indirect subparts of a product.

Data-modifying statements in WITH are executed exactly once, and always to completion, independently of whether the primary query reads all (or indeed any) of their output. Notice that this is different from the rule for SELECT in WITH: as stated in the previous section, execution of a SELECT is carried only as far as the primary query demands its output.

The sub-statements in WITH are executed concurrently with each other and with the main query. Therefore, when using data-modifying statements in WITH, the order in which the specified updates actually happen is unpredictable. All the statements are executed with the same snapshot (see [**Chapter 13**](https://www.postgresql.org/docs/10/mvcc.html)), so they cannot “see” one another's effects on the target tables. This alleviates the effects of the unpredictability of the actual order of row updates, and means that RETURNING data is the only way to communicate changes between different WITH sub-statements and the main query. An example of this is that in

WITH t AS (

UPDATE products SET price = price \* 1.05

RETURNING \*

)

SELECT \* FROM products;

the outer SELECT would return the original prices before the action of the UPDATE, while in

WITH t AS (

UPDATE products SET price = price \* 1.05

RETURNING \*

)

SELECT \* FROM t;

the outer SELECT would return the updated data.

Trying to update the same row twice in a single statement is not supported. Only one of the modifications takes place, but it is not easy (and sometimes not possible) to reliably predict which one. This also applies to deleting a row that was already updated in the same statement: only the update is performed. Therefore you should generally avoid trying to modify a single row twice in a single statement. In particular avoid writing WITH sub-statements that could affect the same rows changed by the main statement or a sibling sub-statement. The effects of such a statement will not be predictable.

At present, any table used as the target of a data-modifying statement in WITH must not have a conditional rule, nor an ALSO rule, nor an INSTEAD rule that expands to multiple statements.

## Chapter 8. Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) command.

[**Table 8.1**](https://www.postgresql.org/docs/10/datatype.html#DATATYPE-TABLE) shows all the built-in general-purpose data types. Most of the alternative names listed in the “Aliases” column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but are not listed here.

**Table 8.1. Data Types**

| **Name** | **Aliases** | **Description** |
| --- | --- | --- |
| bigint | int8 | signed eight-byte integer |
| bigserial | serial8 | autoincrementing eight-byte integer |
| bit [ (***n***) ] |  | fixed-length bit string |
| bit varying [ (***n***) ] | varbit [ (***n***) ] | variable-length bit string |
| boolean | bool | logical Boolean (true/false) |
| box |  | rectangular box on a plane |
| bytea |  | binary data (“byte array”) |
| character [ (***n***) ] | char [ (***n***) ] | fixed-length character string |
| character varying [ (***n***) ] | varchar [ (***n***) ] | variable-length character string |
| cidr |  | IPv4 or IPv6 network address |
| circle |  | circle on a plane |
| date |  | calendar date (year, month, day) |
| double precision | float8 | double precision floating-point number (8 bytes) |
| inet |  | IPv4 or IPv6 host address |
| integer | int, int4 | signed four-byte integer |
| interval [ ***fields*** ] [ (***p***) ] |  | time span |
| json |  | textual JSON data |
| jsonb |  | binary JSON data, decomposed |
| line |  | infinite line on a plane |
| lseg |  | line segment on a plane |
| macaddr |  | MAC (Media Access Control) address |
| macaddr8 |  | MAC (Media Access Control) address (EUI-64 format) |
| money |  | currency amount |
| numeric [ (***p***, ***s***) ] | decimal [ (***p***, ***s***) ] | exact numeric of selectable precision |
| path |  | geometric path on a plane |
| pg\_lsn |  | PostgreSQL Log Sequence Number |
| point |  | geometric point on a plane |
| polygon |  | closed geometric path on a plane |
| real | float4 | single precision floating-point number (4 bytes) |
| smallint | int2 | signed two-byte integer |
| smallserial | serial2 | autoincrementing two-byte integer |
| serial | serial4 | autoincrementing four-byte integer |
| text |  | variable-length character string |
| time [ (***p***) ] [ without time zone ] |  | time of day (no time zone) |
| time [ (***p***) ] with time zone | timetz | time of day, including time zone |
| timestamp [ (***p***) ] [ without time zone ] |  | date and time (no time zone) |
| timestamp [ (***p***) ] with time zone | timestamptz | date and time, including time zone |
| tsquery |  | text search query |
| tsvector |  | text search document |
| txid\_snapshot |  | user-level transaction ID snapshot |
| uuid |  | universally unique identifier |
| xml |  | XML data |

### Compatibility

The following types (or spellings thereof) are specified by SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (with or without time zone), timestamp (with or without time zone), xml.

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possible formats, such as the date and time types. Some of the input and output functions are not invertible, i.e., the result of an output function might lose accuracy when compared to the original input.

## 8.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. [**Table 8.2**](https://www.postgresql.org/docs/10/datatype-numeric.html#DATATYPE-NUMERIC-TABLE) lists the available types.

**Table 8.2. Numeric Types**

| **Name** | **Storage Size** | **Description** | **Range** |
| --- | --- | --- | --- |
| smallint | 2 bytes | small-range integer | -32768 to +32767 |
| integer | 4 bytes | typical choice for integer | -2147483648 to +2147483647 |
| bigint | 8 bytes | large-range integer | -9223372036854775808 to +9223372036854775807 |
| decimal | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| numeric | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| real | 4 bytes | variable-precision, inexact | 6 decimal digits precision |
| double precision | 8 bytes | variable-precision, inexact | 15 decimal digits precision |
| smallserial | 2 bytes | small autoincrementing integer | 1 to 32767 |
| serial | 4 bytes | autoincrementing integer | 1 to 2147483647 |
| bigserial | 8 bytes | large autoincrementing integer | 1 to 9223372036854775807 |

The syntax of constants for the numeric types is described in [**Section 4.1.2**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS). The numeric types have a full set of corresponding arithmetic operators and functions. Refer to [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html) for more information. The following sections describe the types in detail.

### 8.1.1. Integer Types

The types smallint, integer, and bigint store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type integer is the common choice, as it offers the best balance between range, storage size, and performance. The smallint type is generally only used if disk space is at a premium. The biginttype is designed to be used when the range of the integer type is insufficient.

SQL only specifies the integer types integer (or int), smallint, and bigint. The type names int2, int4, and int8 are extensions, which are also used by some other SQL database systems.

### 8.1.2. Arbitrary Precision Numbers

The type numeric can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. Calculations with numeric values yield exact results where possible, e.g. addition, subtraction, multiplication. However, calculations on numeric values are very slow compared to the integer types, or to the floating-point types described in the next section.

We use the following terms below: The precision of a numeric is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The scale of a numeric is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a numeric column can be configured. To declare a column of type numeric use the syntax:

NUMERIC(***precision***, ***scale***)

The precision must be positive, the scale zero or positive. Alternatively:

NUMERIC(***precision***)

selects a scale of 0. Specifying:

NUMERIC

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas numeric columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

Note

The maximum allowed precision when explicitly specified in the type declaration is 1000; NUMERIC without a specified precision is subject to the limits described in [**Table 8.2**](https://www.postgresql.org/docs/10/datatype-numeric.html#DATATYPE-NUMERIC-TABLE).

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the numerictype is more akin to varchar(***n***) than to char(***n***).) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

In addition to ordinary numeric values, the numeric type allows the special value NaN, meaning “not-a-number”. Any operation on NaN yields another NaN. When writing this value as a constant in an SQL command, you must put quotes around it, for example UPDATE table SET x = 'NaN'. On input, the string NaN is recognized in a case-insensitive manner.

Note

In most implementations of the “not-a-number” concept, NaN is not considered equal to any other numeric value (including NaN). In order to allow numeric values to be sorted and used in tree-based indexes, PostgreSQL treats NaN values as equal, and greater than all non-NaN values.

The types decimal and numeric are equivalent. Both types are part of the SQL standard.

When rounding values, the numeric type rounds ties away from zero, while (on most machines) the real and double precision types round ties to the nearest even number. For example:

SELECT x,

round(x::numeric) AS num\_round,

round(x::double precision) AS dbl\_round

FROM generate\_series(-3.5, 3.5, 1) as x;

x | num\_round | dbl\_round

------+-----------+-----------

-3.5 | -4 | -4

-2.5 | -3 | -2

-1.5 | -2 | -2

-0.5 | -1 | -0

0.5 | 1 | 0

1.5 | 2 | 2

2.5 | 3 | 2

3.5 | 4 | 4

(8 rows)

### 8.1.3. Floating-Point Types

The data types real and double precision are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

* If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
* If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
* Comparing two floating-point values for equality might not always work as expected.

On most platforms, the real type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The double precision type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Note

The [**extra\_float\_digits**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-EXTRA-FLOAT-DIGITS) setting controls the number of extra significant digits included when a floating point value is converted to text for output. With the default value of 0, the output is the same on every platform supported by PostgreSQL. Increasing it will produce output that more accurately represents the stored value, but may be unportable.

In addition to ordinary numeric values, the floating-point types have several special values:

Infinity  
-Infinity  
NaN

These represent the IEEE 754 special values “infinity”, “negative infinity”, and “not-a-number”, respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will probably not work as expected.) When writing these values as constants in an SQL command, you must put quotes around them, for example UPDATE table SET x = '-Infinity'. On input, these strings are recognized in a case-insensitive manner.

Note

IEEE754 specifies that NaN should not compare equal to any other floating-point value (including NaN). In order to allow floating-point values to be sorted and used in tree-based indexes, PostgreSQL treats NaN values as equal, and greater than all non-NaN values.

PostgreSQL also supports the SQL-standard notations float and float(***p***) for specifying inexact numeric types. Here, ***p*** specifies the minimum acceptable precision in binary digits. PostgreSQL accepts float(1) to float(24) as selecting the real type, while float(25) to float(53) select double precision. Values of ***p*** outside the allowed range draw an error. float with no precision specified is taken to mean double precision.

Note

The assumption that real and double precision have exactly 24 and 53 bits in the mantissa respectively is correct for IEEE-standard floating point implementations. On non-IEEE platforms it might be off a little, but for simplicity the same ranges of ***p*** are used on all platforms.

### 8.1.4. Serial Types

Note

This section describes a PostgreSQL-specific way to create an autoincrementing column. Another way is to use the SQL-standard identity column feature, described at [**CREATE TABLE**](https://www.postgresql.org/docs/10/sql-createtable.html).

The data types smallserial, serial and bigserial are not true types, but merely a notational convenience for creating unique identifier columns (similar to the AUTO\_INCREMENT property supported by some other databases). In the current implementation, specifying:

CREATE TABLE ***tablename*** (

***colname*** SERIAL

);

is equivalent to specifying:

CREATE SEQUENCE ***tablename***\_***colname***\_seq AS integer;

CREATE TABLE ***tablename*** (

***colname*** integer NOT NULL DEFAULT nextval('***tablename***\_***colname***\_seq')

);

ALTER SEQUENCE ***tablename***\_***colname***\_seq OWNED BY ***tablename***.***colname***;

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. A NOT NULL constraint is applied to ensure that a null value cannot be inserted. (In most cases you would also want to attach a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted by accident, but this is not automatic.) Lastly, the sequence is marked as “owned by” the column, so that it will be dropped if the column or table is dropped.

Note

Because smallserial, serial and bigserial are implemented using sequences, there may be "holes" or gaps in the sequence of values which appears in the column, even if no rows are ever deleted. A value allocated from the sequence is still "used up" even if a row containing that value is never successfully inserted into the table column. This may happen, for example, if the inserting transaction rolls back. See nextval() in [**Section 9.16**](https://www.postgresql.org/docs/10/functions-sequence.html)for details.

To insert the next value of the sequence into the serial column, specify that the serial column should be assigned its default value. This can be done either by excluding the column from the list of columns in the INSERT statement, or through the use of the DEFAULT key word.

The type names serial and serial4 are equivalent: both create integer columns. The type names bigserial and serial8 work the same way, except that they create a bigint column. bigserialshould be used if you anticipate the use of more than 231 identifiers over the lifetime of the table. The type names smallserial and serial2 also work the same way, except that they create a smallintcolumn.

The sequence created for a serial column is automatically dropped when the owning column is dropped. You can drop the sequence without dropping the column, but this will force removal of the column default expression.

**8.2. Monetary Types**

The money type stores a currency amount with a fixed fractional precision; see [**Table 8.3**](https://www.postgresql.org/docs/10/datatype-money.html#DATATYPE-MONEY-TABLE). The fractional precision is determined by the database's [**lc\_monetary**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-LC-MONETARY) setting. The range shown in the table assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting, such as '$1,000.00'. Output is generally in the latter form but depends on the locale.

**Table 8.3. Monetary Types**

| **Name** | **Storage Size** | **Description** | **Range** |
| --- | --- | --- | --- |
| money | 8 bytes | currency amount | -92233720368547758.08 to +92233720368547758.07 |

Since the output of this data type is locale-sensitive, it might not work to load money data into a database that has a different setting of lc\_monetary. To avoid problems, before restoring a dump into a new database make sure lc\_monetary has the same or equivalent value as in the database that was dumped.

Values of the numeric, int, and bigint data types can be cast to money. Conversion from the real and double precision data types can be done by casting to numeric first, for example:

SELECT '12.34'::float8::numeric::money;

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A money value can be cast to numeric without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

SELECT '52093.89'::money::numeric::float8;

Division of a money value by an integer value is performed with truncation of the fractional part towards zero. To get a rounded result, divide by a floating-point value, or cast the money value to numericbefore dividing and back to money afterwards. (The latter is preferable to avoid risking precision loss.) When a money value is divided by another money value, the result is double precision (i.e., a pure number, not money); the currency units cancel each other out in the division.

## 8.3. Character Types

**Table 8.4. Character Types**

| **Name** | **Description** |
| --- | --- |
| character varying(***n***), varchar(***n***) | variable-length with limit |
| character(***n***), char(***n***) | fixed-length, blank padded |
| text | variable unlimited length |

[**Table 8.4**](https://www.postgresql.org/docs/10/datatype-character.html#DATATYPE-CHARACTER-TABLE) shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: character varying(***n***) and character(***n***), where ***n*** is a positive integer. Both of these types can store strings up to ***n*** characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string.

If one explicitly casts a value to character varying(***n***) or character(***n***), then an over-length value will be truncated to ***n*** characters without raising an error. (This too is required by the SQL standard.)

The notations varchar(***n***) and char(***n***) are aliases for character varying(***n***) and character(***n***), respectively. character without length specifier is equivalent to character(1). If character varying is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL provides the text type, which stores strings of any length. Although the type text is not in the SQL standard, several other SQL database management systems have it as well.

Values of type character are physically padded with spaces to the specified width ***n***, and are stored and displayed that way. However, trailing spaces are treated as semantically insignificant and disregarded when comparing two values of type character. In collations where whitespace is significant, this behavior can produce unexpected results; for example SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2) returns true, even though C locale would consider a space to be greater than a newline. Trailing spaces are removed when converting a character value to one of the other string types. Note that trailing spaces are semantically significant in character varying and text values, and when using pattern matching, that is LIKE and regular expressions.

The storage requirement for a short string (up to 126 bytes) is 1 byte plus the actual string, which includes the space padding in the case of character. Longer strings have 4 bytes of overhead instead of 1. Long strings are compressed by the system automatically, so the physical requirement on disk might be less. Very long values are also stored in background tables so that they do not interfere with rapid access to shorter column values. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for ***n*** in the data type declaration is less than that. It wouldn't be useful to change this because with multibyte character encodings the number of characters and bytes can be quite different. If you desire to store long strings with no specific upper limit, use text or character varying without a length specifier, rather than making up an arbitrary length limit.)

Tip

There is no performance difference among these three types, apart from increased storage space when using the blank-padded type, and a few extra CPU cycles to check the length when storing into a length-constrained column. While character(***n***) has performance advantages in some other database systems, there is no such advantage in PostgreSQL; in fact character(***n***) is usually the slowest of the three because of its additional storage costs. In most situations text or character varying should be used instead.

Refer to [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS) for information about the syntax of string literals, and to [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html) for information about available operators and functions. The database character set determines the character set used to store textual values; for more information on character set support, refer to [**Section 23.3**](https://www.postgresql.org/docs/10/multibyte.html).

**Example 8.1. Using the Character Types**

CREATE TABLE test1 (a character(4));

INSERT INTO test1 VALUES ('ok');

SELECT a, char\_length(a) FROM test1; -- (1)

a | char\_length

------+-------------

ok | 2

CREATE TABLE test2 (b varchar(5));

INSERT INTO test2 VALUES ('ok');

INSERT INTO test2 VALUES ('good ');

INSERT INTO test2 VALUES ('too long');

ERROR: value too long for type character varying(5)

INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation

SELECT b, char\_length(b) FROM test2;

b | char\_length

-------+-------------

ok | 2

good | 5

too l | 5

|  |  |
| --- | --- |
| [**(1)**](https://www.postgresql.org/docs/10/datatype-character.html#co.datatype-char) | The char\_length function is discussed in [**Section 9.4**](https://www.postgresql.org/docs/10/functions-string.html). |

There are two other fixed-length character types in PostgreSQL, shown in [**Table 8.5**](https://www.postgresql.org/docs/10/datatype-character.html#DATATYPE-CHARACTER-SPECIAL-TABLE). The name type exists only for the storage of identifiers in the internal system catalogs and is not intended for use by the general user. Its length is currently defined as 64 bytes (63 usable characters plus terminator) but should be referenced using the constant NAMEDATALEN in C source code. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length might change in a future release. The type "char" (note the quotes) is different from char(1) in that it only uses one byte of storage. It is internally used in the system catalogs as a simplistic enumeration type.

**Table 8.5. Special Character Types**

| **Name** | **Storage Size** | **Description** |
| --- | --- | --- |
| "char" | 1 byte | single-byte internal type |
| name | 64 bytes | internal type for object names |

## 8.4. Binary Data Types

The bytea data type allows storage of binary strings; see [**Table 8.6**](https://www.postgresql.org/docs/10/datatype-binary.html#DATATYPE-BINARY-TABLE).

**Table 8.6. Binary Data Types**

| **Name** | **Storage Size** | **Description** |
| --- | --- | --- |
| bytea | 1 or 4 bytes plus the actual binary string | variable-length binary string |

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings in two ways. First, binary strings specifically allow storing octets of value zero and other “non-printable” octets (usually, octets outside the decimal range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database's selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as “raw bytes”, whereas character strings are appropriate for storing text.

The bytea type supports two formats for input and output: “hex” format and PostgreSQL's historical “escape” format. Both of these are always accepted on input. The output format depends on the configuration parameter [**bytea\_output**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-BYTEA-OUTPUT); the default is hex. (Note that the hex format was introduced in PostgreSQL 9.0; earlier versions and some tools don't understand it.)

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. The input format is different from bytea, but the provided functions and operators are mostly the same.

### 8.4.1. bytea Hex Format

The “hex” format encodes binary data as 2 hexadecimal digits per byte, most significant nibble first. The entire string is preceded by the sequence \x (to distinguish it from the escape format). In some contexts, the initial backslash may need to be escaped by doubling it (see [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS)). For input, the hexadecimal digits can be either upper or lower case, and whitespace is permitted between digit pairs (but not within a digit pair nor in the starting \x sequence). The hex format is compatible with a wide range of external applications and protocols, and it tends to be faster to convert than the escape format, so its use is preferred.

Example:

SELECT '\xDEADBEEF';

### 8.4.2. bytea Escape Format

The “escape” format is the traditional PostgreSQL format for the bytea type. It takes the approach of representing a binary string as a sequence of ASCII characters, while converting those bytes that cannot be represented as an ASCII character into special escape sequences. If, from the point of view of the application, representing bytes as characters makes sense, then this representation can be convenient. But in practice it is usually confusing because it fuzzes up the distinction between binary strings and character strings, and also the particular escape mechanism that was chosen is somewhat unwieldy. Therefore, this format should probably be avoided for most new applications.

When entering bytea values in escape format, octets of certain values must be escaped, while all octet values can be escaped. In general, to escape an octet, convert it into its three-digit octal value and precede it by a backslash. Backslash itself (octet decimal value 92) can alternatively be represented by double backslashes. [**Table 8.7**](https://www.postgresql.org/docs/10/datatype-binary.html#DATATYPE-BINARY-SQLESC) shows the characters that must be escaped, and gives the alternative escape sequences where applicable.

**Table 8.7.**bytea**Literal Escaped Octets**

| **Decimal Octet Value** | **Description** | **Escaped Input Representation** | **Example** | **Hex Representation** |
| --- | --- | --- | --- | --- |
| 0 | zero octet | '\000' | SELECT '\000'::bytea; | \x00 |
| 39 | single quote | '''' or '\047' | SELECT ''''::bytea; | \x27 |
| 92 | backslash | '\\' or '\134' | SELECT '\\'::bytea; | \x5c |
| 0 to 31 and 127 to 255 | “non-printable” octets | '\***xxx'*** (octal value) | SELECT '\001'::bytea; | \x01 |

The requirement to escape non-printable octets varies depending on locale settings. In some instances you can get away with leaving them unescaped.

The reason that single quotes must be doubled, as shown in [**Table 8.7**](https://www.postgresql.org/docs/10/datatype-binary.html#DATATYPE-BINARY-SQLESC), is that this is true for any string literal in a SQL command. The generic string-literal parser consumes the outermost single quotes and reduces any pair of single quotes to one data character. What the bytea input function sees is just one single quote, which it treats as a plain data character. However, the bytea input function treats backslashes as special, and the other behaviors shown in [**Table 8.7**](https://www.postgresql.org/docs/10/datatype-binary.html#DATATYPE-BINARY-SQLESC) are implemented by that function.

In some contexts, backslashes must be doubled compared to what is shown above, because the generic string-literal parser will also reduce pairs of backslashes to one data character; see [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS).

Bytea octets are output in hex format by default. If you change [**bytea\_output**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-BYTEA-OUTPUT) to escape, “non-printable” octets are converted to their equivalent three-digit octal value and preceded by one backslash. Most “printable” octets are output by their standard representation in the client character set, e.g.:

SET bytea\_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;

bytea

----------------

abc klm \*\251T

The octet with decimal value 92 (backslash) is doubled in the output. Details are in [**Table 8.8**](https://www.postgresql.org/docs/10/datatype-binary.html#DATATYPE-BINARY-RESESC).

**Table 8.8.**bytea**Output Escaped Octets**

| **Decimal Octet Value** | **Description** | **Escaped Output Representation** | **Example** | **Output Result** |
| --- | --- | --- | --- | --- |
| 92 | backslash | \\ | SELECT '\134'::bytea; | \\ |
| 0 to 31 and 127 to 255 | “non-printable” octets | \***xxx*** (octal value) | SELECT '\001'::bytea; | \001 |
| 32 to 126 | “printable” octets | client character set representation | SELECT '\176'::bytea; | ~ |

Depending on the front end to PostgreSQL you use, you might have additional work to do in terms of escaping and unescaping bytea strings. For example, you might also have to escape line feeds and carriage returns if your interface automatically translates these.

## 8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in [**Table 8.9**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-TABLE). The operations available on these data types are described in [**Section 9.9**](https://www.postgresql.org/docs/10/functions-datetime.html). Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced (see [**Section B.5**](https://www.postgresql.org/docs/10/datetime-units-history.html) for more information).

**Table 8.9. Date/Time Types**

| **Name** | **Storage Size** | **Description** | **Low Value** | **High Value** | **Resolution** |
| --- | --- | --- | --- | --- | --- |
| timestamp [ (***p***) ] [ without time zone ] | 8 bytes | both date and time (no time zone) | 4713 BC | 294276 AD | 1 microsecond |
| timestamp [ (***p***) ] with time zone | 8 bytes | both date and time, with time zone | 4713 BC | 294276 AD | 1 microsecond |
| date | 4 bytes | date (no time of day) | 4713 BC | 5874897 AD | 1 day |
| time [ (***p***) ] [ without time zone ] | 8 bytes | time of day (no date) | 00:00:00 | 24:00:00 | 1 microsecond |
| time [ (***p***) ] with time zone | 12 bytes | time of day (no date), with time zone | 00:00:00+1459 | 24:00:00-1459 | 1 microsecond |
| interval [ ***fields*** ] [ (***p***) ] | 16 bytes | time interval | -178000000 years | 178000000 years | 1 microsecond |

Note

The SQL standard requires that writing just timestamp be equivalent to timestamp without time zone, and PostgreSQL honors that behavior. timestamptz is accepted as an abbreviation for timestamp with time zone; this is a PostgreSQL extension.

time, timestamp, and interval accept an optional precision value ***p*** which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of ***p*** is from 0 to 6.

The interval type has an additional option, which is to restrict the set of stored fields by writing one of these phrases:

YEAR

MONTH

DAY

HOUR

MINUTE

SECOND

YEAR TO MONTH

DAY TO HOUR

DAY TO MINUTE

DAY TO SECOND

HOUR TO MINUTE

HOUR TO SECOND

MINUTE TO SECOND

Note that if both ***fields*** and ***p*** are specified, the ***fields*** must include SECOND, since the precision applies only to the seconds.

The type time with time zone is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of date, time, timestamp without time zone, and timestamp with time zone should provide a complete range of date/time functionality required by any application.

The types abstime and reltime are lower precision types which are used internally. You are discouraged from using these types in applications; these internal types might disappear in a future release.

### 8.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of day, month, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the [**DateStyle**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DATESTYLE) parameter to MDY to select month-day-year interpretation, DMY to select day-month-year interpretation, or YMD to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See [**Appendix B**](https://www.postgresql.org/docs/10/datetime-appendix.html) for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to [**Section 4.1.2.7**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC) for more information. SQL requires the following syntax

***type*** [ (***p***) ] '***value***'

where ***p*** is an optional precision specification giving the number of fractional digits in the seconds field. Precision can be specified for time, timestamp, and interval types, and can range from 0 to 6. If no precision is specified in a constant specification, it defaults to the precision of the literal value (but not more than 6 digits).

#### 8.5.1.1. Dates

**[Table 8.10](https://www.postgresql.org/docs/10/datatype-datetime.html" \l "DATATYPE-DATETIME-DATE-TABLE" \o "Table 8.10. Date Input)** shows some possible inputs for the date type.

**Table 8.10. Date Input**

| **Example** | **Description** |
| --- | --- |
| 1999-01-08 | ISO 8601; January 8 in any mode (recommended format) |
| January 8, 1999 | unambiguous in any datestyle input mode |
| 1/8/1999 | January 8 in MDY mode; August 1 in DMY mode |
| 1/18/1999 | January 18 in MDY mode; rejected in other modes |
| 01/02/03 | January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode |
| 1999-Jan-08 | January 8 in any mode |
| Jan-08-1999 | January 8 in any mode |
| 08-Jan-1999 | January 8 in any mode |
| 99-Jan-08 | January 8 in YMD mode, else error |
| 08-Jan-99 | January 8, except error in YMD mode |
| Jan-08-99 | January 8, except error in YMD mode |
| 19990108 | ISO 8601; January 8, 1999 in any mode |
| 990108 | ISO 8601; January 8, 1999 in any mode |
| 1999.008 | year and day of year |
| J2451187 | Julian date |
| January 8, 99 BC | year 99 BC |

#### 8.5.1.2. Times

The time-of-day types are time [ (***p***) ] without time zone and time [ (***p***) ] with time zone. time alone is equivalent to time without time zone.

Valid input for these types consists of a time of day followed by an optional time zone. (See [**Table 8.11**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-TIME-TABLE) and [**Table 8.12**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-TIMEZONE-TABLE).) If a time zone is specified in the input for time without time zone, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as America/New\_York. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the time with time zone value.

**Table 8.11. Time Input**

| **Example** | **Description** |
| --- | --- |
| 04:05:06.789 | ISO 8601 |
| 04:05:06 | ISO 8601 |
| 04:05 | ISO 8601 |
| 040506 | ISO 8601 |
| 04:05 AM | same as 04:05; AM does not affect value |
| 04:05 PM | same as 16:05; input hour must be <= 12 |
| 04:05:06.789-8 | ISO 8601 |
| 04:05:06-08:00 | ISO 8601 |
| 04:05-08:00 | ISO 8601 |
| 040506-08 | ISO 8601 |
| 04:05:06 PST | time zone specified by abbreviation |
| 2003-04-12 04:05:06 America/New\_York | time zone specified by full name |

**Table 8.12. Time Zone Input**

| **Example** | **Description** |
| --- | --- |
| PST | Abbreviation (for Pacific Standard Time) |
| America/New\_York | Full time zone name |
| PST8PDT | POSIX-style time zone specification |
| -8:00 | ISO-8601 offset for PST |
| -800 | ISO-8601 offset for PST |
| -8 | ISO-8601 offset for PST |
| zulu | Military abbreviation for UTC |
| z | Short form of zulu |

Refer to [**Section 8.5.3**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-TIMEZONES) for more information on how to specify time zones.

#### 8.5.1.3. Time Stamps

Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC. (Alternatively, AD/BC can appear before the time zone, but this is not the preferred ordering.) Thus:

1999-01-08 04:05:06

and:

1999-01-08 04:05:06 -8:00

are valid values, which follow the ISO 8601 standard. In addition, the common format:

January 8 04:05:06 1999 PST

is supported.

The SQL standard differentiates timestamp without time zone and timestamp with time zone literals by the presence of a “+” or “-” symbol and time zone offset after the time. Hence, according to the standard,

TIMESTAMP '2004-10-19 10:23:54'

is a timestamp without time zone, while

TIMESTAMP '2004-10-19 10:23:54+02'

is a timestamp with time zone. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as timestamp without time zone. To ensure that a literal is treated as timestamp with time zone, give it the correct explicit type:

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

In a literal that has been determined to be timestamp without time zone, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For timestamp with time zone, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's [**TimeZone**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-TIMEZONE) parameter, and is converted to UTC using the offset for the timezone zone.

When a timestamp with time zone value is output, it is always converted from UTC to the current timezone zone, and displayed as local time in that zone. To see the time in another time zone, either change timezone or use the AT TIME ZONE construct (see [**Section 9.9.3**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-ZONECONVERT)).

Conversions between timestamp without time zone and timestamp with time zone normally assume that the timestamp without time zone value should be taken or given as timezone local time. A different time zone can be specified for the conversion using AT TIME ZONE.

#### 8.5.1.4. Special Values

PostgreSQL supports several special date/time input values for convenience, as shown in [**Table 8.13**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-SPECIAL-TABLE). The values infinity and -infinity are specially represented inside the system and will be displayed unchanged; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, now and related strings are converted to a specific time value as soon as they are read.) All of these values need to be enclosed in single quotes when used as constants in SQL commands.

**Table 8.13. Special Date/Time Inputs**

| **Input String** | **Valid Types** | **Description** |
| --- | --- | --- |
| epoch | date, timestamp | 1970-01-01 00:00:00+00 (Unix system time zero) |
| infinity | date, timestamp | later than all other time stamps |
| -infinity | date, timestamp | earlier than all other time stamps |
| now | date, time, timestamp | current transaction's start time |
| today | date, timestamp | midnight today |
| tomorrow | date, timestamp | midnight tomorrow |
| yesterday | date, timestamp | midnight yesterday |
| allballs | time | 00:00:00.00 UTC |

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. The latter four accept an optional subsecond precision specification. (See [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT).) Note that these are SQL functions and are not recognized in data input strings.

### 8.5.2. Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES (Unix date format), or German. The default is the ISO format. (The SQLstandard requires the use of the ISO 8601 format. The name of the “SQL” output format is a historical accident.) [**Table 8.14**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-OUTPUT-TABLE) shows examples of each output style. The output of the date and time types is generally only the date or time part in accordance with the given examples. However, the POSTGRES style outputs date-only values in ISO format.

**Table 8.14. Date/Time Output Styles**

| **Style Specification** | **Description** | **Example** |
| --- | --- | --- |
| ISO | ISO 8601, SQL standard | 1997-12-17 07:37:16-08 |
| SQL | traditional style | 12/17/1997 07:37:16.00 PST |
| Postgres | original style | Wed Dec 17 07:37:16 1997 PST |
| German | regional style | 17.12.1997 07:37:16.00 PST |

Note

ISO 8601 specifies the use of uppercase letter T to separate the date and time. PostgreSQLaccepts that format on input, but on output it uses a space rather than T, as shown above. This is for readability and for consistency with RFC 3339 as well as some other database systems.

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See [**Section 8.5.1**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-INPUT) for how this setting also affects interpretation of input values.) [**Table 8.15**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-DATETIME-OUTPUT2-TABLE) shows examples.

**Table 8.15. Date Order Conventions**

| datestyle**Setting** | **Input Ordering** | **Example Output** |
| --- | --- | --- |
| SQL, DMY | ***day***/***month***/***year*** | 17/12/1997 15:37:16.00 CET |
| SQL, MDY | ***month***/***day***/***year*** | 12/17/1997 07:37:16.00 PST |
| Postgres, DMY | ***day***/***month***/***year*** | Wed 17 Dec 07:37:16 1997 PST |

The date/time style can be selected by the user using the SET datestyle command, the [**DateStyle**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DATESTYLE) parameter in the postgresql.conf configuration file, or the PGDATESTYLE environment variable on the server or client.

The formatting function to\_char (see [**Section 9.8**](https://www.postgresql.org/docs/10/functions-formatting.html)) is also available as a more flexible way to format date/time output.

### 8.5.3. Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL uses the widely-used IANA (Olson) time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

PostgreSQL endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

* Although the date type cannot have an associated time zone, the time type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
* The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DSTboundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We do not recommend using the type time with time zone (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the [**TimeZone**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-TIMEZONE) configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

* A full time zone name, for example America/New\_York. The recognized time zone names are listed in the pg\_timezone\_names view (see [**Section 51.90**](https://www.postgresql.org/docs/10/view-pg-timezone-names.html)). PostgreSQL uses the widely-used IANA time zone data for this purpose, so the same time zone names are also recognized by other software.
* A time zone abbreviation, for example PST. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition-date rules as well. The recognized abbreviations are listed in the pg\_timezone\_abbrevs view (see [**Section 51.89**](https://www.postgresql.org/docs/10/view-pg-timezone-abbrevs.html)). You cannot set the configuration parameters [**TimeZone**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-TIMEZONE) or [**log\_timezone**](https://www.postgresql.org/docs/10/runtime-config-logging.html#GUC-LOG-TIMEZONE) to a time zone abbreviation, but you can use abbreviations in date/time input values and with the AT TIME ZONE operator.
* In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications of the form ***STDoffset*** or ***STDoffsetDST***, where ***STD*** is a zone abbreviation, ***offset*** is a numeric offset in hours west from UTC, and ***DST*** is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if EST5EDT were not already a recognized zone name, it would be accepted and would be functionally equivalent to United States East Coast time. In this syntax, a zone abbreviation can be a string of letters, or an arbitrary string surrounded by angle brackets (<>). When a daylight-savings zone abbreviation is present, it is assumed to be used according to the same daylight-savings transition rules used in the IANA time zone database's posixrules entry. In a standard PostgreSQL installation, posixrules is the same as US/Eastern, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the posixrules file.

In short, this is the difference between abbreviations and full names: abbreviations represent a specific offset from UTC, whereas many of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets. As an example, 2014-06-04 12:00 America/New\_York represents noon local time in New York, which for this particular date was Eastern Daylight Time (UTC-4). So 2014-06-04 12:00 EDT specifies that same time instant. But 2014-06-04 12:00 EST specifies noon Eastern Standard Time (UTC-5), regardless of whether daylight savings was nominally in effect on that date.

To complicate matters, some jurisdictions have used the same timezone abbreviation to mean different UTC offsets at different times; for example, in Moscow MSK has meant UTC+3 in some years and UTC+4 in others. PostgreSQL interprets such abbreviations according to whatever they meant (or had most recently meant) on the specified date; but, as with the EST example above, this is not necessarily the same as local civil time on that date.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, SET TIMEZONE TO FOOBAR0 will work, leaving the system effectively using a rather peculiar abbreviation for UTC. Another issue to keep in mind is that in POSIX time zone names, positive offsets are used for locations west of Greenwich. Everywhere else, PostgreSQL follows the ISO-8601 convention that positive timezone offsets are east of Greenwich.

In all cases, timezone names and abbreviations are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts but not others.)

Neither timezone names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under .../share/timezone/ and .../share/timezonesets/ of the installation directory (see [**Section B.4**](https://www.postgresql.org/docs/10/datetime-config-files.html)).

The [**TimeZone**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-TIMEZONE) configuration parameter can be set in the file postgresql.conf, or in any of the other standard ways described in [**Chapter 19**](https://www.postgresql.org/docs/10/runtime-config.html). There are also some special ways to set it:

* The SQL command SET TIME ZONE sets the time zone for the session. This is an alternative spelling of SET TIMEZONE TO with a more SQL-spec-compatible syntax.
* The PGTZ environment variable is used by libpq clients to send a SET TIME ZONE command to the server upon connection.

### 8.5.4. Interval Input

interval values can be written using the following verbose syntax:

[@] ***quantity*** ***unit*** [***quantity*** ***unit***...] [***direction***]

where ***quantity*** is a number (possibly signed); ***unit*** is microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, or abbreviations or plurals of these units; ***direction*** can be ago or empty. The at sign (@) is optional noise. The amounts of the different units are implicitly added with appropriate sign accounting. ago negates all the fields. This syntax is also used for interval output, if [**IntervalStyle**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-INTERVALSTYLE) is set to postgres\_verbose.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, '1 12:59:10' is read the same as '1 day 12 hours 59 min 10 sec'. Also, a combination of years and months can be specified with a dash; for example '200-10' is read the same as '200 years 10 months'. (These shorter forms are in fact the only ones allowed by the SQL standard, and are used for output when IntervalStyle is set to sql\_standard.)

Interval values can also be written as ISO 8601 time intervals, using either the “format with designators” of the standard's section 4.4.3.2 or the “alternative format” of section 4.4.3.3. The format with designators looks like this:

P ***quantity*** ***unit*** [ ***quantity*** ***unit*** ...] [ T [ ***quantity*** ***unit*** ...]]

The string must start with a P, and may include a T that introduces the time-of-day units. The available unit abbreviations are given in [**Table 8.16**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-INTERVAL-ISO8601-UNITS). Units may be omitted, and may be specified in any order, but units smaller than a day must appear after T. In particular, the meaning of M depends on whether it is before or after T.

**Table 8.16. ISO 8601 Interval Unit Abbreviations**

| **Abbreviation** | **Meaning** |
| --- | --- |
| Y | Years |
| M | Months (in the date part) |
| W | Weeks |
| D | Days |
| H | Hours |
| M | Minutes (in the time part) |
| S | Seconds |

In the alternative format:

P [ ***years***-***months***-***days*** ] [ T ***hours***:***minutes***:***seconds*** ]

the string must begin with P, and a T separates the date and time parts of the interval. The values are given as numbers similar to ISO 8601 dates.

When writing an interval constant with a ***fields*** specification, or when assigning a string to an interval column that was defined with a ***fields*** specification, the interpretation of unmarked quantities depends on the ***fields***. For example INTERVAL '1' YEAR is read as 1 year, whereas INTERVAL '1' means 1 second. Also, field values “to the right” of the least significant field allowed by the ***fields***specification are silently discarded. For example, writing INTERVAL '1 day 2:03:04' HOUR TO MINUTE results in dropping the seconds field, but not the day field.

According to the SQL standard all fields of an interval value must have the same sign, so a leading negative sign applies to all fields; for example the negative sign in the interval literal '-1 2:03:04'applies to both the days and hour/minute/second parts. PostgreSQL allows the fields to have different signs, and traditionally treats each field in the textual representation as independently signed, so that the hour/minute/second part is considered positive in this example. If IntervalStyle is set to sql\_standard then a leading sign is considered to apply to all fields (but only if no additional signs appear). Otherwise the traditional PostgreSQL interpretation is used. To avoid ambiguity, it's recommended to attach an explicit sign to each field if any field is negative.

In the verbose input format, and in some fields of the more compact input formats, field values can have fractional parts; for example '1.5 week' or '01:02:03.45'. Such input is converted to the appropriate number of months, days, and seconds for storage. When this would result in a fractional number of months or days, the fraction is added to the lower-order fields using the conversion factors 1 month = 30 days and 1 day = 24 hours. For example, '1.5 month' becomes 1 month and 15 days. Only seconds will ever be shown as fractional on output.

[**Table 8.17**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-INTERVAL-INPUT-EXAMPLES) shows some examples of valid interval input.

**Table 8.17. Interval Input**

| **Example** | **Description** |
| --- | --- |
| 1-2 | SQL standard format: 1 year 2 months |
| 3 4:05:06 | SQL standard format: 3 days 4 hours 5 minutes 6 seconds |
| 1 year 2 months 3 days 4 hours 5 minutes 6 seconds | Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds |
| P1Y2M3DT4H5M6S | ISO 8601 “format with designators”: same meaning as above |
| P0001-02-03T04:05:06 | ISO 8601 “alternative format”: same meaning as above |

Internally interval values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. The months and days fields are integers while the seconds field can store fractions. Because intervals are usually created from constant strings or timestamp subtraction, this storage method works well in most cases, but can cause unexpected results:

SELECT EXTRACT(hours from '80 minutes'::interval);

date\_part

-----------

1

SELECT EXTRACT(days from '80 hours'::interval);

date\_part

-----------

0

Functions justify\_days and justify\_hours are available for adjusting days and hours that overflow their normal ranges.

### 8.5.5. Interval Output

The output format of the interval type can be set to one of the four styles sql\_standard, postgres, postgres\_verbose, or iso\_8601, using the command SET intervalstyle. The default is the postgresformat. [**Table 8.18**](https://www.postgresql.org/docs/10/datatype-datetime.html#INTERVAL-STYLE-OUTPUT-TABLE) shows examples of each output style.

The sql\_standard style produces output that conforms to the SQL standard's specification for interval literal strings, if the interval value meets the standard's restrictions (either year-month only or day-time only, with no mixing of positive and negative components). Otherwise the output looks like a standard year-month literal string followed by a day-time literal string, with explicit signs added to disambiguate mixed-sign intervals.

The output of the postgres style matches the output of PostgreSQL releases prior to 8.4 when the [**DateStyle**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DATESTYLE) parameter was set to ISO.

The output of the postgres\_verbose style matches the output of PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to non-ISO output.

The output of the iso\_8601 style matches the “format with designators” described in section 4.4.3.2 of the ISO 8601 standard.

**Table 8.18. Interval Output Style Examples**

| **Style Specification** | **Year-Month Interval** | **Day-Time Interval** | **Mixed Interval** |
| --- | --- | --- | --- |
| sql\_standard | 1-2 | 3 4:05:06 | -1-2 +3 -4:05:06 |
| postgres | 1 year 2 mons | 3 days 04:05:06 | -1 year -2 mons +3 days -04:05:06 |
| postgres\_verbose | @ 1 year 2 mons | @ 3 days 4 hours 5 mins 6 secs | @ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago |
| iso\_8601 | P1Y2M | P3DT4H5M6S | P-1Y-2M3DT-4H-5M-6S |

**8.6. Boolean Type**

PostgreSQL provides the standard SQL type boolean; see [**Table 8.19**](https://www.postgresql.org/docs/10/datatype-boolean.html#DATATYPE-BOOLEAN-TABLE). The boolean type can have several states: “true”, “false”, and a third state, “unknown”, which is represented by the SQL null value.

**Table 8.19. Boolean Data Type**

| **Name** | **Storage Size** | **Description** |
| --- | --- | --- |
| boolean | 1 byte | state of true or false |

Boolean constants can be represented in SQL queries by the SQL key words TRUE, FALSE, and NULL.

The datatype input function for type boolean accepts these string representations for the “true” state:

|  |
| --- |
| true |
| yes |
| on |
| 1 |

and these representations for the “false” state:

|  |
| --- |
| false |
| no |
| off |
| 0 |

Unique prefixes of these strings are also accepted, for example t or n. Leading or trailing whitespace is ignored, and case does not matter.

The datatype output function for type boolean always emits either t or f, as shown in [**Example 8.2**](https://www.postgresql.org/docs/10/datatype-boolean.html#DATATYPE-BOOLEAN-EXAMPLE).

**Example 8.2. Using the**boolean**Type**

CREATE TABLE test1 (a boolean, b text);

INSERT INTO test1 VALUES (TRUE, 'sic est');

INSERT INTO test1 VALUES (FALSE, 'non est');

SELECT \* FROM test1;

a | b

---+---------

t | sic est

f | non est

SELECT \* FROM test1 WHERE a;

a | b

---+---------

t | sic est

The key words TRUE and FALSE are the preferred (SQL-compliant) method for writing Boolean constants in SQL queries. But you can also use the string representations by following the generic string-literal constant syntax described in [**Section 4.1.2.7**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC), for example 'yes'::boolean.

Note that the parser automatically understands that TRUE and FALSE are of type boolean, but this is not so for NULL because that can have any type. So in some contexts you might have to cast NULL to boolean explicitly, for example NULL::boolean. Conversely, the cast can be omitted from a string-literal Boolean value in contexts where the parser can deduce that the literal must be of type boolean.

## 8.7. Enumerated Types

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the enum types supported in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

### 8.7.1. Declaration of Enumerated Types

Enum types are created using the [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) command, for example:

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

Once created, the enum type can be used in table and function definitions much like any other type:

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

CREATE TABLE person (

name text,

current\_mood mood

);

INSERT INTO person VALUES ('Moe', 'happy');

SELECT \* FROM person WHERE current\_mood = 'happy';

name | current\_mood

------+--------------

Moe | happy

(1 row)

### 8.7.2. Ordering

The ordering of the values in an enum type is the order in which the values were listed when the type was created. All standard comparison operators and related aggregate functions are supported for enums. For example:

INSERT INTO person VALUES ('Larry', 'sad');

INSERT INTO person VALUES ('Curly', 'ok');

SELECT \* FROM person WHERE current\_mood > 'sad';

name | current\_mood

-------+--------------

Moe | happy

Curly | ok

(2 rows)

SELECT \* FROM person WHERE current\_mood > 'sad' ORDER BY current\_mood;

name | current\_mood

-------+--------------

Curly | ok

Moe | happy

(2 rows)

SELECT name

FROM person

WHERE current\_mood = (SELECT MIN(current\_mood) FROM person);

name

-------

Larry

(1 row)

### 8.7.3. Type Safety

Each enumerated data type is separate and cannot be compared with other enumerated types. See this example:

CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');

CREATE TABLE holidays (

num\_weeks integer,

happiness happiness

);

INSERT INTO holidays(num\_weeks,happiness) VALUES (4, 'happy');

INSERT INTO holidays(num\_weeks,happiness) VALUES (6, 'very happy');

INSERT INTO holidays(num\_weeks,happiness) VALUES (8, 'ecstatic');

INSERT INTO holidays(num\_weeks,happiness) VALUES (2, 'sad');

ERROR: invalid input value for enum happiness: "sad"

SELECT person.name, holidays.num\_weeks FROM person, holidays

WHERE person.current\_mood = holidays.happiness;

ERROR: operator does not exist: mood = happiness

If you really need to do something like that, you can either write a custom operator or add explicit casts to your query:

SELECT person.name, holidays.num\_weeks FROM person, holidays

WHERE person.current\_mood::text = holidays.happiness::text;

name | num\_weeks

------+-----------

Moe | 4

(1 row)

### 8.7.4. Implementation Details

Enum labels are case sensitive, so 'happy' is not the same as 'HAPPY'. White space in the labels is significant too.

Although enum types are primarily intended for static sets of values, there is support for adding new values to an existing enum type, and for renaming values (see [**ALTER TYPE**](https://www.postgresql.org/docs/10/sql-altertype.html)). Existing values cannot be removed from an enum type, nor can the sort ordering of such values be changed, short of dropping and re-creating the enum type.

An enum value occupies four bytes on disk. The length of an enum value's textual label is limited by the NAMEDATALEN setting compiled into PostgreSQL; in standard builds this means at most 63 bytes.

The translations from internal enum values to textual labels are kept in the system catalog [pg\_enum](https://www.postgresql.org/docs/10/catalog-pg-enum.html). Querying this catalog directly can be useful.

## 8.8. Geometric Types

Geometric data types represent two-dimensional spatial objects. [**Table 8.20**](https://www.postgresql.org/docs/10/datatype-geometric.html#DATATYPE-GEO-TABLE) shows the geometric types available in PostgreSQL.

**Table 8.20. Geometric Types**

| **Name** | **Storage Size** | **Description** | **Representation** |
| --- | --- | --- | --- |
| point | 16 bytes | Point on a plane | (x,y) |
| line | 32 bytes | Infinite line | {A,B,C} |
| lseg | 32 bytes | Finite line segment | ((x1,y1),(x2,y2)) |
| box | 32 bytes | Rectangular box | ((x1,y1),(x2,y2)) |
| path | 16+16n bytes | Closed path (similar to polygon) | ((x1,y1),...) |
| path | 16+16n bytes | Open path | [(x1,y1),...] |
| polygon | 40+16n bytes | Polygon (similar to closed path) | ((x1,y1),...) |
| circle | 24 bytes | Circle | <(x,y),r> (center point and radius) |

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections. They are explained in [**Section 9.11**](https://www.postgresql.org/docs/10/functions-geometry.html).

### 8.8.1. Points

Points are the fundamental two-dimensional building block for geometric types. Values of type point are specified using either of the following syntaxes:

( ***x*** , ***y*** )

***x*** , ***y***

where ***x*** and ***y*** are the respective coordinates, as floating-point numbers.

Points are output using the first syntax.

### 8.8.2. Lines

Lines are represented by the linear equation ***A***x + ***B***y + ***C*** = 0, where ***A*** and ***B*** are not both zero. Values of type line are input and output in the following form:

{ ***A***, ***B***, ***C*** }

Alternatively, any of the following forms can be used for input:

[ ( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** ) ]

( ( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** ) )

( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** )

***x1*** , ***y1*** , ***x2*** , ***y2***

where (***x1***,***y1***) and (***x2***,***y2***) are two different points on the line.

### 8.8.3. Line Segments

Line segments are represented by pairs of points that are the endpoints of the segment. Values of type lseg are specified using any of the following syntaxes:

[ ( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** ) ]

( ( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** ) )

( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** )

***x1*** , ***y1*** , ***x2*** , ***y2***

where (***x1***,***y1***) and (***x2***,***y2***) are the end points of the line segment.

Line segments are output using the first syntax.

### 8.8.4. Boxes

Boxes are represented by pairs of points that are opposite corners of the box. Values of type box are specified using any of the following syntaxes:

( ( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** ) )

( ***x1*** , ***y1*** ) , ( ***x2*** , ***y2*** )

***x1*** , ***y1*** , ***x2*** , ***y2***

where (***x1***,***y1***) and (***x2***,***y2***) are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but the values will be reordered as needed to store the upper right and lower left corners, in that order.

### 8.8.5. Paths

Paths are represented by lists of connected points. Paths can be open, where the first and last points in the list are considered not connected, or closed, where the first and last points are considered connected.

Values of type path are specified using any of the following syntaxes:

[ ( ***x1*** , ***y1*** ) , ... , ( ***xn*** , ***yn*** ) ]

( ( ***x1*** , ***y1*** ) , ... , ( ***xn*** , ***yn*** ) )

( ***x1*** , ***y1*** ) , ... , ( ***xn*** , ***yn*** )

( ***x1*** , ***y1*** , ... , ***xn*** , ***yn*** )

***x1*** , ***y1*** , ... , ***xn*** , ***yn***

where the points are the end points of the line segments comprising the path. Square brackets ([]) indicate an open path, while parentheses (()) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first or second syntax, as appropriate.

### 8.8.6. Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons are very similar to closed paths, but are stored differently and have their own set of support routines.

Values of type polygon are specified using any of the following syntaxes:

( ( ***x1*** , ***y1*** ) , ... , ( ***xn*** , ***yn*** ) )

( ***x1*** , ***y1*** ) , ... , ( ***xn*** , ***yn*** )

( ***x1*** , ***y1*** , ... , ***xn*** , ***yn*** )

***x1*** , ***y1*** , ... , ***xn*** , ***yn***

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

### 8.8.7. Circles

Circles are represented by a center point and radius. Values of type circle are specified using any of the following syntaxes:

< ( ***x*** , ***y*** ) , ***r*** >

( ( ***x*** , ***y*** ) , ***r*** )

( ***x*** , ***y*** ) , ***r***

***x*** , ***y*** , ***r***

where (***x***,***y***) is the center point and ***r*** is the radius of the circle.

Circles are output using the first syntax.

## 8.9. Network Address Types

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses, as shown in [**Table 8.21**](https://www.postgresql.org/docs/10/datatype-net-types.html#DATATYPE-NET-TYPES-TABLE). It is better to use these types instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions (see [**Section 9.12**](https://www.postgresql.org/docs/10/functions-net.html)).

**Table 8.21. Network Address Types**

| **Name** | **Storage Size** | **Description** |
| --- | --- | --- |
| cidr | 7 or 19 bytes | IPv4 and IPv6 networks |
| inet | 7 or 19 bytes | IPv4 and IPv6 hosts and networks |
| macaddr | 6 bytes | MAC addresses |
| macaddr8 | 8 bytes | MAC addresses (EUI-64 format) |

When sorting inet or cidr data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

### 8.9.1. inet

The inet type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the “netmask”). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address. Note that if you want to accept only networks, you should use the cidr type rather than inet.

The input format for this type is ***address/y*** where ***address*** is an IPv4 or IPv6 address and ***y*** is the number of bits in the netmask. If the ***/y*** portion is missing, the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the ***/y*** portion is suppressed if the netmask specifies a single host.

### 8.9.2. cidr

The cidr type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is ***address/y*** where ***address*** is the network represented as an IPv4 or IPv6 address, and ***y*** is the number of bits in the netmask. If ***y*** is omitted, it is calculated using assumptions from the older classful network numbering system, except it will be at least large enough to include all of the octets written in the input. It is an error to specify a network address that has bits set to the right of the specified netmask.

[**Table 8.22**](https://www.postgresql.org/docs/10/datatype-net-types.html#DATATYPE-NET-CIDR-TABLE) shows some examples.

**Table 8.22.**cidr**Type Input Examples**

| cidr**Input** | cidr**Output** | abbrev(cidr) |
| --- | --- | --- |
| 192.168.100.128/25 | 192.168.100.128/25 | 192.168.100.128/25 |
| 192.168/24 | 192.168.0.0/24 | 192.168.0/24 |
| 192.168/25 | 192.168.0.0/25 | 192.168.0.0/25 |
| 192.168.1 | 192.168.1.0/24 | 192.168.1/24 |
| 192.168 | 192.168.0.0/24 | 192.168.0/24 |
| 128.1 | 128.1.0.0/16 | 128.1/16 |
| 128 | 128.0.0.0/16 | 128.0/16 |
| 128.1.2 | 128.1.2.0/24 | 128.1.2/24 |
| 10.1.2 | 10.1.2.0/24 | 10.1.2/24 |
| 10.1 | 10.1.0.0/16 | 10.1/16 |
| 10 | 10.0.0.0/8 | 10/8 |
| 10.1.2.3/32 | 10.1.2.3/32 | 10.1.2.3/32 |
| 2001:4f8:3:ba::/64 | 2001:4f8:3:ba::/64 | 2001:4f8:3:ba::/64 |
| 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128 | 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128 | 2001:4f8:3:ba:2e0:81ff:fe22:d1f1 |
| ::ffff:1.2.3.0/120 | ::ffff:1.2.3.0/120 | ::ffff:1.2.3/120 |
| ::ffff:1.2.3.0/128 | ::ffff:1.2.3.0/128 | ::ffff:1.2.3.0/128 |

### 8.9.3. inet vs. cidr

The essential difference between inet and cidr data types is that inet accepts values with nonzero bits to the right of the netmask, whereas cidr does not. For example, 192.168.0.1/24 is valid for inet but not for cidr.

Tip

If you do not like the output format for inet or cidr values, try the functions host, text, and abbrev.

### 8.9.4. macaddr

The macaddr type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in the following formats:

|  |
| --- |
| '08:00:2b:01:02:03' |
| '08-00-2b-01-02-03' |
| '08002b:010203' |
| '08002b-010203' |
| '0800.2b01.0203' |
| '0800-2b01-0203' |
| '08002b010203' |

These examples would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the first of the forms shown.

IEEE Std 802-2001 specifies the second shown form (with hyphens) as the canonical form for MAC addresses, and specifies the first form (with colons) as the bit-reversed notation, so that 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. This convention is widely ignored nowadays, and it is relevant only for obsolete network protocols (such as Token Ring). PostgreSQL makes no provisions for bit reversal, and all accepted formats use the canonical LSB order.

The remaining five input formats are not part of any standard.

### 8.9.5. macaddr8

The macaddr8 type stores MAC addresses in EUI-64 format, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). This type can accept both 6 and 8 byte length MAC addresses and stores them in 8 byte length format. MAC addresses given in 6 byte format will be stored in 8 byte length format with the 4th and 5th bytes set to FF and FE, respectively. Note that IPv6 uses a modified EUI-64 format where the 7th bit should be set to one after the conversion from EUI-48. The function macaddr8\_set7bit is provided to make this change. Generally speaking, any input which is comprised of pairs of hex digits (on byte boundaries), optionally separated consistently by one of ':', '-' or '.', is accepted. The number of hex digits must be either 16 (8 bytes) or 12 (6 bytes). Leading and trailing whitespace is ignored. The following are examples of input formats that are accepted:

|  |
| --- |
| '08:00:2b:01:02:03:04:05' |
| '08-00-2b-01-02-03-04-05' |
| '08002b:0102030405' |
| '08002b-0102030405' |
| '0800.2b01.0203.0405' |
| '0800-2b01-0203-0405' |
| '08002b01:02030405' |
| '08002b0102030405' |

These examples would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the first of the forms shown. The last six input formats that are mentioned above are not part of any standard. To convert a traditional 48 bit MAC address in EUI-48 format to modified EUI-64 format to be included as the host portion of an IPv6 address, use macaddr8\_set7bit as shown:

SELECT macaddr8\_set7bit('08:00:2b:01:02:03');

macaddr8\_set7bit

-------------------------

0a:00:2b:ff:fe:01:02:03

(1 row)

## 8.10. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: bit(***n***) and bit varying(***n***), where ***n*** is a positive integer.

bit type data must match the length ***n*** exactly; it is an error to attempt to store shorter or longer bit strings. bit varying data is of variable length up to the maximum length ***n***; longer strings will be rejected. Writing bit without a length is equivalent to bit(1), while bit varying without a length specification means unlimited length.

Note

If one explicitly casts a bit-string value to bit(***n***), it will be truncated or zero-padded on the right to be exactly ***n*** bits, without raising an error. Similarly, if one explicitly casts a bit-string value to bit varying(***n***), it will be truncated on the right if it is more than ***n*** bits.

Refer to [**Section 4.1.2.5**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-BIT-STRINGS) for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see [**Section 9.6**](https://www.postgresql.org/docs/10/functions-bitstring.html).

**Example 8.3. Using the Bit String Types**

CREATE TABLE test (a BIT(3), b BIT VARYING(5));

INSERT INTO test VALUES (B'101', B'00');

INSERT INTO test VALUES (B'10', B'101');

ERROR: bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');

SELECT \* FROM test;

a | b

-----+-----

101 | 00

100 | 101

A bit string value requires 1 byte for each group of 8 bits, plus 5 or 8 bytes overhead depending on the length of the string (but long values may be compressed or moved out-of-line, as explained in [**Section 8.3**](https://www.postgresql.org/docs/10/datatype-character.html) for character strings).

## 8.11. Text Search Types

PostgreSQL provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language documents to locate those that best match a query. The tsvector type represents a document in a form optimized for text search; the tsquery type similarly represents a text query. [**Chapter 12**](https://www.postgresql.org/docs/10/textsearch.html) provides a detailed explanation of this facility, and [**Section 9.13**](https://www.postgresql.org/docs/10/functions-textsearch.html) summarizes the related functions and operators.

### 8.11.1. tsvector

A tsvector value is a sorted list of distinct lexemes, which are words that have been normalized to merge different variants of the same word (see [**Chapter 12**](https://www.postgresql.org/docs/10/textsearch.html) for details). Sorting and duplicate-elimination are done automatically during input, as shown in this example:

SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;

tsvector

----------------------------------------------------

'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'

To represent lexemes containing whitespace or punctuation, surround them with quotes:

SELECT $$the lexeme ' ' contains spaces$$::tsvector;

tsvector

-------------------------------------------

' ' 'contains' 'lexeme' 'spaces' 'the'

(We use dollar-quoted string literals in this example and the next one to avoid the confusion of having to double quote marks within the literals.) Embedded quotes and backslashes must be doubled:

SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;

tsvector

------------------------------------------------

'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'

Optionally, integer positions can be attached to lexemes:

SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;

tsvector

-------------------------------------------------------------------------------

'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4

A position normally indicates the source word's location in the document. Positional information can be used for proximity ranking. Position values can range from 1 to 16383; larger numbers are silently set to 16383. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a weight, which can be A, B, C, or D. D is the default and hence is not shown on output:

SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;

tsvector

----------------------------

'a':1A 'cat':5 'fat':2B,4C

Weights are typically used to reflect document structure, for example by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

It is important to understand that the tsvector type itself does not perform any word normalization; it assumes the words it is given are normalized appropriately for the application. For example,

SELECT 'The Fat Rats'::tsvector;

tsvector

--------------------

'Fat' 'Rats' 'The'

For most English-text-searching applications the above words would be considered non-normalized, but tsvector doesn't care. Raw document text should usually be passed through to\_tsvector to normalize the words appropriately for searching:

SELECT to\_tsvector('english', 'The Fat Rats');

to\_tsvector

-----------------

'fat':2 'rat':3

Again, see [**Chapter 12**](https://www.postgresql.org/docs/10/textsearch.html) for more detail.

### 8.11.2. tsquery

A tsquery value stores lexemes that are to be searched for, and can combine them using the Boolean operators & (AND), | (OR), and ! (NOT), as well as the phrase search operator <-> (FOLLOWED BY). There is also a variant <***N***> of the FOLLOWED BY operator, where ***N*** is an integer constant that specifies the distance between the two lexemes being searched for. <-> is equivalent to <1>.

Parentheses can be used to enforce grouping of these operators. In the absence of parentheses, ! (NOT) binds most tightly, <-> (FOLLOWED BY) next most tightly, then & (AND), with | (OR) binding the least tightly.

Here are some examples:

SELECT 'fat & rat'::tsquery;

tsquery

---------------

'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;

tsquery

---------------------------

'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;

tsquery

------------------------

'fat' & 'rat' & !'cat'

Optionally, lexemes in a tsquery can be labeled with one or more weight letters, which restricts them to match only tsvector lexemes with one of those weights:

SELECT 'fat:ab & cat'::tsquery;

tsquery

------------------

'fat':AB & 'cat'

Also, lexemes in a tsquery can be labeled with \* to specify prefix matching:

SELECT 'super:\*'::tsquery;

tsquery

-----------

'super':\*

This query will match any word in a tsvector that begins with “super”.

Quoting rules for lexemes are the same as described previously for lexemes in tsvector; and, as with tsvector, any required normalization of words must be done before converting to the tsquerytype. The to\_tsquery function is convenient for performing such normalization:

SELECT to\_tsquery('Fat:ab & Cats');

to\_tsquery

------------------

'fat':AB & 'cat'

Note that to\_tsquery will process prefixes in the same way as other words, which means this comparison returns true:

SELECT to\_tsvector( 'postgraduate' ) @@ to\_tsquery( 'postgres:\*' );

?column?

----------

t

because postgres gets stemmed to postgr:

SELECT to\_tsvector( 'postgraduate' ), to\_tsquery( 'postgres:\*' );

to\_tsvector | to\_tsquery

---------------+------------

'postgradu':1 | 'postgr':\*

which will match the stemmed form of postgraduate.

## 8.12. UUID Type

The data type uuid stores Universally Unique Identifiers (UUID) as defined by RFC 4122, ISO/IEC 9834-8:2005, and related standards. (Some systems refer to this data type as a globally unique identifier, or GUID, instead.) This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm. Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

A UUID is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, for a total of 32 digits representing the 128 bits. An example of a UUID in this standard form is:

a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

PostgreSQL also accepts the following alternative forms for input: use of upper-case digits, the standard format surrounded by braces, omitting some or all hyphens, adding a hyphen after any group of four digits. Examples are:

A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11

{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}

a0eebc999c0b4ef8bb6d6bb9bd380a11

a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11

{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}

Output is always in the standard form.

PostgreSQL provides storage and comparison functions for UUIDs, but the core database does not include any function for generating UUIDs, because no single algorithm is well suited for every application. The [**uuid-ossp**](https://www.postgresql.org/docs/10/uuid-ossp.html) module provides functions that implement several standard algorithms. The [**pgcrypto**](https://www.postgresql.org/docs/10/pgcrypto.html) module also provides a generation function for random UUIDs. Alternatively, UUIDs could be generated by client applications or other libraries invoked through a server-side function.

## 8.13. XML Type

The xml data type can be used to store XML data. Its advantage over storing XML data in a text field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it; see [**Section 9.14**](https://www.postgresql.org/docs/10/functions-xml.html). Use of this data type requires the installation to have been built with configure --with-libxml.

The xml type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by reference to the more permissive [**“document node”**](https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode) of the XQuery and XPath data model. Roughly, this means that content fragments can have more than one top-level element or character node. The expression ***xmlvalue*** IS DOCUMENT can be used to evaluate whether a particular xml value is a full document or only a content fragment.

### 8.13.1. Creating XML Values

To produce a value of type xml from character data, use the function xmlparse:

XMLPARSE ( { DOCUMENT | CONTENT } ***value***)

Examples:

XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')

XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')

While this is the only way to convert character strings into XML values according to the SQL standard, the PostgreSQL-specific syntaxes:

xml '<foo>bar</foo>'

'<foo>bar</foo>'::xml

can also be used.

The xml type does not validate input values against a document type declaration (DTD), even when the input value specifies a DTD. There is also currently no built-in support for validating against other XML schema languages such as XML Schema.

The inverse operation, producing a character string value from xml, uses the function xmlserialize:

XMLSERIALIZE ( { DOCUMENT | CONTENT } ***value*** AS ***type*** )

***type*** can be character, character varying, or text (or an alias for one of those). Again, according to the SQL standard, this is the only way to convert between type xml and character types, but PostgreSQL also allows you to simply cast the value.

When a character string value is cast to or from type xml without going through XMLPARSE or XMLSERIALIZE, respectively, the choice of DOCUMENT versus CONTENT is determined by the “XML option” session configuration parameter, which can be set using the standard command:

SET XML OPTION { DOCUMENT | CONTENT };

or the more PostgreSQL-like syntax

SET xmloption TO { DOCUMENT | CONTENT };

The default is CONTENT, so all forms of XML data are allowed.

### 8.13.2. Encoding Handling

Care must be taken when dealing with multiple character encodings on the client, server, and in the XML data passed through them. When using the text mode to pass queries to the server and query results to the client (which is the normal mode), PostgreSQL converts all character data passed between the client and the server and vice versa to the character encoding of the respective end; see [**Section 23.3**](https://www.postgresql.org/docs/10/multibyte.html). This includes string representations of XML values, such as in the above examples. This would ordinarily mean that encoding declarations contained in XML data can become invalid as the character data is converted to other encodings while traveling between client and server, because the embedded encoding declaration is not changed. To cope with this behavior, encoding declarations contained in character strings presented for input to the xml type are ignored, and content is assumed to be in the current server encoding. Consequently, for correct processing, character strings of XML data must be sent from the client in the current client encoding. It is the responsibility of the client to either convert documents to the current client encoding before sending them to the server, or to adjust the client encoding appropriately. On output, values of type xml will not have an encoding declaration, and clients should assume all data is in the current client encoding.

When using binary mode to pass query parameters to the server and query results back to the client, no encoding conversion is performed, so the situation is different. In this case, an encoding declaration in the XML data will be observed, and if it is absent, the data will be assumed to be in UTF-8 (as required by the XML standard; note that PostgreSQL does not support UTF-16). On output, data will have an encoding declaration specifying the client encoding, unless the client encoding is UTF-8, in which case it will be omitted.

Needless to say, processing XML data with PostgreSQL will be less error-prone and more efficient if the XML data encoding, client encoding, and server encoding are the same. Since XML data is internally processed in UTF-8, computations will be most efficient if the server encoding is also UTF-8.

Caution

Some XML-related functions may not work at all on non-ASCII data when the server encoding is not UTF-8. This is known to be an issue for xmltable() and xpath() in particular.

### 8.13.3. Accessing XML Values

The xml data type is unusual in that it does not provide any comparison operators. This is because there is no well-defined and universally useful comparison algorithm for XML data. One consequence of this is that you cannot retrieve rows by comparing an xml column against a search value. XML values should therefore typically be accompanied by a separate key field such as an ID. An alternative solution for comparing XML values is to convert them to character strings first, but note that character string comparison has little to do with a useful XML comparison method.

Since there are no comparison operators for the xml data type, it is not possible to create an index directly on a column of this type. If speedy searches in XML data are desired, possible workarounds include casting the expression to a character string type and indexing that, or indexing an XPath expression. Of course, the actual query would have to be adjusted to search by the indexed expression.

The text-search functionality in PostgreSQL can also be used to speed up full-document searches of XML data. The necessary preprocessing support is, however, not yet available in the PostgreSQL distribution.

## 8.14. JSON Types

JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in [**RFC 7159**](https://tools.ietf.org/html/rfc7159). Such data can also be stored as text, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules. There are also assorted JSON-specific functions and operators available for data stored in these data types; see [**Section 9.15**](https://www.postgresql.org/docs/10/functions-json.html).

There are two JSON data types: json and jsonb. They accept almost identical sets of values as input. The major practical difference is one of efficiency. The json data type stores an exact copy of the input text, which processing functions must reparse on each execution; while jsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. jsonb also supports indexing, which can be a significant advantage.

Because the json type stores an exact copy of the input text, it will preserve semantically-insignificant white space between tokens, as well as the order of keys within JSON objects. Also, if a JSON object within the value contains the same key more than once, all the key/value pairs are kept. (The processing functions consider the last value as the operative one.) By contrast, jsonb does not preserve white space, does not preserve the order of object keys, and does not keep duplicate object keys. If duplicate keys are specified in the input, only the last value is kept.

In general, most applications should prefer to store JSON data as jsonb, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

PostgreSQL allows only one character set encoding per database. It is therefore not possible for the JSON types to conform rigidly to the JSON specification unless the database encoding is UTF8. Attempts to directly include characters that cannot be represented in the database encoding will fail; conversely, characters that can be represented in the database encoding but not in UTF8 will be allowed.

RFC 7159 permits JSON strings to contain Unicode escape sequences denoted by \u***XXXX***. In the input function for the json type, Unicode escapes are allowed regardless of the database encoding, and are checked only for syntactic correctness (that is, that four hex digits follow \u). However, the input function for jsonb is stricter: it disallows Unicode escapes for non-ASCII characters (those above U+007F) unless the database encoding is UTF8. The jsonb type also rejects \u0000 (because that cannot be represented in PostgreSQL's text type), and it insists that any use of Unicode surrogate pairs to designate characters outside the Unicode Basic Multilingual Plane be correct. Valid Unicode escapes are converted to the equivalent ASCII or UTF8 character for storage; this includes folding surrogate pairs into a single character.

Note

Many of the JSON processing functions described in [**Section 9.15**](https://www.postgresql.org/docs/10/functions-json.html) will convert Unicode escapes to regular characters, and will therefore throw the same types of errors just described even if their input is of type json not jsonb. The fact that the json input function does not make these checks may be considered a historical artifact, although it does allow for simple storage (without processing) of JSON Unicode escapes in a non-UTF8 database encoding. In general, it is best to avoid mixing Unicode escapes in JSON with a non-UTF8 database encoding, if possible.

When converting textual JSON input into jsonb, the primitive types described by RFC 7159 are effectively mapped onto native PostgreSQL types, as shown in [**Table 8.23**](https://www.postgresql.org/docs/10/datatype-json.html#JSON-TYPE-MAPPING-TABLE). Therefore, there are some minor additional constraints on what constitutes valid jsonb data that do not apply to the json type, nor to JSON in the abstract, corresponding to limits on what can be represented by the underlying data type. Notably, jsonb will reject numbers that are outside the range of the PostgreSQL numeric data type, while json will not. Such implementation-defined restrictions are permitted by RFC 7159. However, in practice such problems are far more likely to occur in other implementations, as it is common to represent JSON's number primitive type as IEEE 754 double precision floating point (which RFC 7159 explicitly anticipates and allows for). When using JSON as an interchange format with such systems, the danger of losing numeric precision compared to data originally stored by PostgreSQLshould be considered.

Conversely, as noted in the table there are some minor restrictions on the input format of JSON primitive types that do not apply to the corresponding PostgreSQL types.

**Table 8.23. JSON primitive types and corresponding PostgreSQL types**

| **JSON primitive type** | **PostgreSQL type** | **Notes** |
| --- | --- | --- |
| string | text | \u0000 is disallowed, as are non-ASCII Unicode escapes if database encoding is not UTF8 |
| number | numeric | NaN and infinity values are disallowed |
| boolean | boolean | Only lowercase true and false spellings are accepted |
| null | (none) | SQL NULL is a different concept |

### 8.14.1. JSON Input and Output Syntax

The input/output syntax for the JSON data types is as specified in RFC 7159.

The following are all valid json (or jsonb) expressions:

-- Simple scalar/primitive value

-- Primitive values can be numbers, quoted strings, true, false, or null

SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)

SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values

-- Note that object keys must always be quoted strings

SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily

SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;

As previously stated, when a JSON value is input and then printed without any additional processing, json outputs the same text that was input, while jsonb does not preserve semantically-insignificant details such as whitespace. For example, note the differences here:

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;

json

-------------------------------------------------

{"bar": "baz", "balance": 7.77, "active":false}

(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;

jsonb

--------------------------------------------------

{"bar": "baz", "active": false, "balance": 7.77}

(1 row)

One semantically-insignificant detail worth noting is that in jsonb, numbers will be printed according to the behavior of the underlying numeric type. In practice this means that numbers entered with Enotation will be printed without it, for example:

SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;

json | jsonb

-----------------------+-------------------------

{"reading": 1.230e-5} | {"reading": 0.00001230}

(1 row)

However, jsonb will preserve trailing fractional zeroes, as seen in this example, even though those are semantically insignificant for purposes such as equality checks.

### 8.14.2. Designing JSON documents effectively

Representing data as JSON can be considerably more flexible than the traditional relational data model, which is compelling in environments where requirements are fluid. It is quite possible for both approaches to co-exist and complement each other within the same application. However, even for applications where maximal flexibility is desired, it is still recommended that JSON documents have a somewhat fixed structure. The structure is typically unenforced (though enforcing some business rules declaratively is possible), but having a predictable structure makes it easier to write queries that usefully summarize a set of “documents” (datums) in a table.

JSON data is subject to the same concurrency-control considerations as any other data type when stored in a table. Although storing large documents is practicable, keep in mind that any update acquires a row-level lock on the whole row. Consider limiting JSON documents to a manageable size in order to decrease lock contention among updating transactions. Ideally, JSON documents should each represent an atomic datum that business rules dictate cannot reasonably be further subdivided into smaller datums that could be modified independently.

### 8.14.3. jsonb Containment and Existence

Testing containment is an important capability of jsonb. There is no parallel set of facilities for the json type. Containment tests whether one jsonb document has contained within it another one. These examples return true except as noted:

-- Simple scalar/primitive values contain only the identical value:

SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- The array on the right side is contained within the one on the left:

SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Order of array elements is not significant, so this is also true:

SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Duplicate array elements don't matter either:

SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- The object with a single pair on the right side is contained

-- within the object on the left side:

SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.4}'::jsonb;

-- The array on the right side is **not** considered contained within the

-- array on the left, even though a similar array is nested within it:

SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb; -- yields false

-- But with a layer of nesting, it is contained:

SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Similarly, containment is not reported here:

SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- yields false

-- A top-level key and an empty object is contained:

SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;

The general principle is that the contained object must match the containing object as to structure and data contents, possibly after discarding some non-matching array elements or object key/value pairs from the containing object. But remember that the order of array elements is not significant when doing a containment match, and duplicate array elements are effectively considered only once.

As a special exception to the general principle that the structures must match, an array may contain a primitive value:

-- This array contains the primitive string value:

SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- This exception is not reciprocal -- non-containment is reported here:

SELECT '"bar"'::jsonb @> '["bar"]'::jsonb; -- yields false

jsonb also has an existence operator, which is a variation on the theme of containment: it tests whether a string (given as a text value) appears as an object key or array element at the top level of the jsonb value. These examples return true except as noted:

-- String exists as array element:

SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- String exists as object key:

SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Object values are not considered:

SELECT '{"foo": "bar"}'::jsonb ? 'bar'; -- yields false

-- As with containment, existence must match at the top level:

SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- yields false

-- A string is considered to exist if it matches a primitive JSON string:

SELECT '"foo"'::jsonb ? 'foo';

JSON objects are better suited than arrays for testing containment or existence when there are many keys or elements involved, because unlike arrays they are internally optimized for searching, and do not need to be searched linearly.

Tip

Because JSON containment is nested, an appropriate query can skip explicit selection of sub-objects. As an example, suppose that we have a doc column containing objects at the top level, with most objects containing tags fields that contain arrays of sub-objects. This query finds entries in which sub-objects containing both "term":"paris" and "term":"food" appear, while ignoring any such keys outside the tags array:

SELECT doc->'site\_name' FROM websites

WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';

One could accomplish the same thing with, say,

SELECT doc->'site\_name' FROM websites

WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';

but that approach is less flexible, and often less efficient as well.

On the other hand, the JSON existence operator is not nested: it will only look for the specified key or array element at top level of the JSON value.

The various containment and existence operators, along with all other JSON operators and functions are documented in [**Section 9.15**](https://www.postgresql.org/docs/10/functions-json.html).

### 8.14.4. jsonb Indexing

GIN indexes can be used to efficiently search for keys or key/value pairs occurring within a large number of jsonb documents (datums). Two GIN “operator classes” are provided, offering different performance and flexibility trade-offs.

The default GIN operator class for jsonb supports queries with top-level key-exists operators ?, ?& and ?| operators and path/value-exists operator @>. (For details of the semantics that these operators implement, see [**Table 9.44**](https://www.postgresql.org/docs/10/functions-json.html#FUNCTIONS-JSONB-OP-TABLE).) An example of creating an index with this operator class is:

CREATE INDEX idxgin ON api USING GIN (jdoc);

The non-default GIN operator class jsonb\_path\_ops supports indexing the @> operator only. An example of creating an index with this operator class is:

CREATE INDEX idxginp ON api USING GIN (jdoc jsonb\_path\_ops);

Consider the example of a table that stores JSON documents retrieved from a third-party web service, with a documented schema definition. A typical document is:

{

"guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",

"name": "Angela Barton",

"is\_active": true,

"company": "Magnafone",

"address": "178 Howard Place, Gulf, Washington, 702",

"registered": "2009-11-07T08:53:22 +08:00",

"latitude": 19.793713,

"longitude": 86.513373,

"tags": [

"enim",

"aliquip",

"qui"

]

}

We store these documents in a table named api, in a jsonb column named jdoc. If a GIN index is created on this column, queries like the following can make use of the index:

-- Find documents in which the key "company" has value "Magnafone"

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';

However, the index could not be used for queries like the following, because though the operator ? is indexable, it is not applied directly to the indexed column jdoc:

-- Find documents in which the key "tags" contains key or array element "qui"

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';

Still, with appropriate use of expression indexes, the above query can use an index. If querying for particular items within the "tags" key is common, defining an index like this may be worthwhile:

CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));

Now, the WHERE clause jdoc -> 'tags' ? 'qui' will be recognized as an application of the indexable operator ? to the indexed expression jdoc -> 'tags'. (More information on expression indexes can be found in [**Section 11.7**](https://www.postgresql.org/docs/10/indexes-expressional.html).)

Another approach to querying is to exploit containment, for example:

-- Find documents in which the key "tags" contains array element "qui"

SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';

A simple GIN index on the jdoc column can support this query. But note that such an index will store copies of every key and value in the jdoc column, whereas the expression index of the previous example stores only data found under the tags key. While the simple-index approach is far more flexible (since it supports queries about any key), targeted expression indexes are likely to be smaller and faster to search than a simple index.

Although the jsonb\_path\_ops operator class supports only queries with the @> operator, it has notable performance advantages over the default operator class jsonb\_ops. A jsonb\_path\_ops index is usually much smaller than a jsonb\_ops index over the same data, and the specificity of searches is better, particularly when queries contain keys that appear frequently in the data. Therefore search operations typically perform better than with the default operator class.

The technical difference between a jsonb\_ops and a jsonb\_path\_ops GIN index is that the former creates independent index items for each key and value in the data, while the latter creates index items only for each value in the data. [**[6]**](https://www.postgresql.org/docs/10/datatype-json.html#ftn.id-1.5.7.22.17.8.3) Basically, each jsonb\_path\_ops index item is a hash of the value and the key(s) leading to it; for example to index {"foo": {"bar": "baz"}}, a single index item would be created incorporating all three of foo, bar, and baz into the hash value. Thus a containment query looking for this structure would result in an extremely specific index search; but there is no way at all to find out whether foo appears as a key. On the other hand, a jsonb\_ops index would create three index items representing foo, bar, and baz separately; then to do the containment query, it would look for rows containing all three of these items. While GIN indexes can perform such an AND search fairly efficiently, it will still be less specific and slower than the equivalent jsonb\_path\_opssearch, especially if there are a very large number of rows containing any single one of the three index items.

A disadvantage of the jsonb\_path\_ops approach is that it produces no index entries for JSON structures not containing any values, such as {"a": {}}. If a search for documents containing such a structure is requested, it will require a full-index scan, which is quite slow. jsonb\_path\_ops is therefore ill-suited for applications that often perform such searches.

jsonb also supports btree and hash indexes. These are usually useful only if it's important to check equality of complete JSON documents. The btree ordering for jsonb datums is seldom of great interest, but for completeness it is:

***Object*** > ***Array*** > ***Boolean*** > ***Number*** > ***String*** > ***Null***

***Object with n pairs*** > ***object with n - 1 pairs***

***Array with n elements*** > ***array with n - 1 elements***

Objects with equal numbers of pairs are compared in the order:

***key-1***, ***value-1***, ***key-2*** ...

Note that object keys are compared in their storage order; in particular, since shorter keys are stored before longer keys, this can lead to results that might be unintuitive, such as:

{ "aa": 1, "c": 1} > {"b": 1, "d": 1}

Similarly, arrays with equal numbers of elements are compared in the order:

***element-1***, ***element-2*** ...

Primitive JSON values are compared using the same comparison rules as for the underlying PostgreSQL data type. Strings are compared using the default database collation.

[**[6]**](https://www.postgresql.org/docs/10/datatype-json.html#id-1.5.7.22.17.8.3) For this purpose, the term “value” includes array elements, though JSON terminology sometimes considers array elements distinct from values within objects.

## 8.15. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type, enum type, or composite type can be created. Arrays of domains are not yet supported.

### 8.15.1. Declaration of Array Types

To illustrate the use of array types, we create this table:

CREATE TABLE sal\_emp (

name text,

pay\_by\_quarter integer[],

schedule text[][]

);

As shown, an array data type is named by appending square brackets ([]) to the data type name of the array elements. The above command will create a table named sal\_emp with a column of type text (name), a one-dimensional array of type integer (pay\_by\_quarter), which represents the employee's salary by quarter, and a two-dimensional array of text (schedule), which represents the employee's weekly schedule.

The syntax for CREATE TABLE allows the exact size of arrays to be specified, for example:

CREATE TABLE tictactoe (

squares integer[3][3]

);

However, the current implementation ignores any supplied array size limits, i.e., the behavior is the same as for arrays of unspecified length.

The current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring the array size or number of dimensions in CREATE TABLE is simply documentation; it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard by using the keyword ARRAY, can be used for one-dimensional arrays. pay\_by\_quarter could have been defined as:

pay\_by\_quarter integer ARRAY[4],

Or, if no array size is to be specified:

pay\_by\_quarter integer ARRAY,

As before, however, PostgreSQL does not enforce the size restriction in any case.

### 8.15.2. Array Value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You can put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

'{ ***val1*** ***delim*** ***val2*** ***delim*** ... }'

where ***delim*** is the delimiter character for the type, as recorded in its pg\_type entry. Among the standard data types provided in the PostgreSQL distribution, all use a comma (,), except for type boxwhich uses a semicolon (;). Each ***val*** is either a constant of the array element type, or a subarray. An example of an array constant is:

'{{1,2,3},{4,5,6},{7,8,9}}'

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value “NULL”, you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in [**Section 4.1.2.7**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC). The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some INSERT statements:

INSERT INTO sal\_emp

VALUES ('Bill',

'{10000, 10000, 10000, 10000}',

'{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal\_emp

VALUES ('Carol',

'{20000, 25000, 25000, 25000}',

'{{"breakfast", "consulting"}, {"meeting", "lunch"}}');

The result of the previous two inserts looks like this:

SELECT \* FROM sal\_emp;

name | pay\_by\_quarter | schedule

-------+---------------------------+-------------------------------------------

Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}

Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}

(2 rows)

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error, for example:

INSERT INTO sal\_emp

VALUES ('Bill',

'{10000, 10000, 10000, 10000}',

'{{"meeting", "lunch"}, {"meeting"}}');

ERROR: multidimensional arrays must have array expressions with matching dimensions

The ARRAY constructor syntax can also be used:

INSERT INTO sal\_emp

VALUES ('Bill',

ARRAY[10000, 10000, 10000, 10000],

ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal\_emp

VALUES ('Carol',

ARRAY[20000, 25000, 25000, 25000],

ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The ARRAYconstructor syntax is discussed in more detail in [**Section 4.2.12**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ARRAY-CONSTRUCTORS).

### 8.15.3. Accessing Arrays

Now, we can run some queries on the table. First, we show how to access a single element of an array. This query retrieves the names of the employees whose pay changed in the second quarter:

SELECT name FROM sal\_emp WHERE pay\_by\_quarter[1] <> pay\_by\_quarter[2];

name

-------

Carol

(1 row)

The array subscript numbers are written within square brackets. By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of ***n*** elements starts with array[1] and ends with array[***n***].

This query retrieves the third quarter pay of all employees:

SELECT pay\_by\_quarter[3] FROM sal\_emp;

pay\_by\_quarter

----------------

10000

25000

(2 rows)

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing ***lower-bound***:***upper-bound*** for one or more array dimensions. For example, this query retrieves the first item on Bill's schedule for the first two days of the week:

SELECT schedule[1:2][1:1] FROM sal\_emp WHERE name = 'Bill';

schedule

------------------------

{{meeting},{training}}

(1 row)

If any dimension is written as a slice, i.e., contains a colon, then all dimensions are treated as slices. Any dimension that has only a single number (no colon) is treated as being from 1 to the number specified. For example, [2] is treated as [1:2], as in this example:

SELECT schedule[1:2][2] FROM sal\_emp WHERE name = 'Bill';

schedule

-------------------------------------------

{{meeting,lunch},{training,presentation}}

(1 row)

To avoid confusion with the non-slice case, it's best to use slice syntax for all dimensions, e.g., [1:2][1:1], not [2][1:1].

It is possible to omit the ***lower-bound*** and/or ***upper-bound*** of a slice specifier; the missing bound is replaced by the lower or upper limit of the array's subscripts. For example:

SELECT schedule[:2][2:] FROM sal\_emp WHERE name = 'Bill';

schedule

------------------------

{{lunch},{presentation}}

(1 row)

SELECT schedule[:][1:1] FROM sal\_emp WHERE name = 'Bill';

schedule

------------------------

{{meeting},{training}}

(1 row)

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if schedule currently has the dimensions [1:3][1:2] then referencing schedule[3][3] yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.

An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. (This does not match non-slice behavior and is done for historical reasons.) If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region instead of returning null.

The current dimensions of any array value can be retrieved with the array\_dims function:

SELECT array\_dims(schedule) FROM sal\_emp WHERE name = 'Carol';

array\_dims

------------

[1:2][1:2]

(1 row)

array\_dims produces a text result, which is convenient for people to read but perhaps inconvenient for programs. Dimensions can also be retrieved with array\_upper and array\_lower, which return the upper and lower bound of a specified array dimension, respectively:

SELECT array\_upper(schedule, 1) FROM sal\_emp WHERE name = 'Carol';

array\_upper

-------------

2

(1 row)

array\_length will return the length of a specified array dimension:

SELECT array\_length(schedule, 1) FROM sal\_emp WHERE name = 'Carol';

array\_length

--------------

2

(1 row)

cardinality returns the total number of elements in an array across all dimensions. It is effectively the number of rows a call to unnest would yield:

SELECT cardinality(schedule) FROM sal\_emp WHERE name = 'Carol';

cardinality

-------------

4

(1 row)

### 8.15.4. Modifying Arrays

An array value can be replaced completely:

UPDATE sal\_emp SET pay\_by\_quarter = '{25000,25000,27000,27000}'

WHERE name = 'Carol';

or using the ARRAY expression syntax:

UPDATE sal\_emp SET pay\_by\_quarter = ARRAY[25000,25000,27000,27000]

WHERE name = 'Carol';

An array can also be updated at a single element:

UPDATE sal\_emp SET pay\_by\_quarter[4] = 15000

WHERE name = 'Bill';

or updated in a slice:

UPDATE sal\_emp SET pay\_by\_quarter[1:2] = '{27000,27000}'

WHERE name = 'Carol';

The slice syntaxes with omitted ***lower-bound*** and/or ***upper-bound*** can be used too, but only when updating an array value that is not NULL or zero-dimensional (otherwise, there is no existing subscript limit to substitute).

A stored array value can be enlarged by assigning to elements not already present. Any positions between those previously present and the newly assigned elements will be filled with nulls. For example, if array myarray currently has 4 elements, it will have six elements after an update that assigns to myarray[6]; myarray[5] will contain null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to myarray[-2:7] to create an array with subscript values from -2 to 7.

New array values can also be constructed using the concatenation operator, ||:

SELECT ARRAY[1,2] || ARRAY[3,4];

?column?

-----------

{1,2,3,4}

(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];

?column?

---------------------

{{5,6},{1,2},{3,4}}

(1 row)

The concatenation operator allows a single element to be pushed onto the beginning or end of a one-dimensional array. It also accepts two ***N***-dimensional arrays, or an ***N***-dimensional and an ***N+1***-dimensional array.

When a single element is pushed onto either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

SELECT array\_dims(1 || '[0:1]={2,3}'::int[]);

array\_dims

------------

[0:2]

(1 row)

SELECT array\_dims(ARRAY[1,2] || 3);

array\_dims

------------

[1:3]

(1 row)

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand's outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

SELECT array\_dims(ARRAY[1,2] || ARRAY[3,4,5]);

array\_dims

------------

[1:5]

(1 row)

SELECT array\_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);

array\_dims

------------

[1:5][1:2]

(1 row)

When an ***N***-dimensional array is pushed onto the beginning or end of an ***N+1***-dimensional array, the result is analogous to the element-array case above. Each ***N***-dimensional sub-array is essentially an element of the ***N+1***-dimensional array's outer dimension. For example:

SELECT array\_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);

array\_dims

------------

[1:3][1:2]

(1 row)

An array can also be constructed by using the functions array\_prepend, array\_append, or array\_cat. The first two only support one-dimensional arrays, but array\_cat supports multidimensional arrays. Some examples:

SELECT array\_prepend(1, ARRAY[2,3]);

array\_prepend

---------------

{1,2,3}

(1 row)

SELECT array\_append(ARRAY[1,2], 3);

array\_append

--------------

{1,2,3}

(1 row)

SELECT array\_cat(ARRAY[1,2], ARRAY[3,4]);

array\_cat

-----------

{1,2,3,4}

(1 row)

SELECT array\_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);

array\_cat

---------------------

{{1,2},{3,4},{5,6}}

(1 row)

SELECT array\_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);

array\_cat

---------------------

{{5,6},{1,2},{3,4}}

In simple cases, the concatenation operator discussed above is preferred over direct use of these functions. However, because the concatenation operator is overloaded to serve all three cases, there are situations where use of one of the functions is helpful to avoid ambiguity. For example consider:

SELECT ARRAY[1, 2] || '{3, 4}'; -- the untyped literal is taken as an array

?column?

-----------

{1,2,3,4}

SELECT ARRAY[1, 2] || '7'; -- so is this one

ERROR: malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL; -- so is an undecorated NULL

?column?

----------

{1,2}

(1 row)

SELECT array\_append(ARRAY[1, 2], NULL); -- this might have been meant

array\_append

--------------

{1,2,NULL}

In the examples above, the parser sees an integer array on one side of the concatenation operator, and a constant of undetermined type on the other. The heuristic it uses to resolve the constant's type is to assume it's of the same type as the operator's other input — in this case, integer array. So the concatenation operator is presumed to represent array\_cat, not array\_append. When that's the wrong choice, it could be fixed by casting the constant to the array's element type; but explicit use of array\_append might be a preferable solution.

### 8.15.5. Searching in Arrays

To search for a value in an array, each value must be checked. This can be done manually, if you know the size of the array. For example:

SELECT \* FROM sal\_emp WHERE pay\_by\_quarter[1] = 10000 OR

pay\_by\_quarter[2] = 10000 OR

pay\_by\_quarter[3] = 10000 OR

pay\_by\_quarter[4] = 10000;

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. An alternative method is described in [**Section 9.23**](https://www.postgresql.org/docs/10/functions-comparisons.html). The above query could be replaced by:

SELECT \* FROM sal\_emp WHERE 10000 = ANY (pay\_by\_quarter);

In addition, you can find rows where the array has all values equal to 10000 with:

SELECT \* FROM sal\_emp WHERE 10000 = ALL (pay\_by\_quarter);

Alternatively, the generate\_subscripts function can be used. For example:

SELECT \* FROM

(SELECT pay\_by\_quarter,

generate\_subscripts(pay\_by\_quarter, 1) AS s

FROM sal\_emp) AS foo

WHERE pay\_by\_quarter[s] = 10000;

This function is described in [**Table 9.59**](https://www.postgresql.org/docs/10/functions-srf.html#FUNCTIONS-SRF-SUBSCRIPTS).

You can also search an array using the && operator, which checks whether the left operand overlaps with the right operand. For instance:

SELECT \* FROM sal\_emp WHERE pay\_by\_quarter && ARRAY[10000];

This and other array operators are further described in [**Section 9.18**](https://www.postgresql.org/docs/10/functions-array.html). It can be accelerated by an appropriate index, as described in [**Section 11.2**](https://www.postgresql.org/docs/10/indexes-types.html).

You can also search for specific values in an array using the array\_position and array\_positions functions. The former returns the subscript of the first occurrence of a value in an array; the latter returns an array with the subscripts of all occurrences of the value in the array. For example:

SELECT array\_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');

array\_positions

-----------------

2

SELECT array\_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);

array\_positions

-----------------

{1,4,8}

Tip

Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale better for a large number of elements.

### 8.15.6. Array Input and Output Syntax

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces ({ and }) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (,) but can be something else: it is determined by the typdelim setting for the array's element type. Among the standard data types provided in the PostgreSQL distribution, all use a comma, except for type box, which uses a semicolon (;). In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word NULL. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either the presence or absence of quotes.

By default, the lower bound index value of an array's dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets ([]) around each array dimension's lower and upper bounds, with a colon (:) delimiter character in between. The array dimension decoration is followed by an equal sign (=). For example:

SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2

FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

e1 | e2

----+----

1 | 6

(1 row)

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.

If the value written for an element is NULL (in any case variant), the element is taken to be NULL. The presence of any quotes or backslashes disables this and allows the literal string value “NULL” to be entered. Also, for backward compatibility with pre-8.2 versions of PostgreSQL, the [**array\_nulls**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-ARRAY-NULLS) configuration parameter can be turned off to suppress recognition of NULL as a NULL.

As shown previously, when writing an array value you can use double quotes around any individual array element. You must do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or the data type's delimiter character), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word NULL must be quoted, too. To put a double quote or backslash in a quoted array element value, precede it with a backslash. Alternatively, you can avoid quotes and use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You can add whitespace before a left brace or after a right brace. You can also add whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

Tip

The ARRAY constructor syntax (see [**Section 4.2.12**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ARRAY-CONSTRUCTORS)) is often easier to work with than the array-literal syntax when writing array values in SQL commands. In ARRAY, individual element values are written the same way they would be written when not members of an array.

## 8.16. Composite Types

A composite type represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

### 8.16.1. Declaration of Composite Types

Here are two simple examples of defining composite types:

CREATE TYPE complex AS (

r double precision,

i double precision

);

CREATE TYPE inventory\_item AS (

name text,

supplier\_id integer,

price numeric

);

The syntax is comparable to CREATE TABLE, except that only field names and types can be specified; no constraints (such as NOT NULL) can presently be included. Note that the AS keyword is essential; without it, the system will think a different kind of CREATE TYPE command is meant, and you will get odd syntax errors.

Having defined the types, we can use them to create tables:

CREATE TABLE on\_hand (

item inventory\_item,

count integer

);

INSERT INTO on\_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);

or functions:

CREATE FUNCTION price\_extension(inventory\_item, integer) RETURNS numeric

AS 'SELECT $1.price \* $2' LANGUAGE SQL;

SELECT price\_extension(item, 10) FROM on\_hand;

Whenever you create a table, a composite type is also automatically created, with the same name as the table, to represent the table's row type. For example, had we said:

CREATE TABLE inventory\_item (

name text,

supplier\_id integer REFERENCES suppliers,

price numeric CHECK (price > 0)

);

then the same inventory\_item composite type shown above would come into being as a byproduct, and could be used just as above. Note however an important restriction of the current implementation: since no constraints are associated with a composite type, the constraints shown in the table definition do not apply to values of the composite type outside the table. (A partial workaround is to use domain types as members of composite types.)

### 8.16.2. Constructing Composite Values

To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas. You can put double quotes around any field value, and must do so if it contains commas or parentheses. (More details appear [**below**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-IO-SYNTAX).) Thus, the general format of a composite constant is the following:

'( ***val1*** , ***val2*** , ... )'

An example is:

'("fuzzy dice",42,1.99)'

which would be a valid value of the inventory\_item type defined above. To make a field be NULL, write no characters at all in its position in the list. For example, this constant specifies a NULL third field:

'("fuzzy dice",42,)'

If you want an empty string rather than NULL, write double quotes:

'("",42,)'

Here the first field is a non-NULL empty string, the third is NULL.

(These constants are actually only a special case of the generic type constants discussed in [**Section 4.1.2.7**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC). The constant is initially treated as a string and passed to the composite-type input conversion routine. An explicit type specification might be necessary to tell which type to convert the constant to.)

The ROW expression syntax can also be used to construct composite values. In most cases this is considerably simpler to use than the string-literal syntax since you don't have to worry about multiple layers of quoting. We already used this method above:

ROW('fuzzy dice', 42, 1.99)

ROW('', 42, NULL)

The ROW keyword is actually optional as long as you have more than one field in the expression, so these can be simplified to:

('fuzzy dice', 42, 1.99)

('', 42, NULL)

The ROW expression syntax is discussed in more detail in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS).

### 8.16.3. Accessing Composite Types

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name. In fact, it's so much like selecting from a table name that you often have to use parentheses to keep from confusing the parser. For example, you might try to select some subfields from our on\_hand example table with something like:

SELECT item.name FROM on\_hand WHERE item.price > 9.99;

This will not work since the name item is taken to be a table name, not a column name of on\_hand, per SQL syntax rules. You must write it like this:

SELECT (item).name FROM on\_hand WHERE (item).price > 9.99;

or if you need to use the table name as well (for instance in a multitable query), like this:

SELECT (on\_hand.item).name FROM on\_hand WHERE (on\_hand.item).price > 9.99;

Now the parenthesized object is correctly interpreted as a reference to the item column, and then the subfield can be selected from it.

Similar syntactic issues apply whenever you select a field from a composite value. For instance, to select just one field from the result of a function that returns a composite value, you'd need to write something like:

SELECT (my\_func(...)).field FROM ...

Without the extra parentheses, this will generate a syntax error.

The special field name \* means “all fields”, as further explained in [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE).

### 8.16.4. Modifying Composite Types

Here are some examples of the proper syntax for inserting and updating composite columns. First, inserting or updating a whole column:

INSERT INTO mytab (complex\_col) VALUES((1.1,2.2));

UPDATE mytab SET complex\_col = ROW(1.1,2.2) WHERE ...;

The first example omits ROW, the second uses it; we could have done it either way.

We can update an individual subfield of a composite column:

UPDATE mytab SET complex\_col.r = (complex\_col).r + 1 WHERE ...;

Notice here that we don't need to (and indeed cannot) put parentheses around the column name appearing just after SET, but we do need parentheses when referencing the same column in the expression to the right of the equal sign.

And we can specify subfields as targets for INSERT, too:

INSERT INTO mytab (complex\_col.r, complex\_col.i) VALUES(1.1, 2.2);

Had we not supplied values for all the subfields of the column, the remaining subfields would have been filled with null values.

### 8.16.5. Using Composite Types in Queries

There are various special syntax rules and behaviors associated with composite types in queries. These rules provide useful shortcuts, but can be confusing if you don't know the logic behind them.

In PostgreSQL, a reference to a table name (or alias) in a query is effectively a reference to the composite value of the table's current row. For example, if we had a table inventory\_item as shown [**above**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-DECLARING), we could write:

SELECT c FROM inventory\_item c;

This query produces a single composite-valued column, so we might get output like:

c

------------------------

("fuzzy dice",42,1.99)

(1 row)

Note however that simple names are matched to column names before table names, so this example works only because there is no column named c in the query's tables.

The ordinary qualified-column-name syntax ***table\_name***.***column\_name*** can be understood as applying [**field selection**](https://www.postgresql.org/docs/10/sql-expressions.html#FIELD-SELECTION) to the composite value of the table's current row. (For efficiency reasons, it's not actually implemented that way.)

When we write

SELECT c.\* FROM inventory\_item c;

then, according to the SQL standard, we should get the contents of the table expanded into separate columns:

name | supplier\_id | price

------------+-------------+-------

fuzzy dice | 42 | 1.99

(1 row)

as if the query were

SELECT c.name, c.supplier\_id, c.price FROM inventory\_item c;

PostgreSQL will apply this expansion behavior to any composite-valued expression, although as shown [**above**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-ACCESSING), you need to write parentheses around the value that .\* is applied to whenever it's not a simple table name. For example, if myfunc() is a function returning a composite type with columns a, b, and c, then these two queries have the same result:

SELECT (myfunc(x)).\* FROM some\_table;

SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some\_table;

Tip

PostgreSQL handles column expansion by actually transforming the first form into the second. So, in this example, myfunc() would get invoked three times per row with either syntax. If it's an expensive function you may wish to avoid that, which you can do with a query like:

SELECT (m).\* FROM (SELECT myfunc(x) AS m FROM some\_table OFFSET 0) ss;

The OFFSET 0 clause keeps the optimizer from “flattening” the sub-select to arrive at the form with multiple calls of myfunc().

The ***composite\_value***.\* syntax results in column expansion of this kind when it appears at the top level of a [SELECT**output list**](https://www.postgresql.org/docs/10/queries-select-lists.html), a [RETURNING**list**](https://www.postgresql.org/docs/10/dml-returning.html) in INSERT/UPDATE/DELETE, a [VALUES**clause**](https://www.postgresql.org/docs/10/queries-values.html), or a [**row constructor**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). In all other contexts (including when nested inside one of those constructs), attaching .\* to a composite value does not change the value, since it means “all columns” and so the same composite value is produced again. For example, if somefunc() accepts a composite-valued argument, these queries are the same:

SELECT somefunc(c.\*) FROM inventory\_item c;

SELECT somefunc(c) FROM inventory\_item c;

In both cases, the current row of inventory\_item is passed to the function as a single composite-valued argument. Even though .\* does nothing in such cases, using it is good style, since it makes clear that a composite value is intended. In particular, the parser will consider c in c.\* to refer to a table name or alias, not to a column name, so that there is no ambiguity; whereas without .\*, it is not clear whether c means a table name or a column name, and in fact the column-name interpretation will be preferred if there is a column named c.

Another example demonstrating these concepts is that all these queries mean the same thing:

SELECT \* FROM inventory\_item c ORDER BY c;

SELECT \* FROM inventory\_item c ORDER BY c.\*;

SELECT \* FROM inventory\_item c ORDER BY ROW(c.\*);

All of these ORDER BY clauses specify the row's composite value, resulting in sorting the rows according to the rules described in [**Section 9.23.6**](https://www.postgresql.org/docs/10/functions-comparisons.html#COMPOSITE-TYPE-COMPARISON). However, if inventory\_item contained a column named c, the first case would be different from the others, as it would mean to sort by that column only. Given the column names previously shown, these queries are also equivalent to those above:

SELECT \* FROM inventory\_item c ORDER BY ROW(c.name, c.supplier\_id, c.price);

SELECT \* FROM inventory\_item c ORDER BY (c.name, c.supplier\_id, c.price);

(The last case uses a row constructor with the key word ROW omitted.)

Another special syntactical behavior associated with composite values is that we can use functional notation for extracting a field of a composite value. The simple way to explain this is that the notations ***field***(***table***) and ***table***.***field*** are interchangeable. For example, these queries are equivalent:

SELECT c.name FROM inventory\_item c WHERE c.price > 1000;

SELECT name(c) FROM inventory\_item c WHERE price(c) > 1000;

Moreover, if we have a function that accepts a single argument of a composite type, we can call it with either notation. These queries are all equivalent:

SELECT somefunc(c) FROM inventory\_item c;

SELECT somefunc(c.\*) FROM inventory\_item c;

SELECT c.somefunc FROM inventory\_item c;

This equivalence between functional notation and field notation makes it possible to use functions on composite types to implement “computed fields”. An application using the last query above wouldn't need to be directly aware that somefunc isn't a real column of the table.

Tip

Because of this behavior, it's unwise to give a function that takes a single composite-type argument the same name as any of the fields of that composite type. If there is ambiguity, the field-name interpretation will be preferred, so that such a function could not be called without tricks. One way to force the function interpretation is to schema-qualify the function name, that is, write ***schema***.***func***(***compositevalue***).

### 8.16.6. Composite Type Input and Output Syntax

The external text representation of a composite value consists of items that are interpreted according to the I/O conversion rules for the individual field types, plus decoration that indicates the composite structure. The decoration consists of parentheses (( and )) around the whole value, plus commas (,) between adjacent items. Whitespace outside the parentheses is ignored, but within the parentheses it is considered part of the field value, and might or might not be significant depending on the input conversion rules for the field data type. For example, in:

'( 42)'

the whitespace will be ignored if the field type is integer, but not if it is text.

As shown previously, when writing a composite value you can write double quotes around any individual field value. You must do so if the field value would otherwise confuse the composite-value parser. In particular, fields containing parentheses, commas, double quotes, or backslashes must be double-quoted. To put a double quote or backslash in a quoted composite field value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted field value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as composite syntax.

A completely empty field value (no characters at all between the commas or parentheses) represents a NULL. To write a value that is an empty string rather than NULL, write "".

The composite output routine will put double quotes around field values if they are empty strings or contain parentheses, commas, double quotes, backslashes, or white space. (Doing so for white space is not essential, but aids legibility.) Double quotes and backslashes embedded in field values will be doubled.

Note

Remember that what you write in an SQL command will first be interpreted as a string literal, and then as a composite. This doubles the number of backslashes you need (assuming escape string syntax is used). For example, to insert a text field containing a double quote and a backslash in a composite value, you'd need to write:

INSERT ... VALUES ('("\"\\")');

The string-literal processor removes one level of backslashes, so that what arrives at the composite-value parser looks like ("\"\\"). In turn, the string fed to the text data type's input routine becomes "\. (If we were working with a data type whose input routine also treated backslashes specially, bytea for example, we might need as many as eight backslashes in the command to get one backslash into the stored composite field.) Dollar quoting (see [**Section 4.1.2.4**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING)) can be used to avoid the need to double backslashes.

Tip

The ROW constructor syntax is usually easier to work with than the composite-literal syntax when writing composite values in SQL commands. In ROW, individual field values are written the same way they would be written when not members of a composite.

## 8.17. Range Types

Range types are data types representing a range of values of some element type (called the range's subtype). For instance, ranges of timestamp might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is tsrange (short for “timestamp range”), and timestamp is the subtype. The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

### 8.17.1. Built-in Range Types

PostgreSQL comes with the following built-in range types:

* int4range — Range of integer
* int8range — Range of bigint
* numrange — Range of numeric
* tsrange — Range of timestamp without time zone
* tstzrange — Range of timestamp with time zone
* daterange — Range of date

In addition, you can define your own range types; see [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) for more information.

### 8.17.2. Examples

CREATE TABLE reservation (room int, during tsrange);

INSERT INTO reservation VALUES

(1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment

SELECT int4range(10, 20) @> 3;

-- Overlaps

SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound

SELECT upper(int8range(15, 25));

-- Compute the intersection

SELECT int4range(10, 20) \* int4range(15, 25);

-- Is the range empty?

SELECT isempty(numrange(1, 5));

See [**Table 9.50**](https://www.postgresql.org/docs/10/functions-range.html#RANGE-OPERATORS-TABLE) and [**Table 9.51**](https://www.postgresql.org/docs/10/functions-range.html#RANGE-FUNCTIONS-TABLE) for complete lists of operators and functions on range types.

### 8.17.3. Inclusive and Exclusive Bounds

Every non-empty range has two bounds, the lower bound and the upper bound. All points between these values are included in the range. An inclusive bound means that the boundary point itself is included in the range as well, while an exclusive bound means that the boundary point is not included in the range.

In the text form of a range, an inclusive lower bound is represented by “[” while an exclusive lower bound is represented by “(”. Likewise, an inclusive upper bound is represented by “]”, while an exclusive upper bound is represented by “)”. (See [**Section 8.17.5**](https://www.postgresql.org/docs/10/rangetypes.html#RANGETYPES-IO) for more details.)

The functions lower\_inc and upper\_inc test the inclusivity of the lower and upper bounds of a range value, respectively.

### 8.17.4. Infinite (Unbounded) Ranges

The lower bound of a range can be omitted, meaning that all points less than the upper bound are included in the range. Likewise, if the upper bound of the range is omitted, then all points greater than the lower bound are included in the range. If both lower and upper bounds are omitted, all values of the element type are considered to be in the range.

This is equivalent to considering that the lower bound is “minus infinity”, or the upper bound is “plus infinity”, respectively. But note that these infinite values are never values of the range's element type, and can never be part of the range. (So there is no such thing as an inclusive infinite bound — if you try to write one, it will automatically be converted to an exclusive bound.)

Also, some element types have a notion of “infinity”, but that is just another value so far as the range type mechanisms are concerned. For example, in timestamp ranges, [today,] means the same thing as [today,). But [today,infinity] means something different from [today,infinity) — the latter excludes the special timestamp value infinity.

The functions lower\_inf and upper\_inf test for infinite lower and upper bounds of a range, respectively.

### 8.17.5. Range Input/Output

The input for a range value must follow one of the following patterns:

(***lower-bound***,***upper-bound***)

(***lower-bound***,***upper-bound***]

[***lower-bound***,***upper-bound***)

[***lower-bound***,***upper-bound***]

empty

The parentheses or brackets indicate whether the lower and upper bounds are exclusive or inclusive, as described previously. Notice that the final pattern is empty, which represents an empty range (a range that contains no points).

The ***lower-bound*** may be either a string that is valid input for the subtype, or empty to indicate no lower bound. Likewise, ***upper-bound*** may be either a string that is valid input for the subtype, or empty to indicate no upper bound.

Each bound value can be quoted using " (double quote) characters. This is necessary if the bound value contains parentheses, brackets, commas, double quotes, or backslashes, since these characters would otherwise be taken as part of the range syntax. To put a double quote or backslash in a quoted bound value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted bound value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as range syntax. Also, to write a bound value that is an empty string, write "", since writing nothing means an infinite bound.

Whitespace is allowed before and after the range value, but any whitespace between the parentheses or brackets is taken as part of the lower or upper bound value. (Depending on the element type, it might or might not be significant.)

Note

These rules are very similar to those for writing field values in composite-type literals. See [**Section 8.16.6**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-IO-SYNTAX) for additional commentary.

Examples:

-- includes 3, does not include 7, and does include all points in between

SELECT '[3,7)'::int4range;

-- does not include either 3 or 7, but includes all points in between

SELECT '(3,7)'::int4range;

-- includes only the single point 4

SELECT '[4,4]'::int4range;

-- includes no points (and will be normalized to 'empty')

SELECT '[4,4)'::int4range;

### 8.17.6. Constructing Ranges

Each range type has a constructor function with the same name as the range type. Using the constructor function is frequently more convenient than writing a range literal constant, since it avoids the need for extra quoting of the bound values. The constructor function accepts two or three arguments. The two-argument form constructs a range in standard form (lower bound inclusive, upper bound exclusive), while the three-argument form constructs a range with bounds of the form specified by the third argument. The third argument must be one of the strings “()”, “(]”, “[)”, or “[]”. For example:

-- The full form is: lower bound, upper bound, and text argument indicating

-- inclusivity/exclusivity of bounds.

SELECT numrange(1.0, 14.0, '(]');

-- If the third argument is omitted, '[)' is assumed.

SELECT numrange(1.0, 14.0);

-- Although '(]' is specified here, on display the value will be converted to

-- canonical form, since int8range is a discrete range type (see below).

SELECT int8range(1, 14, '(]');

-- Using NULL for either bound causes the range to be unbounded on that side.

SELECT numrange(NULL, 2.2);

### 8.17.7. Discrete Range Types

A discrete range is one whose element type has a well-defined “step”, such as integer or date. In these types two elements can be said to be adjacent, when there are no valid values between them. This contrasts with continuous ranges, where it's always (or almost always) possible to identify other element values between two given values. For example, a range over the numeric type is continuous, as is a range over timestamp. (Even though timestamp has limited precision, and so could theoretically be treated as discrete, it's better to consider it continuous since the step size is normally not of interest.)

Another way to think about a discrete range type is that there is a clear idea of a “next” or “previous” value for each element value. Knowing that, it is possible to convert between inclusive and exclusive representations of a range's bounds, by choosing the next or previous element value instead of the one originally given. For example, in an integer range type [4,8] and (3,9) denote the same set of values; but this would not be so for a range over numeric.

A discrete range type should have a canonicalization function that is aware of the desired step size for the element type. The canonicalization function is charged with converting equivalent values of the range type to have identical representations, in particular consistently inclusive or exclusive bounds. If a canonicalization function is not specified, then ranges with different formatting will always be treated as unequal, even though they might represent the same set of values in reality.

The built-in range types int4range, int8range, and daterange all use a canonical form that includes the lower bound and excludes the upper bound; that is, [). User-defined range types can use other conventions, however.

### 8.17.8. Defining New Range Types

Users can define their own range types. The most common reason to do this is to use ranges over subtypes not provided among the built-in range types. For example, to define a new range type of subtype float8:

CREATE TYPE floatrange AS RANGE (

subtype = float8,

subtype\_diff = float8mi

);

SELECT '[1.234, 5.678]'::floatrange;

Because float8 has no meaningful “step”, we do not define a canonicalization function in this example.

Defining your own range type also allows you to specify a different subtype B-tree operator class or collation to use, so as to change the sort ordering that determines which values fall into a given range.

If the subtype is considered to have discrete rather than continuous values, the CREATE TYPE command should specify a canonical function. The canonicalization function takes an input range value, and must return an equivalent range value that may have different bounds and formatting. The canonical output for two ranges that represent the same set of values, for example the integer ranges [1, 7] and [1, 8), must be identical. It doesn't matter which representation you choose to be the canonical one, so long as two equivalent values with different formattings are always mapped to the same value with the same formatting. In addition to adjusting the inclusive/exclusive bounds format, a canonicalization function might round off boundary values, in case the desired step size is larger than what the subtype is capable of storing. For instance, a range type over timestamp could be defined to have a step size of an hour, in which case the canonicalization function would need to round off bounds that weren't a multiple of an hour, or perhaps throw an error instead.

In addition, any range type that is meant to be used with GiST or SP-GiST indexes should define a subtype difference, or subtype\_diff, function. (The index will still work without subtype\_diff, but it is likely to be considerably less efficient than if a difference function is provided.) The subtype difference function takes two input values of the subtype, and returns their difference (i.e., ***X*** minus ***Y***) represented as a float8 value. In our example above, the function float8mi that underlies the regular float8 minus operator can be used; but for any other subtype, some type conversion would be necessary. Some creative thought about how to represent differences as numbers might be needed, too. To the greatest extent possible, the subtype\_diff function should agree with the sort ordering implied by the selected operator class and collation; that is, its result should be positive whenever its first argument is greater than its second according to the sort ordering.

A less-oversimplified example of a subtype\_diff function is:

CREATE FUNCTION time\_subtype\_diff(x time, y time) RETURNS float8 AS

'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (

subtype = time,

subtype\_diff = time\_subtype\_diff

);

SELECT '[11:10, 23:00]'::timerange;

See [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) for more information about creating range types.

### 8.17.9. Indexing

GiST and SP-GiST indexes can be created for table columns of range types. For instance, to create a GiST index:

CREATE INDEX reservation\_idx ON reservation USING GIST (during);

A GiST or SP-GiST index can accelerate queries involving these range operators: =, &&, <@, @>, <<, >>, -|-, &<, and &> (see [**Table 9.50**](https://www.postgresql.org/docs/10/functions-range.html#RANGE-OPERATORS-TABLE) for more information).

In addition, B-tree and hash indexes can be created for table columns of range types. For these index types, basically the only useful range operation is equality. There is a B-tree sort ordering defined for range values, with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. Range types' B-tree and hash support is primarily meant to allow sorting and hashing internally in queries, rather than creation of actual indexes.

### 8.17.10. Constraints on Ranges

While UNIQUE is a natural constraint for scalar values, it is usually unsuitable for range types. Instead, an exclusion constraint is often more appropriate (see [**CREATE TABLE ... CONSTRAINT ... EXCLUDE**](https://www.postgresql.org/docs/10/sql-createtable.html#SQL-CREATETABLE-EXCLUDE)). Exclusion constraints allow the specification of constraints such as “non-overlapping” on a range type. For example:

CREATE TABLE reservation (

during tsrange,

EXCLUDE USING GIST (during WITH &&)

);

That constraint will prevent any overlapping values from existing in the table at the same time:

INSERT INTO reservation VALUES

('[2010-01-01 11:30, 2010-01-01 15:00)');

INSERT 0 1

INSERT INTO reservation VALUES

('[2010-01-01 14:45, 2010-01-01 15:45)');

ERROR: conflicting key value violates exclusion constraint "reservation\_during\_excl"

DETAIL: Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts

with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).

You can use the [btree\_gist](https://www.postgresql.org/docs/10/btree-gist.html) extension to define exclusion constraints on plain scalar data types, which can then be combined with range exclusions for maximum flexibility. For example, after btree\_gist is installed, the following constraint will reject overlapping ranges only if the meeting room numbers are equal:

CREATE EXTENSION btree\_gist;

CREATE TABLE room\_reservation (

room text,

during tsrange,

EXCLUDE USING GIST (room WITH =, during WITH &&)

);

INSERT INTO room\_reservation VALUES

('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');

INSERT 0 1

INSERT INTO room\_reservation VALUES

('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');

ERROR: conflicting key value violates exclusion constraint "room\_reservation\_room\_during\_excl"

DETAIL: Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts

with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room\_reservation VALUES

('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');

INSERT 0 1

## 8.18. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless WITH OIDS is specified when the table is created, or the [**default\_with\_oids**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-DEFAULT-WITH-OIDS) configuration variable is enabled. Type oid represents an object identifier. There are also several alias types for oid: regproc, regprocedure, regoper, regoperator, regclass, regtype, regrole, regnamespace, regconfig, and regdictionary. [**Table 8.24**](https://www.postgresql.org/docs/10/datatype-oid.html#DATATYPE-OID-TABLE) shows an overview.

The oid type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table's OID column as a primary key is discouraged. OIDs are best used only for references to system tables.

The oid type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type oid would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the pg\_attribute rows related to a table mytable, one could write:

SELECT \* FROM pg\_attribute WHERE attrelid = 'mytable'::regclass;

rather than:

SELECT \* FROM pg\_attribute

WHERE attrelid = (SELECT oid FROM pg\_class WHERE relname = 'mytable');

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named mytable in different schemas. The regclass input converter handles the table lookup according to the schema path setting, and so it does the “right thing” automatically. Similarly, casting a table's OID to regclass is handy for symbolic display of a numeric OID.

**Table 8.24. Object Identifier Types**

| **Name** | **References** | **Description** | **Value Example** |
| --- | --- | --- | --- |
| oid | any | numeric object identifier | 564182 |
| regproc | pg\_proc | function name | sum |
| regprocedure | pg\_proc | function with argument types | sum(int4) |
| regoper | pg\_operator | operator name | + |
| regoperator | pg\_operator | operator with argument types | \*(integer,integer) or -(NONE,integer) |
| regclass | pg\_class | relation name | pg\_type |
| regtype | pg\_type | data type name | integer |
| regrole | pg\_authid | role name | smithee |
| regnamespace | pg\_namespace | namespace name | pg\_catalog |
| regconfig | pg\_ts\_config | text search configuration | english |
| regdictionary | pg\_ts\_dict | text search dictionary | simple |

All of the OID alias types for objects grouped by namespace accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified. The regproc and regoper alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses regprocedure or regoperatorare more appropriate. For regoperator, unary operators are identified by writing NONE for the unused operand.

An additional property of most of the OID alias types is the creation of dependencies. If a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object. For example, if a column has a default expression nextval('my\_seq'::regclass), PostgreSQL understands that the default expression depends on the sequence my\_seq; the system will not let the sequence be dropped without first removing the default expression. regrole is the only exception for the property. Constants of this type are not allowed in such expressions.

Note

The OID alias types do not completely follow transaction isolation rules. The planner also treats them as simple constants, which may result in sub-optimal planning.

Another identifier type used by the system is xid, or transaction (abbreviated xact) identifier. This is the data type of the system columns xmin and xmax. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is cid, or command identifier. This is the data type of the system columns cmin and cmax. Command identifiers are also 32-bit quantities.

A final identifier type used by the system is tid, or tuple identifier (row identifier). This is the data type of the system column ctid. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

(The system columns are further explained in [**Section 5.4**](https://www.postgresql.org/docs/10/ddl-system-columns.html).)

## 8.19. pg\_lsn Type

The pg\_lsn data type can be used to store LSN (Log Sequence Number) data which is a pointer to a location in the WAL. This type is a representation of XLogRecPtr and an internal system type of PostgreSQL.

Internally, an LSN is a 64-bit integer, representing a byte position in the write-ahead log stream. It is printed as two hexadecimal numbers of up to 8 digits each, separated by a slash; for example, 16/B374D848. The pg\_lsn type supports the standard comparison operators, like = and >. Two LSNs can be subtracted using the - operator; the result is the number of bytes separating those write-ahead log locations.

**8.20. Pseudo-Types**

The PostgreSQL type system contains a number of special-purpose entries that are collectively called *pseudo-types*. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQLdata type. [**Table 8.25**](https://www.postgresql.org/docs/10/datatype-pseudo.html#DATATYPE-PSEUDOTYPES-TABLE) lists the existing pseudo-types.

**Table 8.25. Pseudo-Types**

| **Name** | **Description** |
| --- | --- |
| any | Indicates that a function accepts any input data type. |
| anyelement | Indicates that a function accepts any data type (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC)). |
| anyarray | Indicates that a function accepts any array data type (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC)). |
| anynonarray | Indicates that a function accepts any non-array data type (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC)). |
| anyenum | Indicates that a function accepts any enum data type (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC) and [**Section 8.7**](https://www.postgresql.org/docs/10/datatype-enum.html)). |
| anyrange | Indicates that a function accepts any range data type (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC) and [**Section 8.17**](https://www.postgresql.org/docs/10/rangetypes.html)). |
| cstring | Indicates that a function accepts or returns a null-terminated C string. |
| internal | Indicates that a function accepts or returns a server-internal data type. |
| language\_handler | A procedural language call handler is declared to return language\_handler. |
| fdw\_handler | A foreign-data wrapper handler is declared to return fdw\_handler. |
| index\_am\_handler | An index access method handler is declared to return index\_am\_handler. |
| tsm\_handler | A tablesample method handler is declared to return tsm\_handler. |
| record | Identifies a function taking or returning an unspecified row type. |
| trigger | A trigger function is declared to return trigger. |
| event\_trigger | An event trigger function is declared to return event\_trigger. |
| pg\_ddl\_command | Identifies a representation of DDL commands that is available to event triggers. |
| void | Indicates that a function returns no value. |
| unknown | Identifies a not-yet-resolved type, e.g. of an undecorated string literal. |
| opaque | An obsolete type name that formerly served many of the above purposes. |

Functions coded in C (whether built-in or dynamically loaded) can be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present most procedural languages forbid use of a pseudo-type as an argument type, and allow only void and record as a result type (plus trigger or event\_trigger when the function is used as a trigger or event trigger). Some also support polymorphic functions using the types anyelement, anyarray, anynonarray, anyenum, and anyrange.

The internal pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct invocation in an SQL query. If a function has at least one internal-type argument then it cannot be called from SQL. To preserve the type safety of this restriction it is important to follow this coding rule: do not create any function that is declared to return internal unless it has at least one internal argument.

## Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in [**Part V**](https://www.postgresql.org/docs/10/server-programming.html). The psql commands \dfand \do can be used to list all available functions and operators, respectively.

If you are concerned about portability then note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations. This chapter is also not exhaustive; additional functions appear in relevant sections of the manual.

**9.1. Logical Operators**

The usual logical operators are available:

|  |
| --- |
| AND |
| OR |
| NOT |

SQL uses a three-valued logic system with true, false, and null, which represents “unknown”. Observe the following truth tables:

| ***a*** | ***b*** | ***a* AND *b*** | ***a* OR *b*** |
| --- | --- | --- | --- |
| TRUE | TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | NULL | NULL | TRUE |
| FALSE | FALSE | FALSE | FALSE |
| FALSE | NULL | FALSE | NULL |
| NULL | NULL | NULL | NULL |

| ***a*** | **NOT *a*** |
| --- | --- |
| TRUE | FALSE |
| FALSE | TRUE |
| NULL | NULL |

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result. But see [**Section 4.2.14**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-EXPRESS-EVAL) for more information about the order of evaluation of subexpressions.

## 9.2. Comparison Functions and Operators

The usual comparison operators are available, as shown in [**Table 9.1**](https://www.postgresql.org/docs/10/functions-comparison.html#FUNCTIONS-COMPARISON-OP-TABLE).

**Table 9.1. Comparison Operators**

| **Operator** | **Description** |
| --- | --- |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| = | equal |
| <> or != | not equal |

Note

The != operator is converted to <> in the parser stage. It is not possible to implement !=and <> operators that do different things.

Comparison operators are available for all relevant data types. All comparison operators are binary operators that return values of type boolean; expressions like 1 < 2 < 3 are not valid (because there is no < operator to compare a Boolean value with 3).

There are also some comparison predicates, as shown in [**Table 9.2**](https://www.postgresql.org/docs/10/functions-comparison.html#FUNCTIONS-COMPARISON-PRED-TABLE). These behave much like operators, but have special syntax mandated by the SQL standard.

**Table 9.2. Comparison Predicates**

| **Predicate** | **Description** |
| --- | --- |
| ***a*** BETWEEN ***x*** AND ***y*** | between |
| ***a*** NOT BETWEEN ***x*** AND ***y*** | not between |
| ***a*** BETWEEN SYMMETRIC ***x*** AND ***y*** | between, after sorting the comparison values |
| ***a*** NOT BETWEEN SYMMETRIC ***x*** AND ***y*** | not between, after sorting the comparison values |
| ***a*** IS DISTINCT FROM ***b*** | not equal, treating null like an ordinary value |
| ***a*** IS NOT DISTINCT FROM ***b*** | equal, treating null like an ordinary value |
| ***expression*** IS NULL | is null |
| ***expression*** IS NOT NULL | is not null |
| ***expression*** ISNULL | is null (nonstandard syntax) |
| ***expression*** NOTNULL | is not null (nonstandard syntax) |
| ***boolean\_expression*** IS TRUE | is true |
| ***boolean\_expression*** IS NOT TRUE | is false or unknown |
| ***boolean\_expression*** IS FALSE | is false |
| ***boolean\_expression*** IS NOT FALSE | is true or unknown |
| ***boolean\_expression*** IS UNKNOWN | is unknown |
| ***boolean\_expression*** IS NOT UNKNOWN | is true or false |

The BETWEEN predicate simplifies range tests:

***a*** BETWEEN ***x*** AND ***y***

is equivalent to

***a*** >= ***x*** AND ***a*** <= ***y***

Notice that BETWEEN treats the endpoint values as included in the range. NOT BETWEEN does the opposite comparison:

***a*** NOT BETWEEN ***x*** AND ***y***

is equivalent to

***a*** < ***x*** OR ***a*** > ***y***

BETWEEN SYMMETRIC is like BETWEEN except there is no requirement that the argument to the left of AND be less than or equal to the argument on the right. If it is not, those two arguments are automatically swapped, so that a nonempty range is always implied.

Ordinary comparison operators yield null (signifying “unknown”), not true or false, when either input is null. For example, 7 = NULL yields null, as does 7 <> NULL. When this behavior is not suitable, use the IS [ NOT ] DISTINCT FROM predicates:

***a*** IS DISTINCT FROM ***b***

***a*** IS NOT DISTINCT FROM ***b***

For non-null inputs, IS DISTINCT FROM is the same as the <> operator. However, if both inputs are null it returns false, and if only one input is null it returns true. Similarly, IS NOT DISTINCT FROM is identical to = for non-null inputs, but it returns true when both inputs are null, and false when only one input is null. Thus, these predicates effectively act as though null were a normal data value, rather than “unknown”.

To check whether a value is or is not null, use the predicates:

***expression*** IS NULL

***expression*** IS NOT NULL

or the equivalent, but nonstandard, predicates:

***expression*** ISNULL

***expression*** NOTNULL

Do not write ***expression*** = NULL because NULL is not “equal to” NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.)

Tip

Some applications might expect that ***expression*** = NULL returns true if ***expression***evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard. However, if that cannot be done the [**transform\_null\_equals**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-TRANSFORM-NULL-EQUALS) configuration variable is available. If it is enabled, PostgreSQL will convert x = NULL clauses to x IS NULL.

If the ***expression*** is row-valued, then IS NULL is true when the row expression itself is null or when all the row's fields are null, while IS NOT NULL is true when the row expression itself is non-null and all the row's fields are non-null. Because of this behavior, IS NULL and IS NOT NULL do not always return inverse results for row-valued expressions; in particular, a row-valued expression that contains both null and non-null fields will return false for both tests. In some cases, it may be preferable to write ***row*** IS DISTINCT FROM NULL or ***row*** IS NOT DISTINCT FROM NULL, which will simply check whether the overall row value is null without any additional tests on the row fields.

Boolean values can also be tested using the predicates

***boolean\_expression*** IS TRUE

***boolean\_expression*** IS NOT TRUE

***boolean\_expression*** IS FALSE

***boolean\_expression*** IS NOT FALSE

***boolean\_expression*** IS UNKNOWN

***boolean\_expression*** IS NOT UNKNOWN

These will always return true or false, never a null value, even when the operand is null. A null input is treated as the logical value “unknown”. Notice that IS UNKNOWN and IS NOT UNKNOWN are effectively the same as IS NULL and IS NOT NULL, respectively, except that the input expression must be of Boolean type.

Some comparison-related functions are also available, as shown in [**Table 9.3**](https://www.postgresql.org/docs/10/functions-comparison.html#FUNCTIONS-COMPARISON-FUNC-TABLE).

**Table 9.3. Comparison Functions**

| **Function** | **Description** | **Example** | **Example Result** |
| --- | --- | --- | --- |
| num\_nonnulls(VARIADIC "any") | returns the number of non-null arguments | num\_nonnulls(1, NULL, 2) | 2 |
| num\_nulls(VARIADIC "any") | returns the number of null arguments | num\_nulls(1, NULL, 2) | 1 |

## 9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without standard mathematical conventions (e.g., date/time types) we describe the actual behavior in subsequent sections.

[**Table 9.4**](https://www.postgresql.org/docs/10/functions-math.html#FUNCTIONS-MATH-OP-TABLE) shows the available mathematical operators.

**Table 9.4. Mathematical Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | addition | 2 + 3 | 5 |
| - | subtraction | 2 - 3 | -1 |
| \* | multiplication | 2 \* 3 | 6 |
| / | division (integer division truncates the result) | 4 / 2 | 2 |
| % | modulo (remainder) | 5 % 4 | 1 |
| ^ | exponentiation (associates left to right) | 2.0 ^ 3.0 | 8 |
| |/ | square root | |/ 25.0 | 5 |
| ||/ | cube root | ||/ 27.0 | 3 |
| ! | factorial | 5 ! | 120 |
| !! | factorial (prefix operator) | !! 5 | 120 |
| @ | absolute value | @ -5.0 | 5 |
| & | bitwise AND | 91 & 15 | 11 |
| | | bitwise OR | 32 | 3 | 35 |
| # | bitwise XOR | 17 # 5 | 20 |
| ~ | bitwise NOT | ~1 | -2 |
| << | bitwise shift left | 1 << 4 | 16 |
| >> | bitwise shift right | 8 >> 2 | 2 |

The bitwise operators work only on integral data types, whereas the others are available for all numeric data types. The bitwise operators are also available for the bit string types bit and bit varying, as shown in [**Table 9.13**](https://www.postgresql.org/docs/10/functions-bitstring.html#FUNCTIONS-BIT-STRING-OP-TABLE).

[**Table 9.5**](https://www.postgresql.org/docs/10/functions-math.html#FUNCTIONS-MATH-FUNC-TABLE) shows the available mathematical functions. In the table, dp indicates double precision. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with double precision data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases can therefore vary depending on the host system.

**Table 9.5. Mathematical Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| abs(***x***) | (same as input) | absolute value | abs(-17.4) | 17.4 |
| cbrt(dp) | dp | cube root | cbrt(27.0) | 3 |
| ceil(dp or numeric) | (same as input) | nearest integer greater than or equal to argument | ceil(-42.8) | -42 |
| ceiling(dp or numeric) | (same as input) | nearest integer greater than or equal to argument (same as ceil) | ceiling(-95.3) | -95 |
| degrees(dp) | dp | radians to degrees | degrees(0.5) | 28.6478897565412 |
| div(*y* numeric, *x* numeric) | numeric | integer quotient of *y*/*x* | div(9,4) | 2 |
| exp(dp or numeric) | (same as input) | exponential | exp(1.0) | 2.71828182845905 |
| floor(dp or numeric) | (same as input) | nearest integer less than or equal to argument | floor(-42.8) | -43 |
| ln(dp or numeric) | (same as input) | natural logarithm | ln(2.0) | 0.693147180559945 |
| log(dp or numeric) | (same as input) | base 10 logarithm | log(100.0) | 2 |
| log(*b* numeric, *x* numeric) | numeric | logarithm to base *b* | log(2.0, 64.0) | 6.0000000000 |
| mod(*y*, *x*) | (same as argument types) | remainder of *y*/*x* | mod(9,4) | 1 |
| pi() | dp | “π” constant | pi() | 3.14159265358979 |
| power(*a* dp, *b* dp) | dp | *a* raised to the power of *b* | power(9.0, 3.0) | 729 |
| power(*a* numeric, *b*numeric) | numeric | *a* raised to the power of *b* | power(9.0, 3.0) | 729 |
| radians(dp) | dp | degrees to radians | radians(45.0) | 0.785398163397448 |
| round(dp or numeric) | (same as input) | round to nearest integer | round(42.4) | 42 |
| round(*v* numeric, *s* int) | numeric | round to *s* decimal places | round(42.4382, 2) | 42.44 |
| scale(numeric) | integer | scale of the argument (the number of decimal digits in the fractional part) | scale(8.41) | 2 |
| sign(dp or numeric) | (same as input) | sign of the argument (-1, 0, +1) | sign(-8.4) | -1 |
| sqrt(dp or numeric) | (same as input) | square root | sqrt(2.0) | 1.4142135623731 |
| trunc(dp or numeric) | (same as input) | truncate toward zero | trunc(42.8) | 42 |
| trunc(*v* numeric, *s* int) | numeric | truncate to *s* decimal places | trunc(42.4382, 2) | 42.43 |
| width\_bucket(*operand* dp,*b1* dp, *b2* dp, *count* int) | int | return the bucket number to which *operand* would be assigned in a histogram having *count* equal-width buckets spanning the range *b1* to *b2*; returns 0 or *count*+1 for an input outside the range | width\_bucket(5.35, 0.024, 10.06, 5) | 3 |
| width\_bucket(*operand*numeric, *b1* numeric, *b2*numeric, *count* int) | int | return the bucket number to which *operand* would be assigned in a histogram having *count* equal-width buckets spanning the range *b1* to *b2*; returns 0 or *count*+1 for an input outside the range | width\_bucket(5.35, 0.024, 10.06, 5) | 3 |
| width\_bucket(*operand*anyelement, *thresholds*anyarray) | int | return the bucket number to which *operand* would be assigned given an array listing the lower bounds of the buckets; returns 0 for an input less than the first lower bound; the *thresholds* array must be sorted, smallest first, or unexpected results will be obtained | width\_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[]) | 2 |

[**Table 9.6**](https://www.postgresql.org/docs/10/functions-math.html#FUNCTIONS-MATH-RANDOM-TABLE) shows functions for generating random numbers.

**Table 9.6. Random Functions**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| random() | dp | random value in the range 0.0 <= x < 1.0 |
| setseed(dp) | void | set seed for subsequent random() calls (value between -1.0 and 1.0, inclusive) |

The characteristics of the values returned by random() depend on the system implementation. It is not suitable for cryptographic applications; see [**pgcrypto**](https://www.postgresql.org/docs/10/pgcrypto.html) module for an alternative.

Finally, [**Table 9.7**](https://www.postgresql.org/docs/10/functions-math.html#FUNCTIONS-MATH-TRIG-TABLE) shows the available trigonometric functions. All trigonometric functions take arguments and return values of type double precision. Each of the trigonometric functions comes in two variants, one that measures angles in radians and one that measures angles in degrees.

**Table 9.7. Trigonometric Functions**

| **Function (radians)** | **Function (degrees)** | **Description** |
| --- | --- | --- |
| acos(***x***) | acosd(***x***) | inverse cosine |
| asin(***x***) | asind(***x***) | inverse sine |
| atan(***x***) | atand(***x***) | inverse tangent |
| atan2(***y***, ***x***) | atan2d(***y***, ***x***) | inverse tangent of ***y***/***x*** |
| cos(***x***) | cosd(***x***) | cosine |
| cot(***x***) | cotd(***x***) | cotangent |
| sin(***x***) | sind(***x***) | sine |
| tan(***x***) | tand(***x***) | tangent |

Note

Another way to work with angles measured in degrees is to use the unit transformation functions radians() and degrees() shown earlier. However, using the degree-based trigonometric functions is preferred, as that way avoids round-off error for special cases such as sind(30).

## 9.4. String Functions and Operators

[**9.4.1.**format](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-FORMAT)

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types character, character varying, and text. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic space-padding when using the character type. Some functions also exist natively for the bit-string types.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in [**Table 9.8**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-SQL). PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see [**Table 9.9**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-OTHER)).

Note

Before PostgreSQL 8.3, these functions would silently accept values of several non-string data types as well, due to the presence of implicit coercions from those data types to text. Those coercions have been removed because they frequently caused surprising behaviors. However, the string concatenation operator (||) still accepts non-string input, so long as at least one input is of a string type, as shown in [**Table 9.8**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-SQL). For other cases, insert an explicit coercion to text if you need to duplicate the previous behavior.

**Table 9.8. SQL String Functions and Operators**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| *string* || *string* | text | String concatenation | 'Post' || 'greSQL' | PostgreSQL |
| *string* || *non-string* or *non-string* ||*string* | text | String concatenation with one non-string input | 'Value: ' || 42 | Value: 42 |
| bit\_length(*string*) | int | Number of bits in string | bit\_length('jose') | 32 |
| char\_length(*string*) or character\_length(*string*) | int | Number of characters in string | char\_length('jose') | 4 |
| lower(*string*) | text | Convert string to lower case | lower('TOM') | tom |
| octet\_length(*string*) | int | Number of bytes in string | octet\_length('jose') | 4 |
| overlay(*string* placing *string* from int[for int]) | text | Replace substring | overlay('Txxxxas' placing 'hom' from 2 for 4) | Thomas |
| position(*substring* in *string*) | int | Location of specified substring | position('om' in 'Thomas') | 3 |
| substring(*string* [from int] [for int]) | text | Extract substring | substring('Thomas' from 2 for 3) | hom |
| substring(*string* from ***pattern***) | text | Extract substring matching POSIX regular expression. See [**Section 9.7**](https://www.postgresql.org/docs/10/functions-matching.html) for more information on pattern matching. | substring('Thomas' from '...$') | mas |
| substring(*string* from ***pattern*** for***escape***) | text | Extract substring matching SQL regular expression. See [**Section 9.7**](https://www.postgresql.org/docs/10/functions-matching.html) for more information on pattern matching. | substring('Thomas' from '%#"o\_a#"\_' for '#') | oma |
| trim([leading | trailing | both] [*characters*] from *string*) | text | Remove the longest string containing only characters from *characters* (a space by default) from the start, end, or both ends (both is the default) of *string* | trim(both 'xyz' from 'yxTomxx') | Tom |
| trim([leading | trailing | both] [from]*string* [, *characters*] ) | text | Non-standard syntax for trim() | trim(both from 'yxTomxx', 'xyz') | Tom |
| upper(*string*) | text | Convert string to upper case | upper('tom') | TOM |

Additional string manipulation functions are available and are listed in [**Table 9.9**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-OTHER). Some of them are used internally to implement the SQL-standard string functions listed in [**Table 9.8**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-SQL).

**Table 9.9. Other String Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| ascii(*string*) | int | ASCII code of the first character of the argument. For UTF8 returns the Unicode code point of the character. For other multibyte encodings, the argument must be an ASCIIcharacter. | ascii('x') | 120 |
| btrim(*string* text [, *characters*text]) | text | Remove the longest string consisting only of characters in *characters* (a space by default) from the start and end of *string* | btrim('xyxtrimyyx', 'xyz') | trim |
| chr(int) | text | Character with the given code. For UTF8 the argument is treated as a Unicode code point. For other multibyte encodings the argument must designate an ASCII character. The NULL (0) character is not allowed because text data types cannot store such bytes. | chr(65) | A |
| concat(*str* "any" [, *str* "any" [, ...] ]) | text | Concatenate the text representations of all the arguments. NULL arguments are ignored. | concat('abcde', 2, NULL, 22) | abcde222 |
| concat\_ws(*sep* text, *str* "any" [,*str* "any" [, ...] ]) | text | Concatenate all but the first argument with separators. The first argument is used as the separator string. NULL arguments are ignored. | concat\_ws(',', 'abcde', 2, NULL, 22) | abcde,2,22 |
| convert(*string* bytea,*src\_encoding* name, *dest\_encoding*name) | bytea | Convert string to *dest\_encoding*. The original encoding is specified by *src\_encoding*. The *string* must be valid in this encoding. Conversions can be defined by CREATE CONVERSION. Also there are some predefined conversions. See [**Table 9.10**](https://www.postgresql.org/docs/10/functions-string.html#CONVERSION-NAMES) for available conversions. | convert('text\_in\_utf8', 'UTF8', 'LATIN1') | text\_in\_utf8represented in Latin-1 encoding (ISO 8859-1) |
| convert\_from(*string* bytea,*src\_encoding* name) | text | Convert string to the database encoding. The original encoding is specified by *src\_encoding*. The *string* must be valid in this encoding. | convert\_from('text\_in\_utf8', 'UTF8') | text\_in\_utf8represented in the current database encoding |
| convert\_to(*string* text,*dest\_encoding* name) | bytea | Convert string to *dest\_encoding*. | convert\_to('some text', 'UTF8') | some text represented in the UTF8 encoding |
| decode(*string* text, *format* text) | bytea | Decode binary data from textual representation in *string*. Options for *format* are same as in encode. | decode('MTIzAAE=', 'base64') | \x3132330001 |
| encode(*data* bytea, *format* text) | text | Encode binary data into a textual representation. Supported formats are: base64, hex, escape. escape converts zero bytes and high-bit-set bytes to octal sequences (\***nnn***) and doubles backslashes. | encode('123\000\001', 'base64') | MTIzAAE= |
| format(*formatstr* text [,*formatarg* "any" [, ...] ]) | text | Format arguments according to a format string. This function is similar to the C function sprintf. See [**Section 9.4.1**](https://www.postgresql.org/docs/10/functions-string.html#FUNCTIONS-STRING-FORMAT). | format('Hello %s, %1$s', 'World') | Hello World, World |
| initcap(*string*) | text | Convert the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters. | initcap('hi THOMAS') | Hi Thomas |
| left(*str* text, *n* int) | text | Return first ***n*** characters in the string. When ***n*** is negative, return all but last |***n***| characters. | left('abcde', 2) | ab |
| length(*string*) | int | Number of characters in *string* | length('jose') | 4 |
| length(*string* bytea, *encoding*name ) | int | Number of characters in *string* in the given *encoding*. The *string* must be valid in this encoding. | length('jose', 'UTF8') | 4 |
| lpad(*string* text, *length* int [,*fill* text]) | text | Fill up the *string* to length *length* by prepending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated (on the right). | lpad('hi', 5, 'xy') | xyxhi |
| ltrim(*string* text [, *characters*text]) | text | Remove the longest string containing only characters from *characters* (a space by default) from the start of *string* | ltrim('zzzytest', 'xyz') | test |
| md5(*string*) | text | Calculates the MD5 hash of *string*, returning the result in hexadecimal | md5('abc') | 900150983cd24fb0 d6963f7d28e17f72 |
| parse\_ident(*qualified\_identifier*text [, *strictmode* booleanDEFAULT true ] ) | text[] | Split *qualified\_identifier* into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is false, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions.) Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to name[]. | parse\_ident('"SomeSchema".someTable') | {SomeSchema,sometable} |
| pg\_client\_encoding() | name | Current client encoding name | pg\_client\_encoding() | SQL\_ASCII |
| quote\_ident(*string* text) | text | Return the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also [**Example 42.1**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE). | quote\_ident('Foo bar') | "Foo bar" |
| quote\_literal(*string* text) | text | Return the given string suitably quoted to be used as a string literal in an SQLstatement string. Embedded single-quotes and backslashes are properly doubled. Note that quote\_literal returns null on null input; if the argument might be null, quote\_nullable is often more suitable. See also [**Example 42.1**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE). | quote\_literal(E'O\'Reilly') | 'O''Reilly' |
| quote\_literal(*value* anyelement) | text | Coerce the given value to text and then quote it as a literal. Embedded single-quotes and backslashes are properly doubled. | quote\_literal(42.5) | '42.5' |
| quote\_nullable(*string* text) | text | Return the given string suitably quoted to be used as a string literal in an SQLstatement string; or, if the argument is null, return NULL. Embedded single-quotes and backslashes are properly doubled. See also [**Example 42.1**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE). | quote\_nullable(NULL) | NULL |
| quote\_nullable(*value* anyelement) | text | Coerce the given value to text and then quote it as a literal; or, if the argument is null, return NULL. Embedded single-quotes and backslashes are properly doubled. | quote\_nullable(42.5) | '42.5' |
| regexp\_match(*string* text,*pattern* text [, *flags* text]) | text[] | Return captured substring(s) resulting from the first match of a POSIX regular expression to the *string*. See [**Section 9.7.3**](https://www.postgresql.org/docs/10/functions-matching.html#FUNCTIONS-POSIX-REGEXP) for more information. | regexp\_match('foobarbequebaz', '(bar)(beque)') | {bar,beque} |
| regexp\_matches(*string* text,*pattern* text [, *flags* text]) | setof text[] | Return captured substring(s) resulting from matching a POSIX regular expression to the *string*. See [**Section 9.7.3**](https://www.postgresql.org/docs/10/functions-matching.html#FUNCTIONS-POSIX-REGEXP) for more information. | regexp\_matches('foobarbequebaz', 'ba.', 'g') | {bar}  {baz}  (2 rows) |
| regexp\_replace(*string* text,*pattern* text, *replacement* text[, *flags* text]) | text | Replace substring(s) matching a POSIX regular expression. See [**Section 9.7.3**](https://www.postgresql.org/docs/10/functions-matching.html#FUNCTIONS-POSIX-REGEXP) for more information. | regexp\_replace('Thomas', '.[mN]a.', 'M') | ThM |
| regexp\_split\_to\_array(*string*text, *pattern* text [, *flags* text]) | text[] | Split *string* using a POSIX regular expression as the delimiter. See [**Section 9.7.3**](https://www.postgresql.org/docs/10/functions-matching.html#FUNCTIONS-POSIX-REGEXP) for more information. | regexp\_split\_to\_array('hello world', '\s+') | {hello,world} |
| regexp\_split\_to\_table(*string*text, *pattern* text [, *flags*text]) | setof text | Split *string* using a POSIX regular expression as the delimiter. See [**Section 9.7.3**](https://www.postgresql.org/docs/10/functions-matching.html#FUNCTIONS-POSIX-REGEXP) for more information. | regexp\_split\_to\_table('hello world', '\s+') | hello  world  (2 rows) |
| repeat(*string* text, *number* int) | text | Repeat *string* the specified *number* of times | repeat('Pg', 4) | PgPgPgPg |
| replace(*string* text, *from* text,*to* text) | text | Replace all occurrences in *string* of substring *from* with substring *to* | replace('abcdefabcdef', 'cd', 'XX') | abXXefabXXef |
| reverse(*str*) | text | Return reversed string. | reverse('abcde') | edcba |
| right(*str* text, *n* int) | text | Return last ***n*** characters in the string. When ***n*** is negative, return all but first |***n***| characters. | right('abcde', 2) | de |
| rpad(*string* text, *length* int [,*fill* text]) | text | Fill up the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated. | rpad('hi', 5, 'xy') | hixyx |
| rtrim(*string* text [, *characters*text]) | text | Remove the longest string containing only characters from *characters* (a space by default) from the end of *string* | rtrim('testxxzx', 'xyz') | test |
| split\_part(*string* text,*delimiter* text, *field* int) | text | Split *string* on *delimiter* and return the given field (counting from one) | split\_part('abc~@~def~@~ghi', '~@~', 2) | def |
| strpos(*string*, *substring*) | int | Location of specified substring (same as position(*substring* in *string*), but note the reversed argument order) | strpos('high', 'ig') | 2 |
| substr(*string*, *from* [, *count*]) | text | Extract substring (same as substring(*string* from *from* for *count*)) | substr('alphabet', 3, 2) | ph |
| to\_ascii(*string* text [, *encoding*text]) | text | Convert *string* to ASCII from another encoding (only supports conversion from LATIN1, LATIN2, LATIN9, and WIN1250 encodings) | to\_ascii('Karel') | Karel |
| to\_hex(*number* int or bigint) | text | Convert *number* to its equivalent hexadecimal representation | to\_hex(2147483647) | 7fffffff |
| translate(*string* text, *from*text, *to* text) | text | Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set. If *from* is longer than *to*, occurrences of the extra characters in *from* are removed. | translate('12345', '143', 'ax') | a2x5 |

The concat, concat\_ws and format functions are variadic, so it is possible to pass the values to be concatenated or formatted as an array marked with the VARIADIC keyword (see [**Section 37.4.5**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-VARIADIC-FUNCTIONS)). The array's elements are treated as if they were separate ordinary arguments to the function. If the variadic array argument is NULL, concat and concat\_ws return NULL, but format treats a NULL as a zero-element array.

See also the aggregate function string\_agg in [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html).

**Table 9.10. Built-in Conversions**

| **Conversion Name**[**[a]**](https://www.postgresql.org/docs/10/functions-string.html#ftn.id-1.5.8.9.10.2.1.1.1.1) | **Source Encoding** | **Destination Encoding** |
| --- | --- | --- |
| ascii\_to\_mic | SQL\_ASCII | MULE\_INTERNAL |
| ascii\_to\_utf8 | SQL\_ASCII | UTF8 |
| big5\_to\_euc\_tw | BIG5 | EUC\_TW |
| big5\_to\_mic | BIG5 | MULE\_INTERNAL |
| big5\_to\_utf8 | BIG5 | UTF8 |
| euc\_cn\_to\_mic | EUC\_CN | MULE\_INTERNAL |
| euc\_cn\_to\_utf8 | EUC\_CN | UTF8 |
| euc\_jp\_to\_mic | EUC\_JP | MULE\_INTERNAL |
| euc\_jp\_to\_sjis | EUC\_JP | SJIS |
| euc\_jp\_to\_utf8 | EUC\_JP | UTF8 |
| euc\_kr\_to\_mic | EUC\_KR | MULE\_INTERNAL |
| euc\_kr\_to\_utf8 | EUC\_KR | UTF8 |
| euc\_tw\_to\_big5 | EUC\_TW | BIG5 |
| euc\_tw\_to\_mic | EUC\_TW | MULE\_INTERNAL |
| euc\_tw\_to\_utf8 | EUC\_TW | UTF8 |
| gb18030\_to\_utf8 | GB18030 | UTF8 |
| gbk\_to\_utf8 | GBK | UTF8 |
| iso\_8859\_10\_to\_utf8 | LATIN6 | UTF8 |
| iso\_8859\_13\_to\_utf8 | LATIN7 | UTF8 |
| iso\_8859\_14\_to\_utf8 | LATIN8 | UTF8 |
| iso\_8859\_15\_to\_utf8 | LATIN9 | UTF8 |
| iso\_8859\_16\_to\_utf8 | LATIN10 | UTF8 |
| iso\_8859\_1\_to\_mic | LATIN1 | MULE\_INTERNAL |
| iso\_8859\_1\_to\_utf8 | LATIN1 | UTF8 |
| iso\_8859\_2\_to\_mic | LATIN2 | MULE\_INTERNAL |
| iso\_8859\_2\_to\_utf8 | LATIN2 | UTF8 |
| iso\_8859\_2\_to\_windows\_1250 | LATIN2 | WIN1250 |
| iso\_8859\_3\_to\_mic | LATIN3 | MULE\_INTERNAL |
| iso\_8859\_3\_to\_utf8 | LATIN3 | UTF8 |
| iso\_8859\_4\_to\_mic | LATIN4 | MULE\_INTERNAL |
| iso\_8859\_4\_to\_utf8 | LATIN4 | UTF8 |
| iso\_8859\_5\_to\_koi8\_r | ISO\_8859\_5 | KOI8R |
| iso\_8859\_5\_to\_mic | ISO\_8859\_5 | MULE\_INTERNAL |
| iso\_8859\_5\_to\_utf8 | ISO\_8859\_5 | UTF8 |
| iso\_8859\_5\_to\_windows\_1251 | ISO\_8859\_5 | WIN1251 |
| iso\_8859\_5\_to\_windows\_866 | ISO\_8859\_5 | WIN866 |
| iso\_8859\_6\_to\_utf8 | ISO\_8859\_6 | UTF8 |
| iso\_8859\_7\_to\_utf8 | ISO\_8859\_7 | UTF8 |
| iso\_8859\_8\_to\_utf8 | ISO\_8859\_8 | UTF8 |
| iso\_8859\_9\_to\_utf8 | LATIN5 | UTF8 |
| johab\_to\_utf8 | JOHAB | UTF8 |
| koi8\_r\_to\_iso\_8859\_5 | KOI8R | ISO\_8859\_5 |
| koi8\_r\_to\_mic | KOI8R | MULE\_INTERNAL |
| koi8\_r\_to\_utf8 | KOI8R | UTF8 |
| koi8\_r\_to\_windows\_1251 | KOI8R | WIN1251 |
| koi8\_r\_to\_windows\_866 | KOI8R | WIN866 |
| koi8\_u\_to\_utf8 | KOI8U | UTF8 |
| mic\_to\_ascii | MULE\_INTERNAL | SQL\_ASCII |
| mic\_to\_big5 | MULE\_INTERNAL | BIG5 |
| mic\_to\_euc\_cn | MULE\_INTERNAL | EUC\_CN |
| mic\_to\_euc\_jp | MULE\_INTERNAL | EUC\_JP |
| mic\_to\_euc\_kr | MULE\_INTERNAL | EUC\_KR |
| mic\_to\_euc\_tw | MULE\_INTERNAL | EUC\_TW |
| mic\_to\_iso\_8859\_1 | MULE\_INTERNAL | LATIN1 |
| mic\_to\_iso\_8859\_2 | MULE\_INTERNAL | LATIN2 |
| mic\_to\_iso\_8859\_3 | MULE\_INTERNAL | LATIN3 |
| mic\_to\_iso\_8859\_4 | MULE\_INTERNAL | LATIN4 |
| mic\_to\_iso\_8859\_5 | MULE\_INTERNAL | ISO\_8859\_5 |
| mic\_to\_koi8\_r | MULE\_INTERNAL | KOI8R |
| mic\_to\_sjis | MULE\_INTERNAL | SJIS |
| mic\_to\_windows\_1250 | MULE\_INTERNAL | WIN1250 |
| mic\_to\_windows\_1251 | MULE\_INTERNAL | WIN1251 |
| mic\_to\_windows\_866 | MULE\_INTERNAL | WIN866 |
| sjis\_to\_euc\_jp | SJIS | EUC\_JP |
| sjis\_to\_mic | SJIS | MULE\_INTERNAL |
| sjis\_to\_utf8 | SJIS | UTF8 |
| tcvn\_to\_utf8 | WIN1258 | UTF8 |
| uhc\_to\_utf8 | UHC | UTF8 |
| utf8\_to\_ascii | UTF8 | SQL\_ASCII |
| utf8\_to\_big5 | UTF8 | BIG5 |
| utf8\_to\_euc\_cn | UTF8 | EUC\_CN |
| utf8\_to\_euc\_jp | UTF8 | EUC\_JP |
| utf8\_to\_euc\_kr | UTF8 | EUC\_KR |
| utf8\_to\_euc\_tw | UTF8 | EUC\_TW |
| utf8\_to\_gb18030 | UTF8 | GB18030 |
| utf8\_to\_gbk | UTF8 | GBK |
| utf8\_to\_iso\_8859\_1 | UTF8 | LATIN1 |
| utf8\_to\_iso\_8859\_10 | UTF8 | LATIN6 |
| utf8\_to\_iso\_8859\_13 | UTF8 | LATIN7 |
| utf8\_to\_iso\_8859\_14 | UTF8 | LATIN8 |
| utf8\_to\_iso\_8859\_15 | UTF8 | LATIN9 |
| utf8\_to\_iso\_8859\_16 | UTF8 | LATIN10 |
| utf8\_to\_iso\_8859\_2 | UTF8 | LATIN2 |
| utf8\_to\_iso\_8859\_3 | UTF8 | LATIN3 |
| utf8\_to\_iso\_8859\_4 | UTF8 | LATIN4 |
| utf8\_to\_iso\_8859\_5 | UTF8 | ISO\_8859\_5 |
| utf8\_to\_iso\_8859\_6 | UTF8 | ISO\_8859\_6 |
| utf8\_to\_iso\_8859\_7 | UTF8 | ISO\_8859\_7 |
| utf8\_to\_iso\_8859\_8 | UTF8 | ISO\_8859\_8 |
| utf8\_to\_iso\_8859\_9 | UTF8 | LATIN5 |
| utf8\_to\_johab | UTF8 | JOHAB |
| utf8\_to\_koi8\_r | UTF8 | KOI8R |
| utf8\_to\_koi8\_u | UTF8 | KOI8U |
| utf8\_to\_sjis | UTF8 | SJIS |
| utf8\_to\_tcvn | UTF8 | WIN1258 |
| utf8\_to\_uhc | UTF8 | UHC |
| utf8\_to\_windows\_1250 | UTF8 | WIN1250 |
| utf8\_to\_windows\_1251 | UTF8 | WIN1251 |
| utf8\_to\_windows\_1252 | UTF8 | WIN1252 |
| utf8\_to\_windows\_1253 | UTF8 | WIN1253 |
| utf8\_to\_windows\_1254 | UTF8 | WIN1254 |
| utf8\_to\_windows\_1255 | UTF8 | WIN1255 |
| utf8\_to\_windows\_1256 | UTF8 | WIN1256 |
| utf8\_to\_windows\_1257 | UTF8 | WIN1257 |
| utf8\_to\_windows\_866 | UTF8 | WIN866 |
| utf8\_to\_windows\_874 | UTF8 | WIN874 |
| windows\_1250\_to\_iso\_8859\_2 | WIN1250 | LATIN2 |
| windows\_1250\_to\_mic | WIN1250 | MULE\_INTERNAL |
| windows\_1250\_to\_utf8 | WIN1250 | UTF8 |
| windows\_1251\_to\_iso\_8859\_5 | WIN1251 | ISO\_8859\_5 |
| windows\_1251\_to\_koi8\_r | WIN1251 | KOI8R |
| windows\_1251\_to\_mic | WIN1251 | MULE\_INTERNAL |
| windows\_1251\_to\_utf8 | WIN1251 | UTF8 |
| windows\_1251\_to\_windows\_866 | WIN1251 | WIN866 |
| windows\_1252\_to\_utf8 | WIN1252 | UTF8 |
| windows\_1256\_to\_utf8 | WIN1256 | UTF8 |
| windows\_866\_to\_iso\_8859\_5 | WIN866 | ISO\_8859\_5 |
| windows\_866\_to\_koi8\_r | WIN866 | KOI8R |
| windows\_866\_to\_mic | WIN866 | MULE\_INTERNAL |
| windows\_866\_to\_utf8 | WIN866 | UTF8 |
| windows\_866\_to\_windows\_1251 | WIN866 | WIN |
| windows\_874\_to\_utf8 | WIN874 | UTF8 |
| euc\_jis\_2004\_to\_utf8 | EUC\_JIS\_2004 | UTF8 |
| utf8\_to\_euc\_jis\_2004 | UTF8 | EUC\_JIS\_2004 |
| shift\_jis\_2004\_to\_utf8 | SHIFT\_JIS\_2004 | UTF8 |
| utf8\_to\_shift\_jis\_2004 | UTF8 | SHIFT\_JIS\_2004 |
| euc\_jis\_2004\_to\_shift\_jis\_2004 | EUC\_JIS\_2004 | SHIFT\_JIS\_2004 |
| shift\_jis\_2004\_to\_euc\_jis\_2004 | SHIFT\_JIS\_2004 | EUC\_JIS\_2004 |
| [**[a]**](https://www.postgresql.org/docs/10/functions-string.html#id-1.5.8.9.10.2.1.1.1.1) The conversion names follow a standard naming scheme: The official name of the source encoding with all non-alphanumeric characters replaced by underscores, followed by \_to\_, followed by the similarly processed destination encoding name. Therefore, the names might deviate from the customary encoding names. | | |

### 9.4.1. format

The function format produces output formatted according to a format string, in a style similar to the C function sprintf.

format(*formatstr* text [, *formatarg* "any" [, ...] ])

***formatstr*** is a format string that specifies how the result should be formatted. Text in the format string is copied directly to the result, except where format specifiers are used. Format specifiers act as placeholders in the string, defining how subsequent function arguments should be formatted and inserted into the result. Each ***formatarg*** argument is converted to text according to the usual output rules for its data type, and then formatted and inserted into the result string according to the format specifier(s).

Format specifiers are introduced by a % character and have the form

%[***position***][***flags***][***width***]***type***

where the component fields are:

***position*** (optional)

A string of the form ***n***$ where ***n*** is the index of the argument to print. Index 1 means the first argument after ***formatstr***. If the ***position*** is omitted, the default is to use the next argument in sequence.

***flags*** (optional)

Additional options controlling how the format specifier's output is formatted. Currently the only supported flag is a minus sign (-) which will cause the format specifier's output to be left-justified. This has no effect unless the ***width*** field is also specified.

***width*** (optional)

Specifies the minimum number of characters to use to display the format specifier's output. The output is padded on the left or right (depending on the - flag) with spaces as needed to fill the width. A too-small width does not cause truncation of the output, but is simply ignored. The width may be specified using any of the following: a positive integer; an asterisk (\*) to use the next function argument as the width; or a string of the form \****n***$ to use the ***n***th function argument as the width.

If the width comes from a function argument, that argument is consumed before the argument that is used for the format specifier's value. If the width argument is negative, the result is left aligned (as if the - flag had been specified) within a field of length abs(***width***).

***type*** (required)

The type of format conversion to use to produce the format specifier's output. The following types are supported:

* s formats the argument value as a simple string. A null value is treated as an empty string.
* I treats the argument value as an SQL identifier, double-quoting it if necessary. It is an error for the value to be null (equivalent to quote\_ident).
* L quotes the argument value as an SQL literal. A null value is displayed as the string NULL, without quotes (equivalent to quote\_nullable).

In addition to the format specifiers described above, the special sequence %% may be used to output a literal % character.

Here are some examples of the basic format conversions:

SELECT format('Hello %s', 'World');

*Result:* Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');

*Result:* Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');

*Result:* INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');

*Result:* INSERT INTO locations VALUES('C:\Program Files')

Here are examples using ***width*** fields and the - flag:

SELECT format('|%10s|', 'foo');

*Result:* | foo|

SELECT format('|%-10s|', 'foo');

*Result:* |foo |

SELECT format('|%\*s|', 10, 'foo');

*Result:* | foo|

SELECT format('|%\*s|', -10, 'foo');

*Result:* |foo |

SELECT format('|%-\*s|', 10, 'foo');

*Result:* |foo |

SELECT format('|%-\*s|', -10, 'foo');

*Result:* |foo |

These examples show use of ***position*** fields:

SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');

*Result:* Testing three, two, one

SELECT format('|%\*2$s|', 'foo', 10, 'bar');

*Result:* | bar|

SELECT format('|%1$\*2$s|', 'foo', 10, 'bar');

*Result:* | foo|

Unlike the standard C function sprintf, PostgreSQL's format function allows format specifiers with and without ***position*** fields to be mixed in the same format string. A format specifier without a ***position*** field always uses the next argument after the last argument consumed. In addition, the format function does not require all function arguments to be used in the format string. For example:

SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');

*Result:* Testing three, two, three

The %I and %L format specifiers are particularly useful for safely constructing dynamic SQL statements. See [**Example 42.1**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE).

## 9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating values of type bytea.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in [**Table 9.11**](https://www.postgresql.org/docs/10/functions-binarystring.html#FUNCTIONS-BINARYSTRING-SQL). PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see [**Table 9.12**](https://www.postgresql.org/docs/10/functions-binarystring.html#FUNCTIONS-BINARYSTRING-OTHER)).

Note

The sample results shown on this page assume that the server parameter [bytea\_output](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-BYTEA-OUTPUT) is set to escape (the traditional PostgreSQL format).

**Table 9.11. SQL Binary String Functions and Operators**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| *string* || *string* | bytea | String concatenation | '\\Post'::bytea || '\047gres\000'::bytea | \\Post'gres\000 |
| octet\_length(*string*) | int | Number of bytes in binary string | octet\_length('jo\000se'::bytea) | 5 |
| overlay(*string* placing *string* fromint [for int]) | bytea | Replace substring | overlay('Th\000omas'::bytea placing '\002\003'::bytea from 2 for 3) | T\\002\\003mas |
| position(*substring* in *string*) | int | Location of specified substring | position('\000om'::bytea in 'Th\000omas'::bytea) | 3 |
| substring(*string* [from int] [forint]) | bytea | Extract substring | substring('Th\000omas'::bytea from 2 for 3) | h\000o |
| trim([both] *bytes* from *string*) | bytea | Remove the longest string containing only bytes appearing in *bytes* from the start and end of *string* | trim('\000\001'::bytea from '\000Tom\001'::bytea) | Tom |

Additional binary string manipulation functions are available and are listed in [**Table 9.12**](https://www.postgresql.org/docs/10/functions-binarystring.html#FUNCTIONS-BINARYSTRING-OTHER). Some of them are used internally to implement the SQL-standard string functions listed in [**Table 9.11**](https://www.postgresql.org/docs/10/functions-binarystring.html#FUNCTIONS-BINARYSTRING-SQL).

**Table 9.12. Other Binary String Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| btrim(*string* bytea,*bytes* bytea) | bytea | Remove the longest string containing only bytes appearing in *bytes* from the start and end of *string* | btrim('\000trim\001'::bytea, '\000\001'::bytea) | trim |
| decode(*string* text,*format* text) | bytea | Decode binary data from textual representation in *string*. Options for *format* are same as in encode. | decode('123\000456', 'escape') | 123\000456 |
| encode(*data* bytea,*format* text) | text | Encode binary data into a textual representation. Supported formats are: base64, hex, escape. escape converts zero bytes and high-bit-set bytes to octal sequences (\***nnn***) and doubles backslashes. | encode('123\000456'::bytea, 'escape') | 123\000456 |
| get\_bit(*string*,*offset*) | int | Extract bit from string | get\_bit('Th\000omas'::bytea, 45) | 1 |
| get\_byte(*string*,*offset*) | int | Extract byte from string | get\_byte('Th\000omas'::bytea, 4) | 109 |
| length(*string*) | int | Length of binary string | length('jo\000se'::bytea) | 5 |
| md5(*string*) | text | Calculates the MD5 hash of *string*, returning the result in hexadecimal | md5('Th\000omas'::bytea) | 8ab2d3c9689aaf18 b4958c334c82d8b1 |
| set\_bit(*string*,*offset*, *newvalue*) | bytea | Set bit in string | set\_bit('Th\000omas'::bytea, 45, 0) | Th\000omAs |
| set\_byte(*string*,*offset*, *newvalue*) | bytea | Set byte in string | set\_byte('Th\000omas'::bytea, 4, 64) | Th\000o@as |

get\_byte and set\_byte number the first byte of a binary string as byte 0. get\_bit and set\_bit number bits from the right within each byte; for example bit 0 is the least significant bit of the first byte, and bit 15 is the most significant bit of the second byte.

See also the aggregate function string\_agg in [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html) and the large object functions in [**Section 34.4**](https://www.postgresql.org/docs/10/lo-funcs.html).

## 9.6. Bit String Functions and Operators

This section describes functions and operators for examining and manipulating bit strings, that is values of the types bit and bit varying. Aside from the usual comparison operators, the operators shown in [**Table 9.13**](https://www.postgresql.org/docs/10/functions-bitstring.html#FUNCTIONS-BIT-STRING-OP-TABLE) can be used. Bit string operands of &, |, and # must be of equal length. When bit shifting, the original length of the string is preserved, as shown in the examples.

**Table 9.13. Bit String Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| || | concatenation | B'10001' || B'011' | 10001011 |
| & | bitwise AND | B'10001' & B'01101' | 00001 |
| | | bitwise OR | B'10001' | B'01101' | 11101 |
| # | bitwise XOR | B'10001' # B'01101' | 11100 |
| ~ | bitwise NOT | ~ B'10001' | 01110 |
| << | bitwise shift left | B'10001' << 3 | 01000 |
| >> | bitwise shift right | B'10001' >> 2 | 00100 |

The following SQL-standard functions work on bit strings as well as character strings: length, bit\_length, octet\_length, position, substring, overlay.

The following functions work on bit strings as well as binary strings: get\_bit, set\_bit. When working with a bit string, these functions number the first (leftmost) bit of the string as bit 0.

In addition, it is possible to cast integral values to and from type bit. Some examples:

44::bit(10) *0000101100*

44::bit(3) *100*

cast(-44 as bit(12)) *111111010100*

'1110'::bit(4)::integer *14*

Note that casting to just “bit” means casting to bit(1), and so will deliver only the least significant bit of the integer.

Note

Casting an integer to bit(n) copies the rightmost n bits. Casting an integer to a bit string width wider than the integer itself will sign-extend on the left.

## 9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL LIKE operator, the more recent SIMILAR TO operator (added in SQL:1999), and POSIX-style regular expressions. Aside from the basic “does this string match this pattern?” operators, functions are available to extract or replace matching substrings and to split a string at matching locations.

Tip

If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

Caution

While most regular-expression searches can be executed very quickly, regular expressions can be contrived that take arbitrary amounts of time and memory to process. Be wary of accepting regular-expression search patterns from hostile sources. If you must do so, it is advisable to impose a statement timeout.

Searches using SIMILAR TO patterns have the same security hazards, since SIMILAR TOprovides many of the same capabilities as POSIX-style regular expressions.

LIKE searches, being much simpler than the other two options, are safer to use with possibly-hostile pattern sources.

### 9.7.1. LIKE

***string*** LIKE ***pattern*** [ESCAPE ***escape-character***]

***string*** NOT LIKE ***pattern*** [ESCAPE ***escape-character***]

The LIKE expression returns true if the ***string*** matches the supplied ***pattern***. (As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa. An equivalent expression is NOT (***string*** LIKE ***pattern***).)

If ***pattern*** does not contain percent signs or underscores, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (\_) in ***pattern*** stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

Some examples:

'abc' LIKE 'abc' *true*

'abc' LIKE 'a%' *true*

'abc' LIKE '\_b\_' *true*

'abc' LIKE 'c' *false*

LIKE pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in ***pattern*** must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note

If you have [**standard\_conforming\_strings**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-STANDARD-CONFORMING-STRINGS) turned off, any backslashes you write in literal string constants will need to be doubled. See [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS) for more information.

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator ~~ is equivalent to LIKE, and ~~\* corresponds to ILIKE. There are also !~~ and !~~\* operators that represent NOT LIKE and NOT ILIKE, respectively. All of these operators are PostgreSQL-specific.

### 9.7.2. SIMILAR TO Regular Expressions

***string*** SIMILAR TO ***pattern*** [ESCAPE ***escape-character***]

***string*** NOT SIMILAR TO ***pattern*** [ESCAPE ***escape-character***]

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string. It is similar to LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between LIKE notation and common regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like LIKE, SIMILAR TO uses \_ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to . and .\* in POSIX regular expressions).

In addition to these facilities borrowed from LIKE, SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

* | denotes alternation (either of two alternatives).
* \* denotes repetition of the previous item zero or more times.
* + denotes repetition of the previous item one or more times.
* ? denotes repetition of the previous item zero or one time.
* {***m***} denotes repetition of the previous item exactly ***m*** times.
* {***m***,} denotes repetition of the previous item ***m*** or more times.
* {***m***,***n***} denotes repetition of the previous item at least ***m*** and not more than ***n*** times.
* Parentheses () can be used to group items into a single logical item.
* A bracket expression [...] specifies a character class, just as in POSIX regular expressions.

Notice that the period (.) is not a metacharacter for SIMILAR TO.

As with LIKE, a backslash disables the special meaning of any of these metacharacters; or a different escape character can be specified with ESCAPE.

Some examples:

'abc' SIMILAR TO 'abc' *true*

'abc' SIMILAR TO 'a' *false*

'abc' SIMILAR TO '%(b|d)%' *true*

'abc' SIMILAR TO '(b|c)%' *false*

The substring function with three parameters, substring(***string*** from ***pattern*** for ***escape-character***), provides extraction of a substring that matches an SQL regular expression pattern. As with SIMILAR TO, the specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern that should be returned on success, the pattern must contain two occurrences of the escape character followed by a double quote ("). The text matching the portion of the pattern between these markers is returned.

Some examples, with #" delimiting the return string:

substring('foobar' from '%#"o\_b#"%' for '#') *oob*

substring('foobar' from '#"o\_b#"%' for '#') *NULL*

### 9.7.3. POSIX Regular Expressions

**[Table 9.14](https://www.postgresql.org/docs/10/functions-matching.html" \l "FUNCTIONS-POSIX-TABLE" \o "Table 9.14. Regular Expression Match Operators)** lists the available operators for pattern matching using POSIX regular expressions.

**Table 9.14. Regular Expression Match Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| ~ | Matches regular expression, case sensitive | 'thomas' ~ '.\*thomas.\*' |
| ~\* | Matches regular expression, case insensitive | 'thomas' ~\* '.\*Thomas.\*' |
| !~ | Does not match regular expression, case sensitive | 'thomas' !~ '.\*Thomas.\*' |
| !~\* | Does not match regular expression, case insensitive | 'thomas' !~\* '.\*vadim.\*' |

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE and SIMILAR TO operators. Many Unix tools such as egrep, sed, or awk use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with LIKE, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

'abc' ~ 'abc' *true*

'abc' ~ '^a' *true*

'abc' ~ '(b|d)' *true*

'abc' ~ '^(b|c)' *false*

The POSIX pattern language is described in much greater detail below.

The substring function with two parameters, substring(***string*** from ***pattern***), provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

substring('foobar' from 'o.b') *oob*

substring('foobar' from 'o(.)b') *o*

The regexp\_replace function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax regexp\_replace(***source***, ***pattern***, ***replacement*** [, ***flags*** ]). The ***source*** string is returned unchanged if there is no match to the ***pattern***. If there is a match, the ***source*** string is returned with the ***replacement*** string substituted for the matching substring. The ***replacement*** string can contain \***n***, where ***n*** is 1 through 9, to indicate that the source substring matching the ***n***'th parenthesized subexpression of the pattern should be inserted, and it can contain \&to indicate that the substring matching the entire pattern should be inserted. Write \\ if you need to put a literal backslash in the replacement text. The ***flags*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Flag i specifies case-insensitive matching, while flag g specifies replacement of each matching substring rather than only the first one. Supported flags (though not g) are described in [**Table 9.22**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE).

Some examples:

regexp\_replace('foobarbaz', 'b..', 'X')

*fooXbaz*

regexp\_replace('foobarbaz', 'b..', 'X', 'g')

*fooXX*

regexp\_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')

*fooXarYXazY*

The regexp\_match function returns a text array of captured substring(s) resulting from the first match of a POSIX regular expression pattern to a string. It has the syntax regexp\_match(***string***, ***pattern*** [, ***flags*** ]). If there is no match, the result is NULL. If a match is found, and the ***pattern*** contains no parenthesized subexpressions, then the result is a single-element text array containing the substring matching the whole pattern. If a match is found, and the ***pattern*** contains parenthesized subexpressions, then the result is a text array whose ***n***'th element is the substring matching the ***n***'th parenthesized subexpression of the ***pattern*** (not counting “non-capturing” parentheses; see below for details). The ***flags*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in [**Table 9.22**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE).

Some examples:

SELECT regexp\_match('foobarbequebaz', 'bar.\*que');

regexp\_match

--------------

{barbeque}

(1 row)

SELECT regexp\_match('foobarbequebaz', '(bar)(beque)');

regexp\_match

--------------

{bar,beque}

(1 row)

In the common case where you just want the whole matching substring or NULL for no match, write something like

SELECT (regexp\_match('foobarbequebaz', 'bar.\*que'))[1];

regexp\_match

--------------

barbeque

(1 row)

The regexp\_matches function returns a set of text arrays of captured substring(s) resulting from matching a POSIX regular expression pattern to a string. It has the same syntax as regexp\_match. This function returns no rows if there is no match, one row if there is a match and the g flag is not given, or ***N*** rows if there are ***N*** matches and the g flag is given. Each returned row is a text array containing the whole matched substring or the substrings matching parenthesized subexpressions of the ***pattern***, just as described above for regexp\_match. regexp\_matches accepts all the flags shown in [**Table 9.22**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE), plus the g flag which commands it to return all matches, not just the first one.

Some examples:

SELECT regexp\_matches('foo', 'not there');

regexp\_matches

----------------

(0 rows)

SELECT regexp\_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');

regexp\_matches

----------------

{bar,beque}

{bazil,barf}

(2 rows)

Tip

In most cases regexp\_matches() should be used with the g flag, since if you only want the first match, it's easier and more efficient to use regexp\_match(). However, regexp\_match()only exists in PostgreSQL version 10 and up. When working in older versions, a common trick is to place a regexp\_matches() call in a sub-select, for example:

SELECT col1, (SELECT regexp\_matches(col2, '(bar)(beque)')) FROM tab;

This produces a text array if there's a match, or NULL if not, the same as regexp\_match()would do. Without the sub-select, this query would produce no output at all for table rows without a match, which is typically not the desired behavior.

The regexp\_split\_to\_table function splits a string using a POSIX regular expression pattern as a delimiter. It has the syntax regexp\_split\_to\_table(***string***, ***pattern*** [, ***flags*** ]). If there is no match to the ***pattern***, the function returns the ***string***. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string. The ***flags*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. regexp\_split\_to\_table supports the flags described in [**Table 9.22**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE).

The regexp\_split\_to\_array function behaves the same as regexp\_split\_to\_table, except that regexp\_split\_to\_array returns its result as an array of text. It has the syntax regexp\_split\_to\_array(***string***, ***pattern*** [, ***flags*** ]). The parameters are the same as for regexp\_split\_to\_table.

Some examples:

SELECT foo FROM regexp\_split\_to\_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo;

foo

-------

the

quick

brown

fox

jumps

over

the

lazy

dog

(9 rows)

SELECT regexp\_split\_to\_array('the quick brown fox jumps over the lazy dog', '\s+');

regexp\_split\_to\_array

-----------------------------------------------

{the,quick,brown,fox,jumps,over,the,lazy,dog}

(1 row)

SELECT foo FROM regexp\_split\_to\_table('the quick brown fox', '\s\*') AS foo;

foo

-----

t

h

e

q

u

i

c

k

b

r

o

w

n

f

o

x

(16 rows)

As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by regexp\_match and regexp\_matches, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

#### 9.7.3.1. Regular Expression Details

PostgreSQL's regular expressions are implemented using a software package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: extended REs or EREs (roughly those of egrep), and basic REs or BREs (roughly those of ed). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called advanced REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

Note

PostgreSQL always initially presumes that a regular expression follows the ARE rules. However, the more limited ERE or BRE rules can be chosen by prepending an embedded option to the RE pattern, as described in [**Section 9.7.3.4**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-METASYNTAX). This can be useful for compatibility with applications that expect exactly the POSIX 1003.2 rules.

A regular expression is defined as one or more branches, separated by |. It matches anything that matches one of the branches.

A branch is zero or more quantified atoms or constraints, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an atom possibly followed by a single quantifier. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An atom can be any of the possibilities shown in [**Table 9.15**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-ATOMS-TABLE). The possible quantifiers and their meanings are shown in [**Table 9.16**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-QUANTIFIERS-TABLE).

A constraint matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it cannot be followed by a quantifier. The simple constraints are shown in [**Table 9.17**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-CONSTRAINTS-TABLE); some more constraints are described later.

**Table 9.15. Regular Expression Atoms**

| **Atom** | **Description** |
| --- | --- |
| (***re***) | (where ***re*** is any regular expression) matches a match for ***re***, with the match noted for possible reporting |
| (?:***re***) | as above, but the match is not noted for reporting (a “non-capturing” set of parentheses) (AREs only) |
| . | matches any single character |
| [***chars***] | a bracket expression, matching any one of the ***chars*** (see [**Section 9.7.3.2**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-BRACKET-EXPRESSIONS) for more detail) |
| \***k*** | (where ***k*** is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., \\ matches a backslash character |
| \***c*** | where ***c*** is alphanumeric (possibly followed by other characters) is an escape, see [**Section 9.7.3.3**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-ESCAPE-SEQUENCES) (AREs only; in EREs and BREs, this matches ***c***) |
| { | when followed by a character other than a digit, matches the left-brace character {; when followed by a digit, it is the beginning of a ***bound*** (see below) |
| ***x*** | where ***x*** is a single character with no other significance, matches that character |

An RE cannot end with a backslash (\).

Note

If you have [**standard\_conforming\_strings**](https://www.postgresql.org/docs/10/runtime-config-compatible.html#GUC-STANDARD-CONFORMING-STRINGS) turned off, any backslashes you write in literal string constants will need to be doubled. See [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS) for more information.

**Table 9.16. Regular Expression Quantifiers**

| **Quantifier** | **Matches** |
| --- | --- |
| \* | a sequence of 0 or more matches of the atom |
| + | a sequence of 1 or more matches of the atom |
| ? | a sequence of 0 or 1 matches of the atom |
| {***m***} | a sequence of exactly ***m*** matches of the atom |
| {***m***,} | a sequence of ***m*** or more matches of the atom |
| {***m***,***n***} | a sequence of ***m*** through ***n*** (inclusive) matches of the atom; ***m*** cannot exceed ***n*** |
| \*? | non-greedy version of \* |
| +? | non-greedy version of + |
| ?? | non-greedy version of ? |
| {***m***}? | non-greedy version of {***m***} |
| {***m***,}? | non-greedy version of {***m***,} |
| {***m***,***n***}? | non-greedy version of {***m***,***n***} |

The forms using {***...***} are known as bounds. The numbers ***m*** and ***n*** within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

Non-greedy quantifiers (available in AREs only) match the same possibilities as their corresponding normal (greedy) counterparts, but prefer the smallest number rather than the largest number of matches. See [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES) for more detail.

Note

A quantifier cannot immediately follow another quantifier, e.g., \*\* is invalid. A quantifier cannot begin an expression or subexpression or follow ^ or |.

**Table 9.17. Regular Expression Constraints**

| **Constraint** | **Description** |
| --- | --- |
| ^ | matches at the beginning of the string |
| $ | matches at the end of the string |
| (?=***re***) | positive lookahead matches at any point where a substring matching ***re*** begins (AREs only) |
| (?!***re***) | negative lookahead matches at any point where no substring matching ***re*** begins (AREs only) |
| (?<=***re***) | positive lookbehind matches at any point where a substring matching ***re*** ends (AREs only) |
| (?<!***re***) | negative lookbehind matches at any point where no substring matching ***re*** ends (AREs only) |

Lookahead and lookbehind constraints cannot contain back references (see [**Section 9.7.3.3**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-ESCAPE-SEQUENCES)), and all parentheses within them are considered non-capturing.

#### 9.7.3.2. Bracket Expressions

A bracket expression is a list of characters enclosed in []. It normally matches any single character from the list (but see below). If the list begins with ^, it matches any single character not from the rest of the list. If two characters in the list are separated by -, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g., [0-9] in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g., a-c-e. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.

To include a literal ] in the list, make it the first character (after ^, if that is used). To include a literal -, make it the first or last character, or the second endpoint of a range. To use a literal - as the first endpoint of a range, enclose it in [. and .] to make it a collating element (see below). With the exception of these characters, some combinations using [ (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, \ is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in [. and .]stands for the sequence of characters of that collating element. The sequence is treated as a single element of the bracket expression's list. This allows a bracket expression containing a multiple-character collating element to match more than one character, e.g., if the collating sequence includes a ch collating element, then the RE [[.ch.]]\*c matches the first five characters of chchcc.

Note

PostgreSQL currently does not support multi-character collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in [= and =] is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were [. and .].) For example, if o and ^ are the members of an equivalence class, then [[=o=]], [[=^=]], and [o^] are all synonymous. An equivalence class cannot be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in [: and :] stands for the list of all characters belonging to that class. Standard character class names are: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. These stand for the character classes defined in ctype. A locale can provide others. A character class cannot be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions [[:<:]] and [[:>:]] are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable; they are no more standard, but are easier to type.

#### 9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with \ followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and other inconvenient characters in REs. They are shown in [**Table 9.18**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-CHARACTER-ENTRY-ESCAPES-TABLE).

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in [**Table 9.19**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-CLASS-SHORTHAND-ESCAPES-TABLE).

A constraint escape is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in [**Table 9.20**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-CONSTRAINT-ESCAPES-TABLE).

A back reference (\***n***) matches the same string matched by the previous parenthesized subexpression specified by the number ***n*** (see [**Table 9.21**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-CONSTRAINT-BACKREF-TABLE)). For example, ([bc])\1 matches bb or cc but not bc or cb. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

**Table 9.18. Regular Expression Character-entry Escapes**

| **Escape** | **Description** |
| --- | --- |
| \a | alert (bell) character, as in C |
| \b | backspace, as in C |
| \B | synonym for backslash (\) to help reduce the need for backslash doubling |
| \c***X*** | (where ***X*** is any character) the character whose low-order 5 bits are the same as those of ***X***, and whose other bits are all zero |
| \e | the character whose collating-sequence name is ESC, or failing that, the character with octal value 033 |
| \f | form feed, as in C |
| \n | newline, as in C |
| \r | carriage return, as in C |
| \t | horizontal tab, as in C |
| \u***wxyz*** | (where ***wxyz*** is exactly four hexadecimal digits) the character whose hexadecimal value is 0x***wxyz*** |
| \U***stuvwxyz*** | (where ***stuvwxyz*** is exactly eight hexadecimal digits) the character whose hexadecimal value is 0x***stuvwxyz*** |
| \v | vertical tab, as in C |
| \x***hhh*** | (where ***hhh*** is any sequence of hexadecimal digits) the character whose hexadecimal value is 0x***hhh*** (a single character no matter how many hexadecimal digits are used) |
| \0 | the character whose value is 0 (the null byte) |
| \***xy*** | (where ***xy*** is exactly two octal digits, and is not a back reference) the character whose octal value is 0***xy*** |
| \***xyz*** | (where ***xyz*** is exactly three octal digits, and is not a back reference) the character whose octal value is 0***xyz*** |

Hexadecimal digits are 0-9, a-f, and A-F. Octal digits are 0-7.

Numeric character-entry escapes specifying values outside the ASCII range (0-127) have meanings dependent on the database encoding. When the encoding is UTF-8, escape values are equivalent to Unicode code points, for example \u1234 means the character U+1234. For other multibyte encodings, character-entry escapes usually just specify the concatenation of the byte values for the character. If the escape value does not correspond to any legal character in the database encoding, no error will be raised, but it will never match any data.

The character-entry escapes are always taken as ordinary characters. For example, \135 is ] in ASCII, but \135 does not terminate a bracket expression.

**Table 9.19. Regular Expression Class-shorthand Escapes**

| **Escape** | **Description** |
| --- | --- |
| \d | [[:digit:]] |
| \s | [[:space:]] |
| \w | [[:alnum:]\_] (note underscore is included) |
| \D | [^[:digit:]] |
| \S | [^[:space:]] |
| \W | [^[:alnum:]\_] (note underscore is included) |

Within bracket expressions, \d, \s, and \w lose their outer brackets, and \D, \S, and \W are illegal. (So, for example, [a-c\d] is equivalent to [a-c[:digit:]]. Also, [a-c\D], which is equivalent to [a-c^[:digit:]], is illegal.)

**Table 9.20. Regular Expression Constraint Escapes**

| **Escape** | **Description** |
| --- | --- |
| \A | matches only at the beginning of the string (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES) for how this differs from ^) |
| \m | matches only at the beginning of a word |
| \M | matches only at the end of a word |
| \y | matches only at the beginning or end of a word |
| \Y | matches only at a point that is not the beginning or end of a word |
| \Z | matches only at the end of the string (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES) for how this differs from $) |

A word is defined as in the specification of [[:<:]] and [[:>:]] above. Constraint escapes are illegal within bracket expressions.

**Table 9.21. Regular Expression Back References**

| **Escape** | **Description** |
| --- | --- |
| \***m*** | (where ***m*** is a nonzero digit) a back reference to the ***m***'th subexpression |
| \***mnn*** | (where ***m*** is a nonzero digit, and ***nn*** is some more digits, and the decimal value ***mnn*** is not greater than the number of closing capturing parentheses seen so far) a back reference to the ***mnn***'th subexpression |

Note

There is an inherent ambiguity between octal character-entry escapes and back references, which is resolved by the following heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e., the number is in the legal range for a back reference), and otherwise is taken as octal.

#### 9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

An RE can begin with one of two special director prefixes. If an RE begins with \*\*\*:, the rest of the RE is taken as an ARE. (This normally has no effect in PostgreSQL, since REs are assumed to be AREs; but it does have an effect if ERE or BRE mode had been specified by the ***flags*** parameter to a regex function.) If an RE begins with \*\*\*=, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE can begin with embedded options: a sequence (?***xyz***) (where ***xyz*** is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options — in particular, they can override the case-sensitivity behavior implied by a regex operator, or the ***flags*** parameter to a regex function. The available option letters are shown in [**Table 9.22**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-EMBEDDED-OPTIONS-TABLE). Note that these same option letters are used in the ***flags*** parameters of regex functions.

**Table 9.22. ARE Embedded-option Letters**

| **Option** | **Description** |
| --- | --- |
| b | rest of RE is a BRE |
| c | case-sensitive matching (overrides operator type) |
| e | rest of RE is an ERE |
| i | case-insensitive matching (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES)) (overrides operator type) |
| m | historical synonym for n |
| n | newline-sensitive matching (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES)) |
| p | partial newline-sensitive matching (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES)) |
| q | rest of RE is a literal (“quoted”) string, all ordinary characters |
| s | non-newline-sensitive matching (default) |
| t | tight syntax (default; see below) |
| w | inverse partial newline-sensitive (“weird”) matching (see [**Section 9.7.3.5**](https://www.postgresql.org/docs/10/functions-matching.html#POSIX-MATCHING-RULES)) |
| x | expanded syntax (see below) |

Embedded options take effect at the ) terminating the sequence. They can appear only at the start of an ARE (after the \*\*\*: director if any).

In addition to the usual (tight) RE syntax, in which all characters are significant, there is an expanded syntax, available by specifying the embedded x option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a # and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

* a white-space character or # preceded by \ is retained
* white space or # within a bracket expression is retained
* white space and comments cannot appear within multi-character symbols, such as (?:

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the ***space*** character class.

Finally, in an ARE, outside bracket expressions, the sequence (?#***ttt***) (where ***ttt*** is any text not containing a )) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols, like (?:. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial \*\*\*= director has specified that the user's input be treated as a literal string rather than as an RE.

#### 9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is greedy or non-greedy.

Whether an RE is greedy or not is determined by the following rules:

* Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
* Adding parentheses around an RE does not change its greediness.
* A quantified atom with a fixed-repetition quantifier ({***m***} or {***m***}?) has the same greediness (possibly none) as the atom itself.
* A quantified atom with other normal quantifiers (including {***m***,***n***} with ***m*** equal to ***n***) is greedy (prefers longest match).
* A quantified atom with a non-greedy quantifier (including {***m***,***n***}? with ***m*** equal to ***n***) is non-greedy (prefers shortest match).
* A branch — that is, an RE that has no top-level | operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
* An RE consisting of two or more branches connected by the | operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring as a whole. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

SELECT SUBSTRING('XY1234Z', 'Y\*([0-9]{1,3})');

*Result:* 123

SELECT SUBSTRING('XY1234Z', 'Y\*?([0-9]{1,3})');

*Result:* 1

In the first case, the RE as a whole is greedy because Y\* is greedy. It can match beginning at the Y, and it matches the longest possible string starting there, i.e., Y123. The output is the parenthesized part of that, or 123. In the second case, the RE as a whole is non-greedy because Y\*? is non-greedy. It can match beginning at the Y, and it matches the shortest possible string starting there, i.e., Y1. The subexpression [0-9]{1,3} is greedy but it cannot change the decision as to the overall match length; so it is forced to match just 1.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to “eat” relative to each other.

The quantifiers {1,1} and {1,1}? can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE. This is useful when you need the whole RE to have a greediness attribute different from what's deduced from its elements. As an example, suppose that we are trying to separate a string containing some digits into the digits and the parts before and after them. We might try to do that like this:

SELECT regexp\_match('abc01234xyz', '(.\*)(\d+)(.\*)');

*Result:* {abc0123,4,xyz}

That didn't work: the first .\* is greedy so it “eats” as much as it can, leaving the \d+ to match at the last possible place, the last digit. We might try to fix that by making it non-greedy:

SELECT regexp\_match('abc01234xyz', '(.\*?)(\d+)(.\*)');

*Result:* {abc,0,""}

That didn't work either, because now the RE as a whole is non-greedy and so it ends the overall match as soon as possible. We can get what we want by forcing the RE as a whole to be greedy:

SELECT regexp\_match('abc01234xyz', '(?:(.\*?)(\d+)(.\*)){1,1}');

*Result:* {abc,01234,xyz}

Controlling the RE's overall greediness separately from its components' greediness allows great flexibility in handling variable-length patterns.

When deciding what is a longer or shorter match, match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: bb\*matches the three middle characters of abbbc; (week|wee)(night|knights) matches all ten characters of weeknights; when (.\*).\* is matched against abc the parenthesized subexpression matches all three characters; and when (a\*)\* is matched against bc both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g., x becomes [xX]. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g., [x] becomes [xX] and [^x] becomes [^xX].

If newline-sensitive matching is specified, . and bracket expressions using ^ will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and ^and $ will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes \A and \Z continue to match beginning or end of string only.

If partial newline-sensitive matching is specified, this affects . and bracket expressions as with newline-sensitive matching, but not ^ and $.

If inverse partial newline-sensitive matching is specified, this affects ^ and $ as with newline-sensitive matching, but not . and bracket expressions. This isn't very useful but is provided for symmetry.

#### 9.7.3.6. Limits And Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that \ does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the \*\*\* syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include \b, \B, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead/lookbehind constraints, and the longest/shortest-match (rather than first-match) matching semantics.

Two significant incompatibilities exist between AREs and the ERE syntax recognized by pre-7.4 releases of PostgreSQL:

* In AREs, \ followed by an alphanumeric character is either an escape or an error, while in previous releases, it was just another way of writing the alphanumeric. This should not be much of a problem because there was no reason to write such a sequence in earlier releases.
* In AREs, \ remains a special character within [], so a literal \ within a bracket expression must be written \\.

#### 9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. In BREs, |, +, and ? are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are \{ and \}, with { and } by themselves ordinary characters. The parentheses for nested subexpressions are \( and \), with ( and ) by themselves ordinary characters. ^ is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, $ is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and \* is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading ^). Finally, single-digit back references are available, and \< and \> are synonyms for [[:<:]] and [[:>:]] respectively; no other escapes are available in BREs.

## 9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. [**Table 9.23**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-TABLE) lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

**Table 9.23. Formatting Functions**

| **Function** | **Return Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| to\_char(timestamp, text) | text | convert time stamp to string | to\_char(current\_timestamp, 'HH12:MI:SS') |
| to\_char(interval, text) | text | convert interval to string | to\_char(interval '15h 2m 12s', 'HH24:MI:SS') |
| to\_char(int, text) | text | convert integer to string | to\_char(125, '999') |
| to\_char(double precision, text) | text | convert real/double precision to string | to\_char(125.8::real, '999D9') |
| to\_char(numeric, text) | text | convert numeric to string | to\_char(-125.8, '999D99S') |
| to\_date(text, text) | date | convert string to date | to\_date('05 Dec 2000', 'DD Mon YYYY') |
| to\_number(text, text) | numeric | convert string to numeric | to\_number('12,454.8-', '99G999D9S') |
| to\_timestamp(text, text) | timestamp with time zone | convert string to time stamp | to\_timestamp('05 Dec 2000', 'DD Mon YYYY') |

Note

There is also a single-argument to\_timestamp function; see [**Table 9.30**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-TABLE).

Tip

to\_timestamp and to\_date exist to handle input formats that cannot be converted by simple casting. For most standard date/time formats, simply casting the source string to the required data type works, and is much easier. Similarly, to\_number is unnecessary for standard numeric representations.

In a to\_char output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data based on the given value. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for the other functions), template patterns identify the values to be supplied by the input data string.

[**Table 9.24**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-DATETIME-TABLE) shows the template patterns available for formatting date and time values.

**Table 9.24. Template Patterns for Date/Time Formatting**

| **Pattern** | **Description** |
| --- | --- |
| HH | hour of day (01-12) |
| HH12 | hour of day (01-12) |
| HH24 | hour of day (00-23) |
| MI | minute (00-59) |
| SS | second (00-59) |
| MS | millisecond (000-999) |
| US | microsecond (000000-999999) |
| SSSS | seconds past midnight (0-86399) |
| AM, am, PM or pm | meridiem indicator (without periods) |
| A.M., a.m., P.M. or p.m. | meridiem indicator (with periods) |
| Y,YYY | year (4 or more digits) with comma |
| YYYY | year (4 or more digits) |
| YYY | last 3 digits of year |
| YY | last 2 digits of year |
| Y | last digit of year |
| IYYY | ISO 8601 week-numbering year (4 or more digits) |
| IYY | last 3 digits of ISO 8601 week-numbering year |
| IY | last 2 digits of ISO 8601 week-numbering year |
| I | last digit of ISO 8601 week-numbering year |
| BC, bc, AD or ad | era indicator (without periods) |
| B.C., b.c., A.D. or a.d. | era indicator (with periods) |
| MONTH | full upper case month name (blank-padded to 9 chars) |
| Month | full capitalized month name (blank-padded to 9 chars) |
| month | full lower case month name (blank-padded to 9 chars) |
| MON | abbreviated upper case month name (3 chars in English, localized lengths vary) |
| Mon | abbreviated capitalized month name (3 chars in English, localized lengths vary) |
| mon | abbreviated lower case month name (3 chars in English, localized lengths vary) |
| MM | month number (01-12) |
| DAY | full upper case day name (blank-padded to 9 chars) |
| Day | full capitalized day name (blank-padded to 9 chars) |
| day | full lower case day name (blank-padded to 9 chars) |
| DY | abbreviated upper case day name (3 chars in English, localized lengths vary) |
| Dy | abbreviated capitalized day name (3 chars in English, localized lengths vary) |
| dy | abbreviated lower case day name (3 chars in English, localized lengths vary) |
| DDD | day of year (001-366) |
| IDDD | day of ISO 8601 week-numbering year (001-371; day 1 of the year is Monday of the first ISO week) |
| DD | day of month (01-31) |
| D | day of the week, Sunday (1) to Saturday (7) |
| ID | ISO 8601 day of the week, Monday (1) to Sunday (7) |
| W | week of month (1-5) (the first week starts on the first day of the month) |
| WW | week number of year (1-53) (the first week starts on the first day of the year) |
| IW | week number of ISO 8601 week-numbering year (01-53; the first Thursday of the year is in week 1) |
| CC | century (2 digits) (the twenty-first century starts on 2001-01-01) |
| J | Julian Day (integer days since November 24, 4714 BC at midnight UTC) |
| Q | quarter |
| RM | month in upper case Roman numerals (I-XII; I=January) |
| rm | month in lower case Roman numerals (i-xii; i=January) |
| TZ | upper case time-zone abbreviation (only supported in to\_char) |
| tz | lower case time-zone abbreviation (only supported in to\_char) |
| OF | time-zone offset from UTC (only supported in to\_char) |

Modifiers can be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. [**Table 9.25**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-DATETIMEMOD-TABLE) shows the modifier patterns for date/time formatting.

**Table 9.25. Template Pattern Modifiers for Date/Time Formatting**

| **Modifier** | **Description** | **Example** |
| --- | --- | --- |
| FM prefix | fill mode (suppress leading zeroes and padding blanks) | FMMonth |
| TH suffix | upper case ordinal number suffix | DDTH, e.g., 12TH |
| th suffix | lower case ordinal number suffix | DDth, e.g., 12th |
| FX prefix | fixed format global option (see usage notes) | FX Month DD Day |
| TM prefix | translation mode (print localized day and month names based on [**lc\_time**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-LC-TIME)) | TMMonth |
| SP suffix | spell mode (not implemented) | DDSP |

Usage notes for date/time formatting:

* FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width. In PostgreSQL, FM modifies only the next specification, while in Oracle FM affects all subsequent specifications, and repeated FM modifiers toggle fill mode on and off.
* TM does not include trailing blanks. to\_timestamp and to\_date ignore the TM modifier.
* to\_timestamp and to\_date skip multiple blank spaces in the input string unless the FX option is used. For example, to\_timestamp('2000    JUN', 'YYYY MON') works, but to\_timestamp('2000    JUN', 'FXYYYY MON') returns an error because to\_timestamp expects one space only. FX must be specified as the first item in the template.
* Ordinary text is allowed in to\_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. For example, in '"Hello Year "YYYY', the YYYY will be replaced by the year data, but the single Y in Year will not be. In to\_date, to\_number, and to\_timestamp, double-quoted strings skip the number of input characters contained in the string, e.g. "XX" skips two input characters.
* If you want to have a double quote in the output you must precede it with a backslash, for example '\"YYYY Month\"'.
* In to\_timestamp and to\_date, if the year format specification is less than four digits, e.g. YYY, and the supplied year is less than four digits, the year will be adjusted to be nearest to the year 2020, e.g. 95 becomes 1995.
* In to\_timestamp and to\_date, the YYYY conversion has a restriction when processing years with more than 4 digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as 4 digits. For example (with the year 20000): to\_date('200001131', 'YYYYMMDD') will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like to\_date('20000-1131', 'YYYY-MMDD') or to\_date('20000Nov31', 'YYYYMonDD').
* In to\_timestamp and to\_date, the CC (century) field is accepted but ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the result is computed as that year in the specified century. If the century is specified but the year is not, the first year of the century is assumed.
* In to\_timestamp and to\_date, weekday names or numbers (DAY, D, and related field types) are accepted but are ignored for purposes of computing the result. The same is true for quarter (Q) fields.
* In to\_timestamp and to\_date, an ISO 8601 week-numbering date (as distinct from a Gregorian date) can be specified in one of two ways:
  + Year, week number, and weekday: for example to\_date('2006-42-4', 'IYYY-IW-ID') returns the date 2006-10-19. If you omit the weekday it is assumed to be 1 (Monday).
  + Year and day of year: for example to\_date('2006-291', 'IYYY-IDDD') also returns 2006-10-19.

Attempting to enter a date using a mixture of ISO 8601 week-numbering fields and Gregorian date fields is nonsensical, and will cause an error. In the context of an ISO 8601 week-numbering year, the concept of a “month” or “day of month” has no meaning. In the context of a Gregorian year, the ISO week has no meaning.

Caution

While to\_date will reject a mixture of Gregorian and ISO week-numbering date fields, to\_char will not, since output format specifications like YYYY-MM-DD (IYYY-IDDD) can be useful. But avoid writing something like IYYY-MM-DD; that would yield surprising results near the start of the year. (See [**Section 9.9.1**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT) for more information.)

* In to\_timestamp, millisecond (MS) or microsecond (US) fields are used as the seconds digits after the decimal point. For example to\_timestamp('12.3', 'SS.MS') is not 3 milliseconds, but 300, because the conversion treats it as 12 + 0.3 seconds. So, for the format SS.MS, the input values 12.3, 12.30, and 12.300 specify the same number of milliseconds. To get three milliseconds, one must write 12.003, which the conversion treats as 12 + 0.003 = 12.003 seconds.

Here is a more complex example: to\_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US') is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

* to\_char(..., 'ID')'s day of the week numbering matches the extract(isodow from ...) function, but to\_char(..., 'D')'s does not match extract(dow from ...)'s day numbering.
* to\_char(interval) formats HH and HH12 as shown on a 12-hour clock, for example zero hours and 36 hours both output as 12, while HH24 outputs the full hour value, which can exceed 23 in an interval value.

[**Table 9.26**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-NUMERIC-TABLE) shows the template patterns available for formatting numeric values.

**Table 9.26. Template Patterns for Numeric Formatting**

| **Pattern** | **Description** |
| --- | --- |
| 9 | digit position (can be dropped if insignificant) |
| 0 | digit position (will not be dropped, even if insignificant) |
| . (period) | decimal point |
| , (comma) | group (thousands) separator |
| PR | negative value in angle brackets |
| S | sign anchored to number (uses locale) |
| L | currency symbol (uses locale) |
| D | decimal point (uses locale) |
| G | group separator (uses locale) |
| MI | minus sign in specified position (if number < 0) |
| PL | plus sign in specified position (if number > 0) |
| SG | plus/minus sign in specified position |
| RN | Roman numeral (input between 1 and 3999) |
| TH or th | ordinal number suffix |
| V | shift specified number of digits (see notes) |
| EEEE | exponent for scientific notation |

Usage notes for numeric formatting:

* 0 specifies a digit position that will always be printed, even if it contains a leading/trailing zero. 9 also specifies a digit position, but if it is a leading zero then it will be replaced by a space, while if it is a trailing zero and fill mode is specified then it will be deleted. (For to\_number(), these two pattern characters are equivalent.)
* The pattern characters S, L, D, and G represent the sign, currency symbol, decimal point, and thousands separator characters defined by the current locale (see [**lc\_monetary**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-LC-MONETARY) and [**lc\_numeric**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-LC-NUMERIC)). The pattern characters period and comma represent those exact characters, with the meanings of decimal point and thousands separator, regardless of locale.
* If no explicit provision is made for a sign in to\_char()'s pattern, one column will be reserved for the sign, and it will be anchored to (appear just left of) the number. If S appears just left of some 9's, it will likewise be anchored to the number.
* A sign formatted using SG, PL, or MI is not anchored to the number; for example, to\_char(-12, 'MI9999') produces '-  12' but to\_char(-12, 'S9999') produces '  -12'. (The Oracle implementation does not allow the use of MI before 9, but rather requires that 9 precede MI.)
* TH does not convert values less than zero and does not convert fractional numbers.
* PL, SG, and TH are PostgreSQL extensions.
* V with to\_char multiplies the input values by 10^***n***, where ***n*** is the number of digits following V. V with to\_number divides in a similar manner. to\_char and to\_number do not support the use of V combined with a decimal point (e.g., 99.9V99 is not allowed).
* EEEE (scientific notation) cannot be used in combination with any of the other formatting patterns or modifiers other than digit and decimal point patterns, and must be at the end of the format string (e.g., 9.99EEEE is a valid pattern).

Certain modifiers can be applied to any template pattern to alter its behavior. For example, FM99.99 is the 99.99 pattern with the FM modifier. [**Table 9.27**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-NUMERICMOD-TABLE) shows the modifier patterns for numeric formatting.

**Table 9.27. Template Pattern Modifiers for Numeric Formatting**

| **Modifier** | **Description** | **Example** |
| --- | --- | --- |
| FM prefix | fill mode (suppress trailing zeroes and padding blanks) | FM99.99 |
| TH suffix | upper case ordinal number suffix | 999TH |
| th suffix | lower case ordinal number suffix | 999th |

[**Table 9.28**](https://www.postgresql.org/docs/10/functions-formatting.html#FUNCTIONS-FORMATTING-EXAMPLES-TABLE) shows some examples of the use of the to\_char function.

**Table 9.28.**to\_char**Examples**

| **Expression** | **Result** |
| --- | --- |
| to\_char(current\_timestamp, 'Day, DD  HH12:MI:SS') | 'Tuesday  , 06  05:39:18' |
| to\_char(current\_timestamp, 'FMDay, FMDD  HH12:MI:SS') | 'Tuesday, 6  05:39:18' |
| to\_char(-0.1, '99.99') | '  -.10' |
| to\_char(-0.1, 'FM9.99') | '-.1' |
| to\_char(-0.1, 'FM90.99') | '-0.1' |
| to\_char(0.1, '0.9') | ' 0.1' |
| to\_char(12, '9990999.9') | '    0012.0' |
| to\_char(12, 'FM9990999.9') | '0012.' |
| to\_char(485, '999') | ' 485' |
| to\_char(-485, '999') | '-485' |
| to\_char(485, '9 9 9') | ' 4 8 5' |
| to\_char(1485, '9,999') | ' 1,485' |
| to\_char(1485, '9G999') | ' 1 485' |
| to\_char(148.5, '999.999') | ' 148.500' |
| to\_char(148.5, 'FM999.999') | '148.5' |
| to\_char(148.5, 'FM999.990') | '148.500' |
| to\_char(148.5, '999D999') | ' 148,500' |
| to\_char(3148.5, '9G999D999') | ' 3 148,500' |
| to\_char(-485, '999S') | '485-' |
| to\_char(-485, '999MI') | '485-' |
| to\_char(485, '999MI') | '485 ' |
| to\_char(485, 'FM999MI') | '485' |
| to\_char(485, 'PL999') | '+485' |
| to\_char(485, 'SG999') | '+485' |
| to\_char(-485, 'SG999') | '-485' |
| to\_char(-485, '9SG99') | '4-85' |
| to\_char(-485, '999PR') | '<485>' |
| to\_char(485, 'L999') | 'DM 485' |
| to\_char(485, 'RN') | '        CDLXXXV' |
| to\_char(485, 'FMRN') | 'CDLXXXV' |
| to\_char(5.2, 'FMRN') | 'V' |
| to\_char(482, '999th') | ' 482nd' |
| to\_char(485, '"Good number:"999') | 'Good number: 485' |
| to\_char(485.8, '"Pre:"999" Post:" .999') | 'Pre: 485 Post: .800' |
| to\_char(12, '99V999') | ' 12000' |
| to\_char(12.4, '99V999') | ' 12400' |
| to\_char(12.45, '99V9') | ' 125' |
| to\_char(0.0004859, '9.99EEEE') | ' 4.86e-04' |

## 9.9. Date/Time Functions and Operators

[**Table 9.30**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-TABLE) shows the available functions for date/time value processing, with details appearing in the following subsections. [**Table 9.29**](https://www.postgresql.org/docs/10/functions-datetime.html#OPERATORS-DATETIME-TABLE) illustrates the behaviors of the basic arithmetic operators (+, \*, etc.). For formatting functions, refer to [**Section 9.8**](https://www.postgresql.org/docs/10/functions-formatting.html). You should be familiar with the background information on date/time data types from [**Section 8.5**](https://www.postgresql.org/docs/10/datatype-datetime.html).

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time with time zone or timestamp with time zone, and one that takes time without time zone or timestamp without time zone. For brevity, these variants are not shown separately. Also, the + and \* operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

**Table 9.29. Date/Time Operators**

| **Operator** | **Example** | **Result** |
| --- | --- | --- |
| + | date '2001-09-28' + integer '7' | date '2001-10-05' |
| + | date '2001-09-28' + interval '1 hour' | timestamp '2001-09-28 01:00:00' |
| + | date '2001-09-28' + time '03:00' | timestamp '2001-09-28 03:00:00' |
| + | interval '1 day' + interval '1 hour' | interval '1 day 01:00:00' |
| + | timestamp '2001-09-28 01:00' + interval '23 hours' | timestamp '2001-09-29 00:00:00' |
| + | time '01:00' + interval '3 hours' | time '04:00:00' |
| - | - interval '23 hours' | interval '-23:00:00' |
| - | date '2001-10-01' - date '2001-09-28' | integer '3' (days) |
| - | date '2001-10-01' - integer '7' | date '2001-09-24' |
| - | date '2001-09-28' - interval '1 hour' | timestamp '2001-09-27 23:00:00' |
| - | time '05:00' - time '03:00' | interval '02:00:00' |
| - | time '05:00' - interval '2 hours' | time '03:00:00' |
| - | timestamp '2001-09-28 23:00' - interval '23 hours' | timestamp '2001-09-28 00:00:00' |
| - | interval '1 day' - interval '1 hour' | interval '1 day -01:00:00' |
| - | timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' | interval '1 day 15:00:00' |
| \* | 900 \* interval '1 second' | interval '00:15:00' |
| \* | 21 \* interval '1 day' | interval '21 days' |
| \* | double precision '3.5' \* interval '1 hour' | interval '03:30:00' |
| / | interval '1 hour' / double precision '1.5' | interval '00:40:00' |

**Table 9.30. Date/Time Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| age(timestamp, timestamp) | interval | Subtract arguments, producing a “symbolic” result that uses years and months, rather than just days | age(timestamp '2001-04-10', timestamp '1957-06-13') | 43 years 9 mons 27 days |
| age(timestamp) | interval | Subtract from current\_date (at midnight) | age(timestamp '1957-06-13') | 43 years 8 mons 3 days |
| clock\_timestamp() | timestamp with time zone | Current date and time (changes during statement execution); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| current\_date | date | Current date; see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| current\_time | time with time zone | Current time of day; see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| current\_timestamp | timestamp with time zone | Current date and time (start of current transaction); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| date\_part(text, timestamp) | double precision | Get subfield (equivalent to extract); see [**Section 9.9.1**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT) | date\_part('hour', timestamp '2001-02-16 20:38:40') | 20 |
| date\_part(text, interval) | double precision | Get subfield (equivalent to extract); see [**Section 9.9.1**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT) | date\_part('month', interval '2 years 3 months') | 3 |
| date\_trunc(text, timestamp) | timestamp | Truncate to specified precision; see also [**Section 9.9.2**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-TRUNC) | date\_trunc('hour', timestamp '2001-02-16 20:38:40') | 2001-02-16 20:00:00 |
| date\_trunc(text, interval) | interval | Truncate to specified precision; see also [**Section 9.9.2**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-TRUNC) | date\_trunc('hour', interval '2 days 3 hours 40 minutes') | 2 days 03:00:00 |
| extract(*field* from timestamp) | double precision | Get subfield; see [**Section 9.9.1**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT) | extract(hour from timestamp '2001-02-16 20:38:40') | 20 |
| extract(*field* from interval) | double precision | Get subfield; see [**Section 9.9.1**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-EXTRACT) | extract(month from interval '2 years 3 months') | 3 |
| isfinite(date) | boolean | Test for finite date (not +/-infinity) | isfinite(date '2001-02-16') | true |
| isfinite(timestamp) | boolean | Test for finite time stamp (not +/-infinity) | isfinite(timestamp '2001-02-16 21:28:30') | true |
| isfinite(interval) | boolean | Test for finite interval | isfinite(interval '4 hours') | true |
| justify\_days(interval) | interval | Adjust interval so 30-day time periods are represented as months | justify\_days(interval '35 days') | 1 mon 5 days |
| justify\_hours(interval) | interval | Adjust interval so 24-hour time periods are represented as days | justify\_hours(interval '27 hours') | 1 day 03:00:00 |
| justify\_interval(interval) | interval | Adjust interval using justify\_days and justify\_hours, with additional sign adjustments | justify\_interval(interval '1 mon -1 hour') | 29 days 23:00:00 |
| localtime | time | Current time of day; see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| localtimestamp | timestamp | Current date and time (start of current transaction); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| make\_date(*year* int, *month* int, *day* int) | date | Create date from year, month and day fields | make\_date(2013, 7, 15) | 2013-07-15 |
| make\_interval(*years* int DEFAULT 0, *months* int DEFAULT 0, *weeks* intDEFAULT 0, *days* int DEFAULT 0, *hours* int DEFAULT 0, *mins* int DEFAULT 0,*secs* double precision DEFAULT 0.0) | interval | Create interval from years, months, weeks, days, hours, minutes and seconds fields | make\_interval(days => 10) | 10 days |
| make\_time(*hour* int, *min* int, *sec* double precision) | time | Create time from hour, minute and seconds fields | make\_time(8, 15, 23.5) | 08:15:23.5 |
| make\_timestamp(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec*double precision) | timestamp | Create timestamp from year, month, day, hour, minute and seconds fields | make\_timestamp(2013, 7, 15, 8, 15, 23.5) | 2013-07-15 08:15:23.5 |
| make\_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec*double precision, [ *timezone* text ]) | timestamp with time zone | Create timestamp with time zone from year, month, day, hour, minute and seconds fields; if *timezone* is not specified, the current time zone is used | make\_timestamptz(2013, 7, 15, 8, 15, 23.5) | 2013-07-15 08:15:23.5+01 |
| now() | timestamp with time zone | Current date and time (start of current transaction); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| statement\_timestamp() | timestamp with time zone | Current date and time (start of current statement); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| timeofday() | text | Current date and time (like clock\_timestamp, but as a textstring); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| transaction\_timestamp() | timestamp with time zone | Current date and time (start of current transaction); see [**Section 9.9.4**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT) |  |  |
| to\_timestamp(double precision) | timestamp with time zone | Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp | to\_timestamp(1284352323) | 2010-09-13 04:32:03+00 |

In addition to these functions, the SQL OVERLAPS operator is supported:

(***start1***, ***end1***) OVERLAPS (***start2***, ***end2***)

(***start1***, ***length1***) OVERLAPS (***start2***, ***length2***)

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval. When a pair of values is provided, either the start or the end can be written first; OVERLAPS automatically takes the earlier value of the pair as the start. Each time period is considered to represent the half-open interval ***start*** <= ***time*** < ***end***, unless ***start*** and ***end*** are equal in which case it represents that single time instant. This means for instance that two time periods with only an endpoint in common do not overlap.

SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS

(DATE '2001-10-30', DATE '2002-10-30');

*Result:* true

SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS

(DATE '2001-10-30', DATE '2002-10-30');

*Result:* false

SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS

(DATE '2001-10-30', DATE '2001-10-31');

*Result:* false

SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS

(DATE '2001-10-30', DATE '2001-10-31');

*Result:* true

When adding an interval value to (or subtracting an interval value from) a timestamp with time zone value, the days component advances or decrements the date of the timestamp with time zoneby the indicated number of days. Across daylight saving time changes (when the session time zone is set to a time zone that recognizes DST), this means interval '1 day' does not necessarily equal interval '24 hours'. For example, with the session time zone set to CST7CDT, timestamp with time zone '2005-04-02 12:00-07' + interval '1 day' will produce timestamp with time zone '2005-04-03 12:00-06', while adding interval '24 hours' to the same initial timestamp with time zone produces timestamp with time zone '2005-04-03 13:00-06', as there is a change in daylight saving time at 2005-04-03 02:00 in time zone CST7CDT.

Note there can be ambiguity in the months field returned by age because different months have different numbers of days. PostgreSQL's approach uses the month from the earlier of the two dates when calculating partial months. For example, age('2004-06-01', '2004-04-30') uses April to yield 1 mon 1 day, while using May would yield 1 mon 2 days because May has 31 days, while April has only 30.

Subtraction of dates and timestamps can also be complex. One conceptually simple way to perform subtraction is to convert each value to a number of seconds using EXTRACT(EPOCH FROM ...), then subtract the results; this produces the number of seconds between the two values. This will adjust for the number of days in each month, timezone changes, and daylight saving time adjustments. Subtraction of date or timestamp values with the “-” operator returns the number of days (24-hours) and hours/minutes/seconds between the values, making the same adjustments. The age function returns years, months, days, and hours/minutes/seconds, performing field-by-field subtraction and then adjusting for negative field values. The following queries illustrate the differences in these approaches. The sample results were produced with timezone = 'US/Eastern'; there is a daylight saving time change between the two dates used:

SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -

EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');

*Result:* 10537200

SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -

EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))

/ 60 / 60 / 24;

*Result:* 121.958333333333

SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';

*Result:* 121 days 23:00:00

SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');

*Result:* 4 mons

### 9.9.1. EXTRACT, date\_part

EXTRACT(***field*** FROM ***source***)

The extract function retrieves subfields such as year or hour from date/time values. ***source*** must be a value expression of type timestamp, time, or interval. (Expressions of type date are cast to timestamp and can therefore be used as well.) ***field*** is an identifier or string that selects what field to extract from the source value. The extract function returns values of type double precision. The following are valid field names:

century

The century

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');

*Result:* 20

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 21

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 century to 1 century. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

day

For timestamp values, the day (of the month) field (1 - 31) ; for interval values, the number of days

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');

*Result:* 40

decade

The year field divided by 10

SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 200

dow

The day of the week as Sunday (0) to Saturday (6)

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 5

Note that extract's day of the week numbering differs from that of the to\_char(..., 'D') function.

doy

The day of the year (1 - 365/366)

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 47

epoch

For timestamp with time zone values, the number of seconds since 1970-01-01 00:00:00 UTC (can be negative); for date and timestamp values, the number of seconds since 1970-01-01 00:00:00 local time; for interval values, the total number of seconds in the interval

SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');

*Result:* 982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');

*Result:* 442800

You can convert an epoch value back to a time stamp with to\_timestamp:

SELECT to\_timestamp(982384720.12);

*Result:* 2001-02-17 04:38:40.12+00

hour

The hour field (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 20

isodow

The day of the week as Monday (1) to Sunday (7)

SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');

*Result:* 7

This is identical to dow except for Sunday. This matches the ISO 8601 day of the week numbering.

isoyear

The ISO 8601 week-numbering year that the date falls in (not applicable to intervals)

SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');

*Result:* 2005

SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');

*Result:* 2006

Each ISO 8601 week-numbering year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the week field for more information.

This field is not available in PostgreSQL releases prior to 8.3.

microseconds

The seconds field, including fractional parts, multiplied by 1 000 000; note that this includes full seconds

SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');

*Result:* 28500000

millennium

The millennium

SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 3

Years in the 1900s are in the second millennium. The third millennium started January 1, 2001.

milliseconds

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');

*Result:* 28500

minute

The minutes field (0 - 59)

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 38

month

For timestamp values, the number of the month within the year (1 - 12) ; for interval values, the number of months, modulo 12 (0 - 11)

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');

*Result:* 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');

*Result:* 1

quarter

The quarter of the year (1 - 4) that the date is in

SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 1

second

The seconds field, including fractional parts (0 - 59[**[7]**](https://www.postgresql.org/docs/10/functions-datetime.html#ftn.id-1.5.8.14.12.5.11.16.2.1.1))

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');

*Result:* 28.5

timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC. (Technically, PostgreSQL does not use UTC because leap seconds are not handled.)

timezone\_hour

The hour component of the time zone offset

timezone\_minute

The minute component of the time zone offset

week

The number of the ISO 8601 week-numbering week of the year. By definition, ISO weeks start on Mondays and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year.

In the ISO week-numbering system, it is possible for early-January dates to be part of the 52nd or 53rd week of the previous year, and for late-December dates to be part of the first week of the next year. For example, 2005-01-01 is part of the 53rd week of year 2004, and 2006-01-01 is part of the 52nd week of year 2005, while 2012-12-31 is part of the first week of 2013. It's recommended to use the isoyear field together with week to get consistent results.

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 7

year

The year field. Keep in mind there is no 0 AD, so subtracting BC years from AD years should be done with care.

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');

*Result:* 2001

Note

When the input value is +/-Infinity, extract returns +/-Infinity for monotonically-increasing fields (epoch, julian, year, isoyear, decade, century, and millennium). For other fields, NULL is returned. PostgreSQL versions before 9.6 returned zero for all cases of infinite input.

The extract function is primarily intended for computational processing. For formatting date/time values for display, see [**Section 9.8**](https://www.postgresql.org/docs/10/functions-formatting.html).

The date\_part function is modeled on the traditional Ingres equivalent to the SQL-standard function extract:

date\_part('***field***', ***source***)

Note that here the ***field*** parameter needs to be a string value, not a name. The valid field names for date\_part are the same as for extract.

SELECT date\_part('day', TIMESTAMP '2001-02-16 20:38:40');

*Result:* 16

SELECT date\_part('hour', INTERVAL '4 hours 3 minutes');

*Result:* 4

### 9.9.2. date\_trunc

The function date\_trunc is conceptually similar to the trunc function for numbers.

date\_trunc('***field***', ***source***)

***source*** is a value expression of type timestamp or interval. (Values of type date and time are cast automatically to timestamp or interval, respectively.) ***field*** selects to which precision to truncate the input value. The return value is of type timestamp or interval with all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for ***field*** are:

|  |
| --- |
| microseconds |
| milliseconds |
| second |
| minute |
| hour |
| day |
| week |
| month |
| quarter |
| year |
| decade |
| century |
| millennium |

Examples:

SELECT date\_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');

*Result:* 2001-02-16 20:00:00

SELECT date\_trunc('year', TIMESTAMP '2001-02-16 20:38:40');

*Result:* 2001-01-01 00:00:00

### 9.9.3. AT TIME ZONE

The AT TIME ZONE converts time stamp without time zone to/from time stamp with time zone, and time values to different time zones. [**Table 9.31**](https://www.postgresql.org/docs/10/functions-datetime.html#FUNCTIONS-DATETIME-ZONECONVERT-TABLE) shows its variants.

**Table 9.31.**AT TIME ZONE**Variants**

| **Expression** | **Return Type** | **Description** |
| --- | --- | --- |
| timestamp without time zone AT TIME ZONE ***zone*** | timestamp with time zone | Treat given time stamp without time zone as located in the specified time zone |
| timestamp with time zone AT TIME ZONE ***zone*** | timestamp without time zone | Convert given time stamp with time zone to the new time zone, with no time zone designation |
| time with time zone AT TIME ZONE ***zone*** | time with time zone | Convert given time with time zone to the new time zone |

In these expressions, the desired time zone ***zone*** can be specified either as a text string (e.g., 'America/Los\_Angeles') or as an interval (e.g., INTERVAL '-08:00'). In the text case, a time zone name can be specified in any of the ways described in [**Section 8.5.3**](https://www.postgresql.org/docs/10/datatype-datetime.html#DATATYPE-TIMEZONES).

Examples (assuming the local time zone is America/Los\_Angeles):

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';

*Result:* 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';

*Result:* 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago';

*Result:* 2001-02-16 05:38:40

The first example adds a time zone to a value that lacks it, and displays the value using the current TimeZone setting. The second example shifts the time stamp with time zone value to the specified time zone, and returns the value without a time zone. This allows storage and display of values different from the current TimeZone setting. The third example converts Tokyo time to Chicago time. Converting time values to other time zones uses the currently active time zone rules since no date is supplied.

The function timezone(***zone***, ***timestamp***) is equivalent to the SQL-conforming construct ***timestamp*** AT TIME ZONE ***zone***.

### 9.9.4. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

CURRENT\_DATE

CURRENT\_TIME

CURRENT\_TIMESTAMP

CURRENT\_TIME(***precision***)

CURRENT\_TIMESTAMP(***precision***)

LOCALTIME

LOCALTIMESTAMP

LOCALTIME(***precision***)

LOCALTIMESTAMP(***precision***)

CURRENT\_TIME and CURRENT\_TIMESTAMP deliver values with time zone; LOCALTIME and LOCALTIMESTAMP deliver values without time zone.

CURRENT\_TIME, CURRENT\_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP can optionally take a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

SELECT CURRENT\_TIME;

*Result:* 14:39:53.662522-05

SELECT CURRENT\_DATE;

*Result:* 2001-12-23

SELECT CURRENT\_TIMESTAMP;

*Result:* 2001-12-23 14:39:53.662522-05

SELECT CURRENT\_TIMESTAMP(2);

*Result:* 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;

*Result:* 2001-12-23 14:39:53.662522

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp.

Note

Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

transaction\_timestamp()

statement\_timestamp()

clock\_timestamp()

timeofday()

now()

transaction\_timestamp() is equivalent to CURRENT\_TIMESTAMP, but is named to clearly reflect what it returns. statement\_timestamp() returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). statement\_timestamp() and transaction\_timestamp() return the same value during the first command of a transaction, but might differ during subsequent commands. clock\_timestamp() returns the actual current time, and therefore its value changes even within a single SQL command. timeofday() is a historical PostgreSQL function. Like clock\_timestamp(), it returns the actual current time, but as a formatted text string rather than a timestamp with time zone value. now() is a traditional PostgreSQL equivalent to transaction\_timestamp().

All the date/time data types also accept the special literal value now to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

SELECT CURRENT\_TIMESTAMP;

SELECT now();

SELECT TIMESTAMP 'now'; -- incorrect for use with DEFAULT

Tip

You do not want to use the third form when specifying a DEFAULT clause while creating a table. The system will convert now to a timestamp as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion.

### 9.9.5. Delaying Execution

The following functions are available to delay execution of the server process:

pg\_sleep(***seconds***)

pg\_sleep\_for(interval)

pg\_sleep\_until(timestamp with time zone)

pg\_sleep makes the current session's process sleep until ***seconds*** seconds have elapsed. ***seconds*** is a value of type double precision, so fractional-second delays can be specified. pg\_sleep\_for is a convenience function for larger sleep times specified as an interval. pg\_sleep\_until is a convenience function for when a specific wake-up time is desired. For example:

SELECT pg\_sleep(1.5);

SELECT pg\_sleep\_for('5 minutes');

SELECT pg\_sleep\_until('tomorrow 03:00');

Note

The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on factors such as server load. In particular, pg\_sleep\_until is not guaranteed to wake up exactly at the specified time, but it will not wake up any earlier.

Warning

Make sure that your session does not hold more locks than necessary when calling pg\_sleep or its variants. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

[**[7]**](https://www.postgresql.org/docs/10/functions-datetime.html#id-1.5.8.14.12.5.11.16.2.1.1) 60 if leap seconds are implemented by the operating system

**9.10. Enum Support Functions**

For enum types (described in [**Section 8.7**](https://www.postgresql.org/docs/10/datatype-enum.html)), there are several functions that allow cleaner programming without hard-coding particular values of an enum type. These are listed in [**Table 9.32**](https://www.postgresql.org/docs/10/functions-enum.html#FUNCTIONS-ENUM-TABLE). The examples assume an enum type created as:

CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');

**Table 9.32. Enum Support Functions**

| **Function** | **Description** | **Example** | **Example Result** |
| --- | --- | --- | --- |
| enum\_first(anyenum) | Returns the first value of the input enum type | enum\_first(null::rainbow) | red |
| enum\_last(anyenum) | Returns the last value of the input enum type | enum\_last(null::rainbow) | purple |
| enum\_range(anyenum) | Returns all values of the input enum type in an ordered array | enum\_range(null::rainbow) | {red,orange,yellow,green,blue,purple} |
| enum\_range(anyenum, anyenum) | Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type. | enum\_range('orange'::rainbow, 'green'::rainbow) | {orange,yellow,green} |
| enum\_range(NULL, 'green'::rainbow) | {red,orange,yellow,green} |
| enum\_range('orange'::rainbow, NULL) | {orange,yellow,green,blue,purple} |

Notice that except for the two-argument form of enum\_range, these functions disregard the specific value passed to them; they care only about its declared data type. Either null or a specific value of the type can be passed, with the same result. It is more common to apply these functions to a table column or function argument than to a hardwired type name as suggested by the examples.

## 9.11. Geometric Functions and Operators

The geometric types point, box, lseg, line, path, polygon, and circle have a large set of native support functions and operators, shown in [**Table 9.33**](https://www.postgresql.org/docs/10/functions-geometry.html#FUNCTIONS-GEOMETRY-OP-TABLE), [**Table 9.34**](https://www.postgresql.org/docs/10/functions-geometry.html#FUNCTIONS-GEOMETRY-FUNC-TABLE), and [**Table 9.35**](https://www.postgresql.org/docs/10/functions-geometry.html#FUNCTIONS-GEOMETRY-CONV-TABLE).

Caution

Note that the “same as” operator, ~=, represents the usual notion of equality for the point, box, polygon, and circle types. Some of these types also have an = operator, but =compares for equal areas only. The other scalar comparison operators (<= and so on) likewise compare areas for these types.

**Table 9.33. Geometric Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + | Translation | box '((0,0),(1,1))' + point '(2.0,0)' |
| - | Translation | box '((0,0),(1,1))' - point '(2.0,0)' |
| \* | Scaling/rotation | box '((0,0),(1,1))' \* point '(2.0,0)' |
| / | Scaling/rotation | box '((0,0),(2,2))' / point '(2.0,0)' |
| # | Point or box of intersection | box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))' |
| # | Number of points in path or polygon | # path '((1,0),(0,1),(-1,0))' |
| @-@ | Length or circumference | @-@ path '((0,0),(1,0))' |
| @@ | Center | @@ circle '((0,0),10)' |
| ## | Closest point to first operand on second operand | point '(0,0)' ## lseg '((2,0),(0,2))' |
| <-> | Distance between | circle '((0,0),1)' <-> circle '((5,0),1)' |
| && | Overlaps? (One point in common makes this true.) | box '((0,0),(1,1))' && box '((0,0),(2,2))' |
| << | Is strictly left of? | circle '((0,0),1)' << circle '((5,0),1)' |
| >> | Is strictly right of? | circle '((5,0),1)' >> circle '((0,0),1)' |
| &< | Does not extend to the right of? | box '((0,0),(1,1))' &< box '((0,0),(2,2))' |
| &> | Does not extend to the left of? | box '((0,0),(3,3))' &> box '((0,0),(2,2))' |
| <<| | Is strictly below? | box '((0,0),(3,3))' <<| box '((3,4),(5,5))' |
| |>> | Is strictly above? | box '((3,4),(5,5))' |>> box '((0,0),(3,3))' |
| &<| | Does not extend above? | box '((0,0),(1,1))' &<| box '((0,0),(2,2))' |
| |&> | Does not extend below? | box '((0,0),(3,3))' |&> box '((0,0),(2,2))' |
| <^ | Is below (allows touching)? | circle '((0,0),1)' <^ circle '((0,5),1)' |
| >^ | Is above (allows touching)? | circle '((0,5),1)' >^ circle '((0,0),1)' |
| ?# | Intersects? | lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' |
| ?- | Is horizontal? | ?- lseg '((-1,0),(1,0))' |
| ?- | Are horizontally aligned? | point '(1,0)' ?- point '(0,0)' |
| ?| | Is vertical? | ?| lseg '((-1,0),(1,0))' |
| ?| | Are vertically aligned? | point '(0,1)' ?| point '(0,0)' |
| ?-| | Is perpendicular? | lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))' |
| ?|| | Are parallel? | lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))' |
| @> | Contains? | circle '((0,0),2)' @> point '(1,1)' |
| <@ | Contained in or on? | point '(1,1)' <@ circle '((0,0),2)' |
| ~= | Same as? | polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' |

Note

Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called ~and @. These names are still available, but are deprecated and will eventually be removed.

**Table 9.34. Geometric Functions**

| **Function** | **Return Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| area(***object***) | double precision | area | area(box '((0,0),(1,1))') |
| center(***object***) | point | center | center(box '((0,0),(1,2))') |
| diameter(circle) | double precision | diameter of circle | diameter(circle '((0,0),2.0)') |
| height(box) | double precision | vertical size of box | height(box '((0,0),(1,1))') |
| isclosed(path) | boolean | a closed path? | isclosed(path '((0,0),(1,1),(2,0))') |
| isopen(path) | boolean | an open path? | isopen(path '[(0,0),(1,1),(2,0)]') |
| length(***object***) | double precision | length | length(path '((-1,0),(1,0))') |
| npoints(path) | int | number of points | npoints(path '[(0,0),(1,1),(2,0)]') |
| npoints(polygon) | int | number of points | npoints(polygon '((1,1),(0,0))') |
| pclose(path) | path | convert path to closed | pclose(path '[(0,0),(1,1),(2,0)]') |
| popen(path) | path | convert path to open | popen(path '((0,0),(1,1),(2,0))') |
| radius(circle) | double precision | radius of circle | radius(circle '((0,0),2.0)') |
| width(box) | double precision | horizontal size of box | width(box '((0,0),(1,1))') |

**Table 9.35. Geometric Type Conversion Functions**

| **Function** | **Return Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| box(circle) | box | circle to box | box(circle '((0,0),2.0)') |
| box(point) | box | point to empty box | box(point '(0,0)') |
| box(point, point) | box | points to box | box(point '(0,0)', point '(1,1)') |
| box(polygon) | box | polygon to box | box(polygon '((0,0),(1,1),(2,0))') |
| bound\_box(box, box) | box | boxes to bounding box | bound\_box(box '((0,0),(1,1))', box '((3,3),(4,4))') |
| circle(box) | circle | box to circle | circle(box '((0,0),(1,1))') |
| circle(point, double precision) | circle | center and radius to circle | circle(point '(0,0)', 2.0) |
| circle(polygon) | circle | polygon to circle | circle(polygon '((0,0),(1,1),(2,0))') |
| line(point, point) | line | points to line | line(point '(-1,0)', point '(1,0)') |
| lseg(box) | lseg | box diagonal to line segment | lseg(box '((-1,0),(1,0))') |
| lseg(point, point) | lseg | points to line segment | lseg(point '(-1,0)', point '(1,0)') |
| path(polygon) | path | polygon to path | path(polygon '((0,0),(1,1),(2,0))') |
| point(double precision, double precision) | point | construct point | point(23.4, -44.5) |
| point(box) | point | center of box | point(box '((-1,0),(1,0))') |
| point(circle) | point | center of circle | point(circle '((0,0),2.0)') |
| point(lseg) | point | center of line segment | point(lseg '((-1,0),(1,0))') |
| point(polygon) | point | center of polygon | point(polygon '((0,0),(1,1),(2,0))') |
| polygon(box) | polygon | box to 4-point polygon | polygon(box '((0,0),(1,1))') |
| polygon(circle) | polygon | circle to 12-point polygon | polygon(circle '((0,0),2.0)') |
| polygon(***npts***, circle) | polygon | circle to ***npts***-point polygon | polygon(12, circle '((0,0),2.0)') |
| polygon(path) | polygon | path to polygon | polygon(path '((0,0),(1,1),(2,0))') |

It is possible to access the two component numbers of a point as though the point were an array with indexes 0 and 1. For example, if t.p is a point column then SELECT p[0] FROM t retrieves the X coordinate and UPDATE t SET p[1] = ... changes the Y coordinate. In the same way, a value of type box or lseg can be treated as an array of two point values.

The area function works for the types box, circle, and path. The area function only works on the path data type if the points in the path are non-intersecting. For example, the path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH will not work; however, the following visually identical path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH will work. If the concept of an intersecting versus non-intersecting path is confusing, draw both of the above paths side by side on a piece of graph paper.

**9.12. Network Address Functions and Operators**

[**Table 9.36**](https://www.postgresql.org/docs/10/functions-net.html#CIDR-INET-OPERATORS-TABLE) shows the operators available for the cidr and inet types. The operators <<, <<=, >>, >>=, and && test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any host part) and determine whether one network is identical to or a subnet of the other.

**Table 9.36.**cidr**and**inet**Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| < | is less than | inet '192.168.1.5' < inet '192.168.1.6' |
| <= | is less than or equal | inet '192.168.1.5' <= inet '192.168.1.5' |
| = | equals | inet '192.168.1.5' = inet '192.168.1.5' |
| >= | is greater or equal | inet '192.168.1.5' >= inet '192.168.1.5' |
| > | is greater than | inet '192.168.1.5' > inet '192.168.1.4' |
| <> | is not equal | inet '192.168.1.5' <> inet '192.168.1.4' |
| << | is contained by | inet '192.168.1.5' << inet '192.168.1/24' |
| <<= | is contained by or equals | inet '192.168.1/24' <<= inet '192.168.1/24' |
| >> | contains | inet '192.168.1/24' >> inet '192.168.1.5' |
| >>= | contains or equals | inet '192.168.1/24' >>= inet '192.168.1/24' |
| && | contains or is contained by | inet '192.168.1/24' && inet '192.168.1.80/28' |
| ~ | bitwise NOT | ~ inet '192.168.1.6' |
| & | bitwise AND | inet '192.168.1.6' & inet '0.0.0.255' |
| | | bitwise OR | inet '192.168.1.6' | inet '0.0.0.255' |
| + | addition | inet '192.168.1.6' + 25 |
| - | subtraction | inet '192.168.1.43' - 36 |
| - | subtraction | inet '192.168.1.43' - inet '192.168.1.19' |

[**Table 9.37**](https://www.postgresql.org/docs/10/functions-net.html#CIDR-INET-FUNCTIONS-TABLE) shows the functions available for use with the cidr and inet types. The abbrev, host, and text functions are primarily intended to offer alternative display formats.

**Table 9.37.**cidr**and**inet**Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| abbrev(inet) | text | abbreviated display format as text | abbrev(inet '10.1.0.0/16') | 10.1.0.0/16 |
| abbrev(cidr) | text | abbreviated display format as text | abbrev(cidr '10.1.0.0/16') | 10.1/16 |
| broadcast(inet) | inet | broadcast address for network | broadcast('192.168.1.5/24') | 192.168.1.255/24 |
| family(inet) | int | extract family of address; 4 for IPv4, 6 for IPv6 | family('::1') | 6 |
| host(inet) | text | extract IP address as text | host('192.168.1.5/24') | 192.168.1.5 |
| hostmask(inet) | inet | construct host mask for network | hostmask('192.168.23.20/30') | 0.0.0.3 |
| masklen(inet) | int | extract netmask length | masklen('192.168.1.5/24') | 24 |
| netmask(inet) | inet | construct netmask for network | netmask('192.168.1.5/24') | 255.255.255.0 |
| network(inet) | cidr | extract network part of address | network('192.168.1.5/24') | 192.168.1.0/24 |
| set\_masklen(inet, int) | inet | set netmask length for inet value | set\_masklen('192.168.1.5/24', 16) | 192.168.1.5/16 |
| set\_masklen(cidr, int) | cidr | set netmask length for cidr value | set\_masklen('192.168.1.0/24'::cidr, 16) | 192.168.0.0/16 |
| text(inet) | text | extract IP address and netmask length as text | text(inet '192.168.1.5') | 192.168.1.5/32 |
| inet\_same\_family(inet, inet) | boolean | are the addresses from the same family? | inet\_same\_family('192.168.1.5/24', '::1') | false |
| inet\_merge(inet, inet) | cidr | the smallest network which includes both of the given networks | inet\_merge('192.168.1.5/24', '192.168.2.5/24') | 192.168.0.0/22 |

Any cidr value can be cast to inet implicitly or explicitly; therefore, the functions shown above as operating on inet also work on cidr values. (Where there are separate functions for inet and cidr, it is because the behavior should be different for the two cases.) Also, it is permitted to cast an inet value to cidr. When this is done, any bits to the right of the netmask are silently zeroed to create a valid cidr value. In addition, you can cast a text value to inet or cidr using normal casting syntax: for example, inet(***expression***) or ***colname***::cidr.

[**Table 9.38**](https://www.postgresql.org/docs/10/functions-net.html#MACADDR-FUNCTIONS-TABLE) shows the functions available for use with the macaddr type. The function trunc(macaddr) returns a MAC address with the last 3 bytes set to zero. This can be used to associate the remaining prefix with a manufacturer.

**Table 9.38.**macaddr**Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| trunc(macaddr) | macaddr | set last 3 bytes to zero | trunc(macaddr '12:34:56:78:90:ab') | 12:34:56:00:00:00 |

The macaddr type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering, and the bitwise arithmetic operators (~, & and |) for NOT, AND and OR.

[**Table 9.39**](https://www.postgresql.org/docs/10/functions-net.html#MACADDR8-FUNCTIONS-TABLE) shows the functions available for use with the macaddr8 type. The function trunc(macaddr8) returns a MAC address with the last 5 bytes set to zero. This can be used to associate the remaining prefix with a manufacturer.

**Table 9.39.**macaddr8**Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| trunc(macaddr8) | macaddr8 | set last 5 bytes to zero | trunc(macaddr8 '12:34:56:78:90:ab:cd:ef') | 12:34:56:00:00:00:00:00 |
| macaddr8\_set7bit(macaddr8) | macaddr8 | set 7th bit to one, also known as modified EUI-64, for inclusion in an IPv6 address | macaddr8\_set7bit(macaddr8 '00:34:56:ab:cd:ef') | 02:34:56:ff:fe:ab:cd:ef |

The macaddr8 type also supports the standard relational operators (>, <=, etc.) for ordering, and the bitwise arithmetic operators (~, & and |) for NOT, AND and OR.

## 9.13. Text Search Functions and Operators

**[Table 9.40](https://www.postgresql.org/docs/10/functions-textsearch.html" \l "TEXTSEARCH-OPERATORS-TABLE" \o "Table 9.40. Text Search Operators)**, [**Table 9.41**](https://www.postgresql.org/docs/10/functions-textsearch.html#TEXTSEARCH-FUNCTIONS-TABLE) and [**Table 9.42**](https://www.postgresql.org/docs/10/functions-textsearch.html#TEXTSEARCH-FUNCTIONS-DEBUG-TABLE) summarize the functions and operators that are provided for full text searching. See [**Chapter 12**](https://www.postgresql.org/docs/10/textsearch.html) for a detailed explanation of PostgreSQL's text search facility.

**Table 9.40. Text Search Operators**

| **Operator** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| @@ | boolean | tsvector matches tsquery ? | to\_tsvector('fat cats ate rats') @@ to\_tsquery('cat & rat') | t |
| @@@ | boolean | deprecated synonym for @@ | to\_tsvector('fat cats ate rats') @@@ to\_tsquery('cat & rat') | t |
| || | tsvector | concatenate tsvectors | 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector | 'a':1 'b':2,5 'c':3 'd':4 |
| && | tsquery | AND tsquerys together | 'fat | rat'::tsquery && 'cat'::tsquery | ( 'fat' | 'rat' ) & 'cat' |
| || | tsquery | OR tsquerys together | 'fat | rat'::tsquery || 'cat'::tsquery | ( 'fat' | 'rat' ) | 'cat' |
| !! | tsquery | negate a tsquery | !! 'cat'::tsquery | !'cat' |
| <-> | tsquery | tsquery followed by tsquery | to\_tsquery('fat') <-> to\_tsquery('rat') | 'fat' <-> 'rat' |
| @> | boolean | tsquery contains another ? | 'cat'::tsquery @> 'cat & rat'::tsquery | f |
| <@ | boolean | tsquery is contained in ? | 'cat'::tsquery <@ 'cat & rat'::tsquery | t |

Note

The tsquery containment operators consider only the lexemes listed in the two queries, ignoring the combining operators.

In addition to the operators shown in the table, the ordinary B-tree comparison operators (=, <, etc) are defined for types tsvector and tsquery. These are not very useful for text searching but allow, for example, unique indexes to be built on columns of these types.

**Table 9.41. Text Search Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| array\_to\_tsvector(text[]) | tsvector | convert array of lexemes to tsvector | array\_to\_tsvector('{fat,cat,rat}'::text[]) | 'cat' 'fat' 'rat' |
| get\_current\_ts\_config() | regconfig | get default text search configuration | get\_current\_ts\_config() | english |
| length(tsvector) | integer | number of lexemes in tsvector | length('fat:2,4 cat:3 rat:5A'::tsvector) | 3 |
| numnode(tsquery) | integer | number of lexemes plus operators in tsquery | numnode('(fat & rat) | cat'::tsquery) | 5 |
| plainto\_tsquery([ ***config*** regconfig , ] ***query***text) | tsquery | produce tsquery ignoring punctuation | plainto\_tsquery('english', 'The Fat Rats') | 'fat' & 'rat' |
| phraseto\_tsquery([ ***config*** regconfig , ] ***query***text) | tsquery | produce tsquery that searches for a phrase, ignoring punctuation | phraseto\_tsquery('english', 'The Fat Rats') | 'fat' <-> 'rat' |
| querytree(***query*** tsquery) | text | get indexable part of a tsquery | querytree('foo & ! bar'::tsquery) | 'foo' |
| setweight(***vector*** tsvector, ***weight*** "char") | tsvector | assign ***weight*** to each element of ***vector*** | setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A') | 'cat':3A 'fat':2A,4A 'rat':5A |
| setweight(***vector*** tsvector, ***weight*** "char",***lexemes*** text[]) | tsvector | assign ***weight*** to elements of ***vector*** that are listed in ***lexemes*** | setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}') | 'cat':3A 'fat':2,4 'rat':5A |
| strip(tsvector) | tsvector | remove positions and weights from tsvector | strip('fat:2,4 cat:3 rat:5A'::tsvector) | 'cat' 'fat' 'rat' |
| to\_tsquery([ ***config*** regconfig , ] ***query*** text) | tsquery | normalize words and convert to tsquery | to\_tsquery('english', 'The & Fat & Rats') | 'fat' & 'rat' |
| to\_tsvector([ ***config*** regconfig , ] ***document***text) | tsvector | reduce document text to tsvector | to\_tsvector('english', 'The Fat Rats') | 'fat':2 'rat':3 |
| to\_tsvector([ ***config*** regconfig , ] ***document***json(b)) | tsvector | reduce each string value in the document to a tsvector, and then concatenate those in document order to produce a single tsvector | to\_tsvector('english', '{"a": "The Fat Rats"}'::json) | 'fat':2 'rat':3 |
| ts\_delete(***vector*** tsvector, ***lexeme*** text) | tsvector | remove given ***lexeme*** from ***vector*** | ts\_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat') | 'cat':3 'rat':5A |
| ts\_delete(***vector*** tsvector, ***lexemes*** text[]) | tsvector | remove any occurrence of lexemes in ***lexemes*** from ***vector*** | ts\_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat']) | 'cat':3 |
| ts\_filter(***vector*** tsvector, ***weights*** "char"[]) | tsvector | select only elements with given ***weights*** from ***vector*** | ts\_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}') | 'cat':3B 'rat':5A |
| ts\_headline([ ***config*** regconfig, ] ***document***text, ***query*** tsquery [, ***options*** text ]) | text | display a query match | ts\_headline('x y z', 'z'::tsquery) | x y <b>z</b> |
| ts\_headline([ ***config*** regconfig, ] ***document***json(b), ***query*** tsquery [, ***options*** text ]) | text | display a query match | ts\_headline('{"a":"x y z"}'::json, 'z'::tsquery) | {"a":"x y <b>z</b>"} |
| ts\_rank([ ***weights*** float4[], ] ***vector*** tsvector,***query*** tsquery [, ***normalization*** integer ]) | float4 | rank document for query | ts\_rank(textsearch, query) | 0.818 |
| ts\_rank\_cd([ ***weights*** float4[], ] ***vector***tsvector, ***query*** tsquery [, ***normalization***integer ]) | float4 | rank document for query using cover density | ts\_rank\_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query) | 2.01317 |
| ts\_rewrite(***query*** tsquery, ***target*** tsquery,***substitute*** tsquery) | tsquery | replace ***target*** with ***substitute*** within query | ts\_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery) | 'b' & ( 'foo' | 'bar' ) |
| ts\_rewrite(***query*** tsquery, ***select*** text) | tsquery | replace using targets and substitutes from a SELECT command | SELECT ts\_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases') | 'b' & ( 'foo' | 'bar' ) |
| tsquery\_phrase(***query1*** tsquery, ***query2*** tsquery) | tsquery | make query that searches for ***query1*** followed by ***query2*** (same as <-> operator) | tsquery\_phrase(to\_tsquery('fat'), to\_tsquery('cat')) | 'fat' <-> 'cat' |
| tsquery\_phrase(***query1*** tsquery, ***query2*** tsquery,***distance*** integer) | tsquery | make query that searches for ***query1*** followed by ***query2*** at distance ***distance*** | tsquery\_phrase(to\_tsquery('fat'), to\_tsquery('cat'), 10) | 'fat' <10> 'cat' |
| tsvector\_to\_array(tsvector) | text[] | convert tsvector to array of lexemes | tsvector\_to\_array('fat:2,4 cat:3 rat:5A'::tsvector) | {cat,fat,rat} |
| tsvector\_update\_trigger() | trigger | trigger function for automatic tsvector column update | CREATE TRIGGER ... tsvector\_update\_trigger(tsvcol, 'pg\_catalog.swedish', title, body) |  |
| tsvector\_update\_trigger\_column() | trigger | trigger function for automatic tsvector column update | CREATE TRIGGER ... tsvector\_update\_trigger\_column(tsvcol, configcol, title, body) |  |
| unnest(tsvector, OUT ***lexeme*** text, OUT ***positions***smallint[], OUT ***weights*** text) | setof record | expand a tsvector to a set of rows | unnest('fat:2,4 cat:3 rat:5A'::tsvector) | (cat,{3},{D}) ... |

Note

All the text search functions that accept an optional regconfig argument will use the configuration specified by [**default\_text\_search\_config**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DEFAULT-TEXT-SEARCH-CONFIG) when that argument is omitted.

The functions in [**Table 9.42**](https://www.postgresql.org/docs/10/functions-textsearch.html#TEXTSEARCH-FUNCTIONS-DEBUG-TABLE) are listed separately because they are not usually used in everyday text searching operations. They are helpful for development and debugging of new text search configurations.

**Table 9.42. Text Search Debugging Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| ts\_debug([ ***config*** regconfig, ] ***document*** text, OUT ***alias*** text, OUT ***description*** text, OUT ***token***text, OUT ***dictionaries*** regdictionary[], OUT ***dictionary*** regdictionary, OUT ***lexemes*** text[]) | setof record | test a configuration | ts\_debug('english', 'The Brightest supernovaes') | (asciiword,"Word, all ASCII",The,{english\_stem},english\_stem,{}) ... |
| ts\_lexize(***dict*** regdictionary, ***token*** text) | text[] | test a dictionary | ts\_lexize('english\_stem', 'stars') | {star} |
| ts\_parse(***parser\_name*** text, ***document*** text, OUT ***tokid*** integer, OUT ***token*** text) | setof record | test a parser | ts\_parse('default', 'foo - bar') | (1,foo) ... |
| ts\_parse(***parser\_oid*** oid, ***document*** text, OUT ***tokid*** integer, OUT ***token*** text) | setof record | test a parser | ts\_parse(3722, 'foo - bar') | (1,foo) ... |
| ts\_token\_type(***parser\_name*** text, OUT ***tokid*** integer, OUT ***alias*** text, OUT ***description*** text) | setof record | get token types defined by parser | ts\_token\_type('default') | (1,asciiword,"Word, all ASCII") ... |
| ts\_token\_type(***parser\_oid*** oid, OUT ***tokid*** integer, OUT ***alias*** text, OUT ***description*** text) | setof record | get token types defined by parser | ts\_token\_type(3722) | (1,asciiword,"Word, all ASCII") ... |
| ts\_stat(***sqlquery*** text, [ ***weights*** text, ] OUT ***word*** text, OUT ***ndoc*** integer, OUT ***nentry*** integer) | setof record | get statistics of a tsvector column | ts\_stat('SELECT vector from apod') | (foo,10,15) ... |

## 9.14. XML Functions

The functions and function-like expressions described in this section operate on values of type xml. Check [**Section 8.13**](https://www.postgresql.org/docs/10/datatype-xml.html) for information about the xml type. The function-like expressions xmlparse and xmlserialize for converting to and from type xml are not repeated here. Use of most of these functions requires the installation to have been built with configure --with-libxml.

### 9.14.1. Producing XML Content

A set of functions and function-like expressions are available for producing XML content from SQL data. As such, they are particularly suitable for formatting query results into XML documents for processing in client applications.

#### 9.14.1.1. Xmlcomment

xmlcomment(***text***)

The function xmlcomment creates an XML value containing an XML comment with the specified text as content. The text cannot contain “--” or end with a “-” so that the resulting construct is a valid XML comment. If the argument is null, the result is null.

Example:

SELECT xmlcomment('hello');

xmlcomment

--------------

<!--hello-->

#### 9.14.1.2. Xmlconcat

xmlconcat(***xml***[, ...])

The function xmlconcat concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no nonnull arguments.

Example:

SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

xmlconcat

----------------------

<abc/><bar>foo</bar>

XML declarations, if present, are combined as follows. If all argument values have the same XML version declaration, that version is used in the result, else no version is used. If all argument values have the standalone declaration value “yes”, then that value is used in the result. If all argument values have a standalone declaration value and at least one is “no”, then that is used in the result. Else the result will have no standalone declaration. If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration. Encoding declarations are ignored and removed in all cases.

Example:

SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

xmlconcat

-----------------------------------

<?xml version="1.1"?><foo/><bar/>

#### 9.14.1.3. Xmlelement

xmlelement(name ***name*** [, xmlattributes(***value*** [AS ***attname***] [, ... ])] [***, content, ...***])

The xmlelement expression produces an XML element with the given name, attributes, and content.

Examples:

SELECT xmlelement(name foo);

xmlelement

------------

<foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

xmlelement

------------------

<foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current\_date as bar), 'cont', 'ent');

xmlelement

-------------------------------------

<foo bar="2007-01-26">content</foo>

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence \_x***HHHH***\_, where ***HHHH*** is the character's Unicode codepoint in hexadecimal notation. For example:

SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

xmlelement

----------------------------------

<foo\_x0024\_bar a\_x0026\_b="xyz"/>

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column's name will be used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is valid:

CREATE TABLE test (a xml, b xml);

SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;

But these are not:

SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;

SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;

Element content, if specified, will be formatted according to its data type. If the content is itself of type xml, complex XML documents can be constructed. For example:

SELECT xmlelement(name foo, xmlattributes('xyz' as bar),

xmlelement(name abc),

xmlcomment('test'),

xmlelement(name xyz));

xmlelement

----------------------------------------------

<foo bar="xyz"><abc/><!--test--><xyz/></foo>

Content of other types will be formatted into valid XML character data. This means in particular that the characters <, >, and & will be converted to entities. Binary data (data type bytea) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter [**xmlbinary**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-XMLBINARY). The particular behavior for individual data types is expected to evolve in order to align the SQL and PostgreSQL data types with the XML Schema specification, at which point a more precise description will appear.

#### 9.14.1.4. Xmlforest

xmlforest(***content*** [AS ***name***] [, ...])

The xmlforest expression produces an XML forest (sequence) of elements using the given names and content.

Examples:

SELECT xmlforest('abc' AS foo, 123 AS bar);

xmlforest

------------------------------

<foo>abc</foo><bar>123</bar>

SELECT xmlforest(table\_name, column\_name)

FROM information\_schema.columns

WHERE table\_schema = 'pg\_catalog';

xmlforest

-------------------------------------------------------------------------------------------

<table\_name>pg\_authid</table\_name><column\_name>rolname</column\_name>

<table\_name>pg\_authid</table\_name><column\_name>rolsuper</column\_name>

...

As seen in the second example, the element name can be omitted if the content value is a column reference, in which case the column name is used by default. Otherwise, a name must be specified.

Element names that are not valid XML names are escaped as shown for xmlelement above. Similarly, content data is escaped to make valid XML content, unless it is already of type xml.

Note that XML forests are not valid XML documents if they consist of more than one element, so it might be useful to wrap xmlforest expressions in xmlelement.

#### 9.14.1.5. Xmlpi

xmlpi(name ***target*** [, ***content***])

The xmlpi expression creates an XML processing instruction. The content, if present, must not contain the character sequence ?>.

Example:

SELECT xmlpi(name php, 'echo "hello world";');

xmlpi

-----------------------------

<?php echo "hello world";?>

#### 9.14.1.6. Xmlroot

xmlroot(***xml***, version ***text*** | no value [, standalone yes|no|no value])

The xmlroot expression alters the properties of the root node of an XML value. If a version is specified, it replaces the value in the root node's version declaration; if a standalone setting is specified, it replaces the value in the root node's standalone declaration.

SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),

version '1.0', standalone yes);

xmlroot

----------------------------------------

<?xml version="1.0" standalone="yes"?>

<content>abc</content>

#### 9.14.1.7. Xmlagg

xmlagg(***xml***)

The function xmlagg is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, much like xmlconcat does, except that concatenation occurs across rows rather than across expressions in a single row. See [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html) for additional information about aggregate functions.

Example:

CREATE TABLE test (y int, x xml);

INSERT INTO test VALUES (1, '<foo>abc</foo>');

INSERT INTO test VALUES (2, '<bar/>');

SELECT xmlagg(x) FROM test;

xmlagg

----------------------

<foo>abc</foo><bar/>

To determine the order of the concatenation, an ORDER BY clause may be added to the aggregate call as described in [**Section 4.2.7**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-AGGREGATES). For example:

SELECT xmlagg(x ORDER BY y DESC) FROM test;

xmlagg

----------------------

<bar/><foo>abc</foo>

The following non-standard approach used to be recommended in previous versions, and may still be useful in specific cases:

SELECT xmlagg(x) FROM (SELECT \* FROM test ORDER BY y DESC) AS tab;

xmlagg

----------------------

<bar/><foo>abc</foo>

### 9.14.2. XML Predicates

The expressions described in this section check properties of xml values.

#### 9.14.2.1. IS DOCUMENT

***xml*** IS DOCUMENT

The expression IS DOCUMENT returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null. See [**Section 8.13**](https://www.postgresql.org/docs/10/datatype-xml.html) about the difference between documents and content fragments.

#### 9.14.2.2. IS NOT DOCUMENT

***xml*** IS NOT DOCUMENT

The expression IS NOT DOCUMENT returns false if the argument XML value is a proper XML document, true if it is not (that is, it is a content fragment), or null if the argument is null.

#### 9.14.2.3. XMLEXISTS

XMLEXISTS(***text*** PASSING [BY REF] ***xml*** [BY REF])

The function xmlexists returns true if the XPath expression in the first argument returns any nodes, and false otherwise. (If either argument is null, the result is null.)

Example:

SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF '<towns><town>Toronto</town><town>Ottawa</town></towns>');

xmlexists

------------

t

(1 row)

The BY REF clauses have no effect in PostgreSQL, but are allowed for SQL conformance and compatibility with other implementations. Per SQL standard, the first BY REF is required, the second is optional. Also note that the SQL standard specifies the xmlexists construct to take an XQuery expression as first argument, but PostgreSQL currently only supports XPath, which is a subset of XQuery.

#### 9.14.2.4. Xml\_is\_well\_formed

xml\_is\_well\_formed(***text***)

xml\_is\_well\_formed\_document(***text***)

xml\_is\_well\_formed\_content(***text***)

These functions check whether a text string is well-formed XML, returning a Boolean result. xml\_is\_well\_formed\_document checks for a well-formed document, while xml\_is\_well\_formed\_contentchecks for well-formed content. xml\_is\_well\_formed does the former if the [**xmloption**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-XMLOPTION) configuration parameter is set to DOCUMENT, or the latter if it is set to CONTENT. This means that xml\_is\_well\_formedis useful for seeing whether a simple cast to type xml will succeed, whereas the other two functions are useful for seeing whether the corresponding variants of XMLPARSE will succeed.

Examples:

SET xmloption TO DOCUMENT;

SELECT xml\_is\_well\_formed('<>');

xml\_is\_well\_formed

--------------------

f

(1 row)

SELECT xml\_is\_well\_formed('<abc/>');

xml\_is\_well\_formed

--------------------

t

(1 row)

SET xmloption TO CONTENT;

SELECT xml\_is\_well\_formed('abc');

xml\_is\_well\_formed

--------------------

t

(1 row)

SELECT xml\_is\_well\_formed\_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</pg:foo>');

xml\_is\_well\_formed\_document

-----------------------------

t

(1 row)

SELECT xml\_is\_well\_formed\_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</my:foo>');

xml\_is\_well\_formed\_document

-----------------------------

f

(1 row)

The last example shows that the checks include whether namespaces are correctly matched.

### 9.14.3. Processing XML

To process values of data type xml, PostgreSQL offers the functions xpath and xpath\_exists, which evaluate XPath 1.0 expressions, and the XMLTABLE table function.

#### 9.14.3.1. Xpath

xpath(***xpath***, ***xml*** [, ***nsarray***])

The function xpath evaluates the XPath expression ***xpath*** (a text value) against the XML value ***xml***. It returns an array of XML values corresponding to the node set produced by the XPath expression. If the XPath expression returns a scalar value rather than a node set, a single-element array is returned.

The second argument must be a well formed XML document. In particular, it must have a single root node element.

The optional third argument of the function is an array of namespace mappings. This array should be a two-dimensional text array with the length of the second axis being equal to 2 (i.e., it should be an array of arrays, each of which consists of exactly 2 elements). The first element of each array entry is the namespace name (alias), the second the namespace URI. It is not required that aliases provided in this array be the same as those being used in the XML document itself (in other words, both in the XML document and in the xpath function context, aliases are local).

Example:

SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',

ARRAY[ARRAY['my', 'http://example.com']]);

xpath

--------

{test}

(1 row)

To deal with default (anonymous) namespaces, do something like this:

SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',

ARRAY[ARRAY['mydefns', 'http://example.com']]);

xpath

--------

{test}

(1 row)

#### 9.14.3.2. Xpath\_exists

xpath\_exists(***xpath***, ***xml*** [, ***nsarray***])

The function xpath\_exists is a specialized form of the xpath function. Instead of returning the individual XML values that satisfy the XPath, this function returns a Boolean indicating whether the query was satisfied or not. This function is equivalent to the standard XMLEXISTS predicate, except that it also offers support for a namespace mapping argument.

Example:

SELECT xpath\_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',

ARRAY[ARRAY['my', 'http://example.com']]);

xpath\_exists

--------------

t

(1 row)

#### 9.14.3.3. Xmltable

xmltable( [XMLNAMESPACES(***namespace uri*** AS ***namespace name***[, ...]), ]

***row\_expression*** PASSING [BY REF] ***document\_expression*** [BY REF]

COLUMNS ***name*** { ***type*** [PATH ***column\_expression***] [DEFAULT ***default\_expression***] [NOT NULL | NULL]

| FOR ORDINALITY }

[, ...]

)

The xmltable function produces a table based on the given XML value, an XPath filter to extract rows, and an optional set of column definitions.

The optional XMLNAMESPACES clause is a comma-separated list of namespaces. It specifies the XML namespaces used in the document and their aliases. A default namespace specification is not currently supported.

The required ***row\_expression*** argument is an XPath expression that is evaluated against the supplied XML document to obtain an ordered sequence of XML nodes. This sequence is what xmltabletransforms into output rows.

***document\_expression*** provides the XML document to operate on. The BY REF clauses have no effect in PostgreSQL, but are allowed for SQL conformance and compatibility with other implementations. The argument must be a well-formed XML document; fragments/forests are not accepted.

The mandatory COLUMNS clause specifies the list of columns in the output table. If the COLUMNS clause is omitted, the rows in the result set contain a single column of type xml containing the data matched by ***row\_expression***. If COLUMNS is specified, each entry describes a single column. See the syntax summary above for the format. The column name and type are required; the path, default and nullability clauses are optional.

A column marked FOR ORDINALITY will be populated with row numbers matching the order in which the output rows appeared in the original input XML document. At most one column may be marked FOR ORDINALITY.

The column\_expression for a column is an XPath expression that is evaluated for each row, relative to the result of the ***row\_expression***, to find the value of the column. If no column\_expression is given, then the column name is used as an implicit path.

If a column's XPath expression returns multiple elements, an error is raised. If the expression matches an empty tag, the result is an empty string (not NULL). Any xsi:nil attributes are ignored.

The text body of the XML matched by the ***column\_expression*** is used as the column value. Multiple text() nodes within an element are concatenated in order. Any child elements, processing instructions, and comments are ignored, but the text contents of child elements are concatenated to the result. Note that the whitespace-only text() node between two non-text elements is preserved, and that leading whitespace on a text() node is not flattened.

If the path expression does not match for a given row but ***default\_expression*** is specified, the value resulting from evaluating that expression is used. If no DEFAULT clause is given for the column, the field will be set to NULL. It is possible for a ***default\_expression*** to reference the value of output columns that appear prior to it in the column list, so the default of one column may be based on the value of another column.

Columns may be marked NOT NULL. If the ***column\_expression*** for a NOT NULL column does not match anything and there is no DEFAULT or the ***default\_expression*** also evaluates to null, an error is reported.

Unlike regular PostgreSQL functions, ***column\_expression*** and ***default\_expression*** are not evaluated to a simple value before calling the function. ***column\_expression*** is normally evaluated exactly once per input row, and ***default\_expression*** is evaluated each time a default is needed for a field. If the expression qualifies as stable or immutable the repeat evaluation may be skipped. Effectively xmltable behaves more like a subquery than a function call. This means that you can usefully use volatile functions like nextval in ***default\_expression***, and ***column\_expression*** may depend on other parts of the XML document.

Examples:

CREATE TABLE xmldata AS SELECT

xml $$

<ROWS>

<ROW id="1">

<COUNTRY\_ID>AU</COUNTRY\_ID>

<COUNTRY\_NAME>Australia</COUNTRY\_NAME>

</ROW>

<ROW id="5">

<COUNTRY\_ID>JP</COUNTRY\_ID>

<COUNTRY\_NAME>Japan</COUNTRY\_NAME>

<PREMIER\_NAME>Shinzo Abe</PREMIER\_NAME>

<SIZE unit="sq\_mi">145935</SIZE>

</ROW>

<ROW id="6">

<COUNTRY\_ID>SG</COUNTRY\_ID>

<COUNTRY\_NAME>Singapore</COUNTRY\_NAME>

<SIZE unit="sq\_km">697</SIZE>

</ROW>

</ROWS>

$$ AS data;

SELECT xmltable.\*

FROM xmldata,

XMLTABLE('//ROWS/ROW'

PASSING data

COLUMNS id int PATH '@id',

ordinality FOR ORDINALITY,

"COUNTRY\_NAME" text,

country\_id text PATH 'COUNTRY\_ID',

size\_sq\_km float PATH 'SIZE[@unit = "sq\_km"]',

size\_other text PATH

'concat(SIZE[@unit!="sq\_km"], " ", SIZE[@unit!="sq\_km"]/@unit)',

premier\_name text PATH 'PREMIER\_NAME' DEFAULT 'not specified') ;

id | ordinality | COUNTRY\_NAME | country\_id | size\_sq\_km | size\_other | premier\_name

----+------------+--------------+------------+------------+--------------+---------------

1 | 1 | Australia | AU | | | not specified

5 | 2 | Japan | JP | | 145935 sq\_mi | Shinzo Abe

6 | 3 | Singapore | SG | 697 | | not specified

The following example shows concatenation of multiple text() nodes, usage of the column name as XPath filter, and the treatment of whitespace, XML comments and processing instructions:

CREATE TABLE xmlelements AS SELECT

xml $$

<root>

<element> Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x--> bbb<x>xxx</x>CC </element>

</root>

$$ AS data;

SELECT xmltable.\*

FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);

element

----------------------

Hello2a2 bbbCC

The following example illustrates how the XMLNAMESPACES clause can be used to specify a list of namespaces used in the XML document as well as in the XPath expressions:

WITH xmldata(data) AS (VALUES ('

<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">

<item foo="1" B:bar="2"/>

<item foo="3" B:bar="4"/>

<item foo="4" B:bar="5"/>

</example>'::xml)

)

SELECT xmltable.\*

FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,

'http://example.com/b' AS "B"),

'/x:example/x:item'

PASSING (SELECT data FROM xmldata)

COLUMNS foo int PATH '@foo',

bar int PATH '@B:bar');

foo | bar

-----+-----

1 | 2

3 | 4

4 | 5

(3 rows)

### 9.14.4. Mapping Tables to XML

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality:

table\_to\_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)

query\_to\_xml(query text, nulls boolean, tableforest boolean, targetns text)

cursor\_to\_xml(cursor refcursor, count int, nulls boolean,

tableforest boolean, targetns text)

The return type of each function is xml.

table\_to\_xml maps the content of the named table, passed as parameter *tbl*. The regclass type accepts strings identifying tables using the usual notation, including optional schema qualifications and double quotes. query\_to\_xml executes the query whose text is passed as parameter *query* and maps the result set. cursor\_to\_xml fetches the indicated number of rows from the cursor specified by the parameter *cursor*. This variant is recommended if large tables have to be mapped, because the result value is built up in memory by each function.

If *tableforest* is false, then the resulting XML document looks like this:

<tablename>

<row>

<columnname1>data</columnname1>

<columnname2>data</columnname2>

</row>

<row>

...

</row>

...

</tablename>

If *tableforest* is true, the result is an XML content fragment that looks like this:

<tablename>

<columnname1>data</columnname1>

<columnname2>data</columnname2>

</tablename>

<tablename>

...

</tablename>

...

If no table name is available, that is, when mapping a query or a cursor, the string table is used in the first format, row in the second format.

The choice between these formats is up to the user. The first format is a proper XML document, which will be important in many applications. The second format tends to be more useful in the cursor\_to\_xml function if the result values are to be reassembled into one document later on. The functions for producing XML content discussed above, in particular xmlelement, can be used to alter the results to taste.

The data values are mapped in the same way as described for the function xmlelement above.

The parameter *nulls* determines whether null values should be included in the output. If true, null values in columns are represented as:

<columnname xsi:nil="true"/>

where xsi is the XML namespace prefix for XML Schema Instance. An appropriate namespace declaration will be added to the result value. If false, columns containing null values are simply omitted from the output.

The parameter *targetns* specifies the desired XML namespace of the result. If no particular namespace is wanted, an empty string should be passed.

The following functions return XML Schema documents describing the mappings performed by the corresponding functions above:

table\_to\_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

query\_to\_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

cursor\_to\_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)

It is essential that the same parameters are passed in order to obtain matching XML data mappings and XML Schema documents.

The following functions produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together. They can be useful where self-contained and self-describing results are wanted:

table\_to\_xml\_and\_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

query\_to\_xml\_and\_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

In addition, the following functions are available to produce analogous mappings of entire schemas or the entire current database:

schema\_to\_xml(schema name, nulls boolean, tableforest boolean, targetns text)

schema\_to\_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

schema\_to\_xml\_and\_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database\_to\_xml(nulls boolean, tableforest boolean, targetns text)

database\_to\_xmlschema(nulls boolean, tableforest boolean, targetns text)

database\_to\_xml\_and\_xmlschema(nulls boolean, tableforest boolean, targetns text)

Note that these potentially produce a lot of data, which needs to be built up in memory. When requesting content mappings of large schemas or databases, it might be worthwhile to consider mapping the tables separately instead, possibly even through a cursor.

The result of a schema content mapping looks like this:

<schemaname>

table1-mapping

table2-mapping

...

</schemaname>

where the format of a table mapping depends on the *tableforest* parameter as explained above.

The result of a database content mapping looks like this:

<dbname>

<schema1name>

...

</schema1name>

<schema2name>

...

</schema2name>

...

</dbname>

where the schema mapping is as above.

As an example of using the output produced by these functions, [**Figure 9.1**](https://www.postgresql.org/docs/10/functions-xml.html#XSLT-XML-HTML) shows an XSLT stylesheet that converts the output of table\_to\_xml\_and\_xmlschema to an HTML document containing a tabular rendition of the table data. In a similar manner, the results from these functions can be converted into other XML-based formats.

**Figure 9.1. XSLT Stylesheet for Converting SQL/XML Output to HTML**

<?xml version="1.0"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns="http://www.w3.org/1999/xhtml"

>

<xsl:output method="xml"

doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"

doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"

indent="yes"/>

<xsl:template match="/\*">

<xsl:variable name="schema" select="//xsd:schema"/>

<xsl:variable name="tabletypename"

select="$schema/xsd:element[@name=name(current())]/@type"/>

<xsl:variable name="rowtypename"

select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:element[@name='row']/@type"/>

<html>

<head>

<title><xsl:value-of select="name(current())"/></title>

</head>

<body>

<table>

<tr>

<xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/xsd:element/@name">

<th><xsl:value-of select="."/></th>

</xsl:for-each>

</tr>

<xsl:for-each select="row">

<tr>

<xsl:for-each select="\*">

<td><xsl:value-of select="."/></td>

</xsl:for-each>

</tr>

</xsl:for-each>

</table>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

## 9.15. JSON Functions and Operators

**[Table 9.43](https://www.postgresql.org/docs/10/functions-json.html" \l "FUNCTIONS-JSON-OP-TABLE" \o "Table 9.43. json and jsonb Operators)** shows the operators that are available for use with the two JSON data types (see [**Section 8.14**](https://www.postgresql.org/docs/10/datatype-json.html)).

**Table 9.43.**json**and**jsonb**Operators**

| **Operator** | **Right Operand Type** | **Description** | **Example** | **Example Result** |
| --- | --- | --- | --- | --- |
| -> | int | Get JSON array element (indexed from zero, negative integers count from the end) | '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2 | {"c":"baz"} |
| -> | text | Get JSON object field by key | '{"a": {"b":"foo"}}'::json->'a' | {"b":"foo"} |
| ->> | int | Get JSON array element as text | '[1,2,3]'::json->>2 | 3 |
| ->> | text | Get JSON object field as text | '{"a":1,"b":2}'::json->>'b' | 2 |
| #> | text[] | Get JSON object at specified path | '{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}' | {"c": "foo"} |
| #>> | text[] | Get JSON object at specified path as text | '{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}' | 3 |

Note

There are parallel variants of these operators for both the json and jsonb types. The field/element/path extraction operators return the same type as their left-hand input (either json or jsonb), except for those specified as returning text, which coerce the value to text. The field/element/path extraction operators return NULL, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such element exists. The field/element/path extraction operators that accept integer JSON array subscripts all support negative subscripting from the end of arrays.

The standard comparison operators shown in [**Table 9.1**](https://www.postgresql.org/docs/10/functions-comparison.html#FUNCTIONS-COMPARISON-OP-TABLE) are available for jsonb, but not for json. They follow the ordering rules for B-tree operations outlined at [**Section 8.14.4**](https://www.postgresql.org/docs/10/datatype-json.html#JSON-INDEXING).

Some further operators also exist only for jsonb, as shown in [**Table 9.44**](https://www.postgresql.org/docs/10/functions-json.html#FUNCTIONS-JSONB-OP-TABLE). Many of these operators can be indexed by jsonb operator classes. For a full description of jsonb containment and existence semantics, see [**Section 8.14.3**](https://www.postgresql.org/docs/10/datatype-json.html#JSON-CONTAINMENT). [**Section 8.14.4**](https://www.postgresql.org/docs/10/datatype-json.html#JSON-INDEXING) describes how these operators can be used to effectively index jsonb data.

**Table 9.44. Additional**jsonb**Operators**

| **Operator** | **Right Operand Type** | **Description** | **Example** |
| --- | --- | --- | --- |
| @> | jsonb | Does the left JSON value contain the right JSON path/value entries at the top level? | '{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb |
| <@ | jsonb | Are the left JSON path/value entries contained at the top level within the right JSON value? | '{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb |
| ? | text | Does the string exist as a top-level key within the JSON value? | '{"a":1, "b":2}'::jsonb ? 'b' |
| ?| | text[] | Do any of these array strings exist as top-level keys? | '{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c'] |
| ?& | text[] | Do all of these array strings exist as top-level keys? | '["a", "b"]'::jsonb ?& array['a', 'b'] |
| || | jsonb | Concatenate two jsonb values into a new jsonb value | '["a", "b"]'::jsonb || '["c", "d"]'::jsonb |
| - | text | Delete key/value pair or string element from left operand. Key/value pairs are matched based on their key value. | '{"a": "b"}'::jsonb - 'a' |
| - | text[] | Delete multiple key/value pairs or string elements from left operand. Key/value pairs are matched based on their key value. | '{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[] |
| - | integer | Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array. | '["a", "b"]'::jsonb - 1 |
| #- | text[] | Delete the field or element with specified path (for JSON arrays, negative integers count from the end) | '["a", {"b":1}]'::jsonb #- '{1,b}' |

Note

The || operator concatenates the elements at the top level of each of its operands. It does not operate recursively. For example, if both operands are objects with a common key field name, the value of the field in the result will just be the value from the right hand operand.

[**Table 9.45**](https://www.postgresql.org/docs/10/functions-json.html#FUNCTIONS-JSON-CREATION-TABLE) shows the functions that are available for creating json and jsonb values. (There are no equivalent functions for jsonb, of the row\_to\_json and array\_to\_json functions. However, the to\_jsonb function supplies much the same functionality as these functions would.)

**Table 9.45. JSON Creation Functions**

| **Function** | **Description** | **Example** | **Example Result** |
| --- | --- | --- | --- |
| to\_json(anyelement)  to\_jsonb(anyelement) | Returns the value as json or jsonb. Arrays and composites are converted (recursively) to arrays and objects; otherwise, if there is a cast from the type to json, the cast function will be used to perform the conversion; otherwise, a scalar value is produced. For any scalar type other than a number, a Boolean, or a null value, the text representation will be used, in such a fashion that it is a valid json or jsonb value. | to\_json('Fred said "Hi."'::text) | "Fred said \"Hi.\"" |
| array\_to\_json(anyarray [, pretty\_bool]) | Returns the array as a JSON array. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if *pretty\_bool* is true. | array\_to\_json('{{1,5},{99,100}}'::int[]) | [[1,5],[99,100]] |
| row\_to\_json(record [, pretty\_bool]) | Returns the row as a JSON object. Line feeds will be added between level-1 elements if *pretty\_bool* is true. | row\_to\_json(row(1,'foo')) | {"f1":1,"f2":"foo"} |
| json\_build\_array(VARIADIC "any")  jsonb\_build\_array(VARIADIC "any") | Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list. | json\_build\_array(1,2,'3',4,5) | [1, 2, "3", 4, 5] |
| json\_build\_object(VARIADIC "any")  jsonb\_build\_object(VARIADIC "any") | Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating keys and values. | json\_build\_object('foo',1,'bar',2) | {"foo": 1, "bar": 2} |
| json\_object(text[])  jsonb\_object(text[]) | Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating key/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a key/value pair. | json\_object('{a, 1, b, "def", c, 3.5}')  json\_object('{{a, 1},{b, "def"},{c, 3.5}}') | {"a": "1", "b": "def", "c": "3.5"} |
| json\_object(keys text[], values text[])  jsonb\_object(keys text[], values text[]) | This form of json\_object takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form. | json\_object('{a, b}', '{1,2}') | {"a": "1", "b": "2"} |

Note

array\_to\_json and row\_to\_json have the same behavior as to\_json except for offering a pretty-printing option. The behavior described for to\_json likewise applies to each individual value converted by the other JSON creation functions.

Note

The [**hstore**](https://www.postgresql.org/docs/10/hstore.html) extension has a cast from hstore to json, so that hstore values converted via the JSON creation functions will be represented as JSON objects, not as primitive string values.

[**Table 9.46**](https://www.postgresql.org/docs/10/functions-json.html#FUNCTIONS-JSON-PROCESSING-TABLE) shows the functions that are available for processing json and jsonb values.

**Table 9.46. JSON Processing Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Example Result** |
| --- | --- | --- | --- | --- |
| json\_array\_length(json)  jsonb\_array\_length(jsonb) | int | Returns the number of elements in the outermost JSON array. | json\_array\_length('[1,2,3,{"f1":1,"f2":[5,6]},4]') | 5 |
| json\_each(json)  jsonb\_each(jsonb) | setof key text, value json  setof key text, value jsonb | Expands the outermost JSON object into a set of key/value pairs. | select \* from json\_each('{"a":"foo", "b":"bar"}') | key | value  -----+-------  a | "foo"  b | "bar" |
| json\_each\_text(json)  jsonb\_each\_text(jsonb) | setof key text, value text | Expands the outermost JSON object into a set of key/value pairs. The returned values will be of type text. | select \* from json\_each\_text('{"a":"foo", "b":"bar"}') | key | value  -----+-------  a | foo  b | bar |
| json\_extract\_path(from\_json json, VARIADIC path\_elems text[])  jsonb\_extract\_path(from\_json jsonb, VARIADIC path\_elems text[]) | json  jsonb | Returns JSON value pointed to by ***path\_elems*** (equivalent to #> operator). | json\_extract\_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4') | {"f5":99,"f6":"foo"} |
| json\_extract\_path\_text(from\_json json, VARIADIC path\_elems text[])  jsonb\_extract\_path\_text(from\_json jsonb, VARIADIC path\_elems text[]) | text | Returns JSON value pointed to by ***path\_elems*** as text (equivalent to #>>operator). | json\_extract\_path\_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}','f4', 'f6') | foo |
| json\_object\_keys(json)  jsonb\_object\_keys(jsonb) | setof text | Returns set of keys in the outermost JSON object. | json\_object\_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}') | json\_object\_keys  ------------------  f1  f2 |
| json\_populate\_record(base anyelement, from\_json json)  jsonb\_populate\_record(base anyelement, from\_json jsonb) | anyelement | Expands the object in ***from\_json*** to a row whose columns match the record type defined by ***base*** (see note below). | select \* from json\_populate\_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}') | a | b | c  ---+-----------+-------------  1 | {2,"a b"} | (4,"a b c") |
| json\_populate\_recordset(base anyelement, from\_json json)  jsonb\_populate\_recordset(base anyelement, from\_json jsonb) | setof anyelement | Expands the outermost array of objects in ***from\_json*** to a set of rows whose columns match the record type defined by ***base***(see note below). | select \* from json\_populate\_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]') | a | b  ---+---  1 | 2  3 | 4 |
| json\_array\_elements(json)  jsonb\_array\_elements(jsonb) | setof json  setof jsonb | Expands a JSON array to a set of JSON values. | select \* from json\_array\_elements('[1,true, [2,false]]') | value  -----------  1  true  [2,false] |
| json\_array\_elements\_text(json)  jsonb\_array\_elements\_text(jsonb) | setof text | Expands a JSON array to a set of textvalues. | select \* from json\_array\_elements\_text('["foo", "bar"]') | value  -----------  foo  bar |
| json\_typeof(json)  jsonb\_typeof(jsonb) | text | Returns the type of the outermost JSON value as a text string. Possible types are object, array, string, number, boolean, and null. | json\_typeof('-123.4') | number |
| json\_to\_record(json)  jsonb\_to\_record(jsonb) | record | Builds an arbitrary record from a JSON object (see note below). As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause. | select \* from json\_to\_record('{"a":1,"b":[1,2,3],"c":[1,2,3],"e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype) | a | b | c | d | r  ---+---------+---------+---+---------------  1 | [1,2,3] | {1,2,3} | | (123,"a b c") |
| json\_to\_recordset(json)  jsonb\_to\_recordset(jsonb) | setof record | Builds an arbitrary set of records from a JSON array of objects (see note below). As with all functions returning record, the caller must explicitly define the structure of the record with an AS clause. | select \* from json\_to\_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]') as x(a int, b text); | a | b  ---+-----  1 | foo  2 | |
| json\_strip\_nulls(from\_json json)  jsonb\_strip\_nulls(from\_json jsonb) | json  jsonb | Returns ***from\_json*** with all object fields that have null values omitted. Other null values are untouched. | json\_strip\_nulls('[{"f1":1,"f2":null},2,null,3]') | [{"f1":1},2,null,3] |
| jsonb\_set(target jsonb, path text[], new\_value jsonb[,*create\_missing* boolean]) | jsonb | Returns ***target*** with the section designated by ***path*** replaced by ***new\_value***, or with ***new\_value*** added if ***create\_missing*** is true ( default is true) and the item designated by ***path*** does not exist. As with the path orientated operators, negative integers that appear in ***path*** count from the end of JSON arrays. | jsonb\_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false)  jsonb\_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]') | [{"f1":[2,3,4],"f2":null},2,null,3]  [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2] |
| jsonb\_insert(target jsonb, path text[], new\_value jsonb, [*insert\_after* boolean]) | jsonb | Returns ***target*** with ***new\_value*** inserted. If ***target*** section designated by ***path*** is in a JSONB array, ***new\_value*** will be inserted before target or after if ***insert\_after*** is true (default is false). If ***target*** section designated by ***path*** is in JSONB object, ***new\_value*** will be inserted only if ***target***does not exist. As with the path orientated operators, negative integers that appear in ***path*** count from the end of JSON arrays. | jsonb\_insert('{"a": [0,1,2]}', '{a, 1}', '"new\_value"')  jsonb\_insert('{"a": [0,1,2]}', '{a, 1}', '"new\_value"', true) | {"a": [0, "new\_value", 1, 2]}  {"a": [0, 1, "new\_value", 2]} |
| jsonb\_pretty(from\_json jsonb) | text | Returns ***from\_json*** as indented JSON text. | jsonb\_pretty('[{"f1":1,"f2":null},2,null,3]') | [  {  "f1": 1,  "f2": null  },  2,  null,  3  ] |

Note

Many of these functions and operators will convert Unicode escapes in JSON strings to the appropriate single character. This is a non-issue if the input is type jsonb, because the conversion was already done; but for json input, this may result in throwing an error, as noted in [**Section 8.14**](https://www.postgresql.org/docs/10/datatype-json.html).

Note

The functions json[b]\_populate\_record, json[b]\_populate\_recordset, json[b]\_to\_recordand json[b]\_to\_recordset operate on a JSON object, or array of objects, and extract the values associated with keys whose names match column names of the output row type. Object fields that do not correspond to any output column name are ignored, and output columns that do not match any object field will be filled with nulls. To convert a JSON value to the SQL type of an output column, the following rules are applied in sequence:

* A JSON null value is converted to a SQL null in all cases.
* If the output column is of type json or jsonb, the JSON value is just reproduced exactly.
* If the output column is a composite (row) type, and the JSON value is a JSON object, the fields of the object are converted to columns of the output row type by recursive application of these rules.
* Likewise, if the output column is an array type and the JSON value is a JSON array, the elements of the JSON array are converted to elements of the output array by recursive application of these rules.
* Otherwise, if the JSON value is a string literal, the contents of the string are fed to the input conversion function for the column's data type.
* Otherwise, the ordinary text representation of the JSON value is fed to the input conversion function for the column's data type.

While the examples for these functions use constants, the typical use would be to reference a table in the FROM clause and use one of its json or jsonb columns as an argument to the function. Extracted key values can then be referenced in other parts of the query, like WHERE clauses and target lists. Extracting multiple values in this way can improve performance over extracting them separately with per-key operators.

Note

All the items of the path parameter of jsonb\_set as well as jsonb\_insert except the last item must be present in the target. If create\_missing is false, all items of the pathparameter of jsonb\_set must be present. If these conditions are not met the target is returned unchanged.

If the last path item is an object key, it will be created if it is absent and given the new value. If the last path item is an array index, if it is positive the item to set is found by counting from the left, and if negative by counting from the right - -1 designates the rightmost element, and so on. If the item is out of the range -array\_length .. array\_length -1, and create\_missing is true, the new value is added at the beginning of the array if the item is negative, and at the end of the array if it is positive.

Note

The json\_typeof function's null return value should not be confused with a SQL NULL. While calling json\_typeof('null'::json) will return null, calling json\_typeof(NULL::json)will return a SQL NULL.

Note

If the argument to json\_strip\_nulls contains duplicate field names in any object, the result could be semantically somewhat different, depending on the order in which they occur. This is not an issue for jsonb\_strip\_nulls since jsonb values never have duplicate object field names.

See also [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html) for the aggregate function json\_agg which aggregates record values as JSON, and the aggregate function json\_object\_agg which aggregates pairs of values into a JSON object, and their jsonb equivalents, jsonb\_agg and jsonb\_object\_agg.

## 9.16. Sequence Manipulation Functions

This section describes functions for operating on sequence objects, also called sequence generators or just sequences. Sequence objects are special single-row tables created with [**CREATE SEQUENCE**](https://www.postgresql.org/docs/10/sql-createsequence.html). Sequence objects are commonly used to generate unique identifiers for rows of a table. The sequence functions, listed in [**Table 9.47**](https://www.postgresql.org/docs/10/functions-sequence.html#FUNCTIONS-SEQUENCE-TABLE), provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

**Table 9.47. Sequence Functions**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| currval(regclass) | bigint | Return value most recently obtained with nextval for specified sequence |
| lastval() | bigint | Return value most recently obtained with nextval for any sequence |
| nextval(regclass) | bigint | Advance sequence and return new value |
| setval(regclass, bigint) | bigint | Set sequence's current value |
| setval(regclass, bigint, boolean) | bigint | Set sequence's current value and is\_called flag |

The sequence to be operated on by a sequence function is specified by a regclass argument, which is simply the OID of the sequence in the pg\_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type's input converter will do the work for you. Just write the sequence name enclosed in single quotes so that it looks like a literal constant. For compatibility with the handling of ordinary SQL names, the string will be converted to lower case unless it contains double quotes around the sequence name. Thus:

nextval('foo') *operates on sequence foo*

nextval('FOO') *operates on sequence foo*

nextval('"Foo"') *operates on sequence Foo*

The sequence name can be schema-qualified if necessary:

nextval('myschema.foo') *operates on myschema.foo*

nextval('"myschema".foo') *same as above*

nextval('foo') *searches search path for foo*

See [**Section 8.18**](https://www.postgresql.org/docs/10/datatype-oid.html) for more information about regclass.

Note

Before PostgreSQL 8.1, the arguments of the sequence functions were of type text, not regclass, and the above-described conversion from a text string to an OID value would happen at run time during each call. For backward compatibility, this facility still exists, but internally it is now handled as an implicit coercion from text to regclass before the function is invoked.

When you write the argument of a sequence function as an unadorned literal string, it becomes a constant of type regclass. Since this is really just an OID, it will track the originally identified sequence despite later renaming, schema reassignment, etc. This “early binding” behavior is usually desirable for sequence references in column defaults and views. But sometimes you might want “late binding” where the sequence reference is resolved at run time. To get late-binding behavior, force the constant to be stored as a text constant instead of regclass:

nextval('foo'::text) *foo is looked up at runtime*

Note that late binding was the only behavior supported in PostgreSQL releases before 8.1, so you might need to do this to preserve the semantics of old applications.

Of course, the argument of a sequence function can be an expression as well as a constant. If it is a text expression then the implicit coercion will result in a run-time lookup.

The available sequence functions are:

nextval

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute nextval concurrently, each will safely receive a distinct sequence value.

If a sequence object has been created with default parameters, successive nextval calls will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the [**CREATE SEQUENCE**](https://www.postgresql.org/docs/10/sql-createsequence.html) command; see its command reference page for more information.

Important

To avoid blocking concurrent transactions that obtain numbers from the same sequence, a nextval operation is never rolled back; that is, once a value has been fetched it is considered used and will not be returned again. This is true even if the surrounding transaction later aborts, or if the calling query ends up not using the value. For example an INSERT with an ON CONFLICTclause will compute the to-be-inserted tuple, including doing any required nextval calls, before detecting any conflict that would cause it to follow the ON CONFLICT rule instead. Such cases will leave unused “holes” in the sequence of assigned values. Thus, PostgreSQL sequence objects cannot be used to obtain *“gapless”* sequences.

This function requires USAGE or UPDATE privilege on the sequence.

currval

Return the value most recently obtained by nextval for this sequence in the current session. (An error is reported if nextval has never been called for this sequence in this session.) Because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed nextval since the current session did.

This function requires USAGE or SELECT privilege on the sequence.

lastval

Return the value most recently returned by nextval in the current session. This function is identical to currval, except that instead of taking the sequence name as an argument it refers to whichever sequence nextval was most recently applied to in the current session. It is an error to call lastval if nextval has not yet been called in the current session.

This function requires USAGE or SELECT privilege on the last used sequence.

setval

Reset the sequence object's counter value. The two-parameter form sets the sequence's last\_value field to the specified value and sets its is\_called field to true, meaning that the next nextval will advance the sequence before returning a value. The value reported by currval is also set to the specified value. In the three-parameter form, is\_called can be set to either trueor false. true has the same effect as the two-parameter form. If it is set to false, the next nextval will return exactly the specified value, and sequence advancement commences with the following nextval. Furthermore, the value reported by currval is not changed in this case. For example,

SELECT setval('foo', 42); *Next nextval will return 43*

SELECT setval('foo', 42, true); *Same as above*

SELECT setval('foo', 42, false); *Next nextval will return 42*

The result returned by setval is just the value of its second argument.

Important

Because sequences are non-transactional, changes made by setval are not undone if the transaction rolls back.

This function requires UPDATE privilege on the sequence.

## 9.17. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip

If your needs go beyond the capabilities of these conditional expressions, you might want to consider writing a stored procedure in a more expressive programming language.

### 9.17.1. CASE

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other programming languages:

CASE WHEN ***condition*** THEN ***result***

[WHEN ...]

[ELSE ***result***]

END

CASE clauses can be used wherever an expression is valid. Each ***condition*** is an expression that returns a boolean result. If the condition's result is true, the value of the CASE expression is the ***result***that follows the condition, and the remainder of the CASE expression is not processed. If the condition's result is not true, any subsequent WHEN clauses are examined in the same manner. If no WHEN***condition*** yields true, the value of the CASE expression is the ***result*** of the ELSE clause. If the ELSE clause is omitted and no condition is true, the result is null.

An example:

SELECT \* FROM test;

a

---

1

2

3

SELECT a,

CASE WHEN a=1 THEN 'one'

WHEN a=2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

The data types of all the ***result*** expressions must be convertible to a single output type. See [**Section 10.5**](https://www.postgresql.org/docs/10/typeconv-union-case.html) for more details.

There is a “simple” form of CASE expression that is a variant of the general form above:

CASE ***expression***

WHEN ***value*** THEN ***result***

[WHEN ...]

[ELSE ***result***]

END

The first ***expression*** is computed, then compared to each of the ***value*** expressions in the WHEN clauses until one is found that is equal to it. If no match is found, the ***result*** of the ELSE clause (or a null value) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

SELECT a,

CASE a WHEN 1 THEN 'one'

WHEN 2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;

Note

As described in [**Section 4.2.14**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-EXPRESS-EVAL), there are various situations in which subexpressions of an expression are evaluated at different times, so that the principle that “CASE evaluates only necessary subexpressions” is not ironclad. For example a constant 1/0 subexpression will usually result in a division-by-zero failure at planning time, even if it's within a CASE arm that would never be entered at run time.

### 9.17.2. COALESCE

COALESCE(***value*** [, ...])

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

SELECT COALESCE(description, short\_description, '(none)') ...

This returns description if it is not null, otherwise short\_description if it is not null, otherwise (none).

Like a CASE expression, COALESCE only evaluates the arguments that are needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

### 9.17.3. NULLIF

NULLIF(***value1***, ***value2***)

The NULLIF function returns a null value if ***value1*** equals ***value2***; otherwise it returns ***value1***. This can be used to perform the inverse operation of the COALESCE example given above:

SELECT NULLIF(value, '(none)') ...

In this example, if value is (none), null is returned, otherwise the value of value is returned.

### 9.17.4. GREATEST and LEAST

GREATEST(***value*** [, ...])

LEAST(***value*** [, ...])

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result (see [**Section 10.5**](https://www.postgresql.org/docs/10/typeconv-union-case.html) for details). NULL values in the list are ignored. The result will be NULL only if all the expressions evaluate to NULL.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension. Some other databases make them return NULL if any argument is NULL, rather than only when all are NULL.

## 9.18. Array Functions and Operators

[**Table 9.48**](https://www.postgresql.org/docs/10/functions-array.html#ARRAY-OPERATORS-TABLE) shows the operators available for array types.

**Table 9.48. Array Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| = | equal | ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] | t |
| <> | not equal | ARRAY[1,2,3] <> ARRAY[1,2,4] | t |
| < | less than | ARRAY[1,2,3] < ARRAY[1,2,4] | t |
| > | greater than | ARRAY[1,4,3] > ARRAY[1,2,4] | t |
| <= | less than or equal | ARRAY[1,2,3] <= ARRAY[1,2,3] | t |
| >= | greater than or equal | ARRAY[1,4,3] >= ARRAY[1,4,3] | t |
| @> | contains | ARRAY[1,4,3] @> ARRAY[3,1] | t |
| <@ | is contained by | ARRAY[2,7] <@ ARRAY[1,7,4,2,6] | t |
| && | overlap (have elements in common) | ARRAY[1,4,3] && ARRAY[2,1] | t |
| || | array-to-array concatenation | ARRAY[1,2,3] || ARRAY[4,5,6] | {1,2,3,4,5,6} |
| || | array-to-array concatenation | ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] | {{1,2,3},{4,5,6},{7,8,9}} |
| || | element-to-array concatenation | 3 || ARRAY[4,5,6] | {3,4,5,6} |
| || | array-to-element concatenation | ARRAY[4,5,6] || 7 | {4,5,6,7} |

Array comparisons compare the array contents element-by-element, using the default B-tree comparison function for the element data type. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order. (This is a change from versions of PostgreSQL prior to 8.2: older versions would claim that two arrays with the same contents were equal, even if the number of dimensions or subscript ranges were different.)

See [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html) for more details about array operator behavior. See [**Section 11.2**](https://www.postgresql.org/docs/10/indexes-types.html) for more details about which operators support indexed operations.

[**Table 9.49**](https://www.postgresql.org/docs/10/functions-array.html#ARRAY-FUNCTIONS-TABLE) shows the functions available for use with array types. See [**Section 8.15**](https://www.postgresql.org/docs/10/arrays.html) for more information and examples of the use of these functions.

**Table 9.49. Array Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| array\_append(anyarray,anyelement) | anyarray | append an element to the end of an array | array\_append(ARRAY[1,2], 3) | {1,2,3} |
| array\_cat(anyarray,anyarray) | anyarray | concatenate two arrays | array\_cat(ARRAY[1,2,3], ARRAY[4,5]) | {1,2,3,4,5} |
| array\_ndims(anyarray) | int | returns the number of dimensions of the array | array\_ndims(ARRAY[[1,2,3], [4,5,6]]) | 2 |
| array\_dims(anyarray) | text | returns a text representation of array's dimensions | array\_dims(ARRAY[[1,2,3], [4,5,6]]) | [1:2][1:3] |
| array\_fill(anyelement,int[] [, int[]]) | anyarray | returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1 | array\_fill(7, ARRAY[3], ARRAY[2]) | [2:4]={7,7,7} |
| array\_length(anyarray,int) | int | returns the length of the requested array dimension | array\_length(array[1,2,3], 1) | 3 |
| array\_lower(anyarray,int) | int | returns lower bound of the requested array dimension | array\_lower('[0:2]={1,2,3}'::int[], 1) | 0 |
| array\_position(anyarray,anyelement [, int]) | int | returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional) | array\_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon') | 2 |
| array\_positions(anyarray,anyelement) | int[] | returns an array of subscripts of all occurrences of the second argument in the array given as first argument (array must be one-dimensional) | array\_positions(ARRAY['A','A','B','A'], 'A') | {1,2,4} |
| array\_prepend(anyelement,anyarray) | anyarray | append an element to the beginning of an array | array\_prepend(1, ARRAY[2,3]) | {1,2,3} |
| array\_remove(anyarray,anyelement) | anyarray | remove all elements equal to the given value from the array (array must be one-dimensional) | array\_remove(ARRAY[1,2,3,2], 2) | {1,3} |
| array\_replace(anyarray,anyelement, anyelement) | anyarray | replace each array element equal to the given value with a new value | array\_replace(ARRAY[1,2,5,4], 5, 3) | {1,2,3,4} |
| array\_to\_string(anyarray,text [, text]) | text | concatenates array elements using supplied delimiter and optional null string | array\_to\_string(ARRAY[1, 2, 3, NULL, 5], ',', '\*') | 1,2,3,\*,5 |
| array\_upper(anyarray,int) | int | returns upper bound of the requested array dimension | array\_upper(ARRAY[1,8,3,7], 1) | 4 |
| cardinality(anyarray) | int | returns the total number of elements in the array, or 0 if the array is empty | cardinality(ARRAY[[1,2],[3,4]]) | 4 |
| string\_to\_array(text,text [, text]) | text[] | splits string into array elements using supplied delimiter and optional null string | string\_to\_array('xx~^~yy~^~zz', '~^~', 'yy') | {xx,NULL,zz} |
| unnest(anyarray) | setof anyelement | expand an array to a set of rows | unnest(ARRAY[1,2]) | 1  2  (2 rows) |
| unnest(anyarray, anyarray[, ...]) | setof anyelement, anyelement [, ...] | expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see [**Section 7.2.1.4**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLEFUNCTIONS) | unnest(ARRAY[1,2],ARRAY['foo','bar','baz']) | 1 foo  2 bar  NULL baz  (3 rows) |

In array\_position and array\_positions, each array element is compared to the searched value using IS NOT DISTINCT FROM semantics.

In array\_position, NULL is returned if the value is not found.

In array\_positions, NULL is returned only if the array is NULL; if the value is not found in the array, an empty array is returned instead.

In string\_to\_array, if the delimiter parameter is NULL, each character in the input string will become a separate element in the resulting array. If the delimiter is an empty string, then the entire input string is returned as a one-element array. Otherwise the input string is split at each occurrence of the delimiter string.

In string\_to\_array, if the null-string parameter is omitted or NULL, none of the substrings of the input will be replaced by NULL. In array\_to\_string, if the null-string parameter is omitted or NULL, any null elements in the array are simply skipped and not represented in the output string.

Note

There are two differences in the behavior of string\_to\_array from pre-9.1 versions of PostgreSQL. First, it will return an empty (zero-element) array rather than NULL when the input string is of zero length. Second, if the delimiter string is NULL, the function splits the input into individual characters, rather than returning NULL as before.

See also [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html) about the aggregate function array\_agg for use with arrays.

**9.19. Range Functions and Operators**

See [**Section 8.17**](https://www.postgresql.org/docs/10/rangetypes.html) for an overview of range types.

[**Table 9.50**](https://www.postgresql.org/docs/10/functions-range.html#RANGE-OPERATORS-TABLE) shows the operators available for range types.

**Table 9.50. Range Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| = | equal | int4range(1,5) = '[1,4]'::int4range | t |
| <> | not equal | numrange(1.1,2.2) <> numrange(1.1,2.3) | t |
| < | less than | int4range(1,10) < int4range(2,3) | t |
| > | greater than | int4range(1,10) > int4range(1,5) | t |
| <= | less than or equal | numrange(1.1,2.2) <= numrange(1.1,2.2) | t |
| >= | greater than or equal | numrange(1.1,2.2) >= numrange(1.1,2.0) | t |
| @> | contains range | int4range(2,4) @> int4range(2,3) | t |
| @> | contains element | '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp | t |
| <@ | range is contained by | int4range(2,4) <@ int4range(1,7) | t |
| <@ | element is contained by | 42 <@ int4range(1,7) | f |
| && | overlap (have points in common) | int8range(3,7) && int8range(4,12) | t |
| << | strictly left of | int8range(1,10) << int8range(100,110) | t |
| >> | strictly right of | int8range(50,60) >> int8range(20,30) | t |
| &< | does not extend to the right of | int8range(1,20) &< int8range(18,20) | t |
| &> | does not extend to the left of | int8range(7,20) &> int8range(5,10) | t |
| -|- | is adjacent to | numrange(1.1,2.2) -|- numrange(2.2,3.3) | t |
| + | union | numrange(5,15) + numrange(10,20) | [5,20) |
| \* | intersection | int8range(5,15) \* int8range(10,20) | [10,15) |
| - | difference | int8range(5,15) - int8range(10,20) | [5,10) |

The simple comparison operators <, >, <=, and >= compare the lower bounds first, and only if those are equal, compare the upper bounds. These comparisons are not usually very useful for ranges, but are provided to allow B-tree indexes to be constructed on ranges.

The left-of/right-of/adjacent operators always return false when an empty range is involved; that is, an empty range is not considered to be either before or after any other range.

The union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges, as such a range cannot be represented.

[**Table 9.51**](https://www.postgresql.org/docs/10/functions-range.html#RANGE-FUNCTIONS-TABLE) shows the functions available for use with range types.

**Table 9.51. Range Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| lower(anyrange) | range's element type | lower bound of range | lower(numrange(1.1,2.2)) | 1.1 |
| upper(anyrange) | range's element type | upper bound of range | upper(numrange(1.1,2.2)) | 2.2 |
| isempty(anyrange) | boolean | is the range empty? | isempty(numrange(1.1,2.2)) | false |
| lower\_inc(anyrange) | boolean | is the lower bound inclusive? | lower\_inc(numrange(1.1,2.2)) | true |
| upper\_inc(anyrange) | boolean | is the upper bound inclusive? | upper\_inc(numrange(1.1,2.2)) | false |
| lower\_inf(anyrange) | boolean | is the lower bound infinite? | lower\_inf('(,)'::daterange) | true |
| upper\_inf(anyrange) | boolean | is the upper bound infinite? | upper\_inf('(,)'::daterange) | true |
| range\_merge(anyrange, anyrange) | anyrange | the smallest range which includes both of the given ranges | range\_merge('[1,2)'::int4range, '[3,4)'::int4range) | [1,4) |

The lower and upper functions return null if the range is empty or the requested bound is infinite. The lower\_inc, upper\_inc, lower\_inf, and upper\_inf functions all return false for an empty range.

## 9.20. Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in general-purpose aggregate functions are listed in [**Table 9.52**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-AGGREGATE-TABLE) and statistical aggregates in [**Table 9.53**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-AGGREGATE-STATISTICS-TABLE). The built-in within-group ordered-set aggregate functions are listed in [**Table 9.54**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-ORDEREDSET-TABLE) while the built-in within-group hypothetical-set ones are in [**Table 9.55**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-HYPOTHETICAL-TABLE). Grouping operations, which are closely related to aggregate functions, are listed in [**Table 9.56**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-GROUPING-TABLE). The special syntax considerations for aggregate functions are explained in [**Section 4.2.7**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-AGGREGATES). Consult [**Section 2.7**](https://www.postgresql.org/docs/10/tutorial-agg.html) for additional introductory information.

**Table 9.52. General-Purpose Aggregate Functions**

| **Function** | **Argument Type(s)** | **Return Type** | **Partial Mode** | **Description** |
| --- | --- | --- | --- | --- |
| array\_agg(***expression***) | any non-array type | array of the argument type | No | input values, including nulls, concatenated into an array |
| array\_agg(***expression***) | any array type | same as argument data type | No | input arrays concatenated into array of one higher dimension (inputs must all have same dimensionality, and cannot be empty or NULL) |
| avg(***expression***) | smallint, int, bigint, real, double precision, numeric, or interval | numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type | Yes | the average (arithmetic mean) of all input values |
| bit\_and(***expression***) | smallint, int, bigint, or bit | same as argument data type | Yes | the bitwise AND of all non-null input values, or null if none |
| bit\_or(***expression***) | smallint, int, bigint, or bit | same as argument data type | Yes | the bitwise OR of all non-null input values, or null if none |
| bool\_and(***expression***) | bool | bool | Yes | true if all input values are true, otherwise false |
| bool\_or(***expression***) | bool | bool | Yes | true if at least one input value is true, otherwise false |
| count(\*) |  | bigint | Yes | number of input rows |
| count(***expression***) | any | bigint | Yes | number of input rows for which the value of ***expression*** is not null |
| every(***expression***) | bool | bool | Yes | equivalent to bool\_and |
| json\_agg(***expression***) | any | json | No | aggregates values as a JSON array |
| jsonb\_agg(***expression***) | any | jsonb | No | aggregates values as a JSON array |
| json\_object\_agg(***name***,***value***) | (any, any) | json | No | aggregates name/value pairs as a JSON object |
| jsonb\_object\_agg(***name***,***value***) | (any, any) | jsonb | No | aggregates name/value pairs as a JSON object |
| max(***expression***) | any numeric, string, date/time, network, or enum type, or arrays of these types | same as argument type | Yes | maximum value of ***expression*** across all input values |
| min(***expression***) | any numeric, string, date/time, network, or enum type, or arrays of these types | same as argument type | Yes | minimum value of ***expression*** across all input values |
| string\_agg(***expression***,***delimiter***) | (text, text) or (bytea, bytea) | same as argument types | No | input values concatenated into a string, separated by delimiter |
| sum(***expression***) | smallint, int, bigint, real, double precision, numeric, interval, or money | bigint for smallint or int arguments, numeric for bigintarguments, otherwise the same as the argument data type | Yes | sum of ***expression*** across all input values |
| xmlagg(***expression***) | xml | xml | No | concatenation of XML values (see also [**Section 9.14.1.7**](https://www.postgresql.org/docs/10/functions-xml.html#FUNCTIONS-XML-XMLAGG)) |

It should be noted that except for count, these functions return a null value when no rows are selected. In particular, sum of no rows returns null, not zero as one might expect, and array\_agg returns null rather than an empty array when there are no input rows. The coalesce function can be used to substitute zero or an empty array for null when necessary.

Aggregate functions which support Partial Mode are eligible to participate in various optimizations, such as parallel aggregation.

Note

Boolean aggregates bool\_and and bool\_or correspond to standard SQL aggregates everyand any or some. As for any and some, it seems that there is an ambiguity built into the standard syntax:

SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;

Here ANY can be considered either as introducing a subquery, or as being an aggregate function, if the subquery returns one row with a Boolean value. Thus the standard name cannot be given to these aggregates.

Note

Users accustomed to working with other SQL database management systems might be disappointed by the performance of the count aggregate when it is applied to the entire table. A query like:

SELECT count(\*) FROM sometable;

will require effort proportional to the size of the table: PostgreSQL will need to scan either the entire table or the entirety of an index which includes all rows in the table.

The aggregate functions array\_agg, json\_agg, jsonb\_agg, json\_object\_agg, jsonb\_object\_agg, string\_agg, and xmlagg, as well as similar user-defined aggregate functions, produce meaningfully different result values depending on the order of the input values. This ordering is unspecified by default, but can be controlled by writing an ORDER BY clause within the aggregate call, as shown in [**Section 4.2.7**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-AGGREGATES). Alternatively, supplying the input values from a sorted subquery will usually work. For example:

SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;

Beware that this approach can fail if the outer query level contains additional processing, such as a join, because that might cause the subquery's output to be reordered before the aggregate is computed.

[**Table 9.53**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-AGGREGATE-STATISTICS-TABLE) shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions ***N***, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when ***N*** is zero.

**Table 9.53. Aggregate Functions for Statistics**

| **Function** | **Argument Type** | **Return Type** | **Partial Mode** | **Description** |
| --- | --- | --- | --- | --- |
| corr(***Y***, ***X***) | double precision | double precision | Yes | correlation coefficient |
| covar\_pop(***Y***, ***X***) | double precision | double precision | Yes | population covariance |
| covar\_samp(***Y***, ***X***) | double precision | double precision | Yes | sample covariance |
| regr\_avgx(***Y***, ***X***) | double precision | double precision | Yes | average of the independent variable (sum(***X***)/***N***) |
| regr\_avgy(***Y***, ***X***) | double precision | double precision | Yes | average of the dependent variable (sum(***Y***)/***N***) |
| regr\_count(***Y***, ***X***) | double precision | bigint | Yes | number of input rows in which both expressions are nonnull |
| regr\_intercept(***Y***, ***X***) | double precision | double precision | Yes | y-intercept of the least-squares-fit linear equation determined by the (***X***, ***Y***) pairs |
| regr\_r2(***Y***, ***X***) | double precision | double precision | Yes | square of the correlation coefficient |
| regr\_slope(***Y***, ***X***) | double precision | double precision | Yes | slope of the least-squares-fit linear equation determined by the (***X***, ***Y***) pairs |
| regr\_sxx(***Y***, ***X***) | double precision | double precision | Yes | sum(***X***^2) - sum(***X***)^2/***N*** (“sum of squares” of the independent variable) |
| regr\_sxy(***Y***, ***X***) | double precision | double precision | Yes | sum(***X***\****Y***) - sum(***X***) \* sum(***Y***)/***N*** (“sum of products” of independent times dependent variable) |
| regr\_syy(***Y***, ***X***) | double precision | double precision | Yes | sum(***Y***^2) - sum(***Y***)^2/***N*** (“sum of squares” of the dependent variable) |
| stddev(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | historical alias for stddev\_samp |
| stddev\_pop(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | population standard deviation of the input values |
| stddev\_samp(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | sample standard deviation of the input values |
| variance(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | historical alias for var\_samp |
| var\_pop(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | population variance of the input values (square of the population standard deviation) |
| var\_samp(***expression***) | smallint, int, bigint, real, double precision, or numeric | double precision for floating-point arguments, otherwise numeric | Yes | sample variance of the input values (square of the sample standard deviation) |

[**Table 9.54**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-ORDEREDSET-TABLE) shows some aggregate functions that use the ordered-set aggregate syntax. These functions are sometimes referred to as “inverse distribution” functions.

**Table 9.54. Ordered-Set Aggregate Functions**

| **Function** | **Direct Argument Type(s)** | **Aggregated Argument Type(s)** | **Return Type** | **Partial Mode** | **Description** |
| --- | --- | --- | --- | --- | --- |
| mode() WITHIN GROUP (ORDER BY***sort\_expression***) |  | any sortable type | same as sort expression | No | returns the most frequent input value (arbitrarily choosing the first one if there are multiple equally-frequent results) |
| percentile\_cont(***fraction***) WITHIN GROUP (ORDER BY ***sort\_expression***) | double precision | double precision or interval | same as sort expression | No | continuous percentile: returns a value corresponding to the specified fraction in the ordering, interpolating between adjacent input items if needed |
| percentile\_cont(***fractions***) WITHIN GROUP (ORDER BY ***sort\_expression***) | double precision[] | double precision or interval | array of sort expression's type | No | multiple continuous percentile: returns an array of results matching the shape of the ***fractions***parameter, with each non-null element replaced by the value corresponding to that percentile |
| percentile\_disc(***fraction***) WITHIN GROUP (ORDER BY ***sort\_expression***) | double precision | any sortable type | same as sort expression | No | discrete percentile: returns the first input value whose position in the ordering equals or exceeds the specified fraction |
| percentile\_disc(***fractions***) WITHIN GROUP (ORDER BY ***sort\_expression***) | double precision[] | any sortable type | array of sort expression's type | No | multiple discrete percentile: returns an array of results matching the shape of the ***fractions***parameter, with each non-null element replaced by the input value corresponding to that percentile |

All the aggregates listed in [**Table 9.54**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-ORDEREDSET-TABLE) ignore null values in their sorted input. For those that take a ***fraction*** parameter, the fraction value must be between 0 and 1; an error is thrown if not. However, a null fraction value simply produces a null result.

Each of the aggregates listed in [**Table 9.55**](https://www.postgresql.org/docs/10/functions-aggregate.html#FUNCTIONS-HYPOTHETICAL-TABLE) is associated with a window function of the same name defined in [**Section 9.21**](https://www.postgresql.org/docs/10/functions-window.html). In each case, the aggregate result is the value that the associated window function would have returned for the “hypothetical” row constructed from ***args***, if such a row had been added to the sorted group of rows computed from the ***sorted\_args***.

**Table 9.55. Hypothetical-Set Aggregate Functions**

| **Function** | **Direct Argument Type(s)** | **Aggregated Argument Type(s)** | **Return Type** | **Partial Mode** | **Description** |
| --- | --- | --- | --- | --- | --- |
| rank(***args***) WITHIN GROUP (ORDER BY ***sorted\_args***) | VARIADIC "any" | VARIADIC "any" | bigint | No | rank of the hypothetical row, with gaps for duplicate rows |
| dense\_rank(***args***) WITHIN GROUP (ORDER BY***sorted\_args***) | VARIADIC "any" | VARIADIC "any" | bigint | No | rank of the hypothetical row, without gaps |
| percent\_rank(***args***) WITHIN GROUP (ORDER BY***sorted\_args***) | VARIADIC "any" | VARIADIC "any" | double precision | No | relative rank of the hypothetical row, ranging from 0 to 1 |
| cume\_dist(***args***) WITHIN GROUP (ORDER BY ***sorted\_args***) | VARIADIC "any" | VARIADIC "any" | double precision | No | relative rank of the hypothetical row, ranging from 1/***N***to 1 |

For each of these hypothetical-set aggregates, the list of direct arguments given in ***args*** must match the number and types of the aggregated arguments given in ***sorted\_args***. Unlike most built-in aggregates, these aggregates are not strict, that is they do not drop input rows containing nulls. Null values sort according to the rule specified in the ORDER BY clause.

**Table 9.56. Grouping Operations**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| GROUPING(***args...***) | integer | Integer bit mask indicating which arguments are not being included in the current grouping set |

Grouping operations are used in conjunction with grouping sets (see [**Section 7.2.4**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-GROUPING-SETS)) to distinguish result rows. The arguments to the GROUPING operation are not actually evaluated, but they must match exactly expressions given in the GROUP BY clause of the associated query level. Bits are assigned with the rightmost argument being the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the result row, and 1 if it is not. For example:

=> **SELECT \* FROM items\_sold;**

make | model | sales

-------+-------+-------

Foo | GT | 10

Foo | Tour | 20

Bar | City | 15

Bar | Sport | 5

(4 rows)

=> **SELECT make, model, GROUPING(make,model), sum(sales) FROM items\_sold GROUP BY ROLLUP(make,model);**

make | model | grouping | sum

-------+-------+----------+-----

Foo | GT | 0 | 10

Foo | Tour | 0 | 20

Bar | City | 0 | 15

Bar | Sport | 0 | 5

Foo | | 1 | 30

Bar | | 1 | 20

| | 3 | 50

(7 rows)

## 9.21. Window Functions

Window functions provide the ability to perform calculations across sets of rows that are related to the current query row. See [**Section 3.5**](https://www.postgresql.org/docs/10/tutorial-window.html) for an introduction to this feature, and [**Section 4.2.8**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS) for syntax details.

The built-in window functions are listed in [**Table 9.57**](https://www.postgresql.org/docs/10/functions-window.html#FUNCTIONS-WINDOW-TABLE). Note that these functions must be invoked using window function syntax, i.e., an OVER clause is required.

In addition to these functions, any built-in or user-defined general-purpose or statistical aggregate (i.e., not ordered-set or hypothetical-set aggregates) can be used as a window function; see [**Section 9.20**](https://www.postgresql.org/docs/10/functions-aggregate.html) for a list of the built-in aggregates. Aggregate functions act as window functions only when an OVER clause follows the call; otherwise they act as non-window aggregates and return a single row for the entire set.

**Table 9.57. General-Purpose Window Functions**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| row\_number() | bigint | number of the current row within its partition, counting from 1 |
| rank() | bigint | rank of the current row with gaps; same as row\_number of its first peer |
| dense\_rank() | bigint | rank of the current row without gaps; this function counts peer groups |
| percent\_rank() | double precision | relative rank of the current row: (rank - 1) / (total partition rows - 1) |
| cume\_dist() | double precision | cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows |
| ntile(***num\_buckets*** integer) | integer | integer ranging from 1 to the argument value, dividing the partition as equally as possible |
| lag(***value*** anyelement [, ***offset***integer [, ***default*** anyelement]]) | same type as ***value*** | returns ***value*** evaluated at the row that is ***offset*** rows before the current row within the partition; if there is no such row, instead return ***default***(which must be of the same type as ***value***). Both ***offset*** and ***default*** are evaluated with respect to the current row. If omitted, ***offset*** defaults to 1 and ***default*** to null |
| lead(***value*** anyelement [, ***offset***integer [, ***default*** anyelement]]) | same type as ***value*** | returns ***value*** evaluated at the row that is ***offset*** rows after the current row within the partition; if there is no such row, instead return ***default*** (which must be of the same type as ***value***). Both ***offset*** and ***default*** are evaluated with respect to the current row. If omitted, ***offset*** defaults to 1 and ***default*** to null |
| first\_value(***value*** any) | same type as ***value*** | returns ***value*** evaluated at the row that is the first row of the window frame |
| last\_value(***value*** any) | same type as ***value*** | returns ***value*** evaluated at the row that is the last row of the window frame |
| nth\_value(***value*** any, ***nth***integer) | same type as ***value*** | returns ***value*** evaluated at the row that is the ***nth*** row of the window frame (counting from 1); null if no such row |

All of the functions listed in [**Table 9.57**](https://www.postgresql.org/docs/10/functions-window.html#FUNCTIONS-WINDOW-TABLE) depend on the sort ordering specified by the ORDER BY clause of the associated window definition. Rows that are not distinct when considering only the ORDER BYcolumns are said to be peers. The four ranking functions (including cume\_dist) are defined so that they give the same answer for all peer rows.

Note that first\_value, last\_value, and nth\_value consider only the rows within the “window frame”, which by default contains the rows from the start of the partition through the last peer of the current row. This is likely to give unhelpful results for last\_value and sometimes also nth\_value. You can redefine the frame by adding a suitable frame specification (RANGE or ROWS) to the OVER clause. See [**Section 4.2.8**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS) for more information about frame specifications.

When an aggregate function is used as a window function, it aggregates over the rows within the current row's window frame. An aggregate used with ORDER BY and the default window frame definition produces a “running sum” type of behavior, which may or may not be what's wanted. To obtain aggregation over the whole partition, omit ORDER BY or use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Other frame specifications can be used to obtain other effects.

Note

The SQL standard defines a RESPECT NULLS or IGNORE NULLS option for lead, lag, first\_value, last\_value, and nth\_value. This is not implemented in PostgreSQL: the behavior is always the same as the standard's default, namely RESPECT NULLS. Likewise, the standard's FROM FIRST or FROM LAST option for nth\_value is not implemented: only the default FROM FIRST behavior is supported. (You can achieve the result of FROM LAST by reversing the ORDER BY ordering.)

cume\_dist computes the fraction of partition rows that are less than or equal to the current row and its peers, while percent\_rank computes the fraction of partition rows that are less than the current row, assuming the current row does not exist in the partition.

## 9.22. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

### 9.22.1. EXISTS

EXISTS (***subquery***)

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is “true”; if the subquery returns no rows, the result of EXISTS is “false”.

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed long enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has side effects (such as calling sequence functions); whether the side effects occur might be unpredictable.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally unimportant. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on col2, but it produces at most one output row for each tab1 row, even if there are several matching tab2 rows:

SELECT col1

FROM tab1

WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);

### 9.22.2. IN

***expression*** IN (***subquery***)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

***row\_constructor*** IN (***subquery***)

The left-hand side of this form of IN is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of IN is null.

### 9.22.3. NOT IN

***expression*** NOT IN (***subquery***)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT INis “true” if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is “false” if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the NOT IN construct will be null, not true. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

***row\_constructor*** NOT IN (***subquery***)

The left-hand side of this form of NOT IN is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of NOT IN is “true” if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is “false” if any equal row is found.

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of NOT IN is null.

### 9.22.4. ANY/SOME

***expression*** ***operator*** ANY (***subquery***)

***expression*** ***operator*** SOME (***subquery***)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given ***operator***, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

***row\_constructor*** ***operator*** ANY (***subquery***)

***row\_constructor*** ***operator*** SOME (***subquery***)

The left-hand side of this form of ANY is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given ***operator***. The result of ANY is “true” if the comparison returns true for any subquery row. The result is “false” if the comparison returns false for every subquery row (including the case where the subquery returns no rows). The result is NULL if no comparison with a subquery row returns true, and at least one comparison returns NULL.

See [**Section 9.23.5**](https://www.postgresql.org/docs/10/functions-comparisons.html#ROW-WISE-COMPARISON) for details about the meaning of a row constructor comparison.

### 9.22.5. ALL

***expression*** ***operator*** ALL (***subquery***)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given ***operator***, which must yield a Boolean result. The result of ALL is “true” if all rows yield true (including the case where the subquery returns no rows). The result is “false” if any false result is found. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

NOT IN is equivalent to <> ALL.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

***row\_constructor*** ***operator*** ALL (***subquery***)

The left-hand side of this form of ALL is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given ***operator***. The result of ALL is “true” if the comparison returns true for all subquery rows (including the case where the subquery returns no rows). The result is “false” if the comparison returns false for any subquery row. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

See [**Section 9.23.5**](https://www.postgresql.org/docs/10/functions-comparisons.html#ROW-WISE-COMPARISON) for details about the meaning of a row constructor comparison.

### 9.22.6. Single-row Comparison

***row\_constructor*** ***operator*** (***subquery***)

The left-hand side is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See [**Section 9.23.5**](https://www.postgresql.org/docs/10/functions-comparisons.html#ROW-WISE-COMPARISON) for details about the meaning of a row constructor comparison.

## 9.23. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

### 9.23.1. IN

***expression*** IN (***value*** [, ...])

The right-hand side is a parenthesized list of scalar expressions. The result is “true” if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

***expression*** = ***value1***

OR

***expression*** = ***value2***

OR

...

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

### 9.23.2. NOT IN

***expression*** NOT IN (***value*** [, ...])

The right-hand side is a parenthesized list of scalar expressions. The result is “true” if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

***expression*** <> ***value1***

AND

***expression*** <> ***value2***

AND

...

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the NOT IN construct will be null, not true as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of null values.

Tip

x NOT IN y is equivalent to NOT (x IN y) in all cases. However, null values are much more likely to trip up the novice when working with NOT IN than when working with IN. It is best to express your condition positively if possible.

### 9.23.3. ANY/SOME (array)

***expression*** ***operator*** ANY (***array expression***)

***expression*** ***operator*** SOME (***array expression***)

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given ***operator***, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the case where the array has zero elements).

If the array expression yields a null array, the result of ANY will be null. If the left-hand expression yields null, the result of ANY is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of ANY will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

SOME is a synonym for ANY.

### 9.23.4. ALL (array)

***expression*** ***operator*** ALL (***array expression***)

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given ***operator***, which must yield a Boolean result. The result of ALL is “true” if all comparisons yield true (including the case where the array has zero elements). The result is “false” if any false result is found.

If the array expression yields a null array, the result of ALL will be null. If the left-hand expression yields null, the result of ALL is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of ALL will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

### 9.23.5. Row Constructor Comparison

***row\_constructor*** ***operator*** ***row\_constructor***

Each side is a row constructor, as described in [**Section 4.2.13**](https://www.postgresql.org/docs/10/sql-expressions.html#SQL-SYNTAX-ROW-CONSTRUCTORS). The two row values must have the same number of fields. Each side is evaluated and they are compared row-wise. Row constructor comparisons are allowed when the ***operator*** is =, <>, <, <=, > or >=. Every row element must be of a type which has a default B-tree operator class or the attempted comparison may generate an error.

Note

Errors related to the number or types of elements might not occur if the comparison is resolved using earlier columns.

The = and <> cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, ROW(1,2,NULL) < ROW(1,3,0) yields true, not null, because the third pair of elements are not considered.

Note

Prior to PostgreSQL 8.2, the <, <=, > and >= cases were not handled per SQL specification. A comparison like ROW(a,b) < ROW(c,d) was implemented as a < c AND b < d whereas the correct behavior is equivalent to a < c OR (a = c AND b < d).

***row\_constructor*** IS DISTINCT FROM ***row\_constructor***

This construct is similar to a <> row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will either be true or false, never null.

***row\_constructor*** IS NOT DISTINCT FROM ***row\_constructor***

This construct is similar to a = row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

### 9.23.6. Composite Type Comparison

***record*** ***operator*** ***record***

The SQL specification requires row-wise comparison to return NULL if the result depends on comparing two NULL values or a NULL and a non-NULL. PostgreSQL does this only when comparing the results of two row constructors (as in [**Section 9.23.5**](https://www.postgresql.org/docs/10/functions-comparisons.html#ROW-WISE-COMPARISON)) or comparing a row constructor to the output of a subquery (as in [**Section 9.22**](https://www.postgresql.org/docs/10/functions-subquery.html)). In other contexts where two composite-type values are compared, two NULL field values are considered equal, and a NULL is considered larger than a non-NULL. This is necessary in order to have consistent sorting and indexing behavior for composite types.

Each side is evaluated and they are compared row-wise. Composite type comparisons are allowed when the ***operator*** is =, <>, <, <=, > or >=, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-tree operator class, or is the negator of the = member of a B-tree operator class.) The default behavior of the above operators is the same as for IS [ NOT ] DISTINCT FROM for row constructors (see [**Section 9.23.5**](https://www.postgresql.org/docs/10/functions-comparisons.html#ROW-WISE-COMPARISON)).

To support matching of rows which include elements without a default B-tree operator class, the following operators are defined for composite type comparison: \*=, \*<>, \*<, \*<=, \*>, and \*>=. These operators compare the internal binary representation of the two rows. Two rows might have a different binary representation even though comparisons of the two rows with the equality operator is true. The ordering of rows under these comparison operators is deterministic but not otherwise meaningful. These operators are used internally for materialized views and might be useful for other specialized purposes such as replication but are not intended to be generally useful for writing queries.

**9.24. Set Returning Functions**

This section describes functions that possibly return more than one row. The most widely used functions in this class are series generating functions, as detailed in [**Table 9.58**](https://www.postgresql.org/docs/10/functions-srf.html#FUNCTIONS-SRF-SERIES) and [**Table 9.59**](https://www.postgresql.org/docs/10/functions-srf.html#FUNCTIONS-SRF-SUBSCRIPTS). Other, more specialized set-returning functions are described elsewhere in this manual. See [**Section 7.2.1.4**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLEFUNCTIONS) for ways to combine multiple set-returning functions.

**Table 9.58. Series Generating Functions**

| **Function** | **Argument Type** | **Return Type** | **Description** |
| --- | --- | --- | --- |
| generate\_series(*start*, *stop*) | int, bigint or numeric | setof int, setof bigint, or setof numeric (same as argument type) | Generate a series of values, from *start* to *stop* with a step size of one |
| generate\_series(*start*, *stop*, *step*) | int, bigint or numeric | setof int, setof bigint or setof numeric (same as argument type) | Generate a series of values, from *start* to *stop* with a step size of *step* |
| generate\_series(*start*, *stop*, *step*interval) | timestamp or timestamp with time zone | setof timestamp or setof timestamp with time zone (same as argument type) | Generate a series of values, from *start* to *stop* with a step size of *step* |

When *step* is positive, zero rows are returned if *start* is greater than *stop*. Conversely, when *step* is negative, zero rows are returned if *start* is less than *stop*. Zero rows are also returned for NULLinputs. It is an error for *step* to be zero. Some examples follow:

SELECT \* FROM generate\_series(2,4);

generate\_series

-----------------

2

3

4

(3 rows)

SELECT \* FROM generate\_series(5,1,-2);

generate\_series

-----------------

5

3

1

(3 rows)

SELECT \* FROM generate\_series(4,3);

generate\_series

-----------------

(0 rows)

SELECT generate\_series(1.1, 4, 1.3);

generate\_series

-----------------

1.1

2.4

3.7

(3 rows)

-- this example relies on the date-plus-integer operator

SELECT current\_date + s.a AS dates FROM generate\_series(0,14,7) AS s(a);

dates

------------

2004-02-05

2004-02-12

2004-02-19

(3 rows)

SELECT \* FROM generate\_series('2008-03-01 00:00'::timestamp,

'2008-03-04 12:00', '10 hours');

generate\_series

---------------------

2008-03-01 00:00:00

2008-03-01 10:00:00

2008-03-01 20:00:00

2008-03-02 06:00:00

2008-03-02 16:00:00

2008-03-03 02:00:00

2008-03-03 12:00:00

2008-03-03 22:00:00

2008-03-04 08:00:00

(9 rows)

**Table 9.59. Subscript Generating Functions**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| generate\_subscripts(*array anyarray*, *dim int*) | setof int | Generate a series comprising the given array's subscripts. |
| generate\_subscripts(*array anyarray*, *dim int*, *reverse boolean*) | setof int | Generate a series comprising the given array's subscripts. When *reverse* is true, the series is returned in reverse order. |

generate\_subscripts is a convenience function that generates the set of valid subscripts for the specified dimension of the given array. Zero rows are returned for arrays that do not have the requested dimension, or for NULL arrays (but valid subscripts are returned for NULL array elements). Some examples follow:

-- basic usage

SELECT generate\_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;

s

---

1

2

3

4

(4 rows)

-- presenting an array, the subscript and the subscripted

-- value requires a subquery

SELECT \* FROM arrays;

a

--------------------

{-1,-2}

{100,200,300}

(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value

FROM (SELECT generate\_subscripts(a, 1) AS s, a FROM arrays) foo;

array | subscript | value

---------------+-----------+-------

{-1,-2} | 1 | -1

{-1,-2} | 2 | -2

{100,200,300} | 1 | 100

{100,200,300} | 2 | 200

{100,200,300} | 3 | 300

(5 rows)

-- unnest a 2D array

CREATE OR REPLACE FUNCTION unnest2(anyarray)

RETURNS SETOF anyelement AS $$

select $1[i][j]

from generate\_subscripts($1,1) g1(i),

generate\_subscripts($1,2) g2(j);

$$ LANGUAGE sql IMMUTABLE;

CREATE FUNCTION

SELECT \* FROM unnest2(ARRAY[[1,2],[3,4]]);

unnest2

---------

1

2

3

4

(4 rows)

When a function in the FROM clause is suffixed by WITH ORDINALITY, a bigint column is appended to the output which starts from 1 and increments by 1 for each row of the function's output. This is most useful in the case of set returning functions such as unnest().

-- set returning function WITH ORDINALITY

SELECT \* FROM pg\_ls\_dir('.') WITH ORDINALITY AS t(ls,n);

ls | n

-----------------+----

pg\_serial | 1

pg\_twophase | 2

postmaster.opts | 3

pg\_notify | 4

postgresql.conf | 5

pg\_tblspc | 6

logfile | 7

base | 8

postmaster.pid | 9

pg\_ident.conf | 10

global | 11

pg\_xact | 12

pg\_snapshots | 13

pg\_multixact | 14

PG\_VERSION | 15

pg\_wal | 16

pg\_hba.conf | 17

pg\_stat\_tmp | 18

pg\_subtrans | 19

(19 rows)

## 9.25. System Information Functions

[**Table 9.60**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-SESSION-TABLE) shows several functions that extract session and system information.

In addition to the functions listed in this section, there are a number of functions related to the statistics system that also provide system information. See [**Section 28.2.2**](https://www.postgresql.org/docs/10/monitoring-stats.html#MONITORING-STATS-VIEWS) for more information.

**Table 9.60. Session Information Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| current\_catalog | name | name of current database (called “catalog” in the SQL standard) |
| current\_database() | name | name of current database |
| current\_query() | text | text of the currently executing query, as submitted by the client (might contain more than one statement) |
| current\_role | name | equivalent to current\_user |
| current\_schema[()] | name | name of current schema |
| current\_schemas(boolean) | name[] | names of schemas in search path, optionally including implicit schemas |
| current\_user | name | user name of current execution context |
| inet\_client\_addr() | inet | address of the remote connection |
| inet\_client\_port() | int | port of the remote connection |
| inet\_server\_addr() | inet | address of the local connection |
| inet\_server\_port() | int | port of the local connection |
| pg\_backend\_pid() | int | Process ID of the server process attached to the current session |
| pg\_blocking\_pids(int) | int[] | Process ID(s) that are blocking specified server process ID from acquiring a lock |
| pg\_conf\_load\_time() | timestamp with time zone | configuration load time |
| pg\_current\_logfile([text]) | text | Primary log file name, or log in the requested format, currently in use by the logging collector |
| pg\_my\_temp\_schema() | oid | OID of session's temporary schema, or 0 if none |
| pg\_is\_other\_temp\_schema(oid) | boolean | is schema another session's temporary schema? |
| pg\_listening\_channels() | setof text | channel names that the session is currently listening on |
| pg\_notification\_queue\_usage() | double | fraction of the asynchronous notification queue currently occupied (0-1) |
| pg\_postmaster\_start\_time() | timestamp with time zone | server start time |
| pg\_safe\_snapshot\_blocking\_pids(int) | int[] | Process ID(s) that are blocking specified server process ID from acquiring a safe snapshot |
| pg\_trigger\_depth() | int | current nesting level of PostgreSQL triggers (0 if not called, directly or indirectly, from inside a trigger) |
| session\_user | name | session user name |
| user | name | equivalent to current\_user |
| version() | text | PostgreSQL version information. See also [**server\_version\_num**](https://www.postgresql.org/docs/10/runtime-config-preset.html#GUC-SERVER-VERSION-NUM) for a machine-readable version. |

Note

current\_catalog, current\_role, current\_schema, current\_user, session\_user, and user have special syntactic status in SQL: they must be called without trailing parentheses. (In PostgreSQL, parentheses can optionally be used with current\_schema, but not with the others.)

The session\_user is normally the user who initiated the current database connection; but superusers can change this setting with [**SET SESSION AUTHORIZATION**](https://www.postgresql.org/docs/10/sql-set-session-authorization.html). The current\_user is the user identifier that is applicable for permission checking. Normally it is equal to the session user, but it can be changed with [**SET ROLE**](https://www.postgresql.org/docs/10/sql-set-role.html). It also changes during the execution of functions with the attribute SECURITY DEFINER. In Unix parlance, the session user is the “real user” and the current user is the “effective user”. current\_role and user are synonyms for current\_user. (The SQL standard draws a distinction between current\_role and current\_user, but PostgreSQL does not, since it unifies users and roles into a single kind of entity.)

current\_schema returns the name of the schema that is first in the search path (or a null value if the search path is empty). This is the schema that will be used for any tables or other named objects that are created without specifying a target schema. current\_schemas(boolean) returns an array of the names of all schemas presently in the search path. The Boolean option determines whether or not implicitly included system schemas such as pg\_catalog are included in the returned search path.

Note

The search path can be altered at run time. The command is:

SET search\_path TO ***schema*** [, ***schema***, ...]

inet\_client\_addr returns the IP address of the current client, and inet\_client\_port returns the port number. inet\_server\_addr returns the IP address on which the server accepted the current connection, and inet\_server\_port returns the port number. All these functions return NULL if the current connection is via a Unix-domain socket.

pg\_blocking\_pids returns an array of the process IDs of the sessions that are blocking the server process with the specified process ID, or an empty array if there is no such server process or it is not blocked. One server process blocks another if it either holds a lock that conflicts with the blocked process's lock request (hard block), or is waiting for a lock that would conflict with the blocked process's lock request and is ahead of it in the wait queue (soft block). When using parallel queries the result always lists client-visible process IDs (that is, pg\_backend\_pid results) even if the actual lock is held or awaited by a child worker process. As a result of that, there may be duplicated PIDs in the result. Also note that when a prepared transaction holds a conflicting lock, it will be represented by a zero process ID in the result of this function. Frequent calls to this function could have some impact on database performance, because it needs exclusive access to the lock manager's shared state for a short time.

pg\_conf\_load\_time returns the timestamp with time zone when the server configuration files were last loaded. (If the current session was alive at the time, this will be the time when the session itself re-read the configuration files, so the reading will vary a little in different sessions. Otherwise it is the time when the postmaster process re-read the configuration files.)

pg\_current\_logfile returns, as text, the path of the log file(s) currently in use by the logging collector. The path includes the [**log\_directory**](https://www.postgresql.org/docs/10/runtime-config-logging.html#GUC-LOG-DIRECTORY) directory and the log file name. Log collection must be enabled or the return value is NULL. When multiple log files exist, each in a different format, pg\_current\_logfile called without arguments returns the path of the file having the first format found in the ordered list: stderr, csvlog. NULL is returned when no log file has any of these formats. To request a specific file format supply, as text, either csvlog or stderr as the value of the optional parameter. The return value is NULL when the log format requested is not a configured [**log\_destination**](https://www.postgresql.org/docs/10/runtime-config-logging.html#GUC-LOG-DESTINATION). The pg\_current\_logfile reflects the contents of the current\_logfiles file.

pg\_my\_temp\_schema returns the OID of the current session's temporary schema, or zero if it has none (because it has not created any temporary tables). pg\_is\_other\_temp\_schema returns true if the given OID is the OID of another session's temporary schema. (This can be useful, for example, to exclude other sessions' temporary tables from a catalog display.)

pg\_listening\_channels returns a set of names of asynchronous notification channels that the current session is listening to. pg\_notification\_queue\_usage returns the fraction of the total available space for notifications currently occupied by notifications that are waiting to be processed, as a double in the range 0-1. See [**LISTEN**](https://www.postgresql.org/docs/10/sql-listen.html) and [**NOTIFY**](https://www.postgresql.org/docs/10/sql-notify.html) for more information.

pg\_postmaster\_start\_time returns the timestamp with time zone when the server started.

pg\_safe\_snapshot\_blocking\_pids returns an array of the process IDs of the sessions that are blocking the server process with the specified process ID from acquiring a safe snapshot, or an empty array if there is no such server process or it is not blocked. A session running a SERIALIZABLE transaction blocks a SERIALIZABLE READ ONLY DEFERRABLE transaction from acquiring a snapshot until the latter determines that it is safe to avoid taking any predicate locks. See [**Section 13.2.3**](https://www.postgresql.org/docs/10/transaction-iso.html#XACT-SERIALIZABLE) for more information about serializable and deferrable transactions. Frequent calls to this function could have some impact on database performance, because it needs access to the predicate lock manager's shared state for a short time.

version returns a string describing the PostgreSQL server's version. You can also get this information from [**server\_version**](https://www.postgresql.org/docs/10/runtime-config-preset.html#GUC-SERVER-VERSION) or for a machine-readable version, [**server\_version\_num**](https://www.postgresql.org/docs/10/runtime-config-preset.html#GUC-SERVER-VERSION-NUM). Software developers should use server\_version\_num (available since 8.2) or [PQserverVersion](https://www.postgresql.org/docs/10/libpq-status.html#LIBPQ-PQSERVERVERSION) instead of parsing the text version.

**[Table 9.61](https://www.postgresql.org/docs/10/functions-info.html" \l "FUNCTIONS-INFO-ACCESS-TABLE" \o "Table 9.61. Access Privilege Inquiry Functions)** lists functions that allow the user to query object access privileges programmatically. See [**Section 5.6**](https://www.postgresql.org/docs/10/ddl-priv.html) for more information about privileges.

**Table 9.61. Access Privilege Inquiry Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| has\_any\_column\_privilege(*user*, *table*, *privilege*) | boolean | does user have privilege for any column of table |
| has\_any\_column\_privilege(*table*, *privilege*) | boolean | does current user have privilege for any column of table |
| has\_column\_privilege(*user*, *table*, *column*, *privilege*) | boolean | does user have privilege for column |
| has\_column\_privilege(*table*, *column*, *privilege*) | boolean | does current user have privilege for column |
| has\_database\_privilege(*user*, *database*, *privilege*) | boolean | does user have privilege for database |
| has\_database\_privilege(*database*, *privilege*) | boolean | does current user have privilege for database |
| has\_foreign\_data\_wrapper\_privilege(*user*, *fdw*, *privilege*) | boolean | does user have privilege for foreign-data wrapper |
| has\_foreign\_data\_wrapper\_privilege(*fdw*, *privilege*) | boolean | does current user have privilege for foreign-data wrapper |
| has\_function\_privilege(*user*, *function*, *privilege*) | boolean | does user have privilege for function |
| has\_function\_privilege(*function*, *privilege*) | boolean | does current user have privilege for function |
| has\_language\_privilege(*user*, *language*, *privilege*) | boolean | does user have privilege for language |
| has\_language\_privilege(*language*, *privilege*) | boolean | does current user have privilege for language |
| has\_schema\_privilege(*user*, *schema*, *privilege*) | boolean | does user have privilege for schema |
| has\_schema\_privilege(*schema*, *privilege*) | boolean | does current user have privilege for schema |
| has\_sequence\_privilege(*user*, *sequence*, *privilege*) | boolean | does user have privilege for sequence |
| has\_sequence\_privilege(*sequence*, *privilege*) | boolean | does current user have privilege for sequence |
| has\_server\_privilege(*user*, *server*, *privilege*) | boolean | does user have privilege for foreign server |
| has\_server\_privilege(*server*, *privilege*) | boolean | does current user have privilege for foreign server |
| has\_table\_privilege(*user*, *table*, *privilege*) | boolean | does user have privilege for table |
| has\_table\_privilege(*table*, *privilege*) | boolean | does current user have privilege for table |
| has\_tablespace\_privilege(*user*, *tablespace*, *privilege*) | boolean | does user have privilege for tablespace |
| has\_tablespace\_privilege(*tablespace*, *privilege*) | boolean | does current user have privilege for tablespace |
| has\_type\_privilege(*user*, *type*, *privilege*) | boolean | does user have privilege for type |
| has\_type\_privilege(*type*, *privilege*) | boolean | does current user have privilege for type |
| pg\_has\_role(*user*, *role*, *privilege*) | boolean | does user have privilege for role |
| pg\_has\_role(*role*, *privilege*) | boolean | does current user have privilege for role |
| row\_security\_active(*table*) | boolean | does current user have row level security active for table |

has\_table\_privilege checks whether a user can access a table in a particular way. The user can be specified by name, by OID (pg\_authid.oid), public to indicate the PUBLIC pseudo-role, or if the argument is omitted current\_user is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of has\_table\_privilege, which can be distinguished by the number and types of their arguments.) When specifying by name, the name can be schema-qualified if necessary. The desired access privilege type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, or TRIGGER. Optionally, WITH GRANT OPTION can be added to a privilege type to test whether the privilege is held with grant option. Also, multiple privilege types can be listed separated by commas, in which case the result will be true if any of the listed privileges is held. (Case of the privilege string is not significant, and extra whitespace is allowed between but not within privilege names.) Some examples:

SELECT has\_table\_privilege('myschema.mytable', 'select');

SELECT has\_table\_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');

has\_sequence\_privilege checks whether a user can access a sequence in a particular way. The possibilities for its arguments are analogous to has\_table\_privilege. The desired access privilege type must evaluate to one of USAGE, SELECT, or UPDATE.

has\_any\_column\_privilege checks whether a user can access any column of a table in a particular way. Its argument possibilities are analogous to has\_table\_privilege, except that the desired access privilege type must evaluate to some combination of SELECT, INSERT, UPDATE, or REFERENCES. Note that having any of these privileges at the table level implicitly grants it for each column of the table, so has\_any\_column\_privilege will always return true if has\_table\_privilege does for the same arguments. But has\_any\_column\_privilege also succeeds if there is a column-level grant of the privilege for at least one column.

has\_column\_privilege checks whether a user can access a column in a particular way. Its argument possibilities are analogous to has\_table\_privilege, with the addition that the column can be specified either by name or attribute number. The desired access privilege type must evaluate to some combination of SELECT, INSERT, UPDATE, or REFERENCES. Note that having any of these privileges at the table level implicitly grants it for each column of the table.

has\_database\_privilege checks whether a user can access a database in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to some combination of CREATE, CONNECT, TEMPORARY, or TEMP (which is equivalent to TEMPORARY).

has\_function\_privilege checks whether a user can access a function in a particular way. Its argument possibilities are analogous to has\_table\_privilege. When specifying a function by a text string rather than by OID, the allowed input is the same as for the regprocedure data type (see [**Section 8.18**](https://www.postgresql.org/docs/10/datatype-oid.html)). The desired access privilege type must evaluate to EXECUTE. An example is:

SELECT has\_function\_privilege('joeuser', 'myfunc(int, text)', 'execute');

has\_foreign\_data\_wrapper\_privilege checks whether a user can access a foreign-data wrapper in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to USAGE.

has\_language\_privilege checks whether a user can access a procedural language in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to USAGE.

has\_schema\_privilege checks whether a user can access a schema in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to some combination of CREATE or USAGE.

has\_server\_privilege checks whether a user can access a foreign server in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to USAGE.

has\_tablespace\_privilege checks whether a user can access a tablespace in a particular way. Its argument possibilities are analogous to has\_table\_privilege. The desired access privilege type must evaluate to CREATE.

has\_type\_privilege checks whether a user can access a type in a particular way. Its argument possibilities are analogous to has\_table\_privilege. When specifying a type by a text string rather than by OID, the allowed input is the same as for the regtype data type (see [**Section 8.18**](https://www.postgresql.org/docs/10/datatype-oid.html)). The desired access privilege type must evaluate to USAGE.

pg\_has\_role checks whether a user can access a role in a particular way. Its argument possibilities are analogous to has\_table\_privilege, except that public is not allowed as a user name. The desired access privilege type must evaluate to some combination of MEMBER or USAGE. MEMBER denotes direct or indirect membership in the role (that is, the right to do SET ROLE), while USAGE denotes whether the privileges of the role are immediately available without doing SET ROLE.

row\_security\_active checks whether row level security is active for the specified table in the context of the current\_user and environment. The table can be specified by name or by OID.

[**Table 9.62**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-SCHEMA-TABLE) shows functions that determine whether a certain object is visible in the current schema search path. For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. To list the names of all visible tables:

SELECT relname FROM pg\_class WHERE pg\_table\_is\_visible(oid);

**Table 9.62. Schema Visibility Inquiry Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_collation\_is\_visible(*collation\_oid*) | boolean | is collation visible in search path |
| pg\_conversion\_is\_visible(*conversion\_oid*) | boolean | is conversion visible in search path |
| pg\_function\_is\_visible(*function\_oid*) | boolean | is function visible in search path |
| pg\_opclass\_is\_visible(*opclass\_oid*) | boolean | is operator class visible in search path |
| pg\_operator\_is\_visible(*operator\_oid*) | boolean | is operator visible in search path |
| pg\_opfamily\_is\_visible(*opclass\_oid*) | boolean | is operator family visible in search path |
| pg\_statistics\_obj\_is\_visible(*stat\_oid*) | boolean | is statistics object visible in search path |
| pg\_table\_is\_visible(*table\_oid*) | boolean | is table visible in search path |
| pg\_ts\_config\_is\_visible(*config\_oid*) | boolean | is text search configuration visible in search path |
| pg\_ts\_dict\_is\_visible(*dict\_oid*) | boolean | is text search dictionary visible in search path |
| pg\_ts\_parser\_is\_visible(*parser\_oid*) | boolean | is text search parser visible in search path |
| pg\_ts\_template\_is\_visible(*template\_oid*) | boolean | is text search template visible in search path |
| pg\_type\_is\_visible(*type\_oid*) | boolean | is type (or domain) visible in search path |

Each function performs the visibility check for one type of database object. Note that pg\_table\_is\_visible can also be used with views, materialized views, indexes, sequences and foreign tables; pg\_type\_is\_visible can also be used with domains. For functions and operators, an object in the search path is visible if there is no object of the same name and argument data type(s) earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (regclass, regtype, regprocedure, regoperator, regconfig, or regdictionary), for example:

SELECT pg\_type\_is\_visible('myschema.widget'::regtype);

Note that it would not make much sense to test a non-schema-qualified type name in this way — if the name can be recognized at all, it must be visible.

**[Table 9.63](https://www.postgresql.org/docs/10/functions-info.html" \l "FUNCTIONS-INFO-CATALOG-TABLE" \o "Table 9.63. System Catalog Information Functions)** lists functions that extract information from the system catalogs.

**Table 9.63. System Catalog Information Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| format\_type(*type\_oid*, *typemod*) | text | get SQL name of a data type |
| pg\_get\_constraintdef(*constraint\_oid*) | text | get definition of a constraint |
| pg\_get\_constraintdef(*constraint\_oid*, *pretty\_bool*) | text | get definition of a constraint |
| pg\_get\_expr(*pg\_node\_tree*, *relation\_oid*) | text | decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter |
| pg\_get\_expr(*pg\_node\_tree*, *relation\_oid*,*pretty\_bool*) | text | decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter |
| pg\_get\_functiondef(*func\_oid*) | text | get definition of a function |
| pg\_get\_function\_arguments(*func\_oid*) | text | get argument list of function's definition (with default values) |
| pg\_get\_function\_identity\_arguments(*func\_oid*) | text | get argument list to identify a function (without default values) |
| pg\_get\_function\_result(*func\_oid*) | text | get RETURNS clause for function |
| pg\_get\_indexdef(*index\_oid*) | text | get CREATE INDEX command for index |
| pg\_get\_indexdef(*index\_oid*, *column\_no*, *pretty\_bool*) | text | get CREATE INDEX command for index, or definition of just one index column when *column\_no* is not zero |
| pg\_get\_keywords() | setof record | get list of SQL keywords and their categories |
| pg\_get\_ruledef(*rule\_oid*) | text | get CREATE RULE command for rule |
| pg\_get\_ruledef(*rule\_oid*, *pretty\_bool*) | text | get CREATE RULE command for rule |
| pg\_get\_serial\_sequence(*table\_name*, *column\_name*) | text | get name of the sequence that a serial or identity column uses |
| pg\_get\_statisticsobjdef(*statobj\_oid*) | text | get CREATE STATISTICS command for extended statistics object |
| pg\_get\_triggerdef(*trigger\_oid*) | text | get CREATE [ CONSTRAINT ] TRIGGER command for trigger |
| pg\_get\_triggerdef(*trigger\_oid*, *pretty\_bool*) | text | get CREATE [ CONSTRAINT ] TRIGGER command for trigger |
| pg\_get\_userbyid(*role\_oid*) | name | get role name with given OID |
| pg\_get\_viewdef(*view\_name*) | text | get underlying SELECT command for view or materialized view (deprecated) |
| pg\_get\_viewdef(*view\_name*, *pretty\_bool*) | text | get underlying SELECT command for view or materialized view (deprecated) |
| pg\_get\_viewdef(*view\_oid*) | text | get underlying SELECT command for view or materialized view |
| pg\_get\_viewdef(*view\_oid*, *pretty\_bool*) | text | get underlying SELECT command for view or materialized view |
| pg\_get\_viewdef(*view\_oid*, *wrap\_column\_int*) | text | get underlying SELECT command for view or materialized view; lines with fields are wrapped to specified number of columns, pretty-printing is implied |
| pg\_index\_column\_has\_property(*index\_oid*, *column\_no*,*prop\_name*) | boolean | test whether an index column has a specified property |
| pg\_index\_has\_property(*index\_oid*, *prop\_name*) | boolean | test whether an index has a specified property |
| pg\_indexam\_has\_property(*am\_oid*, *prop\_name*) | boolean | test whether an index access method has a specified property |
| pg\_options\_to\_table(*reloptions*) | setof record | get the set of storage option name/value pairs |
| pg\_tablespace\_databases(*tablespace\_oid*) | setof oid | get the set of database OIDs that have objects in the tablespace |
| pg\_tablespace\_location(*tablespace\_oid*) | text | get the path in the file system that this tablespace is located in |
| pg\_typeof(*any*) | regtype | get the data type of any value |
| collation for (*any*) | text | get the collation of the argument |
| to\_regclass(*rel\_name*) | regclass | get the OID of the named relation |
| to\_regproc(*func\_name*) | regproc | get the OID of the named function |
| to\_regprocedure(*func\_name*) | regprocedure | get the OID of the named function |
| to\_regoper(*operator\_name*) | regoper | get the OID of the named operator |
| to\_regoperator(*operator\_name*) | regoperator | get the OID of the named operator |
| to\_regtype(*type\_name*) | regtype | get the OID of the named type |
| to\_regnamespace(*schema\_name*) | regnamespace | get the OID of the named schema |
| to\_regrole(*role\_name*) | regrole | get the OID of the named role |

format\_type returns the SQL name of a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known.

pg\_get\_keywords returns a set of records describing the SQL keywords recognized by the server. The word column contains the keyword. The catcode column contains a category code: U for unreserved, C for column name, T for type or function name, or R for reserved. The catdesc column contains a possibly-localized string describing the category.

pg\_get\_constraintdef, pg\_get\_indexdef, pg\_get\_ruledef, pg\_get\_statisticsobjdef, and pg\_get\_triggerdef, respectively reconstruct the creating command for a constraint, index, rule, extended statistics object, or trigger. (Note that this is a decompiled reconstruction, not the original text of the command.) pg\_get\_expr decompiles the internal form of an individual expression, such as the default value for a column. It can be useful when examining the contents of system catalogs. If the expression might contain Vars, specify the OID of the relation they refer to as the second parameter; if no Vars are expected, zero is sufficient. pg\_get\_viewdef reconstructs the SELECT query that defines a view. Most of these functions come in two variants, one of which can optionally “pretty-print” the result. The pretty-printed format is more readable, but the default format is more likely to be interpreted the same way by future versions of PostgreSQL; avoid using pretty-printed output for dump purposes. Passing false for the pretty-print parameter yields the same result as the variant that does not have the parameter at all.

pg\_get\_functiondef returns a complete CREATE OR REPLACE FUNCTION statement for a function. pg\_get\_function\_arguments returns the argument list of a function, in the form it would need to appear in within CREATE FUNCTION. pg\_get\_function\_result similarly returns the appropriate RETURNS clause for the function. pg\_get\_function\_identity\_arguments returns the argument list necessary to identify a function, in the form it would need to appear in within ALTER FUNCTION, for instance. This form omits default values.

pg\_get\_serial\_sequence returns the name of the sequence associated with a column, or NULL if no sequence is associated with the column. If the column is an identity column, the associated sequence is the sequence internally created for the identity column. For columns created using one of the serial types (serial, smallserial, bigserial), it is the sequence created for that serial column definition. In the latter case, this association can be modified or removed with ALTER SEQUENCE OWNED BY. (The function probably should have been called pg\_get\_owned\_sequence; its current name reflects the fact that it has typically been used with serial or bigserial columns.) The first input parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter is potentially a schema and table, it is not treated as a double-quoted identifier, meaning it is lower cased by default, while the second parameter, being just a column name, is treated as double-quoted and has its case preserved. The function returns a value suitably formatted for passing to sequence functions (see [**Section 9.16**](https://www.postgresql.org/docs/10/functions-sequence.html)). A typical use is in reading the current value of a sequence for an identity or serial column, for example:

SELECT currval(pg\_get\_serial\_sequence('sometable', 'id'));

pg\_get\_userbyid extracts a role's name given its OID.

pg\_index\_column\_has\_property, pg\_index\_has\_property, and pg\_indexam\_has\_property return whether the specified index column, index, or index access method possesses the named property. NULL is returned if the property name is not known or does not apply to the particular object, or if the OID or column number does not identify a valid object. Refer to [**Table 9.64**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-INDEX-COLUMN-PROPS) for column properties, [**Table 9.65**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-INDEX-PROPS) for index properties, and [**Table 9.66**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-INDEXAM-PROPS) for access method properties. (Note that extension access methods can define additional property names for their indexes.)

**Table 9.64. Index Column Properties**

| **Name** | **Description** |
| --- | --- |
| asc | Does the column sort in ascending order on a forward scan? |
| desc | Does the column sort in descending order on a forward scan? |
| nulls\_first | Does the column sort with nulls first on a forward scan? |
| nulls\_last | Does the column sort with nulls last on a forward scan? |
| orderable | Does the column possess any defined sort ordering? |
| distance\_orderable | Can the column be scanned in order by a “distance” operator, for example ORDER BY col <-> constant ? |
| returnable | Can the column value be returned by an index-only scan? |
| search\_array | Does the column natively support col = ANY(array) searches? |
| search\_nulls | Does the column support IS NULL and IS NOT NULL searches? |

**Table 9.65. Index Properties**

| **Name** | **Description** |
| --- | --- |
| clusterable | Can the index be used in a CLUSTER command? |
| index\_scan | Does the index support plain (non-bitmap) scans? |
| bitmap\_scan | Does the index support bitmap scans? |
| backward\_scan | Can the scan direction be changed in mid-scan (to support FETCH BACKWARD on a cursor without needing materialization)? |

**Table 9.66. Index Access Method Properties**

| **Name** | **Description** |
| --- | --- |
| can\_order | Does the access method support ASC, DESC and related keywords in CREATE INDEX? |
| can\_unique | Does the access method support unique indexes? |
| can\_multi\_col | Does the access method support indexes with multiple columns? |
| can\_exclude | Does the access method support exclusion constraints? |

pg\_options\_to\_table returns the set of storage option name/value pairs (***option\_name***/***option\_value***) when passed pg\_class.reloptions or pg\_attribute.attoptions.

pg\_tablespace\_databases allows a tablespace to be examined. It returns the set of OIDs of databases that have objects stored in the tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To display the specific objects populating the tablespace, you will need to connect to the databases identified by pg\_tablespace\_databases and query their pg\_classcatalogs.

pg\_typeof returns the OID of the data type of the value that is passed to it. This can be helpful for troubleshooting or dynamically constructing SQL queries. The function is declared as returning regtype, which is an OID alias type (see [**Section 8.18**](https://www.postgresql.org/docs/10/datatype-oid.html)); this means that it is the same as an OID for comparison purposes but displays as a type name. For example:

SELECT pg\_typeof(33);

pg\_typeof

-----------

integer

(1 row)

SELECT typlen FROM pg\_type WHERE oid = pg\_typeof(33);

typlen

--------

4

(1 row)

The expression collation for returns the collation of the value that is passed to it. Example:

SELECT collation for (description) FROM pg\_description LIMIT 1;

pg\_collation\_for

------------------

"default"

(1 row)

SELECT collation for ('foo' COLLATE "de\_DE");

pg\_collation\_for

------------------

"de\_DE"

(1 row)

The value might be quoted and schema-qualified. If no collation is derived for the argument expression, then a null value is returned. If the argument is not of a collatable data type, then an error is raised.

The to\_regclass, to\_regproc, to\_regprocedure, to\_regoper, to\_regoperator, to\_regtype, to\_regnamespace, and to\_regrole functions translate relation, function, operator, type, schema, and role names (given as text) to objects of type regclass, regproc, regprocedure, regoper, regoperator, regtype, regnamespace, and regrole respectively. These functions differ from a cast from text in that they don't accept a numeric OID, and that they return null rather than throwing an error if the name is not found (or, for to\_regproc and to\_regoper, if the given name matches multiple objects).

**[Table 9.67](https://www.postgresql.org/docs/10/functions-info.html" \l "FUNCTIONS-INFO-OBJECT-TABLE" \o "Table 9.67. Object Information and Addressing Functions)** lists functions related to database object identification and addressing.

**Table 9.67. Object Information and Addressing Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_describe\_object(*classid* oid, *objid* oid, *objsubid* integer) | text | get description of a database object |
| pg\_identify\_object(*classid* oid, *objid* oid, *objsubid* integer) | *type* text, *schema* text, *name* text, *identity* text | get identity of a database object |
| pg\_identify\_object\_as\_address(*classid* oid, *objid* oid, *objsubid* integer) | *type* text, *object\_names* text[], *object\_args* text[] | get external representation of a database object's address |
| pg\_get\_object\_address(*type* text, *name* text[], *args* text[]) | *classid* oid, *objid* oid, *objsubid* integer | get address of a database object from its external representation |

pg\_describe\_object returns a textual description of a database object specified by catalog OID, object OID, and sub-object ID (such as a column number within a table; the sub-object ID is zero when referring to a whole object). This description is intended to be human-readable, and might be translated, depending on server configuration. This is useful to determine the identity of an object as stored in the pg\_depend catalog.

pg\_identify\_object returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. This information is intended to be machine-readable, and is never translated. *type* identifies the type of database object; *schema* is the schema name that the object belongs in, or NULL for object types that do not belong to schemas; *name* is the name of the object, quoted if necessary, if the name (along with schema name, if pertinent) is sufficient to uniquely identify the object, otherwise NULL; *identity* is the complete object identity, with the precise format depending on object type, and each name within the format being schema-qualified and quoted as necessary.

pg\_identify\_object\_as\_address returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. The returned information is independent of the current server, that is, it could be used to identify an identically named object in another server. *type* identifies the type of database object; *object\_names* and *object\_args* are text arrays that together form a reference to the object. These three values can be passed to pg\_get\_object\_address to obtain the internal address of the object. This function is the inverse of pg\_get\_object\_address.

pg\_get\_object\_address returns a row containing enough information to uniquely identify the database object specified by its type and object name and argument arrays. The returned values are the ones that would be used in system catalogs such as pg\_depend and can be passed to other system functions such as pg\_identify\_object or pg\_describe\_object. *classid* is the OID of the system catalog containing the object; *objid* is the OID of the object itself, and *objsubid* is the sub-object ID, or zero if none. This function is the inverse of pg\_identify\_object\_as\_address.

The functions shown in [**Table 9.68**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-INFO-COMMENT-TABLE) extract comments previously stored with the [**COMMENT**](https://www.postgresql.org/docs/10/sql-comment.html) command. A null value is returned if no comment could be found for the specified parameters.

**Table 9.68. Comment Information Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| col\_description(*table\_oid*, *column\_number*) | text | get comment for a table column |
| obj\_description(*object\_oid*, *catalog\_name*) | text | get comment for a database object |
| obj\_description(*object\_oid*) | text | get comment for a database object (deprecated) |
| shobj\_description(*object\_oid*, *catalog\_name*) | text | get comment for a shared database object |

col\_description returns the comment for a table column, which is specified by the OID of its table and its column number. (obj\_description cannot be used for table columns since columns do not have OIDs of their own.)

The two-parameter form of obj\_description returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, obj\_description(123456,'pg\_class') would retrieve the comment for the table with OID 123456. The one-parameter form of obj\_description requires only the object OID. It is deprecated since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment might be returned.

shobj\_description is used just like obj\_description except it is used for retrieving comments on shared objects. Some system catalogs are global to all databases within each cluster, and the descriptions for objects in them are stored globally as well.

The functions shown in [**Table 9.69**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-TXID-SNAPSHOT) provide server transaction information in an exportable form. The main use of these functions is to determine which transactions were committed between two snapshots.

**Table 9.69. Transaction IDs and Snapshots**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| txid\_current() | bigint | get current transaction ID, assigning a new one if the current transaction does not have one |
| txid\_current\_if\_assigned() | bigint | same as txid\_current() but returns null instead of assigning a new transaction ID if none is already assigned |
| txid\_current\_snapshot() | txid\_snapshot | get current snapshot |
| txid\_snapshot\_xip(*txid\_snapshot*) | setof bigint | get in-progress transaction IDs in snapshot |
| txid\_snapshot\_xmax(*txid\_snapshot*) | bigint | get xmax of snapshot |
| txid\_snapshot\_xmin(*txid\_snapshot*) | bigint | get xmin of snapshot |
| txid\_visible\_in\_snapshot(*bigint*, *txid\_snapshot*) | boolean | is transaction ID visible in snapshot? (do not use with subtransaction ids) |
| txid\_status(*bigint*) | text | report the status of the given transaction: committed, aborted, in progress, or null if the transaction ID is too old |

The internal transaction ID type (xid) is 32 bits wide and wraps around every 4 billion transactions. However, these functions export a 64-bit format that is extended with an “epoch” counter so it will not wrap around during the life of an installation. The data type used by these functions, txid\_snapshot, stores information about transaction ID visibility at a particular moment in time. Its components are described in [**Table 9.70**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-TXID-SNAPSHOT-PARTS).

**Table 9.70. Snapshot Components**

| **Name** | **Description** |
| --- | --- |
| xmin | Earliest transaction ID (txid) that is still active. All earlier transactions will either be committed and visible, or rolled back and dead. |
| xmax | First as-yet-unassigned txid. All txids greater than or equal to this are not yet started as of the time of the snapshot, and thus invisible. |
| xip\_list | Active txids at the time of the snapshot. The list includes only those active txids between xmin and xmax; there might be active txids higher than xmax. A txid that is xmin <= txid < xmax and not in this list was already completed at the time of the snapshot, and thus either visible or dead according to its commit status. The list does not include txids of subtransactions. |

txid\_snapshot's textual representation is ***xmin***:***xmax***:***xip\_list***. For example 10:20:10,14,15 means xmin=10, xmax=20, xip\_list=10, 14, 15.

txid\_status(bigint) reports the commit status of a recent transaction. Applications may use it to determine whether a transaction committed or aborted when the application and database server become disconnected while a COMMIT is in progress. The status of a transaction will be reported as either in progress, committed, or aborted, provided that the transaction is recent enough that the system retains the commit status of that transaction. If is old enough that no references to that transaction survive in the system and the commit status information has been discarded, this function will return NULL. Note that prepared transactions are reported as in progress; applications must check [pg\_prepared\_xacts](https://www.postgresql.org/docs/10/view-pg-prepared-xacts.html) if they need to determine whether the txid is a prepared transaction.

The functions shown in [**Table 9.71**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-COMMIT-TIMESTAMP) provide information about transactions that have been already committed. These functions mainly provide information about when the transactions were committed. They only provide useful data when [**track\_commit\_timestamp**](https://www.postgresql.org/docs/10/runtime-config-replication.html#GUC-TRACK-COMMIT-TIMESTAMP) configuration option is enabled and only for transactions that were committed after it was enabled.

**Table 9.71. Committed transaction information**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_xact\_commit\_timestamp(*xid*) | timestamp with time zone | get commit timestamp of a transaction |
| pg\_last\_committed\_xact() | *xid* xid, *timestamp* timestamp with time zone | get transaction ID and commit timestamp of latest committed transaction |

The functions shown in [**Table 9.72**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-CONTROLDATA) print information initialized during initdb, such as the catalog version. They also show information about write-ahead logging and checkpoint processing. This information is cluster-wide, and not specific to any one database. They provide most of the same information, from the same source, as [**pg\_controldata**](https://www.postgresql.org/docs/10/app-pgcontroldata.html), although in a form better suited to SQLfunctions.

**Table 9.72. Control Data Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_control\_checkpoint() | record | Returns information about current checkpoint state. |
| pg\_control\_system() | record | Returns information about current control file state. |
| pg\_control\_init() | record | Returns information about cluster initialization state. |
| pg\_control\_recovery() | record | Returns information about recovery state. |

pg\_control\_checkpoint returns a record, shown in [**Table 9.73**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-PG-CONTROL-CHECKPOINT)

**Table 9.73.**pg\_control\_checkpoint**Columns**

| **Column Name** | **Data Type** |
| --- | --- |
| checkpoint\_lsn | pg\_lsn |
| prior\_lsn | pg\_lsn |
| redo\_lsn | pg\_lsn |
| redo\_wal\_file | text |
| timeline\_id | integer |
| prev\_timeline\_id | integer |
| full\_page\_writes | boolean |
| next\_xid | text |
| next\_oid | oid |
| next\_multixact\_id | xid |
| next\_multi\_offset | xid |
| oldest\_xid | xid |
| oldest\_xid\_dbid | oid |
| oldest\_active\_xid | xid |
| oldest\_multi\_xid | xid |
| oldest\_multi\_dbid | oid |
| oldest\_commit\_ts\_xid | xid |
| newest\_commit\_ts\_xid | xid |
| checkpoint\_time | timestamp with time zone |

pg\_control\_system returns a record, shown in [**Table 9.74**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-PG-CONTROL-SYSTEM)

**Table 9.74.**pg\_control\_system**Columns**

| **Column Name** | **Data Type** |
| --- | --- |
| pg\_control\_version | integer |
| catalog\_version\_no | integer |
| system\_identifier | bigint |
| pg\_control\_last\_modified | timestamp with time zone |

pg\_control\_init returns a record, shown in [**Table 9.75**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-PG-CONTROL-INIT)

**Table 9.75.**pg\_control\_init**Columns**

| **Column Name** | **Data Type** |
| --- | --- |
| max\_data\_alignment | integer |
| database\_block\_size | integer |
| blocks\_per\_segment | integer |
| wal\_block\_size | integer |
| bytes\_per\_wal\_segment | integer |
| max\_identifier\_length | integer |
| max\_index\_columns | integer |
| max\_toast\_chunk\_size | integer |
| large\_object\_chunk\_size | integer |
| float4\_pass\_by\_value | boolean |
| float8\_pass\_by\_value | boolean |
| data\_page\_checksum\_version | integer |

pg\_control\_recovery returns a record, shown in [**Table 9.76**](https://www.postgresql.org/docs/10/functions-info.html#FUNCTIONS-PG-CONTROL-RECOVERY)

**Table 9.76.**pg\_control\_recovery**Columns**

| **Column Name** | **Data Type** |
| --- | --- |
| min\_recovery\_end\_lsn | pg\_lsn |
| min\_recovery\_end\_timeline | integer |
| backup\_start\_lsn | pg\_lsn |
| backup\_end\_lsn | pg\_lsn |
| end\_of\_backup\_record\_required | boolean |

## 9.26. System Administration Functions

The functions described in this section are used to control and monitor a PostgreSQL installation.

### 9.26.1. Configuration Settings Functions

[**Table 9.77**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-SET-TABLE) shows the functions available to query and alter run-time configuration parameters.

**Table 9.77. Configuration Settings Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| current\_setting(*setting\_name* [, *missing\_ok* ]) | text | get current value of setting |
| set\_config(*setting\_name*, *new\_value*, *is\_local*) | text | set parameter and return new value |

The function current\_setting yields the current value of the setting *setting\_name*. It corresponds to the SQL command SHOW. An example:

SELECT current\_setting('datestyle');

current\_setting

-----------------

ISO, MDY

(1 row)

If there is no setting named *setting\_name*, current\_setting throws an error unless *missing\_ok* is supplied and is true.

set\_config sets the parameter *setting\_name* to *new\_value*. If *is\_local* is true, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use false instead. The function corresponds to the SQL command SET. An example:

SELECT set\_config('log\_statement\_stats', 'off', false);

set\_config

------------

off

(1 row)

### 9.26.2. Server Signaling Functions

The functions shown in [**Table 9.78**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-SIGNAL-TABLE) send control signals to other server processes. Use of these functions is restricted to superusers by default but access may be granted to others using GRANT, with noted exceptions.

**Table 9.78. Server Signaling Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_cancel\_backend(*pid*int) | boolean | Cancel a backend's current query. This is also allowed if the calling role is a member of the role whose backend is being canceled or the calling role has been granted pg\_signal\_backend, however only superusers can cancel superuser backends. |
| pg\_reload\_conf() | boolean | Cause server processes to reload their configuration files |
| pg\_rotate\_logfile() | boolean | Rotate server's log file |
| pg\_terminate\_backend(*pid*int) | boolean | Terminate a backend. This is also allowed if the calling role is a member of the role whose backend is being terminated or the calling role has been granted pg\_signal\_backend, however only superusers can terminate superuser backends. |

Each of these functions returns true if successful and false otherwise.

pg\_cancel\_backend and pg\_terminate\_backend send signals (SIGINT or SIGTERM respectively) to backend processes identified by process ID. The process ID of an active backend can be found from the pid column of the pg\_stat\_activity view, or by listing the postgres processes on the server (using ps on Unix or the Task Manager on Windows). The role of an active backend can be found from the usename column of the pg\_stat\_activity view.

pg\_reload\_conf sends a SIGHUP signal to the server, causing configuration files to be reloaded by all server processes.

pg\_rotate\_logfile signals the log-file manager to switch to a new output file immediately. This works only when the built-in log collector is running, since otherwise there is no log-file manager subprocess.

### 9.26.3. Backup Control Functions

The functions shown in [**Table 9.79**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-BACKUP-TABLE) assist in making on-line backups. These functions cannot be executed during recovery (except non-exclusive pg\_start\_backup, non-exclusive pg\_stop\_backup, pg\_is\_in\_backup, pg\_backup\_start\_time and pg\_wal\_lsn\_diff).

**Table 9.79. Backup Control Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_create\_restore\_point(*name* text) | pg\_lsn | Create a named point for performing restore (restricted to superusers by default, but other users can be granted EXECUTE to run the function) |
| pg\_current\_wal\_flush\_lsn() | pg\_lsn | Get current write-ahead log flush location |
| pg\_current\_wal\_insert\_lsn() | pg\_lsn | Get current write-ahead log insert location |
| pg\_current\_wal\_lsn() | pg\_lsn | Get current write-ahead log write location |
| pg\_start\_backup(*label* text [, *fast* boolean [,*exclusive* boolean ]]) | pg\_lsn | Prepare for performing on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function) |
| pg\_stop\_backup() | pg\_lsn | Finish performing exclusive on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function) |
| pg\_stop\_backup(*exclusive* boolean [,*wait\_for\_archive* boolean ]) | setof record | Finish performing exclusive or non-exclusive on-line backup (restricted to superusers by default, but other users can be granted EXECUTE to run the function) |
| pg\_is\_in\_backup() | bool | True if an on-line exclusive backup is still in progress. |
| pg\_backup\_start\_time() | timestamp with time zone | Get start time of an on-line exclusive backup in progress. |
| pg\_switch\_wal() | pg\_lsn | Force switch to a new write-ahead log file (restricted to superusers by default, but other users can be granted EXECUTE to run the function) |
| pg\_walfile\_name(*lsn* pg\_lsn) | text | Convert write-ahead log location to file name |
| pg\_walfile\_name\_offset(*lsn* pg\_lsn) | text, integer | Convert write-ahead log location to file name and decimal byte offset within file |
| pg\_wal\_lsn\_diff(*lsn* pg\_lsn, *lsn* pg\_lsn) | numeric | Calculate the difference between two write-ahead log locations |

pg\_start\_backup accepts an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) When used in exclusive mode, the function writes a backup label file (backup\_label) and, if there are any links in the pg\_tblspc/ directory, a tablespace map file (tablespace\_map) into the database cluster's data directory, performs a checkpoint, and then returns the backup's starting write-ahead log location as text. The user can ignore this result value, but it is provided in case it is useful. When used in non-exclusive mode, the contents of these files are instead returned by the pg\_stop\_backup function, and should be written to the backup by the caller.

postgres=# select pg\_start\_backup('label\_goes\_here');

pg\_start\_backup

-----------------

0/D4445B8

(1 row)

There is an optional second parameter of type boolean. If true, it specifies executing pg\_start\_backup as quickly as possible. This forces an immediate checkpoint which will cause a spike in I/O operations, slowing any concurrently executing queries.

In an exclusive backup, pg\_stop\_backup removes the label file and, if it exists, the tablespace\_map file created by pg\_start\_backup. In a non-exclusive backup, the contents of the backup\_label and tablespace\_map are returned in the result of the function, and should be written to files in the backup (and not in the data directory). There is an optional second parameter of type boolean. If false, the pg\_stop\_backup will return immediately after the backup is completed without waiting for WAL to be archived. This behavior is only useful for backup software which independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. When this parameter is set to true, pg\_stop\_backup will wait for WAL to be archived when archiving is enabled; on the standby, this means that it will wait only when archive\_mode = always. If write activity on the primary is low, it may be useful to run pg\_switch\_wal on the primary in order to trigger an immediate segment switch.

When executed on a primary, the function also creates a backup history file in the write-ahead log archive area. The history file includes the label given to pg\_start\_backup, the starting and ending write-ahead log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending write-ahead log location (which again can be ignored). After recording the ending location, the current write-ahead log insertion point is automatically advanced to the next write-ahead log file, so that the ending write-ahead log file can be archived immediately to complete the backup.

pg\_switch\_wal moves to the next write-ahead log file, allowing the current file to be archived (assuming you are using continuous archiving). The return value is the ending write-ahead log location + 1 within the just-completed write-ahead log file. If there has been no write-ahead log activity since the last write-ahead log switch, pg\_switch\_wal does nothing and returns the start location of the write-ahead log file currently in use.

pg\_create\_restore\_point creates a named write-ahead log record that can be used as recovery target, and returns the corresponding write-ahead log location. The given name can then be used with [**recovery\_target\_name**](https://www.postgresql.org/docs/10/recovery-target-settings.html#RECOVERY-TARGET-NAME) to specify the point up to which recovery will proceed. Avoid creating multiple restore points with the same name, since recovery will stop at the first one whose name matches the recovery target.

pg\_current\_wal\_lsn displays the current write-ahead log write location in the same format used by the above functions. Similarly, pg\_current\_wal\_insert\_lsn displays the current write-ahead log insertion location and pg\_current\_wal\_flush\_lsn displays the current write-ahead log flush location. The insertion location is the “logical” end of the write-ahead log at any instant, while the write location is the end of what has actually been written out from the server's internal buffers and flush location is the location guaranteed to be written to durable storage. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete write-ahead log files. The insertion and flush locations are made available primarily for server debugging purposes. These are both read-only operations and do not require superuser permissions.

You can use pg\_walfile\_name\_offset to extract the corresponding write-ahead log file name and byte offset from the results of any of the above functions. For example:

postgres=# SELECT \* FROM pg\_walfile\_name\_offset(pg\_stop\_backup());

file\_name | file\_offset

--------------------------+-------------

00000001000000000000000D | 4039624

(1 row)

Similarly, pg\_walfile\_name extracts just the write-ahead log file name. When the given write-ahead log location is exactly at a write-ahead log file boundary, both these functions return the name of the preceding write-ahead log file. This is usually the desired behavior for managing write-ahead log archiving behavior, since the preceding file is the last one that currently needs to be archived.

pg\_wal\_lsn\_diff calculates the difference in bytes between two write-ahead log locations. It can be used with pg\_stat\_replication or some functions shown in [**Table 9.79**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-BACKUP-TABLE) to get the replication lag.

For details about proper usage of these functions, see [**Section 25.3**](https://www.postgresql.org/docs/10/continuous-archiving.html).

### 9.26.4. Recovery Control Functions

The functions shown in [**Table 9.80**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-RECOVERY-INFO-TABLE) provide information about the current status of the standby. These functions may be executed both during recovery and in normal running.

**Table 9.80. Recovery Information Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_is\_in\_recovery() | bool | True if recovery is still in progress. |
| pg\_last\_wal\_receive\_lsn() | pg\_lsn | Get last write-ahead log location received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed this will remain static at the value of the last WAL record received and synced to disk during recovery. If streaming replication is disabled, or if it has not yet started, the function returns NULL. |
| pg\_last\_wal\_replay\_lsn() | pg\_lsn | Get last write-ahead log location replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last WAL record applied during that recovery. When the server has been started normally without recovery the function returns NULL. |
| pg\_last\_xact\_replay\_timestamp() | timestamp with time zone | Get time stamp of last transaction replayed during recovery. This is the time at which the commit or abort WAL record for that transaction was generated on the primary. If no transactions have been replayed during recovery, this function returns NULL. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without recovery the function returns NULL. |

The functions shown in [**Table 9.81**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-RECOVERY-CONTROL-TABLE) control the progress of recovery. These functions may be executed only during recovery.

**Table 9.81. Recovery Control Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_is\_wal\_replay\_paused() | bool | True if recovery is paused. |
| pg\_wal\_replay\_pause() | void | Pauses recovery immediately (restricted to superusers by default, but other users can be granted EXECUTE to run the function). |
| pg\_wal\_replay\_resume() | void | Restarts recovery if it was paused (restricted to superusers by default, but other users can be granted EXECUTE to run the function). |

While recovery is paused no further database changes are applied. If in hot standby, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until recovery is resumed.

If streaming replication is disabled, the paused state may continue indefinitely without problem. While streaming replication is in progress WAL records will continue to be received, which will eventually fill available disk space, depending upon the duration of the pause, the rate of WAL generation and available disk space.

### 9.26.5. Snapshot Synchronization Functions

PostgreSQL allows database sessions to synchronize their snapshots. A snapshot determines which data is visible to the transaction that is using the snapshot. Synchronized snapshots are necessary when two or more sessions need to see identical content in the database. If two sessions just start their transactions independently, there is always a possibility that some third transaction commits between the executions of the two START TRANSACTION commands, so that one session sees the effects of that transaction and the other does not.

To solve this problem, PostgreSQL allows a transaction to export the snapshot it is using. As long as the exporting transaction remains open, other transactions can import its snapshot, and thereby be guaranteed that they see exactly the same view of the database that the first transaction sees. But note that any database changes made by any one of these transactions remain invisible to the other transactions, as is usual for changes made by uncommitted transactions. So the transactions are synchronized with respect to pre-existing data, but act normally for changes they make themselves.

Snapshots are exported with the pg\_export\_snapshot function, shown in [**Table 9.82**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION-TABLE), and imported with the [**SET TRANSACTION**](https://www.postgresql.org/docs/10/sql-set-transaction.html) command.

**Table 9.82. Snapshot Synchronization Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_export\_snapshot() | text | Save the current snapshot and return its identifier |

The function pg\_export\_snapshot saves the current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it. A transaction can export more than one snapshot, if needed. Note that doing so is only useful in READ COMMITTED transactions, since in REPEATABLE READ and higher isolation levels, transactions use the same snapshot throughout their lifetime. Once a transaction has exported any snapshots, it cannot be prepared with [**PREPARE TRANSACTION**](https://www.postgresql.org/docs/10/sql-prepare-transaction.html).

See [**SET TRANSACTION**](https://www.postgresql.org/docs/10/sql-set-transaction.html) for details of how to use an exported snapshot.

### 9.26.6. Replication Functions

The functions shown in [**Table 9.83**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-REPLICATION-TABLE) are for controlling and interacting with replication features. See [**Section 26.2.5**](https://www.postgresql.org/docs/10/warm-standby.html#STREAMING-REPLICATION), [**Section 26.2.6**](https://www.postgresql.org/docs/10/warm-standby.html#STREAMING-REPLICATION-SLOTS), and [**Chapter 49**](https://www.postgresql.org/docs/10/replication-origins.html) for information about the underlying features. Use of functions for replication origin is restricted to superusers. Use of functions for replication slot is restricted to superusers and users having REPLICATION privilege.

Many of these functions have equivalent commands in the replication protocol; see [**Section 52.4**](https://www.postgresql.org/docs/10/protocol-replication.html).

The functions described in [**Section 9.26.3**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-BACKUP), [**Section 9.26.4**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-RECOVERY-CONTROL), and [**Section 9.26.5**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION) are also relevant for replication.

**Table 9.83. Replication SQL Functions**

| **Function** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_create\_physical\_replication\_slot(*slot\_name*name [, *immediately\_reserve* boolean,*temporary* boolean]) | (*slot\_name*name, *lsn*pg\_lsn) | Creates a new physical replication slot named *slot\_name*. The optional second parameter, when true, specifies that the LSN for this replication slot be reserved immediately; otherwise the LSN is reserved on first connection from a streaming replication client. Streaming changes from a physical slot is only possible with the streaming-replication protocol — see [**Section 52.4**](https://www.postgresql.org/docs/10/protocol-replication.html). The optional third parameter, *temporary*, when set to true, specifies that the slot should not be permanently stored to disk and is only meant for use by current session. Temporary slots are also released upon any error. This function corresponds to the replication protocol command CREATE\_REPLICATION\_SLOT ... PHYSICAL. |
| pg\_drop\_replication\_slot(*slot\_name* name) | void | Drops the physical or logical replication slot named *slot\_name*. Same as replication protocol command DROP\_REPLICATION\_SLOT. For logical slots, this must be called when connected to the same database the slot was created on. |
| pg\_create\_logical\_replication\_slot(*slot\_name*name, *plugin* name [, *temporary* boolean]) | (*slot\_name*name, *lsn*pg\_lsn) | Creates a new logical (decoding) replication slot named *slot\_name* using the output plugin *plugin*. The optional third parameter, *temporary*, when set to true, specifies that the slot should not be permanently stored to disk and is only meant for use by current session. Temporary slots are also released upon any error. A call to this function has the same effect as the replication protocol command CREATE\_REPLICATION\_SLOT ... LOGICAL. |
| pg\_logical\_slot\_get\_changes(*slot\_name* name,*upto\_lsn* pg\_lsn, *upto\_nchanges* int, VARIADIC*options* text[]) | (*lsn*pg\_lsn, *xid* xid, *data* text) | Returns changes in the slot *slot\_name*, starting from the point at which since changes have been consumed last. If *upto\_lsn* and *upto\_nchanges* are NULL, logical decoding will continue until end of WAL. If *upto\_lsn* is non-NULL, decoding will include only those transactions which commit prior to the specified LSN. If *upto\_nchanges* is non-NULL, decoding will stop when the number of rows produced by decoding exceeds the specified value. Note, however, that the actual number of rows returned may be larger, since this limit is only checked after adding the rows produced when decoding each new transaction commit. |
| pg\_logical\_slot\_peek\_changes(*slot\_name* name,*upto\_lsn* pg\_lsn, *upto\_nchanges* int, VARIADIC*options* text[]) | (*lsn*pg\_lsn, *xid* xid, *data* text) | Behaves just like the pg\_logical\_slot\_get\_changes() function, except that changes are not consumed; that is, they will be returned again on future calls. |
| pg\_logical\_slot\_get\_binary\_changes(*slot\_name*name, *upto\_lsn* pg\_lsn, *upto\_nchanges* int, VARIADIC *options* text[]) | (*lsn*pg\_lsn, *xid* xid, *data*bytea) | Behaves just like the pg\_logical\_slot\_get\_changes() function, except that changes are returned as bytea. |
| pg\_logical\_slot\_peek\_binary\_changes(*slot\_name*name, *upto\_lsn* pg\_lsn, *upto\_nchanges* int, VARIADIC *options* text[]) | (*lsn*pg\_lsn, *xid* xid, *data*bytea) | Behaves just like the pg\_logical\_slot\_get\_changes() function, except that changes are returned as bytea and that changes are not consumed; that is, they will be returned again on future calls. |
| pg\_replication\_origin\_create(*node\_name* text) | oid | Create a replication origin with the given external name, and return the internal id assigned to it. |
| pg\_replication\_origin\_drop(*node\_name* text) | void | Delete a previously created replication origin, including any associated replay progress. |
| pg\_replication\_origin\_oid(*node\_name* text) | oid | Lookup a replication origin by name and return the internal id. If no corresponding replication origin is found an error is thrown. |
| pg\_replication\_origin\_session\_setup(*node\_name*text) | void | Mark the current session as replaying from the given origin, allowing replay progress to be tracked. Use pg\_replication\_origin\_session\_reset to revert. Can only be used if no previous origin is configured. |
| pg\_replication\_origin\_session\_reset() | void | Cancel the effects of pg\_replication\_origin\_session\_setup(). |
| pg\_replication\_origin\_session\_is\_setup() | bool | Has a replication origin been configured in the current session? |
| pg\_replication\_origin\_session\_progress(*flush*bool) | pg\_lsn | Return the replay location for the replication origin configured in the current session. The parameter *flush* determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not. |
| pg\_replication\_origin\_xact\_setup(*origin\_lsn*pg\_lsn, *origin\_timestamp* timestamptz) | void | Mark the current transaction as replaying a transaction that has committed at the given LSN and timestamp. Can only be called when a replication origin has previously been configured using pg\_replication\_origin\_session\_setup(). |
| pg\_replication\_origin\_xact\_reset() | void | Cancel the effects of pg\_replication\_origin\_xact\_setup(). |
| pg\_replication\_origin\_advance(*node\_name* text,*lsn* pg\_lsn) | void | Set replication progress for the given node to the given location. This primarily is useful for setting up the initial location or a new location after configuration changes and similar. Be aware that careless use of this function can lead to inconsistently replicated data. |
| pg\_replication\_origin\_progress(*node\_name*text, *flush* bool) | pg\_lsn | Return the replay location for the given replication origin. The parameter *flush* determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not. |
| pg\_logical\_emit\_message(*transactional* bool,*prefix* text, *content* text) | pg\_lsn | Emit text logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The parameter *transactional* specifies if the message should be part of current transaction or if it should be written immediately and decoded as soon as the logical decoding reads the record. The *prefix* is textual prefix used by the logical decoding plugins to easily recognize interesting messages for them. The *content* is the text of the message. |
| pg\_logical\_emit\_message(*transactional* bool,*prefix* text, *content* bytea) | pg\_lsn | Emit binary logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The parameter *transactional* specifies if the message should be part of current transaction or if it should be written immediately and decoded as soon as the logical decoding reads the record. The *prefix* is textual prefix used by the logical decoding plugins to easily recognize interesting messages for them. The *content* is the binary content of the message. |

### 9.26.7. Database Object Management Functions

The functions shown in [**Table 9.84**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-DBSIZE) calculate the disk space usage of database objects.

**Table 9.84. Database Object Size Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_column\_size(any) | int | Number of bytes used to store a particular value (possibly compressed) |
| pg\_database\_size(oid) | bigint | Disk space used by the database with the specified OID |
| pg\_database\_size(name) | bigint | Disk space used by the database with the specified name |
| pg\_indexes\_size(regclass) | bigint | Total disk space used by indexes attached to the specified table |
| pg\_relation\_size(*relation* regclass, *fork* text) | bigint | Disk space used by the specified fork ('main', 'fsm', 'vm', or 'init') of the specified table or index |
| pg\_relation\_size(*relation* regclass) | bigint | Shorthand for pg\_relation\_size(..., 'main') |
| pg\_size\_bytes(text) | bigint | Converts a size in human-readable format with size units into bytes |
| pg\_size\_pretty(bigint) | text | Converts a size in bytes expressed as a 64-bit integer into a human-readable format with size units |
| pg\_size\_pretty(numeric) | text | Converts a size in bytes expressed as a numeric value into a human-readable format with size units |
| pg\_table\_size(regclass) | bigint | Disk space used by the specified table, excluding indexes (but including TOAST, free space map, and visibility map) |
| pg\_tablespace\_size(oid) | bigint | Disk space used by the tablespace with the specified OID |
| pg\_tablespace\_size(name) | bigint | Disk space used by the tablespace with the specified name |
| pg\_total\_relation\_size(regclass) | bigint | Total disk space used by the specified table, including all indexes and TOAST data |

pg\_column\_size shows the space used to store any individual data value.

pg\_total\_relation\_size accepts the OID or name of a table or toast table, and returns the total on-disk space used for that table, including all associated indexes. This function is equivalent to pg\_table\_size + pg\_indexes\_size.

pg\_table\_size accepts the OID or name of a table and returns the disk space needed for that table, exclusive of indexes. (TOAST space, free space map, and visibility map are included.)

pg\_indexes\_size accepts the OID or name of a table and returns the total disk space used by all the indexes attached to that table.

pg\_database\_size and pg\_tablespace\_size accept the OID or name of a database or tablespace, and return the total disk space used therein. To use pg\_database\_size, you must have CONNECTpermission on the specified database (which is granted by default), or be a member of the pg\_read\_all\_stats role. To use pg\_tablespace\_size, you must have CREATE permission on the specified tablespace, or be a member of the pg\_read\_all\_stats role unless it is the default tablespace for the current database.

pg\_relation\_size accepts the OID or name of a table, index or toast table, and returns the on-disk size in bytes of one fork of that relation. (Note that for most purposes it is more convenient to use the higher-level functions pg\_total\_relation\_size or pg\_table\_size, which sum the sizes of all forks.) With one argument, it returns the size of the main data fork of the relation. The second argument can be provided to specify which fork to examine:

* 'main' returns the size of the main data fork of the relation.
* 'fsm' returns the size of the Free Space Map (see [**Section 66.3**](https://www.postgresql.org/docs/10/storage-fsm.html)) associated with the relation.
* 'vm' returns the size of the Visibility Map (see [**Section 66.4**](https://www.postgresql.org/docs/10/storage-vm.html)) associated with the relation.
* 'init' returns the size of the initialization fork, if any, associated with the relation.

pg\_size\_pretty can be used to format the result of one of the other functions in a human-readable way, using bytes, kB, MB, GB or TB as appropriate.

pg\_size\_bytes can be used to get the size in bytes from a string in human-readable format. The input may have units of bytes, kB, MB, GB or TB, and is parsed case-insensitively. If no units are specified, bytes are assumed.

Note

The units kB, MB, GB and TB used by the functions pg\_size\_pretty and pg\_size\_bytes are defined using powers of 2 rather than powers of 10, so 1kB is 1024 bytes, 1MB is 10242 = 1048576 bytes, and so on.

The functions above that operate on tables or indexes accept a regclass argument, which is simply the OID of the table or index in the pg\_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type's input converter will do the work for you. Just write the table name enclosed in single quotes so that it looks like a literal constant. For compatibility with the handling of ordinary SQL names, the string will be converted to lower case unless it contains double quotes around the table name.

If an OID that does not represent an existing object is passed as argument to one of the above functions, NULL is returned.

The functions shown in [**Table 9.85**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-DBLOCATION) assist in identifying the specific disk files associated with database objects.

**Table 9.85. Database Object Location Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_relation\_filenode(*relation* regclass) | oid | Filenode number of the specified relation |
| pg\_relation\_filepath(*relation* regclass) | text | File path name of the specified relation |
| pg\_filenode\_relation(*tablespace* oid, *filenode* oid) | regclass | Find the relation associated with a given tablespace and filenode |

pg\_relation\_filenode accepts the OID or name of a table, index, sequence, or toast table, and returns the “filenode” number currently assigned to it. The filenode is the base component of the file name(s) used for the relation (see [**Section 66.1**](https://www.postgresql.org/docs/10/storage-file-layout.html) for more information). For most tables the result is the same as pg\_class.relfilenode, but for certain system catalogs relfilenode is zero and this function must be used to get the correct value. The function returns NULL if passed a relation that does not have storage, such as a view.

pg\_relation\_filepath is similar to pg\_relation\_filenode, but it returns the entire file path name (relative to the database cluster's data directory PGDATA) of the relation.

pg\_filenode\_relation is the reverse of pg\_relation\_filenode. Given a “tablespace” OID and a “filenode”, it returns the associated relation's OID. For a table in the database's default tablespace, the tablespace can be specified as 0.

[**Table 9.86**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-COLLATION) lists functions used to manage collations.

**Table 9.86. Collation Management Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_collation\_actual\_version(oid) | text | Return actual version of collation from operating system |
| pg\_import\_system\_collations(*schema* regnamespace) | integer | Import operating system collations |

pg\_collation\_actual\_version returns the actual version of the collation object as it is currently installed in the operating system. If this is different from the value in pg\_collation.collversion, then objects depending on the collation might need to be rebuilt. See also [**ALTER COLLATION**](https://www.postgresql.org/docs/10/sql-altercollation.html).

pg\_import\_system\_collations adds collations to the system catalog pg\_collation based on all the locales it finds in the operating system. This is what initdb uses; see [**Section 23.2.2**](https://www.postgresql.org/docs/10/collation.html#COLLATION-MANAGING) for more details. If additional locales are installed into the operating system later on, this function can be run again to add collations for the new locales. Locales that match existing entries in pg\_collation will be skipped. (But collation objects based on locales that are no longer present in the operating system are not removed by this function.) The *schema* parameter would typically be pg\_catalog, but that is not a requirement; the collations could be installed into some other schema as well. The function returns the number of new collation objects it created.

### 9.26.8. Index Maintenance Functions

**[Table 9.87](https://www.postgresql.org/docs/10/functions-admin.html" \l "FUNCTIONS-ADMIN-INDEX-TABLE" \o "Table 9.87. Index Maintenance Functions)** shows the functions available for index maintenance tasks. These functions cannot be executed during recovery. Use of these functions is restricted to superusers and the owner of the given index.

**Table 9.87. Index Maintenance Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| brin\_summarize\_new\_values(*index* regclass) | integer | summarize page ranges not already summarized |
| brin\_summarize\_range(*index* regclass, *blockNumber* bigint) | integer | summarize the page range covering the given block, if not already summarized |
| brin\_desummarize\_range(*index* regclass, *blockNumber* bigint) | integer | de-summarize the page range covering the given block, if summarized |
| gin\_clean\_pending\_list(*index* regclass) | bigint | move GIN pending list entries into main index structure |

brin\_summarize\_new\_values accepts the OID or name of a BRIN index and inspects the index to find page ranges in the base table that are not currently summarized by the index; for any such range it creates a new summary index tuple by scanning the table pages. It returns the number of new page range summaries that were inserted into the index. brin\_summarize\_range does the same, except it only summarizes the range that covers the given block number.

gin\_clean\_pending\_list accepts the OID or name of a GIN index and cleans up the pending list of the specified index by moving entries in it to the main GIN data structure in bulk. It returns the number of pages removed from the pending list. Note that if the argument is a GIN index built with the fastupdate option disabled, no cleanup happens and the return value is 0, because the index doesn't have a pending list. Please see [**Section 64.4.1**](https://www.postgresql.org/docs/10/gin-implementation.html#GIN-FAST-UPDATE) and [**Section 64.5**](https://www.postgresql.org/docs/10/gin-tips.html) for details of the pending list and fastupdate option.

### 9.26.9. Generic File Access Functions

The functions shown in [**Table 9.88**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADMIN-GENFILE-TABLE) provide native access to files on the machine hosting the server. Only files within the database cluster directory and the log\_directory can be accessed. Use a relative path for files in the cluster directory, and a path matching the log\_directory configuration setting for log files. Use of these functions is restricted to superusers except where stated otherwise.

**Table 9.88. Generic File Access Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_ls\_dir(*dirname* text [, *missing\_ok* boolean,*include\_dot\_dirs* boolean]) | setof text | List the contents of a directory. |
| pg\_ls\_logdir() | setof record | List the name, size, and last modification time of files in the log directory. Access is granted to members of the pg\_monitor role and may be granted to other non-superuser roles. |
| pg\_ls\_waldir() | setof record | List the name, size, and last modification time of files in the WAL directory. Access is granted to members of the pg\_monitor role and may be granted to other non-superuser roles. |
| pg\_read\_file(*filename* text [, *offset* bigint, *length* bigint [,*missing\_ok* boolean] ]) | text | Return the contents of a text file. |
| pg\_read\_binary\_file(*filename* text [, *offset* bigint, *length*bigint [, *missing\_ok* boolean] ]) | bytea | Return the contents of a file. |
| pg\_stat\_file(*filename* text[, *missing\_ok* boolean]) | record | Return information about a file. |

Some of these functions take an optional *missing\_ok* parameter, which specifies the behavior when the file or directory does not exist. If true, the function returns NULL (except pg\_ls\_dir, which returns an empty result set). If false, an error is raised. The default is false.

pg\_ls\_dir returns the names of all files (and directories and other special files) in the specified directory. The *include\_dot\_dirs* indicates whether “.” and “..” are included in the result set. The default is to exclude them (false), but including them can be useful when *missing\_ok* is true, to distinguish an empty directory from an non-existent directory.

pg\_ls\_logdir returns the name, size, and last modified time (mtime) of each file in the log directory. By default, only superusers and members of the pg\_monitor role can use this function. Access may be granted to others using GRANT.

pg\_ls\_waldir returns the name, size, and last modified time (mtime) of each file in the write ahead log (WAL) directory. By default only superusers and members of the pg\_monitor role can use this function. Access may be granted to others using GRANT.

pg\_read\_file returns part of a text file, starting at the given *offset*, returning at most *length* bytes (less if the end of file is reached first). If *offset* is negative, it is relative to the end of the file. If *offset*and *length* are omitted, the entire file is returned. The bytes read from the file are interpreted as a string in the server encoding; an error is thrown if they are not valid in that encoding.

pg\_read\_binary\_file is similar to pg\_read\_file, except that the result is a bytea value; accordingly, no encoding checks are performed. In combination with the convert\_from function, this function can be used to read a file in a specified encoding:

SELECT convert\_from(pg\_read\_binary\_file('file\_in\_utf8.txt'), 'UTF8');

pg\_stat\_file returns a record containing the file size, last accessed time stamp, last modified time stamp, last file status change time stamp (Unix platforms only), file creation time stamp (Windows only), and a boolean indicating if it is a directory. Typical usages include:

SELECT \* FROM pg\_stat\_file('filename');

SELECT (pg\_stat\_file('filename')).modification;

### 9.26.10. Advisory Lock Functions

The functions shown in [**Table 9.89**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADVISORY-LOCKS-TABLE) manage advisory locks. For details about proper use of these functions, see [**Section 13.3.5**](https://www.postgresql.org/docs/10/explicit-locking.html#ADVISORY-LOCKS).

**Table 9.89. Advisory Lock Functions**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_advisory\_lock(*key* bigint) | void | Obtain exclusive session level advisory lock |
| pg\_advisory\_lock(*key1* int, *key2* int) | void | Obtain exclusive session level advisory lock |
| pg\_advisory\_lock\_shared(*key* bigint) | void | Obtain shared session level advisory lock |
| pg\_advisory\_lock\_shared(*key1* int, *key2* int) | void | Obtain shared session level advisory lock |
| pg\_advisory\_unlock(*key* bigint) | boolean | Release an exclusive session level advisory lock |
| pg\_advisory\_unlock(*key1* int, *key2* int) | boolean | Release an exclusive session level advisory lock |
| pg\_advisory\_unlock\_all() | void | Release all session level advisory locks held by the current session |
| pg\_advisory\_unlock\_shared(*key* bigint) | boolean | Release a shared session level advisory lock |
| pg\_advisory\_unlock\_shared(*key1* int, *key2* int) | boolean | Release a shared session level advisory lock |
| pg\_advisory\_xact\_lock(*key* bigint) | void | Obtain exclusive transaction level advisory lock |
| pg\_advisory\_xact\_lock(*key1* int, *key2* int) | void | Obtain exclusive transaction level advisory lock |
| pg\_advisory\_xact\_lock\_shared(*key* bigint) | void | Obtain shared transaction level advisory lock |
| pg\_advisory\_xact\_lock\_shared(*key1* int, *key2* int) | void | Obtain shared transaction level advisory lock |
| pg\_try\_advisory\_lock(*key* bigint) | boolean | Obtain exclusive session level advisory lock if available |
| pg\_try\_advisory\_lock(*key1* int, *key2* int) | boolean | Obtain exclusive session level advisory lock if available |
| pg\_try\_advisory\_lock\_shared(*key* bigint) | boolean | Obtain shared session level advisory lock if available |
| pg\_try\_advisory\_lock\_shared(*key1* int, *key2* int) | boolean | Obtain shared session level advisory lock if available |
| pg\_try\_advisory\_xact\_lock(*key* bigint) | boolean | Obtain exclusive transaction level advisory lock if available |
| pg\_try\_advisory\_xact\_lock(*key1* int, *key2* int) | boolean | Obtain exclusive transaction level advisory lock if available |
| pg\_try\_advisory\_xact\_lock\_shared(*key* bigint) | boolean | Obtain shared transaction level advisory lock if available |
| pg\_try\_advisory\_xact\_lock\_shared(*key1* int, *key2* int) | boolean | Obtain shared transaction level advisory lock if available |

pg\_advisory\_lock locks an application-defined resource, which can be identified either by a single 64-bit key value or two 32-bit key values (note that these two key spaces do not overlap). If another session already holds a lock on the same resource identifier, this function will wait until the resource becomes available. The lock is exclusive. Multiple lock requests stack, so that if the same resource is locked three times it must then be unlocked three times to be released for other sessions' use.

pg\_advisory\_lock\_shared works the same as pg\_advisory\_lock, except the lock can be shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

pg\_try\_advisory\_lock is similar to pg\_advisory\_lock, except the function will not wait for the lock to become available. It will either obtain the lock immediately and return true, or return false if the lock cannot be acquired immediately.

pg\_try\_advisory\_lock\_shared works the same as pg\_try\_advisory\_lock, except it attempts to acquire a shared rather than an exclusive lock.

pg\_advisory\_unlock will release a previously-acquired exclusive session level advisory lock. It returns true if the lock is successfully released. If the lock was not held, it will return false, and in addition, an SQL warning will be reported by the server.

pg\_advisory\_unlock\_shared works the same as pg\_advisory\_unlock, except it releases a shared session level advisory lock.

pg\_advisory\_unlock\_all will release all session level advisory locks held by the current session. (This function is implicitly invoked at session end, even if the client disconnects ungracefully.)

pg\_advisory\_xact\_lock works the same as pg\_advisory\_lock, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg\_advisory\_xact\_lock\_shared works the same as pg\_advisory\_lock\_shared, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg\_try\_advisory\_xact\_lock works the same as pg\_try\_advisory\_lock, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

pg\_try\_advisory\_xact\_lock\_shared works the same as pg\_try\_advisory\_lock\_shared, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

## 9.27. Trigger Functions

Currently PostgreSQL provides one built in trigger function, suppress\_redundant\_updates\_trigger, which will prevent any update that does not actually change the data in the row from taking place, in contrast to the normal behavior which always performs the update regardless of whether or not the data has changed. (This normal behavior makes updates run faster, since no checking is required, and is also useful in certain cases.)

Ideally, you should normally avoid running updates that don't actually change the data in the record. Redundant updates can cost considerable unnecessary time, especially if there are lots of indexes to alter, and space in dead rows that will eventually have to be vacuumed. However, detecting such situations in client code is not always easy, or even possible, and writing expressions to detect them can be error-prone. An alternative is to use suppress\_redundant\_updates\_trigger, which will skip updates that don't change the data. You should use this with care, however. The trigger takes a small but non-trivial time for each record, so if most of the records affected by an update are actually changed, use of this trigger will actually make the update run slower.

The suppress\_redundant\_updates\_trigger function can be added to a table like this:

CREATE TRIGGER z\_min\_update

BEFORE UPDATE ON tablename

FOR EACH ROW EXECUTE PROCEDURE suppress\_redundant\_updates\_trigger();

In most cases, you would want to fire this trigger last for each row. Bearing in mind that triggers fire in name order, you would then choose a trigger name that comes after the name of any other trigger you might have on the table.

For more information about creating triggers, see [**CREATE TRIGGER**](https://www.postgresql.org/docs/10/sql-createtrigger.html).

## 9.28. Event Trigger Functions

PostgreSQL provides these helper functions to retrieve information from event triggers.

For more information about event triggers, see [**Chapter 39**](https://www.postgresql.org/docs/10/event-triggers.html).

### 9.28.1. Capturing Changes at Command End

pg\_event\_trigger\_ddl\_commands returns a list of DDL commands executed by each user action, when invoked in a function attached to a ddl\_command\_end event trigger. If called in any other context, an error is raised. pg\_event\_trigger\_ddl\_commands returns one row for each base command executed; some commands that are a single SQL sentence may return more than one row. This function returns the following columns:

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| classid | oid | OID of catalog the object belongs in |
| objid | oid | OID of the object itself |
| objsubid | integer | Sub-object ID (e.g. attribute number for a column) |
| command\_tag | text | Command tag |
| object\_type | text | Type of the object |
| schema\_name | text | Name of the schema the object belongs in, if any; otherwise NULL. No quoting is applied. |
| object\_identity | text | Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary. |
| in\_extension | bool | True if the command is part of an extension script |
| command | pg\_ddl\_command | A complete representation of the command, in internal format. This cannot be output directly, but it can be passed to other functions to obtain different pieces of information about the command. |

### 9.28.2. Processing Objects Dropped by a DDL Command

pg\_event\_trigger\_dropped\_objects returns a list of all objects dropped by the command in whose sql\_drop event it is called. If called in any other context, pg\_event\_trigger\_dropped\_objects raises an error. pg\_event\_trigger\_dropped\_objects returns the following columns:

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| classid | oid | OID of catalog the object belonged in |
| objid | oid | OID of the object itself |
| objsubid | integer | Sub-object ID (e.g. attribute number for a column) |
| original | bool | True if this was one of the root object(s) of the deletion |
| normal | bool | True if there was a normal dependency relationship in the dependency graph leading to this object |
| is\_temporary | bool | True if this was a temporary object |
| object\_type | text | Type of the object |
| schema\_name | text | Name of the schema the object belonged in, if any; otherwise NULL. No quoting is applied. |
| object\_name | text | Name of the object, if the combination of schema and name can be used as a unique identifier for the object; otherwise NULL. No quoting is applied, and name is never schema-qualified. |
| object\_identity | text | Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary. |
| address\_names | text[] | An array that, together with object\_type and address\_args, can be used by the pg\_get\_object\_address() function to recreate the object address in a remote server containing an identically named object of the same kind |
| address\_args | text[] | Complement for address\_names |

The pg\_event\_trigger\_dropped\_objects function can be used in an event trigger like this:

CREATE FUNCTION test\_event\_trigger\_for\_drops()

RETURNS event\_trigger LANGUAGE plpgsql AS $$

DECLARE

obj record;

BEGIN

FOR obj IN SELECT \* FROM pg\_event\_trigger\_dropped\_objects()

LOOP

RAISE NOTICE '% dropped object: % %.% %',

tg\_tag,

obj.object\_type,

obj.schema\_name,

obj.object\_name,

obj.object\_identity;

END LOOP;

END

$$;

CREATE EVENT TRIGGER test\_event\_trigger\_for\_drops

ON sql\_drop

EXECUTE PROCEDURE test\_event\_trigger\_for\_drops();

### 9.28.3. Handling a Table Rewrite Event

The functions shown in [**Table 9.90**](https://www.postgresql.org/docs/10/functions-event-triggers.html#FUNCTIONS-EVENT-TRIGGER-TABLE-REWRITE) provide information about a table for which a table\_rewrite event has just been called. If called in any other context, an error is raised.

**Table 9.90. Table Rewrite information**

| **Name** | **Return Type** | **Description** |
| --- | --- | --- |
| pg\_event\_trigger\_table\_rewrite\_oid() | Oid | The OID of the table about to be rewritten. |
| pg\_event\_trigger\_table\_rewrite\_reason() | int | The reason code(s) explaining the reason for rewriting. The exact meaning of the codes is release dependent. |

The pg\_event\_trigger\_table\_rewrite\_oid function can be used in an event trigger like this:

CREATE FUNCTION test\_event\_trigger\_table\_rewrite\_oid()

RETURNS event\_trigger

LANGUAGE plpgsql AS

$$

BEGIN

RAISE NOTICE 'rewriting table % for reason %',

pg\_event\_trigger\_table\_rewrite\_oid()::regclass,

pg\_event\_trigger\_table\_rewrite\_reason();

END;

$$;

CREATE EVENT TRIGGER test\_table\_rewrite\_oid

ON table\_rewrite

EXECUTE PROCEDURE test\_event\_trigger\_table\_rewrite\_oid();

## Chapter 10. Type Conversion

SQL statements can, intentionally or not, require the mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user does not need to understand the details of the type conversion mechanism. However, implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by using explicit type conversion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in [**Chapter 8**](https://www.postgresql.org/docs/10/datatype.html) and [**Chapter 9**](https://www.postgresql.org/docs/10/functions.html) for more information on specific data types and allowed functions and operators.

## 10.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is more general and flexible than other SQL implementations. Hence, most type conversion behavior in PostgreSQL is governed by general rules rather than by ad hoc heuristics. This allows the use of mixed-type expressions even with user-defined types.

The PostgreSQL scanner/parser divides lexical elements into five fundamental categories: integers, non-integer numbers, strings, identifiers, and key words. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query:

SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

label | value

--------+-------

Origin | (0,0)

(1 row)

has two literal constants, of type text and point. If a type is not specified for a string literal, then the placeholder type unknown is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Functions can have one or more arguments. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions, operators can be overloaded, so the same problem of selecting the right operator exists.

Value Storage

SQL INSERT and UPDATE statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

UNION, CASE, and related constructs

Since all query results from a unionized SELECT statement must appear in a single set of columns, the types of the results of each SELECT clause must be matched up and converted to a uniform set. Similarly, the result expressions of a CASE construct must be converted to a common type so that the CASE expression as a whole has a known output type. The same holds for ARRAY constructs, and for the GREATEST and LEAST functions.

The system catalogs store information about which conversions, or casts, exist between which data types, and how to perform those conversions. Additional casts can be added by the user with the [**CREATE CAST**](https://www.postgresql.org/docs/10/sql-createcast.html) command. (This is usually done in conjunction with defining new data types. The set of casts between built-in types has been carefully crafted and is best not altered.)

An additional heuristic provided by the parser allows improved determination of the proper casting behavior among groups of types that have implicit casts. Data types are divided into several basic type categories, including boolean, numeric, string, bitstring, datetime, timespan, geometric, network, and user-defined. (For a list see [**Table 51.63**](https://www.postgresql.org/docs/10/catalog-pg-type.html#CATALOG-TYPCATEGORY-TABLE); but note it is also possible to create custom type categories.) Within each category there can be one or more preferred types, which are preferred when there is a choice of possible types. With careful selection of preferred types and available implicit casts, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in a useful way.

All type conversion rules are designed with several principles in mind:

* Implicit conversions should never have surprising or unpredictable outcomes.
* There should be no extra overhead in the parser or executor if a query does not need implicit type conversion. That is, if a query is well-formed and the types already match, then the query should execute without spending extra time in the parser and without introducing unnecessary implicit conversion calls in the query.
* Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function and no longer do implicit conversion to use the old function.

## 10.2. Operators

The specific operator that is referenced by an operator expression is determined using the following procedure. Note that this procedure is indirectly affected by the precedence of the operators involved, since that will determine which sub-expressions are taken to be the inputs of which operators. See [**Section 4.1.6**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-PRECEDENCE) for more information.

**Operator Type Resolution**

1. Select the operators to be considered from the pg\_operator system catalog. If a non-schema-qualified operator name was used (the usual case), the operators considered are those with the matching name and argument count that are visible in the current search path (see [**Section 5.8.3**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATH)). If a qualified operator name was given, only operators in the specified schema are considered.
   1. If the search path finds multiple operators with identical argument types, only the one appearing earliest in the path is considered. Operators with different argument types are considered on an equal footing regardless of search path position.
2. Check for an operator accepting exactly the input argument types. If one exists (there can be only one exact match in the set of operators considered), use it. Lack of an exact match creates a security hazard when calling, via qualified name [**[8]**](https://www.postgresql.org/docs/10/typeconv-oper.html#ftn.OP-QUALIFIED-SECURITY) (not typical), any operator found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match.
   1. If one argument of a binary operator invocation is of the unknown type, then assume it is the same type as the other argument for this check. Invocations involving two unknown inputs, or a unary operator with an unknown input, will never find a match at this step.
   2. If one argument of a binary operator invocation is of the unknown type and the other is of a domain type, next check to see if there is an operator accepting exactly the domain's base type on both sides; if so, use it.
3. Look for the best match.
   1. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. unknown literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
   2. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-operator resolution.
   3. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.
   4. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.
   5. If any input arguments are unknown, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.
   6. If there are both unknown and known-type arguments, and all the known-type arguments have the same type, assume that the unknown arguments are also of that type, and check which candidates can accept that type at the unknown-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Some examples follow.

**Example 10.1. Factorial Operator Type Resolution**

There is only one factorial operator (postfix !) defined in the standard catalog, and it takes an argument of type bigint. The scanner assigns an initial type of integer to the argument in this query expression:

SELECT 40 ! AS "40 factorial";

40 factorial

--------------------------------------------------

815915283247897734345611269596115894272000000000

(1 row)

So the parser does a type conversion on the operand and the query is equivalent to:

SELECT CAST(40 AS bigint) ! AS "40 factorial";

**Example 10.2. String Concatenation Operator Type Resolution**

A string-like syntax is used for working with string types and for working with complex extension types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

SELECT text 'abc' || 'def' AS "text and unknown";

text and unknown

------------------

abcdef

(1 row)

In this case the parser looks to see if there is an operator taking text for both arguments. Since there is, it assumes that the second argument should be interpreted as type text.

Here is a concatenation of two values of unspecified types:

SELECT 'abc' || 'def' AS "unspecified";

unspecified

-------------

abcdef

(1 row)

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, text, is used as the specific type to resolve the unknown-type literals as.

**Example 10.3. Absolute-Value and Negation Operator Type Resolution**

The PostgreSQL operator catalog has several entries for the prefix operator @, all of which implement absolute-value operations for various numeric data types. One of these entries is for type float8, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with an unknown input:

SELECT @ '-4.5' AS "abs";

abs

-----

4.5

(1 row)

Here the system has implicitly resolved the unknown-type literal as type float8 before applying the chosen operator. We can verify that float8 and not some other type was used:

SELECT @ '-4.5e500' AS "abs";

ERROR: "-4.5e500" is out of range for type double precision

On the other hand, the prefix operator ~ (bitwise negation) is defined only for integer data types, not for float8. So, if we try a similar case with ~, we get:

SELECT ~ '20' AS "negation";

ERROR: operator is not unique: ~ "unknown"

HINT: Could not choose a best candidate operator. You might need to add

explicit type casts.

This happens because the system cannot decide which of the several possible ~ operators should be preferred. We can help it out with an explicit cast:

SELECT ~ CAST('20' AS int8) AS "negation";

negation

----------

-21

(1 row)

**Example 10.4. Array Inclusion Operator Type Resolution**

Here is another example of resolving an operator with one known and one unknown input:

SELECT array[1,2] <@ '{1,2,3}' as "is subset";

is subset

-----------

t

(1 row)

The PostgreSQL operator catalog has several entries for the infix operator <@, but the only two that could possibly accept an integer array on the left-hand side are array inclusion (anyarray <@anyarray) and range inclusion (anyelement <@ anyrange). Since none of these polymorphic pseudo-types (see [**Section 8.20**](https://www.postgresql.org/docs/10/datatype-pseudo.html)) are considered preferred, the parser cannot resolve the ambiguity on that basis. However, [**Step 3.f**](https://www.postgresql.org/docs/10/typeconv-oper.html#OP-RESOL-LAST-UNKNOWN) tells it to assume that the unknown-type literal is of the same type as the other input, that is, integer array. Now only one of the two operators can match, so array inclusion is selected. (Had range inclusion been selected, we would have gotten an error, because the string does not have the right format to be a range literal.)

**Example 10.5. Custom Operator on a Domain Type**

Users sometimes try to declare operators applying just to a domain type. This is possible but is not nearly as useful as it might seem, because the operator resolution rules are designed to select operators applying to the domain's base type. As an example consider

CREATE DOMAIN mytext AS text CHECK(...);

CREATE FUNCTION mytext\_eq\_text (mytext, text) RETURNS boolean AS ...;

CREATE OPERATOR = (procedure=mytext\_eq\_text, leftarg=mytext, rightarg=text);

CREATE TABLE mytable (val mytext);

SELECT \* FROM mytable WHERE val = 'foo';

This query will not use the custom operator. The parser will first see if there is a mytext = mytext operator ([**Step 2.a**](https://www.postgresql.org/docs/10/typeconv-oper.html#OP-RESOL-EXACT-UNKNOWN)), which there is not; then it will consider the domain's base type text, and see if there is a text = text operator ([**Step 2.b**](https://www.postgresql.org/docs/10/typeconv-oper.html#OP-RESOL-EXACT-DOMAIN)), which there is; so it resolves the unknown-type literal as text and uses the text = text operator. The only way to get the custom operator to be used is to explicitly cast the literal:

SELECT \* FROM mytable WHERE val = text 'foo';

so that the mytext = text operator is found immediately according to the exact-match rule. If the best-match rules are reached, they actively discriminate against operators on domain types. If they did not, such an operator would create too many ambiguous-operator failures, because the casting rules always consider a domain as castable to or from its base type, and so the domain operator would be considered usable in all the same cases as a similarly-named operator on the base type.

[**[8]**](https://www.postgresql.org/docs/10/typeconv-oper.html#OP-QUALIFIED-SECURITY) The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a [**secure schema usage pattern**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATTERNS).

## 10.3. Functions

The specific function that is referenced by a function call is determined using the following procedure.

**Function Type Resolution**

1. Select the functions to be considered from the pg\_proc system catalog. If a non-schema-qualified function name was used, the functions considered are those with the matching name and argument count that are visible in the current search path (see [**Section 5.8.3**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATH)). If a qualified function name was given, only functions in the specified schema are considered.
   1. If the search path finds multiple functions of identical argument types, only the one appearing earliest in the path is considered. Functions of different argument types are considered on an equal footing regardless of search path position.
   2. If a function is declared with a VARIADIC array parameter, and the call does not use the VARIADIC keyword, then the function is treated as if the array parameter were replaced by one or more occurrences of its element type, as needed to match the call. After such expansion the function might have effective argument types identical to some non-variadic function. In that case the function appearing earlier in the search path is used, or if the two functions are in the same schema, the non-variadic one is preferred.

This creates a security hazard when calling, via qualified name [**[9]**](https://www.postgresql.org/docs/10/typeconv-func.html#ftn.FUNC-QUALIFIED-SECURITY), a variadic function found in a schema that permits untrusted users to create objects. A malicious user can take control and execute arbitrary SQL functions as though you executed them. Substitute a call bearing the VARIADIC keyword, which bypasses this hazard. Calls populating VARIADIC "any"parameters often have no equivalent formulation containing the VARIADIC keyword. To issue those calls safely, the function's schema must permit only trusted users to create objects.

* 1. Functions that have default values for parameters are considered to match any call that omits zero or more of the defaultable parameter positions. If more than one such function matches a call, the one appearing earliest in the search path is used. If there are two or more such functions in the same schema with identical parameter types in the non-defaulted positions (which is possible if they have different sets of defaultable parameters), the system will not be able to determine which to prefer, and so an “ambiguous function call” error will result if no better match to the call can be found.

This creates an availability hazard when calling, via qualified name[**[9]**](https://www.postgresql.org/docs/10/typeconv-func.html#ftn.FUNC-QUALIFIED-SECURITY), any function found in a schema that permits untrusted users to create objects. A malicious user can create a function with the name of an existing function, replicating that function's parameters and appending novel parameters having default values. This precludes new calls to the original function. To forestall this hazard, place functions in schemas that permit only trusted users to create objects.

1. Check for a function accepting exactly the input argument types. If one exists (there can be only one exact match in the set of functions considered), use it. Lack of an exact match creates a security hazard when calling, via qualified name[**[9]**](https://www.postgresql.org/docs/10/typeconv-func.html#ftn.FUNC-QUALIFIED-SECURITY), a function found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match. (Cases involving unknown will never find a match at this step.)
2. If no exact match is found, see if the function call appears to be a special type conversion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal, or a type that is binary-coercible to the named data type, or a type that could be converted to the named data type by applying that type's I/O functions (that is, the conversion is either to or from one of the standard string types). When these conditions are met, the function call is treated as a form of CAST specification. [**[10]**](https://www.postgresql.org/docs/10/typeconv-func.html#ftn.id-1.5.9.8.4.4.1.2)
3. Look for the best match.
   1. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. unknown literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
   2. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-function resolution.
   3. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.
   4. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.
   5. If any input arguments are unknown, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.
   6. If there are both unknown and known-type arguments, and all the known-type arguments have the same type, assume that the unknown arguments are also of that type, and check which candidates can accept that type at the unknown-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Note that the “best match” rules are identical for operator and function type resolution. Some examples follow.

**Example 10.6. Rounding Function Argument Type Resolution**

There is only one round function that takes two arguments; it takes a first argument of type numeric and a second argument of type integer. So the following query automatically converts the first argument of type integer to numeric:

SELECT round(4, 4);

round

--------

4.0000

(1 row)

That query is actually transformed by the parser to:

SELECT round(CAST (4 AS numeric), 4);

Since numeric constants with decimal points are initially assigned the type numeric, the following query will require no type conversion and therefore might be slightly more efficient:

SELECT round(4.0, 4);

**Example 10.7. Variadic Function Resolution**

CREATE FUNCTION public.variadic\_example(VARIADIC numeric[]) RETURNS int

LANGUAGE sql AS 'SELECT 1';

CREATE FUNCTION

This function accepts, but does not require, the VARIADIC keyword. It tolerates both integer and numeric arguments:

SELECT public.variadic\_example(0),

public.variadic\_example(0.0),

public.variadic\_example(VARIADIC array[0.0]);

variadic\_example | variadic\_example | variadic\_example

------------------+------------------+------------------

1 | 1 | 1

(1 row)

However, the first and second calls will prefer more-specific functions, if available:

CREATE FUNCTION public.variadic\_example(numeric) RETURNS int

LANGUAGE sql AS 'SELECT 2';

CREATE FUNCTION

CREATE FUNCTION public.variadic\_example(int) RETURNS int

LANGUAGE sql AS 'SELECT 3';

CREATE FUNCTION

SELECT public.variadic\_example(0),

public.variadic\_example(0.0),

public.variadic\_example(VARIADIC array[0.0]);

variadic\_example | variadic\_example | variadic\_example

------------------+------------------+------------------

3 | 2 | 1

(1 row)

Given the default configuration and only the first function existing, the first and second calls are insecure. Any user could intercept them by creating the second or third function. By matching the argument type exactly and using the VARIADIC keyword, the third call is secure.

**Example 10.8. Substring Function Type Resolution**

There are several substr functions, one of which takes types text and integer. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category string (namely of type text).

SELECT substr('1234', 3);

substr

--------

34

(1 row)

If the string is declared to be of type varchar, as might be the case if it comes from a table, then the parser will try to convert it to become text:

SELECT substr(varchar '1234', 3);

substr

--------

34

(1 row)

This is transformed by the parser to effectively become:

SELECT substr(CAST (varchar '1234' AS text), 3);

Note

The parser learns from the pg\_cast catalog that text and varchar are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion call is really inserted in this case.

And, if the function is called with an argument of type integer, the parser will try to convert that to text:

SELECT substr(1234, 3);

ERROR: function substr(integer, integer) does not exist

HINT: No function matches the given name and argument types. You might need

to add explicit type casts.

This does not work because integer does not have an implicit cast to text. An explicit cast will work, however:

SELECT substr(CAST (1234 AS text), 3);

substr

--------

34

(1 row)

[**[9]**](https://www.postgresql.org/docs/10/typeconv-func.html#FUNC-QUALIFIED-SECURITY) The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a [**secure schema usage pattern**](https://www.postgresql.org/docs/10/ddl-schemas.html#DDL-SCHEMAS-PATTERNS).

[**[10]**](https://www.postgresql.org/docs/10/typeconv-func.html#id-1.5.9.8.4.4.1.2) The reason for this step is to support function-style cast specifications in cases where there is not an actual cast function. If there is a cast function, it is conventionally named after its output type, and so there is no need to have a special case. See [**CREATE CAST**](https://www.postgresql.org/docs/10/sql-createcast.html) for additional commentary.

## 10.4. Value Storage

Values to be inserted into a table are converted to the destination column's data type according to the following steps.

**Value Storage Type Conversion**

1. Check for an exact match with the target.
2. Otherwise, try to convert the expression to the target type. This is possible if an assignment cast between the two types is registered in the pg\_cast catalog (see [**CREATE CAST**](https://www.postgresql.org/docs/10/sql-createcast.html)). Alternatively, if the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.
3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the pg\_cast catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type integer, which receives the destination column's atttypmod value (typically its declared length, although the interpretation of atttypmod varies for different data types), and it may take a third boolean parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

**Example 10.9.**character**Storage Type Conversion**

For a target column declared as character(20) the following statement shows that the stored value is sized correctly:

CREATE TABLE vv (v character(20));

INSERT INTO vv SELECT 'abc' || 'def';

SELECT v, octet\_length(v) FROM vv;

v | octet\_length

----------------------+--------------

abcdef | 20

(1 row)

What has really happened here is that the two unknown literals are resolved to text by default, allowing the || operator to be resolved as text concatenation. Then the text result of the operator is converted to bpchar (“blank-padded char”, the internal name of the character data type) to match the target column type. (Since the conversion from text to bpchar is binary-coercible, this conversion does not insert any real function call.) Finally, the sizing function bpchar(bpchar, integer, boolean) is found in the system catalog and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

## 10.5. UNION, CASE, and Related Constructs

SQL UNION constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The INTERSECT and EXCEPT constructs resolve dissimilar types in the same way as UNION. The CASE, ARRAY, VALUES, GREATEST and LEAST constructs use the identical algorithm to match up their component expressions and select a result data type.

**Type Resolution for**UNION**,**CASE**, and Related Constructs**

1. If all inputs are of the same type, and it is not unknown, resolve as that type.
2. If any input is of a domain type, treat it as being of the domain's base type for all subsequent steps. [**[11]**](https://www.postgresql.org/docs/10/typeconv-union-case.html#ftn.id-1.5.9.10.9.3.1.1)
3. If all inputs are of type unknown, resolve as type text (the preferred type of the string category). Otherwise, unknown inputs are ignored for the purposes of the remaining rules.
4. If the non-unknown inputs are not all of the same type category, fail.
5. Choose the first non-unknown input type which is a preferred type in that category, if there is one.
6. Otherwise, choose the last non-unknown input type that allows all the preceding non-unknown inputs to be implicitly converted to it. (There always is such a type, since at least the first type in the list must satisfy this condition.)
7. Convert all inputs to the selected type. Fail if there is not a conversion from a given input to the selected type.

Some examples follow.

**Example 10.10. Type Resolution with Underspecified Types in a Union**

SELECT text 'a' AS "text" UNION SELECT 'b';

text

------

a

b

(2 rows)

Here, the unknown-type literal 'b' will be resolved to type text.

**Example 10.11. Type Resolution in a Simple Union**

SELECT 1.2 AS "numeric" UNION SELECT 1;

numeric

---------

1

1.2

(2 rows)

The literal 1.2 is of type numeric, and the integer value 1 can be cast implicitly to numeric, so that type is used.

**Example 10.12. Type Resolution in a Transposed Union**

SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

real

------

1

2.2

(2 rows)

Here, since type real cannot be implicitly cast to integer, but integer can be implicitly cast to real, the union result type is resolved as real.

**Example 10.13. Type Resolution in a Nested Union**

SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR: UNION types text and integer cannot be matched

This failure occurs because PostgreSQL treats multiple UNIONs as a nest of pairwise operations; that is, this input is the same as

(SELECT NULL UNION SELECT NULL) UNION SELECT 1;

The inner UNION is resolved as emitting type text, according to the rules given above. Then the outer UNION has inputs of types text and integer, leading to the observed error. The problem can be fixed by ensuring that the leftmost UNION has at least one input of the desired result type.

INTERSECT and EXCEPT operations are likewise resolved pairwise. However, the other constructs described in this section consider all of their inputs in one resolution step.

[**[11]**](https://www.postgresql.org/docs/10/typeconv-union-case.html#id-1.5.9.10.9.3.1.1) Somewhat like the treatment of domain inputs for operators and functions, this behavior allows a domain type to be preserved through a UNION or similar construct, so long as the user is careful to ensure that all inputs are implicitly or explicitly of that exact type. Otherwise the domain's base type will be preferred.

## 10.6. SELECT Output Columns

The rules given in the preceding sections will result in assignment of non-unknown data types to all expressions in a SQL query, except for unspecified-type literals that appear as simple output columns of a SELECT command. For example, in

SELECT 'Hello World';

there is nothing to identify what type the string literal should be taken as. In this situation PostgreSQL will fall back to resolving the literal's type as text.

When the SELECT is one arm of a UNION (or INTERSECT or EXCEPT) construct, or when it appears within INSERT ... SELECT, this rule is not applied since rules given in preceding sections take precedence. The type of an unspecified-type literal can be taken from the other UNION arm in the first case, or from the destination column in the second case.

RETURNING lists are treated the same as SELECT output lists for this purpose.

Note

Prior to PostgreSQL 10, this rule did not exist, and unspecified-type literals in a SELECToutput list were left as type unknown. That had assorted bad consequences, so it's been changed.

## Chapter 11. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

## 11.1. Introduction

Suppose we have a table similar to this:

CREATE TABLE test1 (

id integer,

content varchar

);

and the application issues many queries of the form:

SELECT content FROM test1 WHERE id = ***constant***;

With no advance preparation, the system would have to scan the entire test1 table, row by row, to find all matching entries. If there are many rows in test1 and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method. But if the system has been instructed to maintain an index on the id column, it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most non-fiction books: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest. Just as it is the task of the author to anticipate the items that readers are likely to look up, it is the task of the database programmer to foresee which indexes will be useful.

The following command can be used to create an index on the id column, as discussed:

CREATE INDEX test1\_id\_index ON test1 (id);

The name test1\_id\_index can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the DROP INDEX command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks doing so would be more efficient than a sequential table scan. But you might have to run the ANALYZE command regularly to update statistics to allow the query planner to make educated decisions. See [**Chapter 14**](https://www.postgresql.org/docs/10/performance-tips.html) for information about how to find out whether an index is used and when and why the planner might choose not to use an index.

Indexes can also benefit UPDATE and DELETE commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can also significantly speed up queries with joins.

Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (SELECT statements) to occur on the table in parallel with index creation, but writes (INSERT, UPDATE, DELETE) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see [**Building Indexes Concurrently**](https://www.postgresql.org/docs/10/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY).

After an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are seldom or never used in queries should be removed.

**11.2. Index Types**

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

|  |
| --- |
| < |
| <= |
| = |
| >= |
| > |

Constructs equivalent to combinations of these operators, such as BETWEEN and IN, can also be implemented with a B-tree index search. Also, an IS NULL or IS NOT NULL condition on an index column can be used with a B-tree index.

The optimizer can also use a B-tree index for queries involving the pattern matching operators LIKE and ~ *if* the pattern is a constant and is anchored to the beginning of the string — for example, col LIKE 'foo%' or col ~ '^foo', but not col LIKE '%bar'. However, if your database does not use the C locale you will need to create the index with a special operator class to support indexing of pattern-matching queries; see [**Section 11.9**](https://www.postgresql.org/docs/10/indexes-opclass.html) below. It is also possible to use B-tree indexes for ILIKE and ~\*, but only if the pattern starts with non-alphabetic characters, i.e., characters that are not affected by upper/lower case conversion.

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

CREATE INDEX ***name*** ON ***table*** USING HASH (***column***);

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the *operator class*). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

|  |
| --- |
| << |
| &< |
| &> |
| >> |
| <<| |
| &<| |
| |&> |
| |>> |
| @> |
| <@ |
| ~= |
| && |

(See [**Section 9.11**](https://www.postgresql.org/docs/10/functions-geometry.html) for the meaning of these operators.) The GiST operator classes included in the standard distribution are documented in [**Table 62.1**](https://www.postgresql.org/docs/10/gist-builtin-opclasses.html#GIST-BUILTIN-OPCLASSES-TABLE). Many other GiST operator classes are available in the contrib collection or as separate projects. For more information see [**Chapter 62**](https://www.postgresql.org/docs/10/gist.html).

GiST indexes are also capable of optimizing “nearest-neighbor” searches, such as

SELECT \* FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;

which finds the ten places closest to a given target point. The ability to do this is again dependent on the particular operator class being used. In [**Table 62.1**](https://www.postgresql.org/docs/10/gist-builtin-opclasses.html#GIST-BUILTIN-OPCLASSES-TABLE), operators that can be used in this way are listed in the column “Ordering Operators”.

SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

|  |
| --- |
| << |
| >> |
| ~= |
| <@ |
| <^ |
| >^ |

(See [**Section 9.11**](https://www.postgresql.org/docs/10/functions-geometry.html) for the meaning of these operators.) The SP-GiST operator classes included in the standard distribution are documented in [**Table 63.1**](https://www.postgresql.org/docs/10/spgist-builtin-opclasses.html#SPGIST-BUILTIN-OPCLASSES-TABLE). For more information see [**Chapter 63**](https://www.postgresql.org/docs/10/spgist.html).

GIN indexes are “inverted indexes” which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.

Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies, and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators:

|  |
| --- |
| <@ |
| @> |
| = |
| && |

(See [**Section 9.18**](https://www.postgresql.org/docs/10/functions-array.html) for the meaning of these operators.) The GIN operator classes included in the standard distribution are documented in [**Table 64.1**](https://www.postgresql.org/docs/10/gin-builtin-opclasses.html#GIN-BUILTIN-OPCLASSES-TABLE). Many other GIN operator classes are available in the contrib collection or as separate projects. For more information see [**Chapter 64**](https://www.postgresql.org/docs/10/gin.html).

BRIN indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range. This supports indexed queries using these operators:

|  |
| --- |
| < |
| <= |
| = |
| >= |
| > |

The BRIN operator classes included in the standard distribution are documented in [**Table 65.1**](https://www.postgresql.org/docs/10/brin-builtin-opclasses.html#BRIN-BUILTIN-OPCLASSES-TABLE). For more information see [**Chapter 65**](https://www.postgresql.org/docs/10/brin.html).

## 11.3. Multicolumn Indexes

An index can be defined on more than one column of a table. For example, if you have a table of this form:

CREATE TABLE test2 (

major int,

minor int,

name varchar

);

(say, you keep your /dev directory in a database...) and you frequently issue queries like:

SELECT name FROM test2 WHERE major = ***constant*** AND minor = ***constant***;

then it might be appropriate to define an index on the columns major and minor together, e.g.:

CREATE INDEX test2\_mm\_idx ON test2 (major, minor);

Currently, only the B-tree, GiST, GIN, and BRIN index types support multicolumn indexes. Up to 32 columns can be specified. (This limit can be altered when building PostgreSQL; see the file pg\_config\_manual.h.)

A multicolumn B-tree index can be used with query conditions that involve any subset of the index's columns, but the index is most efficient when there are constraints on the leading (leftmost) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will be used to limit the portion of the index that is scanned. Constraints on columns to the right of these columns are checked in the index, so they save visits to the table proper, but they do not reduce the portion of the index that has to be scanned. For example, given an index on (a, b, c) and a query condition WHERE a = 5 AND b >= 42 AND c < 77, the index would have to be scanned from the first entry with a = 5 and b = 42 up through the last entry with a = 5. Index entries with c >= 77 would be skipped, but they'd still have to be scanned through. This index could in principle be used for queries that have constraints on b and/or c with no constraint on a — but the entire index would have to be scanned, so in most cases the planner would prefer a sequential table scan over using the index.

A multicolumn GiST index can be used with query conditions that involve any subset of the index's columns. Conditions on additional columns restrict the entries returned by the index, but the condition on the first column is the most important one for determining how much of the index needs to be scanned. A GiST index will be relatively ineffective if its first column has only a few distinct values, even if there are many distinct values in additional columns.

A multicolumn GIN index can be used with query conditions that involve any subset of the index's columns. Unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use.

A multicolumn BRIN index can be used with query conditions that involve any subset of the index's columns. Like GIN and unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use. The only reason to have multiple BRIN indexes instead of one multicolumn BRIN index on a single table is to have a different pages\_per\_range storage parameter.

Of course, each column must be used with operators appropriate to the index type; clauses that involve other operators will not be considered.

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized. See also [**Section 11.5**](https://www.postgresql.org/docs/10/indexes-bitmap-scans.html) and [**Section 11.11**](https://www.postgresql.org/docs/10/indexes-index-only-scans.html) for some discussion of the merits of different index configurations.

## 11.4. Indexes and ORDER BY

In addition to simply finding the rows to be returned by a query, an index may be able to deliver them in a specific sorted order. This allows a query's ORDER BY specification to be honored without a separate sorting step. Of the index types currently supported by PostgreSQL, only B-tree can produce sorted output — the other index types return matching rows in an unspecified, implementation-dependent order.

The planner will consider satisfying an ORDER BY specification either by scanning an available index that matches the specification, or by scanning the table in physical order and doing an explicit sort. For a query that requires scanning a large fraction of the table, an explicit sort is likely to be faster than using an index because it requires less disk I/O due to following a sequential access pattern. Indexes are more useful when only a few rows need be fetched. An important special case is ORDER BY in combination with LIMIT ***n***: an explicit sort will have to process all the data to identify the first ***n***rows, but if there is an index matching the ORDER BY, the first ***n*** rows can be retrieved directly, without scanning the remainder at all.

By default, B-tree indexes store their entries in ascending order with nulls last. This means that a forward scan of an index on column x produces output satisfying ORDER BY x (or more verbosely, ORDER BY x ASC NULLS LAST). The index can also be scanned backward, producing output satisfying ORDER BY x DESC (or more verbosely, ORDER BY x DESC NULLS FIRST, since NULLS FIRST is the default for ORDER BY DESC).

You can adjust the ordering of a B-tree index by including the options ASC, DESC, NULLS FIRST, and/or NULLS LAST when creating the index; for example:

CREATE INDEX test2\_info\_nulls\_low ON test2 (info NULLS FIRST);

CREATE INDEX test3\_desc\_index ON test3 (id DESC NULLS LAST);

An index stored in ascending order with nulls first can satisfy either ORDER BY x ASC NULLS FIRST or ORDER BY x DESC NULLS LAST depending on which direction it is scanned in.

You might wonder why bother providing all four options, when two options together with the possibility of backward scan would cover all the variants of ORDER BY. In single-column indexes the options are indeed redundant, but in multicolumn indexes they can be useful. Consider a two-column index on (x, y): this can satisfy ORDER BY x, y if we scan forward, or ORDER BY x DESC, y DESC if we scan backward. But it might be that the application frequently needs to use ORDER BY x ASC, y DESC. There is no way to get that ordering from a plain index, but it is possible if the index is defined as (x ASC, y DESC) or (x DESC, y ASC).

Obviously, indexes with non-default sort orderings are a fairly specialized feature, but sometimes they can produce tremendous speedups for certain queries. Whether it's worth maintaining such an index depends on how often you use queries that require a special sort ordering.

## 11.5. Combining Multiple Indexes

A single index scan can only use query clauses that use the index's columns with operators of its operator class and are joined with AND. For example, given an index on (a, b) a query condition like WHERE a = 5 AND b = 6 could use the index, but a query like WHERE a = 5 OR b = 6 could not directly use the index.

Fortunately, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form AND and OR conditions across several index scans. For example, a query like WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 could be broken down into four separate scans of an index on x, each scan using one of the query clauses. The results of these scans are then ORed together to produce the result. Another example is that if we have separate indexes on x and y, one possible implementation of a query like WHERE x = 5 AND y = 6 is to use each index with the appropriate query clause and then AND together the index results to identify the result rows.

To combine multiple indexes, the system scans each needed index and prepares a bitmap in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then ANDed and ORed together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost, and so a separate sort step will be needed if the query has an ORDER BY clause. For this reason, and because each additional index scan adds extra time, the planner will sometimes choose to use a simple index scan even though additional indexes are available that could have been used as well.

In all but the simplest applications, there are various combinations of indexes that might be useful, and the database developer must make trade-offs to decide which indexes to provide. Sometimes multicolumn indexes are best, but sometimes it's better to create separate indexes and rely on the index-combination feature. For example, if your workload includes a mix of queries that sometimes involve only column x, sometimes only column y, and sometimes both columns, you might choose to create two separate indexes on x and y, relying on index combination to process the queries that use both columns. You could also create a multicolumn index on (x, y). This index would typically be more efficient than index combination for queries involving both columns, but as discussed in [**Section 11.3**](https://www.postgresql.org/docs/10/indexes-multicolumn.html), it would be almost useless for queries involving only y, so it should not be the only index. A combination of the multicolumn index and a separate index on y would serve reasonably well. For queries involving only x, the multicolumn index could be used, though it would be larger and hence slower than an index on x alone. The last alternative is to create all three indexes, but this is probably only reasonable if the table is searched much more often than it is updated and all three types of query are common. If one of the types of query is much less common than the others, you'd probably settle for creating just the two indexes that best match the common types.

## 11.6. Unique Indexes

Indexes can also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

CREATE UNIQUE INDEX ***name*** ON ***table*** (***column*** [, ...]);

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values are not allowed. Null values are not considered equal. A multicolumn unique index will only reject cases where all indexed columns are equal in multiple rows.

PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table. The index covers the columns that make up the primary key or unique constraint (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

Note

There's no need to manually create indexes on unique columns; doing so would just duplicate the automatically-created index.

## 11.7. Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the lower function:

SELECT \* FROM test1 WHERE lower(col1) = 'value';

This query can use an index if one has been defined on the result of the lower(col1) function:

CREATE INDEX test1\_lower\_col1\_idx ON test1 (lower(col1));

If we were to declare this index UNIQUE, it would prevent creation of rows whose col1 values differ only in case, as well as rows whose col1 values are actually identical. Thus, indexes on expressions can be used to enforce constraints that are not definable as simple unique constraints.

As another example, if one often does queries like:

SELECT \* FROM people WHERE (first\_name || ' ' || last\_name) = 'John Smith';

then it might be worth creating an index like this:

CREATE INDEX people\_names ON people ((first\_name || ' ' || last\_name));

The syntax of the CREATE INDEX command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses can be omitted when the expression is just a function call, as in the first example.

Index expressions are relatively expensive to maintain, because the derived expression(s) must be computed for each row upon insertion and whenever it is updated. However, the index expressions are not recomputed during an indexed search, since they are already stored in the index. In both examples above, the system sees the query as just WHERE indexedcolumn = 'constant' and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

**11.8. Partial Indexes**

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries only for those table rows that satisfy the predicate. Partial indexes are a specialized feature, but there are several situations in which they are useful.

One major reason for using a partial index is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up those queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. [**Example 11.1**](https://www.postgresql.org/docs/10/indexes-partial.html#INDEXES-PARTIAL-EX1) shows a possible application of this idea.

**Example 11.1. Setting up a Partial Index to Exclude Common Values**

Suppose you are storing web server access logs in a database. Most accesses originate from the IP address range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

CREATE TABLE access\_log (

url varchar,

client\_ip inet,

...

);

To create a partial index that suits our example, use a command such as this:

CREATE INDEX access\_log\_client\_ip\_ix ON access\_log (client\_ip)

WHERE NOT (client\_ip > inet '192.168.100.0' AND

client\_ip < inet '192.168.100.255');

A typical query that can use this index would be:

SELECT \*

FROM access\_log

WHERE url = '/index.html' AND client\_ip = inet '212.78.10.32';

A query that cannot use this index is:

SELECT \*

FROM access\_log

WHERE client\_ip = inet '192.168.100.23';

Observe that this kind of partial index requires that the common values be predetermined, so such partial indexes are best used for data distributions that do not change. The indexes can be recreated occasionally to adjust for new data distributions, but this adds maintenance effort.

Another possible use for a partial index is to exclude values from the index that the typical query workload is not interested in; this is shown in [**Example 11.2**](https://www.postgresql.org/docs/10/indexes-partial.html#INDEXES-PARTIAL-EX2). This results in the same advantages as listed above, but it prevents the “uninteresting” values from being accessed via that index, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

**Example 11.2. Setting up a Partial Index to Exclude Uninteresting Values**

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

CREATE INDEX orders\_unbilled\_index ON orders (order\_nr)

WHERE billed is not true;

A possible query to use this index would be:

SELECT \* FROM orders WHERE billed is not true AND order\_nr < 10000;

However, the index can also be used in queries that do not involve order\_nr at all, e.g.:

SELECT \* FROM orders WHERE billed is not true AND amount > 5000.00;

This is not as efficient as a partial index on the amount column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

SELECT \* FROM orders WHERE order\_nr = 3501;

The order 3501 might be among the billed or unbilled orders.

[**Example 11.2**](https://www.postgresql.org/docs/10/indexes-partial.html#INDEXES-PARTIAL-EX2) also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the WHERE condition of the query mathematically implies the predicate of the index. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent expressions that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example “x < 1” implies “x < 2”; otherwise the predicate condition must exactly match part of the query's WHERE condition or the index will not be recognized as usable. Matching takes place at query planning time, not at run time. As a result, parameterized query clauses do not work with a partial index. For example a prepared query with a parameter might specify “x < ?” which will never imply “x < 2” for all possible values of the parameter.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in [**Example 11.3**](https://www.postgresql.org/docs/10/indexes-partial.html#INDEXES-PARTIAL-EX3). This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

**Example 11.3. Setting up a Partial Unique Index**

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one “successful” entry for a given subject and target combination, but there might be any number of “unsuccessful” entries. Here is one way to do it:

CREATE TABLE tests (

subject text,

target text,

success boolean,

...

);

CREATE UNIQUE INDEX tests\_success\_constraint ON tests (subject, target)

WHERE success;

This is a particularly efficient approach when there are few successful tests and many unsuccessful ones.

Finally, a partial index can also be used to override the system's query plan choices. Also, data sets with peculiar distributions might cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will be minimal.

More information about partial indexes can be found in [**[ston89b]**](https://www.postgresql.org/docs/10/biblio.html#STON89B), [**[olson93]**](https://www.postgresql.org/docs/10/biblio.html#OLSON93), and [**[seshadri95]**](https://www.postgresql.org/docs/10/biblio.html#SESHADRI95).

## 11.9. Operator Classes and Operator Families

An index definition can specify an operator class for each column of an index.

CREATE INDEX ***name*** ON ***table*** (***column*** ***opclass*** [***sort options***] [, ...]);

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type int4 would use the int4\_ops class; this operator class includes comparison functions for values of type int4. In practice the default operator class for the column's data type is usually sufficient. The main reason for having operator classes is that for some data types, there could be more than one meaningful index behavior. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. The operator class determines the basic sort ordering (which can then be modified by adding sort options COLLATE, ASC/DESC and/or NULLS FIRST/NULLS LAST).

There are also some built-in operator classes besides the default ones:

* The operator classes text\_pattern\_ops, varchar\_pattern\_ops, and bpchar\_pattern\_ops support B-tree indexes on the types text, varchar, and char respectively. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern matching expressions (LIKE or POSIX regular expressions) when the database does not use the standard “C” locale. As an example, you might index a varchar column like this:

CREATE INDEX test\_index ON test\_table (col varchar\_pattern\_ops);

Note that you should also create an index with the default operator class if you want queries involving ordinary <, <=, >, or >= comparisons to use an index. Such queries cannot use the ***xxx***\_pattern\_ops operator classes. (Ordinary equality comparisons can use these operator classes, however.) It is possible to create multiple indexes on the same column with different operator classes. If you do use the C locale, you do not need the ***xxx***\_pattern\_ops operator classes, because an index with the default operator class is usable for pattern-matching queries in the C locale.

The following query shows all defined operator classes:

SELECT am.amname AS index\_method,

opc.opcname AS opclass\_name,

opc.opcintype::regtype AS indexed\_type,

opc.opcdefault AS is\_default

FROM pg\_am am, pg\_opclass opc

WHERE opc.opcmethod = am.oid

ORDER BY index\_method, opclass\_name;

An operator class is actually just a subset of a larger structure called an operator family. In cases where several data types have similar behaviors, it is frequently useful to define cross-data-type operators and allow these to work with indexes. To do this, the operator classes for each of the types must be grouped into the same operator family. The cross-type operators are members of the family, but are not associated with any single class within the family.

This expanded version of the previous query shows the operator family each operator class belongs to:

SELECT am.amname AS index\_method,

opc.opcname AS opclass\_name,

opf.opfname AS opfamily\_name,

opc.opcintype::regtype AS indexed\_type,

opc.opcdefault AS is\_default

FROM pg\_am am, pg\_opclass opc, pg\_opfamily opf

WHERE opc.opcmethod = am.oid AND

opc.opcfamily = opf.oid

ORDER BY index\_method, opclass\_name;

This query shows all defined operator families and all the operators included in each family:

SELECT am.amname AS index\_method,

opf.opfname AS opfamily\_name,

amop.amopopr::regoperator AS opfamily\_operator

FROM pg\_am am, pg\_opfamily opf, pg\_amop amop

WHERE opf.opfmethod = am.oid AND

amop.amopfamily = opf.oid

ORDER BY index\_method, opfamily\_name, opfamily\_operator;

## 11.10. Indexes and Collations

An index can support only one collation per index column. If multiple collations are of interest, multiple indexes may be needed.

Consider these statements:

CREATE TABLE test1c (

id integer,

content varchar COLLATE "x"

);

CREATE INDEX test1c\_content\_index ON test1c (content);

The index automatically uses the collation of the underlying column. So a query of the form

SELECT \* FROM test1c WHERE content > ***constant***;

could use the index, because the comparison will by default use the collation of the column. However, this index cannot accelerate queries that involve some other collation. So if queries of the form, say,

SELECT \* FROM test1c WHERE content > ***constant*** COLLATE "y";

are also of interest, an additional index could be created that supports the "y" collation, like this:

CREATE INDEX test1c\_content\_y\_index ON test1c (content COLLATE "y");

## 11.11. Index-Only Scans

All indexes in PostgreSQL are secondary indexes, meaning that each index is stored separately from the table's main data area (which is called the table's heap in PostgreSQL terminology). This means that in an ordinary index scan, each row retrieval requires fetching data from both the index and the heap. Furthermore, while the index entries that match a given indexable WHERE condition are usually close together in the index, the table rows they reference might be anywhere in the heap. The heap-access portion of an index scan thus involves a lot of random access into the heap, which can be slow, particularly on traditional rotating media. (As described in [**Section 11.5**](https://www.postgresql.org/docs/10/indexes-bitmap-scans.html), bitmap scans try to alleviate this cost by doing the heap accesses in sorted order, but that only goes so far.)

To solve this performance problem, PostgreSQL supports index-only scans, which can answer queries from an index alone without any heap access. The basic idea is to return values directly out of each index entry instead of consulting the associated heap entry. There are two fundamental restrictions on when this method can be used:

1. The index type must support index-only scans. B-tree indexes always do. GiST and SP-GiST indexes support index-only scans for some operator classes but not others. Other index types have no support. The underlying requirement is that the index must physically store, or else be able to reconstruct, the original data value for each index entry. As a counterexample, GIN indexes cannot support index-only scans because each index entry typically holds only part of the original data value.
2. The query must reference only columns stored in the index. For example, given an index on columns x and y of a table that also has a column z, these queries could use index-only scans:
3. SELECT x, y FROM tab WHERE x = 'key';

SELECT x FROM tab WHERE x = 'key' AND y < 42;

but these queries could not:

SELECT x, z FROM tab WHERE x = 'key';

SELECT x FROM tab WHERE x = 'key' AND z < 42;

(Expression indexes and partial indexes complicate this rule, as discussed below.)

If these two fundamental requirements are met, then all the data values required by the query are available from the index, so an index-only scan is physically possible. But there is an additional requirement for any table scan in PostgreSQL: it must verify that each retrieved row be “visible” to the query's MVCC snapshot, as discussed in [**Chapter 13**](https://www.postgresql.org/docs/10/mvcc.html). Visibility information is not stored in index entries, only in heap entries; so at first glance it would seem that every row retrieval would require a heap access anyway. And this is indeed the case, if the table row has been modified recently. However, for seldom-changing data there is a way around this problem. PostgreSQL tracks, for each page in a table's heap, whether all rows stored in that page are old enough to be visible to all current and future transactions. This information is stored in a bit in the table's visibility map. An index-only scan, after finding a candidate index entry, checks the visibility map bit for the corresponding heap page. If it's set, the row is known visible and so the data can be returned with no further work. If it's not set, the heap entry must be visited to find out whether it's visible, so no performance advantage is gained over a standard index scan. Even in the successful case, this approach trades visibility map accesses for heap accesses; but since the visibility map is four orders of magnitude smaller than the heap it describes, far less physical I/O is needed to access it. In most situations the visibility map remains cached in memory all the time.

In short, while an index-only scan is possible given the two fundamental requirements, it will be a win only if a significant fraction of the table's heap pages have their all-visible map bits set. But tables in which a large fraction of the rows are unchanging are common enough to make this type of scan very useful in practice.

To make effective use of the index-only scan feature, you might choose to create indexes in which only the leading columns are meant to match WHERE clauses, while the trailing columns hold “payload” data to be returned by a query. For example, if you commonly run queries like

SELECT y FROM tab WHERE x = 'key';

the traditional approach to speeding up such queries would be to create an index on x only. However, an index on (x, y) would offer the possibility of implementing this query as an index-only scan. As previously discussed, such an index would be larger and hence more expensive than an index on x alone, so this is attractive only if the table is known to be mostly static. Note it's important that the index be declared on (x, y) not (y, x), as for most index types (particularly B-trees) searches that do not constrain the leading index columns are not very efficient.

In principle, index-only scans can be used with expression indexes. For example, given an index on f(x) where x is a table column, it should be possible to execute

SELECT f(x) FROM tab WHERE f(x) < 1;

as an index-only scan; and this is very attractive if f() is an expensive-to-compute function. However, PostgreSQL's planner is currently not very smart about such cases. It considers a query to be potentially executable by index-only scan only when all columns needed by the query are available from the index. In this example, x is not needed except in the context f(x), but the planner does not notice that and concludes that an index-only scan is not possible. If an index-only scan seems sufficiently worthwhile, this can be worked around by declaring the index to be on (f(x), x), where the second column is not expected to be used in practice but is just there to convince the planner that an index-only scan is possible. An additional caveat, if the goal is to avoid recalculating f(x), is that the planner won't necessarily match uses of f(x) that aren't in indexable WHERE clauses to the index column. It will usually get this right in simple queries such as shown above, but not in queries that involve joins. These deficiencies may be remedied in future versions of PostgreSQL.

Partial indexes also have interesting interactions with index-only scans. Consider the partial index shown in [**Example 11.3**](https://www.postgresql.org/docs/10/indexes-partial.html#INDEXES-PARTIAL-EX3):

CREATE UNIQUE INDEX tests\_success\_constraint ON tests (subject, target)

WHERE success;

In principle, we could do an index-only scan on this index to satisfy a query like

SELECT target FROM tests WHERE subject = 'some-subject' AND success;

But there's a problem: the WHERE clause refers to success which is not available as a result column of the index. Nonetheless, an index-only scan is possible because the plan does not need to recheck that part of the WHERE clause at run time: all entries found in the index necessarily have success = true so this need not be explicitly checked in the plan. PostgreSQL versions 9.6 and later will recognize such cases and allow index-only scans to be generated, but older versions will not.

## 11.12. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance or tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the [**EXPLAIN**](https://www.postgresql.org/docs/10/sql-explain.html) command; its application for this purpose is illustrated in [**Section 14.1**](https://www.postgresql.org/docs/10/using-explain.html). It is also possible to gather overall statistics about index usage in a running server, as described in [**Section 28.2**](https://www.postgresql.org/docs/10/monitoring-stats.html).

It is difficult to formulate a general procedure for determining which indexes to create. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation is often necessary. The rest of this section gives some tips for that:

* Always run [**ANALYZE**](https://www.postgresql.org/docs/10/sql-analyze.html) first. This command collects statistics about the distribution of the values in the table. This information is required to estimate the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run ANALYZE is therefore a lost cause. See [**Section 24.1.3**](https://www.postgresql.org/docs/10/routine-vacuuming.html#VACUUM-FOR-STATISTICS) and [**Section 24.1.6**](https://www.postgresql.org/docs/10/routine-vacuuming.html#AUTOVACUUM) for more information.
* Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

It is especially fatal to use very small test data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

Also be careful when making up test data, which is often unavoidable when the application is not yet in production. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

* When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (see [**Section 19.7.1**](https://www.postgresql.org/docs/10/runtime-config-query.html#RUNTIME-CONFIG-QUERY-ENABLE)). For instance, turning off sequential scans (enable\_seqscan) and nested-loop joins (enable\_nestloop), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental reason why the index is not being used; for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)
* If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The EXPLAIN ANALYZE command can be useful here.
* If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs estimated for the plan nodes can be adjusted via run-time parameters (described in [**Section 19.7.2**](https://www.postgresql.org/docs/10/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS)). An inaccurate selectivity estimate is due to insufficient statistics. It might be possible to improve this by tuning the statistics-gathering parameters (see [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html)).

If you do not succeed in adjusting the costs to be more appropriate, then you might have to resort to forcing index usage explicitly. You might also want to contact the PostgreSQLdevelopers to examine the issue.

## Chapter 12. Full Text Search

## 12.1. Introduction

Full Text Searching (or just text search) provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query. The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query. Notions of query and similarity are very flexible and depend on the specific application. The simplest search considers query as a set of words and similarity as the frequency of query words in the document.

Textual search operators have existed in databases for years. PostgreSQL has ~, ~\*, LIKE, and ILIKE operators for textual data types, but they lack many essential properties required by modern information systems:

* There is no linguistic support, even for English. Regular expressions are not sufficient because they cannot easily handle derived words, e.g., satisfies and satisfy. You might miss documents that contain satisfies, although you probably would like to find them when searching for satisfy. It is possible to use OR to search for multiple derived forms, but this is tedious and error-prone (some words can have several thousand derivatives).
* They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
* They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be preprocessed and an index saved for later rapid searching. Preprocessing includes:

* Parsing documents into tokens. It is useful to identify various classes of tokens, e.g., numbers, words, complex words, email addresses, so that they can be processed differently. In principle token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes. PostgreSQL uses a parser to perform this step. A standard parser is provided, and custom parsers can be created for specific needs.
* Converting tokens into lexemes. A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often involves removal of suffixes (such as s or es in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates stop words, which are words that are so common that they are useless for searching. (In short, then, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) PostgreSQL uses dictionaries to perform this step. Various standard dictionaries are provided, and custom ones can be created for specific needs.
* Storing preprocessed documents optimized for searching. For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for proximity ranking, so that a document that contains a more “dense” region of query words is assigned a higher rank than one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can:

* Define stop words that should not be indexed.
* Map synonyms to a single word using Ispell.
* Map phrases to a single word using a thesaurus.
* Map different variations of a word to a canonical form using an Ispell dictionary.
* Map different variations of a word to a canonical form using Snowball stemmer rules.

A data type tsvector is provided for storing preprocessed documents, along with a type tsquery for representing processed queries ([**Section 8.11**](https://www.postgresql.org/docs/10/datatype-textsearch.html)). There are many functions and operators available for these data types ([**Section 9.13**](https://www.postgresql.org/docs/10/functions-textsearch.html)), the most important of which is the match operator @@, which we introduce in [**Section 12.1.2**](https://www.postgresql.org/docs/10/textsearch-intro.html#TEXTSEARCH-MATCHING). Full text searches can be accelerated using indexes ([**Section 12.9**](https://www.postgresql.org/docs/10/textsearch-indexes.html)).

### 12.1.1. What Is a Document?

A document is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (key words) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within PostgreSQL, a document is normally a textual field within a row of a database table, or possibly a combination (concatenation) of such fields, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

SELECT title || ' ' || author || ' ' || abstract || ' ' || body AS document

FROM messages

WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document

FROM messages m, docs d

WHERE mid = did AND mid = 12;

Note

Actually, in these example queries, coalesce should be used to prevent a single NULLattribute from causing a NULL result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires superuser permissions or special function support, so this is usually less convenient than keeping all the data inside PostgreSQL. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed tsvector format. Searching and ranking are performed entirely on the tsvector representation of a document — the original text need only be retrieved when the document has been selected for display to a user. We therefore often speak of the tsvector as being the document, but of course it is only a compact representation of the full document.

### 12.1.2. Basic Text Matching

Full text searching in PostgreSQL is based on the match operator @@, which returns true if a tsvector (document) matches a tsquery (query). It doesn't matter which data type is written first:

SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;

?column?

----------

t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;

?column?

----------

f

As the above example suggests, a tsquery is not just raw text, any more than a tsvector is. A tsquery contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, NOT, and FOLLOWED BY operators. (For syntax details see [**Section 8.11.2**](https://www.postgresql.org/docs/10/datatype-textsearch.html#DATATYPE-TSQUERY).) There are functions to\_tsquery, plainto\_tsquery, and phraseto\_tsquery that are helpful in converting user-written text into a proper tsquery, primarily by normalizing words appearing in the text. Similarly, to\_tsvector is used to parse and normalize a document string. So in practice a text search match would look more like this:

SELECT to\_tsvector('fat cats ate fat rats') @@ to\_tsquery('fat & rat');

?column?

----------

t

Observe that this match would not succeed if written as

SELECT 'fat cats ate fat rats'::tsvector @@ to\_tsquery('fat & rat');

?column?

----------

f

since here no normalization of the word rats will occur. The elements of a tsvector are lexemes, which are assumed already normalized, so rats does not match rat.

The @@ operator also supports text input, allowing explicit conversion of a text string to tsvector or tsquery to be skipped in simple cases. The variants available are:

tsvector @@ tsquery

tsquery @@ tsvector

text @@ tsquery

text @@ text

The first two of these we saw already. The form text @@ tsquery is equivalent to to\_tsvector(x) @@ y. The form text @@ text is equivalent to to\_tsvector(x) @@ plainto\_tsquery(y).

Within a tsquery, the & (AND) operator specifies that both its arguments must appear in the document to have a match. Similarly, the | (OR) operator specifies that at least one of its arguments must appear, while the ! (NOT) operator specifies that its argument must not appear in order to have a match. For example, the query fat & ! rat matches documents that contain fat but not rat.

Searching for phrases is possible with the help of the <-> (FOLLOWED BY) tsquery operator, which matches only if its arguments have matches that are adjacent and in the given order. For example:

SELECT to\_tsvector('fatal error') @@ to\_tsquery('fatal <-> error');

?column?

----------

t

SELECT to\_tsvector('error is not fatal') @@ to\_tsquery('fatal <-> error');

?column?

----------

f

There is a more general version of the FOLLOWED BY operator having the form <***N***>, where ***N*** is an integer standing for the difference between the positions of the matching lexemes. <1> is the same as <->, while <2> allows exactly one other lexeme to appear between the matches, and so on. The phraseto\_tsquery function makes use of this operator to construct a tsquery that can match a multi-word phrase when some of the words are stop words. For example:

SELECT phraseto\_tsquery('cats ate rats');

phraseto\_tsquery

-------------------------------

'cat' <-> 'ate' <-> 'rat'

SELECT phraseto\_tsquery('the cats ate the rats');

phraseto\_tsquery

-------------------------------

'cat' <-> 'ate' <2> 'rat'

A special case that's sometimes useful is that <0> can be used to require that two patterns match the same word.

Parentheses can be used to control nesting of the tsquery operators. Without parentheses, | binds least tightly, then &, then <->, and ! most tightly.

It's worth noticing that the AND/OR/NOT operators mean something subtly different when they are within the arguments of a FOLLOWED BY operator than when they are not, because within FOLLOWED BY the exact position of the match is significant. For example, normally !x matches only documents that do not contain x anywhere. But !x <-> y matches y if it is not immediately after an x; an occurrence of x elsewhere in the document does not prevent a match. Another example is that x & y normally only requires that x and y both appear somewhere in the document, but (x & y) <-> z requires x and y to match at the same place, immediately before a z. Thus this query behaves differently from x <-> z & y <-> z, which will match a document containing two separate sequences x z and y z. (This specific query is useless as written, since x and y could not match at the same place; but with more complex situations such as prefix-match patterns, a query of this form could be useful.)

### 12.1.3. Configurations

The above are all simple text search examples. As mentioned before, full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, e.g., parse based on more than just white space. This functionality is controlled by text search configurations. PostgreSQL comes with predefined configurations for many languages, and you can easily create your own configurations. (psql's \dF command shows all available configurations.)

During installation an appropriate configuration is selected and [**default\_text\_search\_config**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DEFAULT-TEXT-SEARCH-CONFIG) is set accordingly in postgresql.conf. If you are using the same text search configuration for the entire cluster you can use the value in postgresql.conf. To use different configurations throughout the cluster but the same configuration within any one database, use ALTER DATABASE ... SET. Otherwise, you can set default\_text\_search\_config in each session.

Each text search function that depends on a configuration has an optional regconfig argument, so that the configuration to use can be specified explicitly. default\_text\_search\_config is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. PostgreSQL's text search facility provides four types of configuration-related database objects:

* Text search parsers break documents into tokens and classify each token (for example, as words or numbers).
* Text search dictionaries convert tokens to normalized form and reject stop words.
* Text search templates provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
* Text search configurations select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Text search parsers and templates are built from low-level C functions; therefore it requires C programming ability to develop new ones, and superuser privileges to install one into a database. (There are examples of add-on parsers and templates in the contrib/ area of the PostgreSQL distribution.) Since dictionaries and configurations just parameterize and connect together some underlying parsers and templates, no special privilege is needed to create a new dictionary or configuration. Examples of creating custom dictionaries and configurations appear later in this chapter.

## 12.2. Tables and Indexes

The examples in the previous section illustrated full text matching using simple constant strings. This section shows how to search table data, optionally using indexes.

### 12.2.1. Searching a Table

It is possible to do a full text search without an index. A simple query to print the title of each row that contains the word friend in its body field is:

SELECT title

FROM pgweb

WHERE to\_tsvector('english', body) @@ to\_tsquery('english', 'friend');

This will also find related words such as friends and friendly, since all these are reduced to the same normalized lexeme.

The query above specifies that the english configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

SELECT title

FROM pgweb

WHERE to\_tsvector(body) @@ to\_tsquery('friend');

This query will use the configuration set by [**default\_text\_search\_config**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DEFAULT-TEXT-SEARCH-CONFIG).

A more complex example is to select the ten most recent documents that contain create and table in the title or body:

SELECT title

FROM pgweb

WHERE to\_tsvector(title || ' ' || body) @@ to\_tsquery('create & table')

ORDER BY last\_mod\_date DESC

LIMIT 10;

For clarity we omitted the coalesce function calls which would be needed to find rows that contain NULL in one of the two fields.

Although these queries will work without an index, most applications will find this approach too slow, except perhaps for occasional ad-hoc searches. Practical use of text searching usually requires creating an index.

### 12.2.2. Creating Indexes

We can create a GIN index ([**Section 12.9**](https://www.postgresql.org/docs/10/textsearch-indexes.html)) to speed up text searches:

CREATE INDEX pgweb\_idx ON pgweb USING GIN (to\_tsvector('english', body));

Notice that the 2-argument version of to\_tsvector is used. Only text search functions that specify a configuration name can be used in expression indexes ([**Section 11.7**](https://www.postgresql.org/docs/10/indexes-expressional.html)). This is because the index contents must be unaffected by [**default\_text\_search\_config**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DEFAULT-TEXT-SEARCH-CONFIG). If they were affected, the index contents might be inconsistent because different entries could contain tsvectors that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of to\_tsvector was used in the index above, only a query reference that uses the 2-argument version of to\_tsvector with the same configuration name will use that index. That is, WHERE to\_tsvector('english', body) @@ 'a & b' can use the index, but WHERE to\_tsvector(body) @@ 'a & b' cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column, e.g.:

CREATE INDEX pgweb\_idx ON pgweb USING GIN (to\_tsvector(config\_name, body));

where config\_name is a column in the pgweb table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, e.g., WHERE to\_tsvector(config\_name, body) @@ 'a & b'.

Indexes can even concatenate columns:

CREATE INDEX pgweb\_idx ON pgweb USING GIN (to\_tsvector('english', title || ' ' || body));

Another approach is to create a separate tsvector column to hold the output of to\_tsvector. This example is a concatenation of title and body, using coalesce to ensure that one field will still be indexed when the other is NULL:

ALTER TABLE pgweb ADD COLUMN textsearchable\_index\_col tsvector;

UPDATE pgweb SET textsearchable\_index\_col =

to\_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));

Then we create a GIN index to speed up the search:

CREATE INDEX textsearch\_idx ON pgweb USING GIN (textsearchable\_index\_col);

Now we are ready to perform a fast full text search:

SELECT title

FROM pgweb

WHERE textsearchable\_index\_col @@ to\_tsquery('create & table')

ORDER BY last\_mod\_date DESC

LIMIT 10;

When using a separate column to store the tsvector representation, it is necessary to create a trigger to keep the tsvector column current anytime title or body changes. [**Section 12.4.3**](https://www.postgresql.org/docs/10/textsearch-features.html#TEXTSEARCH-UPDATE-TRIGGERS) explains how to do that.

One advantage of the separate-column approach over an expression index is that it is not necessary to explicitly specify the text search configuration in queries in order to make use of the index. As shown in the example above, the query can depend on default\_text\_search\_config. Another advantage is that searches will be faster, since it will not be necessary to redo the to\_tsvector calls to verify index matches. (This is more important when using a GiST index than a GIN index; see [**Section 12.9**](https://www.postgresql.org/docs/10/textsearch-indexes.html).) The expression-index approach is simpler to set up, however, and it requires less disk space since the tsvector representation is not stored explicitly.

## 12.3. Controlling Text Search

To implement full text searching there must be a function to create a tsvector from a document and a tsquery from a user query. Also, we need to return results in a useful order, so we need a function that compares documents with respect to their relevance to the query. It's also important to be able to display the results nicely. PostgreSQL provides support for all of these functions.

### 12.3.1. Parsing Documents

PostgreSQL provides the function to\_tsvector for converting a document to the tsvector data type.

to\_tsvector([ ***config*** regconfig, ] ***document*** text) returns tsvector

to\_tsvector parses a textual document into tokens, reduces the tokens to lexemes, and returns a tsvector which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

SELECT to\_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');

to\_tsvector

-----------------------------------------------------

'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4

In the example above we see that the resulting tsvector does not contain the words a, on, or it, the word rats became rat, and the punctuation sign - was ignored.

The to\_tsvector function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries ([**Section 12.6**](https://www.postgresql.org/docs/10/textsearch-dictionaries.html)) is consulted, where the list can vary depending on the token type. The first dictionary that recognizes the token emits one or more normalized lexemes to represent the token. For example, rats became rat because one of the dictionaries recognized that the word rats is a plural form of rat. Some words are recognized as stop words ([**Section 12.6.1**](https://www.postgresql.org/docs/10/textsearch-dictionaries.html#TEXTSEARCH-STOPWORDS)), which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are a, on, and it. If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign - because there are in fact no dictionaries assigned for its token type (Space symbols), meaning space tokens will never be indexed. The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration ([**Section 12.7**](https://www.postgresql.org/docs/10/textsearch-configuration.html)). It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration english for the English language.

The function setweight can be used to label the entries of a tsvector with a given weight, where a weight is one of the letters A, B, C, or D. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because to\_tsvector(NULL) will return NULL, it is recommended to use coalesce whenever a field might be null. Here is the recommended method for creating a tsvector from a structured document:

UPDATE tt SET ti =

setweight(to\_tsvector(coalesce(title,'')), 'A') ||

setweight(to\_tsvector(coalesce(keyword,'')), 'B') ||

setweight(to\_tsvector(coalesce(abstract,'')), 'C') ||

setweight(to\_tsvector(coalesce(body,'')), 'D');

Here we have used setweight to label the source of each lexeme in the finished tsvector, and then merged the labeled tsvector values using the tsvector concatenation operator ||. ([**Section 12.4.1**](https://www.postgresql.org/docs/10/textsearch-features.html#TEXTSEARCH-MANIPULATE-TSVECTOR)gives details about these operations.)

### 12.3.2. Parsing Queries

PostgreSQL provides the functions to\_tsquery, plainto\_tsquery, and phraseto\_tsquery for converting a query to the tsquery data type. to\_tsquery offers access to more features than either plainto\_tsquery or phraseto\_tsquery, but it is less forgiving about its input.

to\_tsquery([ ***config*** regconfig, ] ***querytext*** text) returns tsquery

to\_tsquery creates a tsquery value from ***querytext***, which must consist of single tokens separated by the tsquery operators & (AND), | (OR), ! (NOT), and <-> (FOLLOWED BY), possibly grouped using parentheses. In other words, the input to to\_tsquery must already follow the general rules for tsquery input, as described in [**Section 8.11.2**](https://www.postgresql.org/docs/10/datatype-textsearch.html#DATATYPE-TSQUERY). The difference is that while basic tsquery input takes the tokens at face value, to\_tsquery normalizes each token into a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

SELECT to\_tsquery('english', 'The & Fat & Rats');

to\_tsquery

---------------

'fat' & 'rat'

As in basic tsquery input, weight(s) can be attached to each lexeme to restrict it to match only tsvector lexemes of those weight(s). For example:

SELECT to\_tsquery('english', 'Fat | Rats:AB');

to\_tsquery

------------------

'fat' | 'rat':AB

Also, \* can be attached to a lexeme to specify prefix matching:

SELECT to\_tsquery('supern:\*A & star:A\*B');

to\_tsquery

--------------------------

'supern':\*A & 'star':\*AB

Such a lexeme will match any word in a tsvector that begins with the given string.

to\_tsquery can also accept single-quoted phrases. This is primarily useful when the configuration includes a thesaurus dictionary that may trigger on such phrases. In the example below, a thesaurus contains the rule supernovae stars : sn:

SELECT to\_tsquery('''supernovae stars'' & !crab');

to\_tsquery

---------------

'sn' & !'crab'

Without quotes, to\_tsquery will generate a syntax error for tokens that are not separated by an AND, OR, or FOLLOWED BY operator.

plainto\_tsquery([ ***config*** regconfig, ] ***querytext*** text) returns tsquery

plainto\_tsquery transforms the unformatted text ***querytext*** to a tsquery value. The text is parsed and normalized much as for to\_tsvector, then the & (AND) tsquery operator is inserted between surviving words.

Example:

SELECT plainto\_tsquery('english', 'The Fat Rats');

plainto\_tsquery

-----------------

'fat' & 'rat'

Note that plainto\_tsquery will not recognize tsquery operators, weight labels, or prefix-match labels in its input:

SELECT plainto\_tsquery('english', 'The Fat & Rats:C');

plainto\_tsquery

---------------------

'fat' & 'rat' & 'c'

Here, all the input punctuation was discarded as being space symbols.

phraseto\_tsquery([ ***config*** regconfig, ] ***querytext*** text) returns tsquery

phraseto\_tsquery behaves much like plainto\_tsquery, except that it inserts the <-> (FOLLOWED BY) operator between surviving words instead of the & (AND) operator. Also, stop words are not simply discarded, but are accounted for by inserting <***N***> operators rather than <-> operators. This function is useful when searching for exact lexeme sequences, since the FOLLOWED BY operators check lexeme order not just the presence of all the lexemes.

Example:

SELECT phraseto\_tsquery('english', 'The Fat Rats');

phraseto\_tsquery

------------------

'fat' <-> 'rat'

Like plainto\_tsquery, the phraseto\_tsquery function will not recognize tsquery operators, weight labels, or prefix-match labels in its input:

SELECT phraseto\_tsquery('english', 'The Fat & Rats:C');

phraseto\_tsquery

-----------------------------

'fat' <-> 'rat' <-> 'c'

### 12.3.3. Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and very application-specific. Different applications might require additional information for ranking, e.g., document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

ts\_rank([ ***weights*** float4[], ] ***vector*** tsvector, ***query*** tsquery [, ***normalization*** integer ]) returns float4

Ranks vectors based on the frequency of their matching lexemes.

ts\_rank\_cd([ ***weights*** float4[], ] ***vector*** tsvector, ***query*** tsquery [, ***normalization*** integer ]) returns float4

This function computes the cover density ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope's "Relevance Ranking for One to Three Term Queries" in the journal "Information Processing and Management", 1999. Cover density is similar to ts\_rank ranking except that the proximity of matching lexemes to each other is taken into consideration.

This function requires lexeme positional information to perform its calculation. Therefore, it ignores any “stripped” lexemes in the tsvector. If there are no unstripped lexemes in the input, the result will be zero. (See [**Section 12.4.1**](https://www.postgresql.org/docs/10/textsearch-features.html#TEXTSEARCH-MANIPULATE-TSVECTOR) for more information about the strip function and positional information in tsvectors.)

For both these functions, the optional ***weights*** argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

{D-weight, C-weight, B-weight, A-weight}

If no ***weights*** are provided, then these defaults are used:

{0.1, 0.2, 0.4, 1.0}

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size, e.g., a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer ***normalization*** option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using | (for example, 2|4).

* 0 (the default) ignores the document length
* 1 divides the rank by 1 + the logarithm of the document length
* 2 divides the rank by the document length
* 4 divides the rank by the mean harmonic distance between extents (this is implemented only by ts\_rank\_cd)
* 8 divides the rank by the number of unique words in document
* 16 divides the rank by 1 + the logarithm of the number of unique words in document
* 32 divides the rank by itself + 1

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 (rank/(rank+1)) can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

SELECT title, ts\_rank\_cd(textsearch, query) AS rank

FROM apod, to\_tsquery('neutrino|(dark & matter)') query

WHERE query @@ textsearch

ORDER BY rank DESC

LIMIT 10;

title | rank

-----------------------------------------------+----------

Neutrinos in the Sun | 3.1

The Sudbury Neutrino Detector | 2.4

A MACHO View of Galactic Dark Matter | 2.01317

Hot Gas and Dark Matter | 1.91171

The Virgo Cluster: Hot Plasma and Dark Matter | 1.90953

Rafting for Solar Neutrinos | 1.9

NGC 4650A: Strange Galaxy and Dark Matter | 1.85774

Hot Gas and Dark Matter | 1.6123

Ice Fishing for Cosmic Neutrinos | 1.6

Weak Lensing Distorts the Universe | 0.818218

This is the same example using normalized ranking:

SELECT title, ts\_rank\_cd(textsearch, query, 32 /\* rank/(rank+1) \*/ ) AS rank

FROM apod, to\_tsquery('neutrino|(dark & matter)') query

WHERE query @@ textsearch

ORDER BY rank DESC

LIMIT 10;

title | rank

-----------------------------------------------+-------------------

Neutrinos in the Sun | 0.756097569485493

The Sudbury Neutrino Detector | 0.705882361190954

A MACHO View of Galactic Dark Matter | 0.668123210574724

Hot Gas and Dark Matter | 0.65655958650282

The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973

Rafting for Solar Neutrinos | 0.655172410958162

NGC 4650A: Strange Galaxy and Dark Matter | 0.650072921219637

Hot Gas and Dark Matter | 0.617195790024749

Ice Fishing for Cosmic Neutrinos | 0.615384618911517

Weak Lensing Distorts the Universe | 0.450010798361481

Ranking can be expensive since it requires consulting the tsvector of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

### 12.3.4. Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. PostgreSQL provides a function ts\_headline that implements this functionality.

ts\_headline([ ***config*** regconfig, ] ***document*** text, ***query*** tsquery [, ***options*** text ]) returns text

ts\_headline accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by ***config***; if ***config*** is omitted, the default\_text\_search\_config configuration is used.

If an ***options*** string is specified it must consist of a comma-separated list of one or more ***option***=***value*** pairs. The available options are:

* StartSel, StopSel: the strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. You must double-quote these strings if they contain spaces or commas.
* MaxWords, MinWords: these numbers determine the longest and shortest headlines to output.
* ShortWord: words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates common English articles.
* HighlightAll: Boolean flag; if true the whole document will be used as the headline, ignoring the preceding three parameters.
* MaxFragments: maximum number of text excerpts or fragments to display. The default value of zero selects a non-fragment-oriented headline generation method. A value greater than zero selects fragment-based headline generation. This method finds text fragments with as many query words as possible and stretches those fragments around the query words. As a result query words are close to the middle of each fragment and have words on each side. Each fragment will be of at most MaxWords and words of length ShortWord or less are dropped at the start and end of each fragment. If not all query words are found in the document, then a single fragment of the first MinWords in the document will be displayed.
* FragmentDelimiter: When more than one fragment is displayed, the fragments will be separated by this string.

Any unspecified options receive these defaults:

StartSel=<b>, StopSel=</b>,

MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,

MaxFragments=0, FragmentDelimiter=" ... "

For example:

SELECT ts\_headline('english',

'The most common type of search

is to find all documents containing given query terms

and return them in order of their similarity to the

query.',

to\_tsquery('query & similarity'));

ts\_headline

------------------------------------------------------------

containing given <b>query</b> terms

and return them in order of their <b>similarity</b> to the

<b>query</b>.

SELECT ts\_headline('english',

'The most common type of search

is to find all documents containing given query terms

and return them in order of their similarity to the

query.',

to\_tsquery('query & similarity'),

'StartSel = <, StopSel = >');

ts\_headline

-------------------------------------------------------

containing given <query> terms

and return them in order of their <similarity> to the

<query>.

ts\_headline uses the original document, not a tsvector summary, so it can be slow and should be used with care.

## 12.4. Additional Features

This section describes additional functions and operators that are useful in connection with text search.

### 12.4.1. Manipulating Documents

[**Section 12.3.1**](https://www.postgresql.org/docs/10/textsearch-controls.html#TEXTSEARCH-PARSING-DOCUMENTS) showed how raw textual documents can be converted into tsvector values. PostgreSQL also provides functions and operators that can be used to manipulate documents that are already in tsvector form.

tsvector || tsvector

The tsvector concatenation operator returns a vector which combines the lexemes and positional information of the two vectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand vector are offset by the largest position mentioned in the left-hand vector, so that the result is nearly equivalent to the result of performing to\_tsvector on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the vector form, rather than concatenating text before applying to\_tsvector, is that you can use different configurations to parse different sections of the document. Also, because the setweight function marks all lexemes of the given vector the same way, it is necessary to parse the text and do setweight before concatenating if you want to label different parts of the document with different weights.

setweight(***vector*** tsvector, ***weight*** "char") returns tsvector

setweight returns a copy of the input vector in which every position has been labeled with the given ***weight***, either A, B, C, or D. (D is the default for new vectors and as such is not displayed on output.) These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

Note that weight labels apply to positions, not lexemes. If the input vector has been stripped of positions then setweight does nothing.

length(***vector*** tsvector) returns integer

Returns the number of lexemes stored in the vector.

strip(***vector*** tsvector) returns tsvector

Returns a vector that lists the same lexemes as the given vector, but lacks any position or weight information. The result is usually much smaller than an unstripped vector, but it is also less useful. Relevance ranking does not work as well on stripped vectors as unstripped ones. Also, the <-> (FOLLOWED BY) tsquery operator will never match stripped input, since it cannot determine the distance between lexeme occurrences.

A full list of tsvector-related functions is available in [**Table 9.41**](https://www.postgresql.org/docs/10/functions-textsearch.html#TEXTSEARCH-FUNCTIONS-TABLE).

### 12.4.2. Manipulating Queries

[**Section 12.3.2**](https://www.postgresql.org/docs/10/textsearch-controls.html#TEXTSEARCH-PARSING-QUERIES) showed how raw textual queries can be converted into tsquery values. PostgreSQL also provides functions and operators that can be used to manipulate queries that are already in tsquery form.

tsquery && tsquery

Returns the AND-combination of the two given queries.

tsquery || tsquery

Returns the OR-combination of the two given queries.

!! tsquery

Returns the negation (NOT) of the given query.

tsquery <-> tsquery

Returns a query that searches for a match to the first given query immediately followed by a match to the second given query, using the <-> (FOLLOWED BY) tsquery operator. For example:

SELECT to\_tsquery('fat') <-> to\_tsquery('cat | rat');

?column?

-----------------------------------

'fat' <-> 'cat' | 'fat' <-> 'rat'

tsquery\_phrase(***query1*** tsquery, ***query2*** tsquery [, ***distance*** integer ]) returns tsquery

Returns a query that searches for a match to the first given query followed by a match to the second given query at a distance of at ***distance*** lexemes, using the <***N***> tsquery operator. For example:

SELECT tsquery\_phrase(to\_tsquery('fat'), to\_tsquery('cat'), 10);

tsquery\_phrase

------------------

'fat' <10> 'cat'

numnode(***query*** tsquery) returns integer

Returns the number of nodes (lexemes plus operators) in a tsquery. This function is useful to determine if the ***query*** is meaningful (returns > 0), or contains only stop words (returns 0). Examples:

SELECT numnode(plainto\_tsquery('the any'));

NOTICE: query contains only stopword(s) or doesn't contain lexeme(s), ignored

numnode

---------

0

SELECT numnode('foo & bar'::tsquery);

numnode

---------

3

querytree(***query*** tsquery) returns text

Returns the portion of a tsquery that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

SELECT querytree(to\_tsquery('!defined'));

querytree

-----------

#### 12.4.2.1. Query Rewriting

The ts\_rewrite family of functions search a given tsquery for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a tsquery-specific version of substring replacement. A target and substitute combination can be thought of as a query rewrite rule. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (e.g., new york, big apple, nyc, gotham) or narrow the search to direct the user to some hot topic. There is some overlap in functionality between this feature and thesaurus dictionaries ([**Section 12.6.4**](https://www.postgresql.org/docs/10/textsearch-dictionaries.html#TEXTSEARCH-THESAURUS)). However, you can modify a set of rewrite rules on-the-fly without reindexing, whereas updating a thesaurus requires reindexing to be effective.

ts\_rewrite (***query*** tsquery, ***target*** tsquery, ***substitute*** tsquery) returns tsquery

This form of ts\_rewrite simply applies a single rewrite rule: ***target*** is replaced by ***substitute*** wherever it appears in ***query***. For example:

SELECT ts\_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);

ts\_rewrite

------------

'b' & 'c'

ts\_rewrite (***query*** tsquery, ***select*** text) returns tsquery

This form of ts\_rewrite accepts a starting ***query*** and a SQL ***select*** command, which is given as a text string. The ***select*** must yield two columns of tsquery type. For each row of the ***select***result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current ***query*** value. For example:

CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);

INSERT INTO aliases VALUES('a', 'c');

SELECT ts\_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');

ts\_rewrite

------------

'b' & 'c'

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to ORDER BY some ordering key.

Let's consider a real-life astronomical example. We'll expand query supernovae using table-driven rewriting rules:

CREATE TABLE aliases (t tsquery primary key, s tsquery);

INSERT INTO aliases VALUES(to\_tsquery('supernovae'), to\_tsquery('supernovae|sn'));

SELECT ts\_rewrite(to\_tsquery('supernovae & crab'), 'SELECT \* FROM aliases');

ts\_rewrite

---------------------------------

'crab' & ( 'supernova' | 'sn' )

We can change the rewriting rules just by updating the table:

UPDATE aliases

SET s = to\_tsquery('supernovae|sn & !nebulae')

WHERE t = to\_tsquery('supernovae');

SELECT ts\_rewrite(to\_tsquery('supernovae & crab'), 'SELECT \* FROM aliases');

ts\_rewrite

---------------------------------------------

'crab' & ( 'supernova' | 'sn' & !'nebula' )

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the tsquery type. In the example below, we select only those rules which might match the original query:

SELECT ts\_rewrite('a & b'::tsquery,

'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');

ts\_rewrite

------------

'b' & 'c'

### 12.4.3. Triggers for Automatic Updates

When using a separate column to store the tsvector representation of your documents, it is necessary to create a trigger to update the tsvector column when the document content columns change. Two built-in trigger functions are available for this, or you can write your own.

tsvector\_update\_trigger(***tsvector\_column\_name***, ***config\_name***, ***text\_column\_name*** [, ... ])

tsvector\_update\_trigger\_column(***tsvector\_column\_name***, ***config\_column\_name***, ***text\_column\_name*** [, ... ])

These trigger functions automatically compute a tsvector column from one or more textual columns, under the control of parameters specified in the CREATE TRIGGER command. An example of their use is:

CREATE TABLE messages (

title text,

body text,

tsv tsvector

);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE

ON messages FOR EACH ROW EXECUTE PROCEDURE

tsvector\_update\_trigger(tsv, 'pg\_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT \* FROM messages;

title | body | tsv

------------+-----------------------+----------------------------

title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to\_tsquery('title & body');

title | body

------------+-----------------------

title here | the body text is here

Having created this trigger, any change in title or body will automatically be reflected into tsv, without the application having to worry about it.

The first trigger argument must be the name of the tsvector column to be updated. The second argument specifies the text search configuration to be used to perform the conversion. For tsvector\_update\_trigger, the configuration name is simply given as the second trigger argument. It must be schema-qualified as shown above, so that the trigger behavior will not change with changes in search\_path. For tsvector\_update\_trigger\_column, the second trigger argument is the name of another table column, which must be of type regconfig. This allows a per-row selection of configuration to be made. The remaining argument(s) are the names of textual columns (of type text, varchar, or char). These will be included in the document in the order given. NULL values will be skipped (but the other columns will still be indexed).

A limitation of these built-in triggers is that they treat all the input columns alike. To process columns differently — for example, to weight title differently from body — it is necessary to write a custom trigger. Here is an example using PL/pgSQL as the trigger language:

CREATE FUNCTION messages\_trigger() RETURNS trigger AS $$

begin

new.tsv :=

setweight(to\_tsvector('pg\_catalog.english', coalesce(new.title,'')), 'A') ||

setweight(to\_tsvector('pg\_catalog.english', coalesce(new.body,'')), 'D');

return new;

end

$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE

ON messages FOR EACH ROW EXECUTE PROCEDURE messages\_trigger();

Keep in mind that it is important to specify the configuration name explicitly when creating tsvector values inside triggers, so that the column's contents will not be affected by changes to default\_text\_search\_config. Failure to do this is likely to lead to problems such as search results changing after a dump and reload.

### 12.4.4. Gathering Document Statistics

The function ts\_stat is useful for checking your configuration and for finding stop-word candidates.

ts\_stat(***sqlquery*** text, [ ***weights*** text, ]

OUT ***word*** text, OUT ***ndoc*** integer,

OUT ***nentry*** integer) returns setof record

***sqlquery*** is a text value containing an SQL query which must return a single tsvector column. ts\_stat executes the query and returns statistics about each distinct lexeme (word) contained in the tsvector data. The columns returned are

* ***word*** text — the value of a lexeme
* ***ndoc*** integer — number of documents (tsvectors) the word occurred in
* ***nentry*** integer — total number of occurrences of the word

If ***weights*** is supplied, only occurrences having one of those weights are counted.

For example, to find the ten most frequent words in a document collection:

SELECT \* FROM ts\_stat('SELECT vector FROM apod')

ORDER BY nentry DESC, ndoc DESC, word

LIMIT 10;

The same, but counting only word occurrences with weight A or B:

SELECT \* FROM ts\_stat('SELECT vector FROM apod', 'ab')

ORDER BY nentry DESC, ndoc DESC, word

LIMIT 10;

## 12.5. Parsers

Text search parsers are responsible for splitting raw document text into tokens and identifying each token's type, where the set of possible types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries. At present PostgreSQL provides just one built-in parser, which has been found to be useful for a wide range of applications.

The built-in parser is named pg\_catalog.default. It recognizes 23 token types, shown in [**Table 12.1**](https://www.postgresql.org/docs/10/textsearch-parsers.html#TEXTSEARCH-DEFAULT-PARSER).

**Table 12.1. Default Parser's Token Types**

| **Alias** | **Description** | **Example** |
| --- | --- | --- |
| asciiword | Word, all ASCII letters | elephant |
| word | Word, all letters | mañana |
| numword | Word, letters and digits | beta1 |
| asciihword | Hyphenated word, all ASCII | up-to-date |
| hword | Hyphenated word, all letters | lógico-matemática |
| numhword | Hyphenated word, letters and digits | postgresql-beta1 |
| hword\_asciipart | Hyphenated word part, all ASCII | postgresql in the context postgresql-beta1 |
| hword\_part | Hyphenated word part, all letters | lógico or matemática in the context lógico-matemática |
| hword\_numpart | Hyphenated word part, letters and digits | beta1 in the context postgresql-beta1 |
| email | Email address | foo@example.com |
| protocol | Protocol head | http:// |
| url | URL | example.com/stuff/index.html |
| host | Host | example.com |
| url\_path | URL path | /stuff/index.html, in the context of a URL |
| file | File or path name | /usr/local/foo.txt, if not within a URL |
| sfloat | Scientific notation | -1.234e56 |
| float | Decimal notation | -1.234 |
| int | Signed integer | -1234 |
| uint | Unsigned integer | 1234 |
| version | Version number | 8.3.0 |
| tag | XML tag | <a href="dictionaries.html"> |
| entity | XML entity | &amp; |
| blank | Space symbols | (any whitespace or punctuation not otherwise recognized) |

Note

The parser's notion of a “letter” is determined by the database's locale setting, specifically lc\_ctype. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types word and asciiword should be treated alike.

email does not support all valid email characters as defined by RFC 5322. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to produce overlapping tokens from the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

SELECT alias, description, token FROM ts\_debug('foo-bar-beta1');

alias | description | token

-----------------+------------------------------------------+---------------

numhword | Hyphenated word, letters and digits | foo-bar-beta1

hword\_asciipart | Hyphenated word part, all ASCII | foo

blank | Space symbols | -

hword\_asciipart | Hyphenated word part, all ASCII | bar

blank | Space symbols | -

hword\_numpart | Hyphenated word part, letters and digits | beta1

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

SELECT alias, description, token FROM ts\_debug('http://example.com/stuff/index.html');

alias | description | token

----------+---------------+------------------------------

protocol | Protocol head | http://

url | URL | example.com/stuff/index.html

host | Host | example.com

url\_path | URL path | /stuff/index.html

## 12.6. Dictionaries

Dictionaries are used to eliminate words that should not be considered in a search (stop words), and to normalize words so that different derived forms of the same word will match. A successfully normalized word is called a lexeme. Aside from improving search quality, normalization and removal of stop words reduce the size of the tsvector representation of a document, thereby improving performance. Normalization does not always have linguistic meaning and usually depends on application semantics.

Some examples of normalization:

* Linguistic - Ispell dictionaries try to reduce input words to a normalized form; stemmer dictionaries remove word endings
* URL locations can be canonicalized to make equivalent URLs match:
  + http://www.pgsql.ru/db/mw/index.html
  + http://www.pgsql.ru/db/mw/
  + http://www.pgsql.ru/db/../db/mw/index.html
* Color names can be replaced by their hexadecimal values, e.g., red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
* If indexing numbers, we can remove some fractional digits to reduce the range of possible numbers, so for example 3.14159265359, 3.1415926, 3.14 will be the same after normalization if only two digits are kept after the decimal point.

A dictionary is a program that accepts a token as input and returns:

* an array of lexemes if the input token is known to the dictionary (notice that one token can produce more than one lexeme)
* a single lexeme with the TSL\_FILTER flag set, to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a filtering dictionary)
* an empty array if the dictionary knows the token, but it is a stop word
* NULL if the dictionary does not recognize the input token

PostgreSQL provides predefined dictionaries for many languages. There are also several predefined templates that can be used to create new dictionaries with custom parameters. Each predefined dictionary template is described below. If no existing template is suitable, it is possible to create new ones; see the contrib/ area of the PostgreSQL distribution for examples.

A text search configuration binds a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until some dictionary recognizes it as a known word. If it is identified as a stop word, or if no dictionary recognizes the token, it will be discarded and not indexed or searched for. Normally, the first dictionary that returns a non-NULL output determines the result, and any remaining dictionaries are not consulted; but a filtering dictionary can replace the given word with a modified word, which is then passed to subsequent dictionaries.

The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a Snowball stemmer or simple, which recognizes everything. For example, for an astronomy-specific search (astro\_en configuration) one could bind token type asciiword (ASCII word) to a synonym dictionary of astronomical terms, a general English dictionary and a Snowball English stemmer:

ALTER TEXT SEARCH CONFIGURATION astro\_en

ADD MAPPING FOR asciiword WITH astrosyn, english\_ispell, english\_stem;

A filtering dictionary can be placed anywhere in the list, except at the end where it'd be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries. For example, a filtering dictionary could be used to remove accents from accented letters, as is done by the [**unaccent**](https://www.postgresql.org/docs/10/unaccent.html) module.

### 12.6.1. Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. For example, every English text contains words like a and the, so it is useless to store them in an index. However, stop words do affect the positions in tsvector, which in turn affect ranking:

SELECT to\_tsvector('english','in the list of stop words');

to\_tsvector

----------------------------

'list':3 'stop':5 'word':6

The missing positions 1,2,4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

SELECT ts\_rank\_cd (to\_tsvector('english','in the list of stop words'), to\_tsquery('list & stop'));

ts\_rank\_cd

------------

0.05

SELECT ts\_rank\_cd (to\_tsvector('english','list stop words'), to\_tsquery('list & stop'));

ts\_rank\_cd

------------

0.1

It is up to the specific dictionary how it treats stop words. For example, ispell dictionaries first normalize words and then look at the list of stop words, while Snowball stemmers first check the list of stop words. The reason for the different behavior is an attempt to decrease noise.

### 12.6.2. Simple Dictionary

The simple dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lower-cased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Here is an example of a dictionary definition using the simple template:

CREATE TEXT SEARCH DICTIONARY public.simple\_dict (

TEMPLATE = pg\_catalog.simple,

STOPWORDS = english

);

Here, english is the base name of a file of stop words. The file's full name will be $SHAREDIR/tsearch\_data/english.stop, where $SHAREDIR means the PostgreSQL installation's shared-data directory, often /usr/local/share/postgresql (use pg\_config --sharedir to determine it if you're not sure). The file format is simply a list of words, one per line. Blank lines and trailing spaces are ignored, and upper case is folded to lower case, but no other processing is done on the file contents.

Now we can test our dictionary:

SELECT ts\_lexize('public.simple\_dict','YeS');

ts\_lexize

-----------

{yes}

SELECT ts\_lexize('public.simple\_dict','The');

ts\_lexize

-----------

{}

We can also choose to return NULL, instead of the lower-cased word, if it is not found in the stop words file. This behavior is selected by setting the dictionary's Accept parameter to false. Continuing the example:

ALTER TEXT SEARCH DICTIONARY public.simple\_dict ( Accept = false );

SELECT ts\_lexize('public.simple\_dict','YeS');

ts\_lexize

-----------

SELECT ts\_lexize('public.simple\_dict','The');

ts\_lexize

-----------

{}

With the default setting of Accept = true, it is only useful to place a simple dictionary at the end of a list of dictionaries, since it will never pass on any token to a following dictionary. Conversely, Accept= false is only useful when there is at least one following dictionary.

Caution

Most types of dictionaries rely on configuration files, such as files of stop words. These files must be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.

Caution

Normally, a database session will read a dictionary configuration file only once, when it is first used within the session. If you modify a configuration file and want to force existing sessions to pick up the new contents, issue an ALTER TEXT SEARCH DICTIONARY command on the dictionary. This can be a “dummy” update that doesn't actually change any parameter values.

### 12.6.3. Synonym Dictionary

This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported (use the thesaurus template ([**Section 12.6.4**](https://www.postgresql.org/docs/10/textsearch-dictionaries.html#TEXTSEARCH-THESAURUS)) for that). A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word “Paris” to “pari”. It is enough to have a Paris paris line in the synonym dictionary and put it before the english\_stem dictionary. For example:

SELECT \* FROM ts\_debug('english', 'Paris');

alias | description | token | dictionaries | dictionary | lexemes

-----------+-----------------+-------+----------------+--------------+---------

asciiword | Word, all ASCII | Paris | {english\_stem} | english\_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my\_synonym (

TEMPLATE = synonym,

SYNONYMS = my\_synonyms

);

ALTER TEXT SEARCH CONFIGURATION english

ALTER MAPPING FOR asciiword

WITH my\_synonym, english\_stem;

SELECT \* FROM ts\_debug('english', 'Paris');

alias | description | token | dictionaries | dictionary | lexemes

-----------+-----------------+-------+---------------------------+------------+---------

asciiword | Word, all ASCII | Paris | {my\_synonym,english\_stem} | my\_synonym | {paris}

The only parameter required by the synonym template is SYNONYMS, which is the base name of its configuration file — my\_synonyms in the above example. The file's full name will be $SHAREDIR/tsearch\_data/my\_synonyms.syn (where $SHAREDIR means the PostgreSQL installation's shared-data directory). The file format is just one line per word to be substituted, with the word followed by its synonym, separated by white space. Blank lines and trailing spaces are ignored.

The synonym template also has an optional parameter CaseSensitive, which defaults to false. When CaseSensitive is false, words in the synonym file are folded to lower case, as are input tokens. When it is true, words and tokens are not folded to lower case, but are compared as-is.

An asterisk (\*) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in to\_tsvector(), but when it is used in to\_tsquery(), the result will be a query item with the prefix match marker (see [**Section 12.3.2**](https://www.postgresql.org/docs/10/textsearch-controls.html#TEXTSEARCH-PARSING-QUERIES)). For example, suppose we have these entries in $SHAREDIR/tsearch\_data/synonym\_sample.syn:

postgres pgsql

postgresql pgsql

postgre pgsql

gogle googl

indices index\*

Then we will get these results:

mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym\_sample');

mydb=# SELECT ts\_lexize('syn','indices');

ts\_lexize

-----------

{index}

(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);

mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;

mydb=# SELECT to\_tsvector('tst','indices');

to\_tsvector

-------------

'index':1

(1 row)

mydb=# SELECT to\_tsquery('tst','indices');

to\_tsquery

------------

'index':\*

(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;

tsvector

---------------------------------

'are' 'indexes' 'useful' 'very'

(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to\_tsquery('tst','indices');

?column?

----------

t

(1 row)

### 12.6.4. Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that includes information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, related terms, etc.

Basically a thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. PostgreSQL's current implementation of the thesaurus dictionary is an extension of the synonym dictionary with added phrase support. A thesaurus dictionary requires a configuration file of the following format:

# this is a comment

sample word(s) : indexed word(s)

more sample word(s) : more indexed word(s)

...

where the colon (:) symbol acts as a delimiter between a phrase and its replacement.

A thesaurus dictionary uses a subdictionary (which is specified in the dictionary's configuration) to normalize the input text before checking for phrase matches. It is only possible to select one subdictionary. An error is reported if the subdictionary fails to recognize a word. In that case, you should remove the use of the word or teach the subdictionary about it. You can place an asterisk (\*) at the beginning of an indexed word to skip applying the subdictionary to it, but all sample words must be known to the subdictionary.

The thesaurus dictionary chooses the longest match if there are multiple phrases matching the input, and ties are broken by using the last definition.

Specific stop words recognized by the subdictionary cannot be specified; instead use ? to mark the location where any stop word can appear. For example, assuming that a and the are stop words according to the subdictionary:

? one ? two : swsw

matches a one the two and the one a two; both would be replaced by swsw.

Since a thesaurus dictionary has the capability to recognize phrases it must remember its state and interact with the parser. A thesaurus dictionary uses these assignments to check if it should handle the next word or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the asciiword token, then a thesaurus dictionary definition like one 7 will not work since token type uint is not assigned to the thesaurus dictionary.

Caution

Thesauruses are used during indexing so any change in the thesaurus dictionary's parameters requires reindexing. For most other dictionary types, small changes such as adding or removing stopwords does not force reindexing.

#### 12.6.4.1. Thesaurus Configuration

To define a new thesaurus dictionary, use the thesaurus template. For example:

CREATE TEXT SEARCH DICTIONARY thesaurus\_simple (

TEMPLATE = thesaurus,

DictFile = mythesaurus,

Dictionary = pg\_catalog.english\_stem

);

Here:

* thesaurus\_simple is the new dictionary's name
* mythesaurus is the base name of the thesaurus configuration file. (Its full name will be $SHAREDIR/tsearch\_data/mythesaurus.ths, where $SHAREDIR means the installation shared-data directory.)
* pg\_catalog.english\_stem is the subdictionary (here, a Snowball English stemmer) to use for thesaurus normalization. Notice that the subdictionary will have its own configuration (for example, stop words), which is not shown here.

Now it is possible to bind the thesaurus dictionary thesaurus\_simple to the desired token types in a configuration, for example:

ALTER TEXT SEARCH CONFIGURATION russian

ALTER MAPPING FOR asciiword, asciihword, hword\_asciipart

WITH thesaurus\_simple;

#### 12.6.4.2. Thesaurus Example

Consider a simple astronomical thesaurus thesaurus\_astro, which contains some astronomical word combinations:

supernovae stars : sn

crab nebulae : crab

Below we create a dictionary and bind some token types to an astronomical thesaurus and English stemmer:

CREATE TEXT SEARCH DICTIONARY thesaurus\_astro (

TEMPLATE = thesaurus,

DictFile = thesaurus\_astro,

Dictionary = english\_stem

);

ALTER TEXT SEARCH CONFIGURATION russian

ALTER MAPPING FOR asciiword, asciihword, hword\_asciipart

WITH thesaurus\_astro, english\_stem;

Now we can see how it works. ts\_lexize is not very useful for testing a thesaurus, because it treats its input as a single token. Instead we can use plainto\_tsquery and to\_tsvector which will break their input strings into multiple tokens:

SELECT plainto\_tsquery('supernova star');

plainto\_tsquery

-----------------

'sn'

SELECT to\_tsvector('supernova star');

to\_tsvector

-------------

'sn':1

In principle, one can use to\_tsquery if you quote the argument:

SELECT to\_tsquery('''supernova star''');

to\_tsquery

------------

'sn'

Notice that supernova star matches supernovae stars in thesaurus\_astro because we specified the english\_stem stemmer in the thesaurus definition. The stemmer removed the e and s.

To index the original phrase as well as the substitute, just include it in the right-hand part of the definition:

supernovae stars : sn supernovae stars

SELECT plainto\_tsquery('supernova star');

plainto\_tsquery

-----------------------------

'sn' & 'supernova' & 'star'

### 12.6.5. Ispell Dictionary

The Ispell dictionary template supports morphological dictionaries, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all declensions and conjugations of the search term bank, e.g., banking, banked, banks, banks', and bank's.

The standard PostgreSQL distribution does not include any Ispell configuration files. Dictionaries for a large number of languages are available from [**Ispell**](http://ficus-www.cs.ucla.edu/geoff/ispell.html). Also, some more modern dictionary file formats are supported — [**MySpell**](http://en.wikipedia.org/wiki/MySpell) (OO < 2.0.1) and [**Hunspell**](http://sourceforge.net/projects/hunspell/) (OO >= 2.0.2). A large list of dictionaries is available on the [**OpenOffice Wiki**](http://wiki.services.openoffice.org/wiki/Dictionaries).

To create an Ispell dictionary perform these steps:

* download dictionary configuration files. OpenOffice extension files have the .oxt extension. It is necessary to extract .aff and .dic files, change extensions to .affix and .dict. For some dictionary files it is also needed to convert characters to the UTF-8 encoding with commands (for example, for a Norwegian language dictionary):
* iconv -f ISO\_8859-1 -t UTF-8 -o nn\_no.affix nn\_NO.aff

iconv -f ISO\_8859-1 -t UTF-8 -o nn\_no.dict nn\_NO.dic

* copy files to the $SHAREDIR/tsearch\_data directory
* load files into PostgreSQL with the following command:
* CREATE TEXT SEARCH DICTIONARY english\_hunspell (
* TEMPLATE = ispell,
* DictFile = en\_us,
* AffFile = en\_us,

Stopwords = english);

Here, DictFile, AffFile, and StopWords specify the base names of the dictionary, affixes, and stop-words files. The stop-words file has the same format explained above for the simple dictionary type. The format of the other files is not specified here but is available from the above-mentioned web sites.

Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary; for example, a Snowball dictionary, which recognizes everything.

The .affix file of Ispell has the following structure:

prefixes

flag \*A:

. > RE # As in enter > reenter

suffixes

flag T:

E > ST # As in late > latest

[^AEIOU]Y > -Y,IEST # As in dirty > dirtiest

[AEIOU]Y > EST # As in gray > grayest

[^EY] > EST # As in small > smallest

And the .dict file has the following structure:

lapse/ADGRS

lard/DGRS

large/PRTY

lark/MRS

Format of the .dict file is:

basic\_form/affix\_class\_name

In the .affix file every affix flag is described in the following format:

condition > [-stripping\_letters,] adding\_affix

Here, condition has a format similar to the format of regular expressions. It can use groupings [...] and [^...]. For example, [AEIOU]Y means that the last letter of the word is "y" and the penultimate letter is "a", "e", "i", "o" or "u". [^EY] means that the last letter is neither "e" nor "y".

Ispell dictionaries support splitting compound words; a useful feature. Notice that the affix file should specify a special flag using the compoundwords controlled statement that marks dictionary words that can participate in compound formation:

compoundwords controlled z

Here are some examples for the Norwegian language:

SELECT ts\_lexize('norwegian\_ispell', 'overbuljongterningpakkmesterassistent');

{over,buljong,terning,pakk,mester,assistent}

SELECT ts\_lexize('norwegian\_ispell', 'sjokoladefabrikk');

{sjokoladefabrikk,sjokolade,fabrikk}

MySpell format is a subset of Hunspell. The .affix file of Hunspell has the following structure:

PFX A Y 1

PFX A 0 re .

SFX T N 4

SFX T 0 st e

SFX T y iest [^aeiou]y

SFX T 0 est [aeiou]y

SFX T 0 est [^ey]

The first line of an affix class is the header. Fields of an affix rules are listed after the header:

* parameter name (PFX or SFX)
* flag (name of the affix class)
* stripping characters from beginning (at prefix) or end (at suffix) of the word
* adding affix
* condition that has a format similar to the format of regular expressions.

The .dict file looks like the .dict file of Ispell:

larder/M

lardy/RT

large/RSPMYT

largehearted

Note

MySpell does not support compound words. Hunspell has sophisticated support for compound words. At present, PostgreSQL implements only the basic compound word operations of Hunspell.

### 12.6.6. Snowball Dictionary

The Snowball dictionary template is based on a project by Martin Porter, inventor of the popular Porter's stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the [**Snowball site**](http://snowballstem.org/) for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary requires a language parameter to identify which stemmer to use, and optionally can specify a stopword file name that gives a list of words to eliminate. (PostgreSQL's standard stopword lists are also provided by the Snowball project.) For example, there is a built-in definition equivalent to

CREATE TEXT SEARCH DICTIONARY english\_stem (

TEMPLATE = snowball,

Language = english,

StopWords = english

);

The stopword file format is the same as already explained.

A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary.

## 12.7. Configuration Example

A text search configuration specifies all options necessary to transform a document into a tsvector: the parser to use to break text into tokens, and the dictionaries to use to transform each token into a lexeme. Every call of to\_tsvector or to\_tsquery needs a text search configuration to perform its processing. The configuration parameter [**default\_text\_search\_config**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DEFAULT-TEXT-SEARCH-CONFIG) specifies the name of the default configuration, which is the one used by text search functions if an explicit configuration parameter is omitted. It can be set in postgresql.conf, or set for an individual session using the SETcommand.

Several predefined text search configurations are available, and you can create custom configurations easily. To facilitate management of text search objects, a set of SQL commands is available, and there are several psql commands that display information about text search objects ([**Section 12.10**](https://www.postgresql.org/docs/10/textsearch-psql.html)).

As an example we will create a configuration pg, starting by duplicating the built-in english configuration:

CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg\_catalog.english );

We will use a PostgreSQL-specific synonym list and store it in $SHAREDIR/tsearch\_data/pg\_dict.syn. The file contents look like:

postgres pg

pgsql pg

postgresql pg

We define the synonym dictionary like this:

CREATE TEXT SEARCH DICTIONARY pg\_dict (

TEMPLATE = synonym,

SYNONYMS = pg\_dict

);

Next we register the Ispell dictionary english\_ispell, which has its own configuration files:

CREATE TEXT SEARCH DICTIONARY english\_ispell (

TEMPLATE = ispell,

DictFile = english,

AffFile = english,

StopWords = english

);

Now we can set up the mappings for words in configuration pg:

ALTER TEXT SEARCH CONFIGURATION pg

ALTER MAPPING FOR asciiword, asciihword, hword\_asciipart,

word, hword, hword\_part

WITH pg\_dict, english\_ispell, english\_stem;

We choose not to index or search some token types that the built-in configuration does handle:

ALTER TEXT SEARCH CONFIGURATION pg

DROP MAPPING FOR email, url, url\_path, sfloat, float;

Now we can test our configuration:

SELECT \* FROM ts\_debug('public.pg', '

PostgreSQL, the highly scalable, SQL compliant, open source object-relational

database management system, is now undergoing beta testing of the next

version of our software.

');

The next step is to set the session to use the new configuration, which was created in the public schema:

=> \dF

List of text search configurations

Schema | Name | Description

---------+------+-------------

public | pg |

SET default\_text\_search\_config = 'public.pg';

SET

SHOW default\_text\_search\_config;

default\_text\_search\_config

----------------------------

public.pg

## 12.8. Testing and Debugging Text Search

The behavior of a custom text search configuration can easily become confusing. The functions described in this section are useful for testing text search objects. You can test a complete configuration, or test parsers and dictionaries separately.

### 12.8.1. Configuration Testing

The function ts\_debug allows easy testing of a text search configuration.

ts\_debug([ ***config*** regconfig, ] ***document*** text,

OUT ***alias*** text,

OUT ***description*** text,

OUT ***token*** text,

OUT ***dictionaries*** regdictionary[],

OUT ***dictionary*** regdictionary,

OUT ***lexemes*** text[])

returns setof record

ts\_debug displays information about every token of ***document*** as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by ***config***, or default\_text\_search\_config if that argument is omitted.

ts\_debug returns one row for each token identified in the text by the parser. The columns returned are

* ***alias*** text — short name of the token type
* ***description*** text — description of the token type
* ***token*** text — text of the token
* ***dictionaries*** regdictionary[] — the dictionaries selected by the configuration for this token type
* ***dictionary*** regdictionary — the dictionary that recognized the token, or NULL if none did
* ***lexemes*** text[] — the lexeme(s) produced by the dictionary that recognized the token, or NULL if none did; an empty array ({}) means it was recognized as a stop word

Here is a simple example:

SELECT \* FROM ts\_debug('english','a fat cat sat on a mat - it ate a fat rats');

alias | description | token | dictionaries | dictionary | lexemes

-----------+-----------------+-------+----------------+--------------+---------

asciiword | Word, all ASCII | a | {english\_stem} | english\_stem | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | fat | {english\_stem} | english\_stem | {fat}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | cat | {english\_stem} | english\_stem | {cat}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | sat | {english\_stem} | english\_stem | {sat}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | on | {english\_stem} | english\_stem | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | a | {english\_stem} | english\_stem | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | mat | {english\_stem} | english\_stem | {mat}

blank | Space symbols | | {} | |

blank | Space symbols | - | {} | |

asciiword | Word, all ASCII | it | {english\_stem} | english\_stem | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | ate | {english\_stem} | english\_stem | {ate}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | a | {english\_stem} | english\_stem | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | fat | {english\_stem} | english\_stem | {fat}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | rats | {english\_stem} | english\_stem | {rat}

For a more extensive demonstration, we first create a public.english configuration and Ispell dictionary for the English language:

CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg\_catalog.english );

CREATE TEXT SEARCH DICTIONARY english\_ispell (

TEMPLATE = ispell,

DictFile = english,

AffFile = english,

StopWords = english

);

ALTER TEXT SEARCH CONFIGURATION public.english

ALTER MAPPING FOR asciiword WITH english\_ispell, english\_stem;

SELECT \* FROM ts\_debug('public.english','The Brightest supernovaes');

alias | description | token | dictionaries | dictionary | lexemes

-----------+-----------------+-------------+-------------------------------+----------------+-------------

asciiword | Word, all ASCII | The | {english\_ispell,english\_stem} | english\_ispell | {}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | Brightest | {english\_ispell,english\_stem} | english\_ispell | {bright}

blank | Space symbols | | {} | |

asciiword | Word, all ASCII | supernovaes | {english\_ispell,english\_stem} | english\_stem | {supernova}

In this example, the word Brightest was recognized by the parser as an ASCII word (alias asciiword). For this token type the dictionary list is english\_ispell and english\_stem. The word was recognized by english\_ispell, which reduced it to the noun bright. The word supernovaes is unknown to the english\_ispell dictionary so it was passed to the next dictionary, and, fortunately, was recognized (in fact, english\_stem is a Snowball dictionary which recognizes everything; that is why it was placed at the end of the dictionary list).

The word The was recognized by the english\_ispell dictionary as a stop word ([**Section 12.6.1**](https://www.postgresql.org/docs/10/textsearch-dictionaries.html#TEXTSEARCH-STOPWORDS)) and will not be indexed. The spaces are discarded too, since the configuration provides no dictionaries at all for them.

You can reduce the width of the output by explicitly specifying which columns you want to see:

SELECT alias, token, dictionary, lexemes

FROM ts\_debug('public.english','The Brightest supernovaes');

alias | token | dictionary | lexemes

-----------+-------------+----------------+-------------

asciiword | The | english\_ispell | {}

blank | | |

asciiword | Brightest | english\_ispell | {bright}

blank | | |

asciiword | supernovaes | english\_stem | {supernova}

### 12.8.2. Parser Testing

The following functions allow direct testing of a text search parser.

ts\_parse(***parser\_name*** text, ***document*** text,

OUT ***tokid*** integer, OUT ***token*** text) returns setof record

ts\_parse(***parser\_oid*** oid, ***document*** text,

OUT ***tokid*** integer, OUT ***token*** text) returns setof record

ts\_parse parses the given ***document*** and returns a series of records, one for each token produced by parsing. Each record includes a tokid showing the assigned token type and a token which is the text of the token. For example:

SELECT \* FROM ts\_parse('default', '123 - a number');

tokid | token

-------+--------

22 | 123

12 |

12 | -

1 | a

12 |

1 | number

ts\_token\_type(***parser\_name*** text, OUT ***tokid*** integer,

OUT ***alias*** text, OUT ***description*** text) returns setof record

ts\_token\_type(***parser\_oid*** oid, OUT ***tokid*** integer,

OUT ***alias*** text, OUT ***description*** text) returns setof record

ts\_token\_type returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer tokid that the parser uses to label a token of that type, the alias that names the token type in configuration commands, and a short description. For example:

SELECT \* FROM ts\_token\_type('default');

tokid | alias | description

-------+-----------------+------------------------------------------

1 | asciiword | Word, all ASCII

2 | word | Word, all letters

3 | numword | Word, letters and digits

4 | email | Email address

5 | url | URL

6 | host | Host

7 | sfloat | Scientific notation

8 | version | Version number

9 | hword\_numpart | Hyphenated word part, letters and digits

10 | hword\_part | Hyphenated word part, all letters

11 | hword\_asciipart | Hyphenated word part, all ASCII

12 | blank | Space symbols

13 | tag | XML tag

14 | protocol | Protocol head

15 | numhword | Hyphenated word, letters and digits

16 | asciihword | Hyphenated word, all ASCII

17 | hword | Hyphenated word, all letters

18 | url\_path | URL path

19 | file | File or path name

20 | float | Decimal notation

21 | int | Signed integer

22 | uint | Unsigned integer

23 | entity | XML entity

### 12.8.3. Dictionary Testing

The ts\_lexize function facilitates dictionary testing.

ts\_lexize(***dict*** regdictionary, ***token*** text) returns text[]

ts\_lexize returns an array of lexemes if the input ***token*** is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or NULL if it is an unknown word.

Examples:

SELECT ts\_lexize('english\_stem', 'stars');

ts\_lexize

-----------

{star}

SELECT ts\_lexize('english\_stem', 'a');

ts\_lexize

-----------

{}

Note

The ts\_lexize function expects a single token, not text. Here is a case where this can be confusing:

SELECT ts\_lexize('thesaurus\_astro','supernovae stars') is null;

?column?

----------

t

The thesaurus dictionary thesaurus\_astro does know the phrase supernovae stars, but ts\_lexize fails since it does not parse the input text but treats it as a single token. Use plainto\_tsquery or to\_tsvector to test thesaurus dictionaries, for example:

SELECT plainto\_tsquery('supernovae stars');

plainto\_tsquery

-----------------

'sn'

**12.9. GIN and GiST Index Types**

There are two kinds of indexes that can be used to speed up full text searches. Note that indexes are not mandatory for full text searching, but in cases where a column is searched on a regular basis, an index is usually desirable.

CREATE INDEX ***name*** ON ***table*** USING GIN (***column***);

Creates a GIN (Generalized Inverted Index)-based index. The ***column*** must be of tsvector type.

CREATE INDEX ***name*** ON ***table*** USING GIST (***column***);

Creates a GiST (Generalized Search Tree)-based index. The ***column*** can be of tsvector or tsquery type.

GIN indexes are the preferred text search index type. As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations. Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words. GIN indexes store only the words (lexemes) of tsvector values, and not their weight labels. Thus a table row recheck is needed when using a query that involves weights.

A GiST index is *lossy*, meaning that the index might produce false matches, and it is necessary to check the actual table row to eliminate such false matches. (PostgreSQL does this automatically when needed.) GiST indexes are lossy because each document is represented in the index by a fixed-length signature. The signature is generated by hashing each word into a single bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit position there will be a false match. If all words in the query have matches (real or false) then the table row must be retrieved to see if the match is correct.

Lossiness causes performance degradation due to unnecessary fetches of table records that turn out to be false matches. Since random access to table records is slow, this limits the usefulness of GiST indexes. The likelihood of false matches depends on several factors, in particular the number of unique words, so using dictionaries to reduce this number is recommended.

Note that GIN index build time can often be improved by increasing [**maintenance\_work\_mem**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAINTENANCE-WORK-MEM), while GiST index build time is not sensitive to that parameter.

Partitioning of big collections and the proper use of GIN and GiST indexes allows the implementation of very fast searches with online update. Partitioning can be done at the database level using table inheritance, or by distributing documents over servers and collecting search results using the [**dblink**](https://www.postgresql.org/docs/10/dblink.html) module. The latter is possible because ranking functions use only local information.

**12.10. psql Support**

Information about text search configuration objects can be obtained in psql using a set of commands:

\dF{d,p,t}[+] [PATTERN]

An optional + produces more details.

The optional parameter ***PATTERN*** can be the name of a text search object, optionally schema-qualified. If ***PATTERN*** is omitted then information about all visible objects will be displayed. ***PATTERN*** can be a regular expression and can provide *separate* patterns for the schema and object names. The following examples illustrate this:

=> \dF \*fulltext\*

List of text search configurations

Schema | Name | Description

--------+--------------+-------------

public | fulltext\_cfg |

=> \dF \*.fulltext\*

List of text search configurations

Schema | Name | Description

----------+----------------------------

fulltext | fulltext\_cfg |

public | fulltext\_cfg |

The available commands are:

\dF[+] [PATTERN]

List text search configurations (add + for more detail).

=> \dF russian

List of text search configurations

Schema | Name | Description

------------+---------+------------------------------------

pg\_catalog | russian | configuration for russian language

=> \dF+ russian

Text search configuration "pg\_catalog.russian"

Parser: "pg\_catalog.default"

Token | Dictionaries

-----------------+--------------

asciihword | english\_stem

asciiword | english\_stem

email | simple

file | simple

float | simple

host | simple

hword | russian\_stem

hword\_asciipart | english\_stem

hword\_numpart | simple

hword\_part | russian\_stem

int | simple

numhword | simple

numword | simple

sfloat | simple

uint | simple

url | simple

url\_path | simple

version | simple

word | russian\_stem

\dFd[+] [PATTERN]

List text search dictionaries (add + for more detail).

=> \dFd

List of text search dictionaries

Schema | Name | Description

------------+-----------------+-----------------------------------------------------------

pg\_catalog | danish\_stem | snowball stemmer for danish language

pg\_catalog | dutch\_stem | snowball stemmer for dutch language

pg\_catalog | english\_stem | snowball stemmer for english language

pg\_catalog | finnish\_stem | snowball stemmer for finnish language

pg\_catalog | french\_stem | snowball stemmer for french language

pg\_catalog | german\_stem | snowball stemmer for german language

pg\_catalog | hungarian\_stem | snowball stemmer for hungarian language

pg\_catalog | italian\_stem | snowball stemmer for italian language

pg\_catalog | norwegian\_stem | snowball stemmer for norwegian language

pg\_catalog | portuguese\_stem | snowball stemmer for portuguese language

pg\_catalog | romanian\_stem | snowball stemmer for romanian language

pg\_catalog | russian\_stem | snowball stemmer for russian language

pg\_catalog | simple | simple dictionary: just lower case and check for stopword

pg\_catalog | spanish\_stem | snowball stemmer for spanish language

pg\_catalog | swedish\_stem | snowball stemmer for swedish language

pg\_catalog | turkish\_stem | snowball stemmer for turkish language

\dFp[+] [PATTERN]

List text search parsers (add + for more detail).

=> \dFp

List of text search parsers

Schema | Name | Description

------------+---------+---------------------

pg\_catalog | default | default word parser

=> \dFp+

Text search parser "pg\_catalog.default"

Method | Function | Description

-----------------+----------------+-------------

Start parse | prsd\_start |

Get next token | prsd\_nexttoken |

End parse | prsd\_end |

Get headline | prsd\_headline |

Get token types | prsd\_lextype |

Token types for parser "pg\_catalog.default"

Token name | Description

-----------------+------------------------------------------

asciihword | Hyphenated word, all ASCII

asciiword | Word, all ASCII

blank | Space symbols

email | Email address

entity | XML entity

file | File or path name

float | Decimal notation

host | Host

hword | Hyphenated word, all letters

hword\_asciipart | Hyphenated word part, all ASCII

hword\_numpart | Hyphenated word part, letters and digits

hword\_part | Hyphenated word part, all letters

int | Signed integer

numhword | Hyphenated word, letters and digits

numword | Word, letters and digits

protocol | Protocol head

sfloat | Scientific notation

tag | XML tag

uint | Unsigned integer

url | URL

url\_path | URL path

version | Version number

word | Word, all letters

(23 rows)

\dFt[+] [PATTERN]

List text search templates (add + for more detail).

=> \dFt

List of text search templates

Schema | Name | Description

------------+-----------+-----------------------------------------------------------

pg\_catalog | ispell | ispell dictionary

pg\_catalog | simple | simple dictionary: just lower case and check for stopword

pg\_catalog | snowball | snowball stemmer

pg\_catalog | synonym | synonym dictionary: replace word by its synonym

pg\_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution

## 12.11. Limitations

The current limitations of PostgreSQL's text search features are:

* The length of each lexeme must be less than 2K bytes
* The length of a tsvector (lexemes + positions) must be less than 1 megabyte
* The number of lexemes must be less than 264
* Position values in tsvector must be greater than 0 and no more than 16,383
* The match distance in a <***N***> (FOLLOWED BY) tsquery operator cannot be more than 16,384
* No more than 256 positions per lexeme
* The number of nodes (lexemes + operators) in a tsquery must be less than 32,768

For comparison, the PostgreSQL 8.1 documentation contained 10,441 unique words, a total of 335,420 words, and the most frequent word “postgresql” was mentioned 6,127 times in 655 documents.

Another example — the PostgreSQL mailing list archives contained 910,989 unique words with 57,491,343 lexemes in 461,020 messages.

## Chapter 13. Concurrency Control

This chapter describes the behavior of the PostgreSQL database system when two or more sessions try to access the same data at the same time. The goals in that situation are to allow efficient access for all sessions while maintaining strict data integrity. Every developer of database applications should be familiar with the topics covered in this chapter.

## 13.1. Introduction

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that each SQL statement sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, providing transaction isolation for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage of using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading. PostgreSQL maintains this guarantee even when providing the strictest level of transaction isolation through the use of an innovative Serializable Snapshot Isolation (SSI) level.

Table- and row-level locking facilities are also available in PostgreSQL for applications which don't generally need full transaction isolation and prefer to explicitly manage particular points of conflict. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

## 13.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation. The most strict is Serializable, which is defined by the standard in a paragraph which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level. The standard notes that due to the definition of Serializable, none of these phenomena are possible at that level. (This is hardly surprising -- if the effect of the transactions must be consistent with having been run one at a time, how could you see any phenomena caused by interactions?)

The phenomena which are prohibited at various levels are:

dirty read

A transaction reads data written by a concurrent uncommitted transaction.

nonrepeatable read

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

serialization anomaly

The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in [**Table 13.1**](https://www.postgresql.org/docs/10/transaction-iso.html#MVCC-ISOLEVEL-TABLE).

**Table 13.1. Transaction Isolation Levels**

| **Isolation Level** | **Dirty Read** | **Nonrepeatable Read** | **Phantom Read** | **Serialization Anomaly** |
| --- | --- | --- | --- | --- |
| Read uncommitted | Allowed, but not in PG | Possible | Possible | Possible |
| Read committed | Not possible | Possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Allowed, but not in PG | Possible |
| Serializable | Not possible | Not possible | Not possible | Not possible |

In PostgreSQL, you can request any of the four standard transaction isolation levels, but internally only three distinct isolation levels are implemented, i.e. PostgreSQL's Read Uncommitted mode behaves like Read Committed. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture.

The table also shows that PostgreSQL's Repeatable Read implementation does not allow phantom reads. Stricter behavior is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, not which phenomena must happen. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command [**SET TRANSACTION**](https://www.postgresql.org/docs/10/sql-set-transaction.html).

Important

Some PostgreSQL data types and functions have special rules regarding transactional behavior. In particular, changes made to a sequence (and therefore the counter of a column declared using serial) are immediately visible to all other transactions and are not rolled back if the transaction that made the changes aborts. See [**Section 9.16**](https://www.postgresql.org/docs/10/functions-sequence.html) and [**Section 8.1.4**](https://www.postgresql.org/docs/10/datatype-numeric.html#DATATYPE-SERIAL).

### 13.2.1. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction uses this isolation level, a SELECT query (without a FOR UPDATE/SHARE clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. In effect, a SELECT query sees a snapshot of the database as of the instant the query begins to run. However, SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive SELECTcommands can see different data, even though they are within a single transaction, if other transactions commit changes after the first SELECT starts and before the second SELECT starts.

UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands behave the same as SELECT in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the WHERE clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation using the updated version of the row. In the case of SELECT FOR UPDATE and SELECT FOR SHARE, this means it is the updated version of the row that is locked and returned to the client.

INSERT with an ON CONFLICT DO UPDATE clause behaves similarly. In Read Committed mode, each row proposed for insertion will either insert or update. Unless there are unrelated errors, one of those two outcomes is guaranteed. If a conflict originates in another transaction whose effects are not yet visible to the INSERT , the UPDATE clause will affect that row, even though possibly no version of that row is conventionally visible to the command.

INSERT with an ON CONFLICT DO NOTHING clause may have insertion not proceed for a row due to the outcome of another transaction whose effects are not visible to the INSERT snapshot. Again, this is only the case in Read Committed mode.

Because of the above rules, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions; however, it is just right for simpler cases. For example, consider updating bank balances with transactions like:

BEGIN;

UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;

UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;

COMMIT;

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start with the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

More complex usage can produce undesirable results in Read Committed mode. For example, consider a DELETE command operating on data that is being both added and removed from its restriction criteria by another command, e.g., assume website is a two-row table with website.hits equaling 9 and 10:

BEGIN;

UPDATE website SET hits = hits + 1;

-- run from another session: DELETE FROM website WHERE hits = 10;

COMMIT;

The DELETE will have no effect even though there is a website.hits = 10 row before and after the UPDATE. This occurs because the pre-update row value 9 is skipped, and when the UPDATE completes and DELETE obtains a lock, the new row value is no longer 10 but 11, which no longer matches the criteria.

Because Read Committed mode starts each command with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue above is whether or not a single command sees an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use; however, it is not sufficient for all cases. Applications that do complex queries and updates might require a more rigorously consistent view of the database than Read Committed mode provides.

### 13.2.2. Repeatable Read Isolation Level

The Repeatable Read isolation level only sees data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena described in [**Table 13.1**](https://www.postgresql.org/docs/10/transaction-iso.html#MVCC-ISOLEVEL-TABLE) except for serialization anomalies. As mentioned above, this is specifically allowed by the standard, which only describes the minimum protections each isolation level must provide.

This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the first non-transaction-control statement in the transaction, not as of the start of the current statement within the transaction. Thus, successive SELECT commands within a single transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started.

Applications using this level must be prepared to retry transactions due to serialization failures.

UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands behave the same as SELECT in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the repeatable read transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the repeatable read transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the repeatable read transaction will be rolled back with the message

ERROR: could not serialize access due to concurrent update

because a repeatable read transaction cannot modify or lock rows changed by other transactions after the repeatable read transaction began.

When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning. The second time through, the transaction will see the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

The Repeatable Read mode provides a rigorous guarantee that each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions of the same level. For example, even a read only transaction at this level may see a control record updated to show that a batch has been completed but not see one of the detail records which is logically part of the batch because it read an earlier revision of the control record. Attempts to enforce business rules by transactions running at this isolation level are not likely to work correctly without careful use of explicit locks to block conflicting transactions.

Note

Prior to PostgreSQL version 9.1, a request for the Serializable transaction isolation level provided exactly the same behavior described here. To retain the legacy Serializable behavior, Repeatable Read should now be requested.

### 13.2.3. Serializable Isolation Level

The Serializable isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions had been executed one after another, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must be prepared to retry transactions due to serialization failures. In fact, this isolation level works exactly the same as Repeatable Read except that it monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a serialization anomaly will trigger a serialization failure.

As an example, consider a table mytab, initially containing:

class | value

-------+-------

1 | 10

1 | 20

2 | 100

2 | 200

Suppose that serializable transaction A computes:

SELECT SUM(value) FROM mytab WHERE class = 1;

and then inserts the result (30) as the value in a new row with class = 2. Concurrently, serializable transaction B computes:

SELECT SUM(value) FROM mytab WHERE class = 2;

and obtains the result 300, which it inserts in a new row with class = 1. Then both transactions try to commit. If either transaction were running at the Repeatable Read isolation level, both would be allowed to commit; but since there is no serial order of execution consistent with the result, using Serializable transactions will allow one transaction to commit and will roll the other back with this message:

ERROR: could not serialize access due to read/write dependencies among transactions

This is because if A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

When relying on Serializable transactions to prevent anomalies, it is important that any data read from a permanent user table not be considered valid until the transaction which read it has successfully committed. This is true even for read-only transactions, except that data read within a deferrable read-only transaction is known to be valid as soon as it is read, because such a transaction waits until it can acquire a snapshot guaranteed to be free from such problems before starting to read any data. In all other cases applications must not depend on results read during a transaction that later aborted; instead, they should retry the transaction until it succeeds.

To guarantee true serializability PostgreSQL uses predicate locking, which means that it keeps locks which allow it to determine when a write would have had an impact on the result of a previous read from a concurrent transaction, had it run first. In PostgreSQL these locks do not cause any blocking and therefore can not play any part in causing a deadlock. They are used to identify and flag dependencies among concurrent Serializable transactions which in certain combinations can lead to serialization anomalies. In contrast, a Read Committed or Repeatable Read transaction which wants to ensure data consistency may need to take out a lock on an entire table, which could block other users attempting to use that table, or it may use SELECT FOR UPDATE or SELECT FOR SHAREwhich not only can block other transactions but cause disk access.

Predicate locks in PostgreSQL, like in most other database systems, are based on data actually accessed by a transaction. These will show up in the [pg\_locks](https://www.postgresql.org/docs/10/view-pg-locks.html) system view with a mode of SIReadLock. The particular locks acquired during execution of a query will depend on the plan used by the query, and multiple finer-grained locks (e.g., tuple locks) may be combined into fewer coarser-grained locks (e.g., page locks) during the course of the transaction to prevent exhaustion of the memory used to track the locks. A READ ONLY transaction may be able to release its SIRead locks before completion, if it detects that no conflicts can still occur which could lead to a serialization anomaly. In fact, READ ONLY transactions will often be able to establish that fact at startup and avoid taking any predicate locks. If you explicitly request a SERIALIZABLE READ ONLY DEFERRABLE transaction, it will block until it can establish this fact. (This is the only case where Serializable transactions block but Repeatable Read transactions don't.) On the other hand, SIRead locks often need to be kept past transaction commit, until overlapping read write transactions complete.

Consistent use of Serializable transactions can simplify development. The guarantee that any set of successfully committed concurrent Serializable transactions will have the same effect as if they were run one at a time means that if you can demonstrate that a single transaction, as written, will do the right thing when run by itself, you can have confidence that it will do the right thing in any mix of Serializable transactions, even without any information about what those other transactions might do, or it will not successfully commit. It is important that an environment which uses this technique have a generalized way of handling serialization failures (which always return with a SQLSTATE value of '40001'), because it will be very hard to predict exactly which transactions might contribute to the read/write dependencies and need to be rolled back to prevent serialization anomalies. The monitoring of read/write dependencies has a cost, as does the restart of transactions which are terminated with a serialization failure, but balanced against the cost and blocking involved in use of explicit locks and SELECT FOR UPDATE or SELECT FOR SHARE, Serializable transactions are the best performance choice for some environments.

While PostgreSQL's Serializable transaction isolation level only allows concurrent transactions to commit if it can prove there is a serial order of execution that would produce the same effect, it doesn't always prevent errors from being raised that would not occur in true serial execution. In particular, it is possible to see unique constraint violations caused by conflicts with overlapping Serializable transactions even after explicitly checking that the key isn't present before attempting to insert it. This can be avoided by making sure that all Serializable transactions that insert potentially conflicting keys explicitly check if they can do so first. For example, imagine an application that asks the user for a new key and then checks that it doesn't exist already by trying to select it first, or generates a new key by selecting the maximum existing key and adding one. If some Serializable transactions insert new keys directly without following this protocol, unique constraints violations might be reported even in cases where they could not occur in a serial execution of the concurrent transactions.

For optimal performance when relying on Serializable transactions for concurrency control, these issues should be considered:

* Declare transactions as READ ONLY when possible.
* Control the number of active connections, using a connection pool if needed. This is always an important performance consideration, but it can be particularly important in a busy system using Serializable transactions.
* Don't put more into a single transaction than needed for integrity purposes.
* Don't leave connections dangling “idle in transaction” longer than necessary. The configuration parameter [**idle\_in\_transaction\_session\_timeout**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-IDLE-IN-TRANSACTION-SESSION-TIMEOUT) may be used to automatically disconnect lingering sessions.
* Eliminate explicit locks, SELECT FOR UPDATE, and SELECT FOR SHARE where no longer needed due to the protections automatically provided by Serializable transactions.
* When the system is forced to combine multiple page-level predicate locks into a single relation-level predicate lock because the predicate lock table is short of memory, an increase in the rate of serialization failures may occur. You can avoid this by increasing [**max\_pred\_locks\_per\_transaction**](https://www.postgresql.org/docs/10/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-TRANSACTION), [**max\_pred\_locks\_per\_relation**](https://www.postgresql.org/docs/10/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-RELATION), and/or [**max\_pred\_locks\_per\_page**](https://www.postgresql.org/docs/10/runtime-config-locks.html#GUC-MAX-PRED-LOCKS-PER-PAGE).
* A sequential scan will always necessitate a relation-level predicate lock. This can result in an increased rate of serialization failures. It may be helpful to encourage the use of index scans by reducing [**random\_page\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-RANDOM-PAGE-COST) and/or increasing [**cpu\_tuple\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CPU-TUPLE-COST). Be sure to weigh any decrease in transaction rollbacks and restarts against any overall change in query execution time.

## 13.3. Explicit Locking

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, TRUNCATE cannot safely be executed concurrently with other operations on the same table, so it obtains an exclusive lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the [pg\_locks](https://www.postgresql.org/docs/10/view-pg-locks.html) system view. For more information on monitoring the status of the lock manager subsystem, refer to [**Chapter 28**](https://www.postgresql.org/docs/10/monitoring.html).

### 13.3.1. Table-level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command [**LOCK**](https://www.postgresql.org/docs/10/sql-lock.html). Remember that all of these lock modes are table-level locks, even if the name contains the word “row”; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts (see [**Table 13.2**](https://www.postgresql.org/docs/10/explicit-locking.html#TABLE-LOCK-COMPATIBILITY)). Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it might acquire ACCESS EXCLUSIVE lock and later acquire ACCESS SHARE lock on the same table.) Non-conflicting lock modes can be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an ACCESS SHARE lock can be held by multiple transactions).

**Table-level Lock Modes**

ACCESS SHARE

Conflicts with the ACCESS EXCLUSIVE lock mode only.

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

ROW SHARE

Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that modifies data in a table.

SHARE UPDATE EXCLUSIVE

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs.

Acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS and ALTER TABLE VALIDATE and other ALTER TABLE variants (for full details see [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html)).

SHARE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

Acquired by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time.

Acquired by CREATE COLLATION, CREATE TRIGGER, and many forms of ALTER TABLE (see [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html)).

EXCLUSIVE

Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARElocks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

Acquired by REFRESH MATERIALIZED VIEW CONCURRENTLY.

ACCESS EXCLUSIVE

Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, and REFRESH MATERIALIZED VIEW (without CONCURRENTLY) commands. Many forms of ALTER TABLE also acquire a lock at this level. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Tip

Only an ACCESS EXCLUSIVE lock blocks a SELECT (without FOR UPDATE/SHARE) statement.

Once acquired, a lock is normally held till end of transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if the savepoint is rolled back to. This is consistent with the principle that ROLLBACK cancels all effects of the commands since the savepoint. The same holds for locks acquired within a PL/pgSQL exception block: an error escape from the block releases locks acquired within it.

**Table 13.2.  Conflicting Lock Modes**

| **Requested Lock Mode** | **Current Lock Mode** | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ACCESS SHARE** | **ROW SHARE** | **ROW EXCLUSIVE** | **SHARE UPDATE EXCLUSIVE** | **SHARE** | **SHARE ROW EXCLUSIVE** | **EXCLUSIVE** | **ACCESS EXCLUSIVE** |
| ACCESS SHARE |  |  |  |  |  |  |  | X |
| ROW SHARE |  |  |  |  |  |  | X | X |
| ROW EXCLUSIVE |  |  |  |  | X | X | X | X |
| SHARE UPDATE EXCLUSIVE |  |  |  | X | X | X | X | X |
| SHARE |  |  | X | X |  | X | X | X |
| SHARE ROW EXCLUSIVE |  |  | X | X | X | X | X | X |
| EXCLUSIVE |  | X | X | X | X | X | X | X |
| ACCESS EXCLUSIVE | X | X | X | X | X | X | X | X |

### 13.3.2. Row-level Locks

In addition to table-level locks, there are row-level locks, which are listed as below with the contexts in which they are used automatically by PostgreSQL. See [**Table 13.3**](https://www.postgresql.org/docs/10/explicit-locking.html#ROW-LOCK-COMPATIBILITY) for a complete table of row-level lock conflicts. Note that a transaction can hold conflicting locks on the same row, even in different subtransactions; but other than that, two transactions can never hold conflicting locks on the same row. Row-level locks do not affect data querying; they block only writers and lockers to the same row.

**Row-level Lock Modes**

FOR UPDATE

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being locked, modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE or SELECT FOR KEY SHARE of these rows will be blocked until the current transaction ends; conversely, SELECT FOR UPDATE will wait for a concurrent transaction that has run any of those commands on the same row, and will then lock and return the updated row (or no row, if the row was deleted). Within a REPEATABLE READ or SERIALIZABLE transaction, however, an error will be thrown if a row to be locked has changed since the transaction started. For further discussion see [**Section 13.4**](https://www.postgresql.org/docs/10/applevel-consistency.html).

The FOR UPDATE lock mode is also acquired by any DELETE on a row, and also by an UPDATE that modifies the values on certain columns. Currently, the set of columns considered for the UPDATEcase are those that have a unique index on them that can be used in a foreign key (so partial indexes and expressional indexes are not considered), but this may change in the future.

FOR NO KEY UPDATE

Behaves similarly to FOR UPDATE, except that the lock acquired is weaker: this lock will not block SELECT FOR KEY SHARE commands that attempt to acquire a lock on the same rows. This lock mode is also acquired by any UPDATE that does not acquire a FOR UPDATE lock.

FOR SHARE

Behaves similarly to FOR NO KEY UPDATE, except that it acquires a shared lock rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing UPDATE, DELETE, SELECT FOR UPDATE or SELECT FOR NO KEY UPDATE on these rows, but it does not prevent them from performing SELECT FOR SHARE or SELECT FOR KEY SHARE.

FOR KEY SHARE

Behaves similarly to FOR SHARE, except that the lock is weaker: SELECT FOR UPDATE is blocked, but not SELECT FOR NO KEY UPDATE. A key-shared lock blocks other transactions from performing DELETE or any UPDATE that changes the key values, but not other UPDATE, and neither does it prevent SELECT FOR NO KEY UPDATE, SELECT FOR SHARE, or SELECT FOR KEY SHARE.

PostgreSQL doesn't remember any information about modified rows in memory, so there is no limit on the number of rows locked at one time. However, locking a row might cause a disk write, e.g., SELECT FOR UPDATE modifies selected rows to mark them locked, and so will result in disk writes.

**Table 13.3. Conflicting Row-level Locks**

| **Requested Lock Mode** | **Current Lock Mode** | | | |
| --- | --- | --- | --- | --- |
| **FOR KEY SHARE** | **FOR SHARE** | **FOR NO KEY UPDATE** | **FOR UPDATE** |
| FOR KEY SHARE |  |  |  | X |
| FOR SHARE |  |  | X | X |
| FOR NO KEY UPDATE |  | X | X | X |
| FOR UPDATE | X | X | X | X |

### 13.3.3. Page-level Locks

In addition to table and row locks, page-level share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers normally need not be concerned with page-level locks, but they are mentioned here for completeness.

### 13.3.4. Deadlocks

The use of explicit locking can increase the likelihood of deadlocks, wherein two (or more) transactions each hold locks that the other wants. For example, if transaction 1 acquires an exclusive lock on table A and then tries to acquire an exclusive lock on table B, while transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A, then neither one can proceed. PostgreSQL automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. (Exactly which transaction will be aborted is difficult to predict and should not be relied upon.)

Note that deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used). Consider the case in which two concurrent transactions modify a table. The first transaction executes:

UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;

This acquires a row-level lock on the row with the specified account number. Then, the second transaction executes:

UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;

UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;

The first UPDATE statement successfully acquires a row-level lock on the specified row, so it succeeds in updating that row. However, the second UPDATE statement finds that the row it is attempting to update has already been locked, so it waits for the transaction that acquired the lock to complete. Transaction two is now waiting on transaction one to complete before it continues execution. Now, transaction one executes:

UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;

Transaction one attempts to acquire a row-level lock on the specified row, but it cannot: transaction two already holds such a lock. So it waits for transaction two to complete. Thus, transaction one is blocked on transaction two, and transaction two is blocked on transaction one: a deadlock condition. PostgreSQL will detect this situation and abort one of the transactions.

The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred. One should also ensure that the first lock acquired on an object in a transaction is the most restrictive mode that will be needed for that object. If it is not feasible to verify this in advance, then deadlocks can be handled on-the-fly by retrying transactions that abort due to deadlocks.

So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

### 13.3.5. Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called advisory locks, because the system does not enforce their use — it is up to the application to use them correctly. Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model. For example, a common use of advisory locks is to emulate pessimistic locking strategies typical of so-called “flat file” data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid table bloat, and are automatically cleaned up by the server at the end of the session.

There are two ways to acquire an advisory lock in PostgreSQL: at session level or at transaction level. Once acquired at session level, an advisory lock is held until explicitly released or the session ends. Unlike standard lock requests, session-level advisory lock requests do not honor transaction semantics: a lock acquired during a transaction that is later rolled back will still be held following the rollback, and likewise an unlock is effective even if the calling transaction fails later. A lock can be acquired multiple times by its owning process; for each completed lock request there must be a corresponding unlock request before the lock is actually released. Transaction-level lock requests, on the other hand, behave more like regular lock requests: they are automatically released at the end of the transaction, and there is no explicit unlock operation. This behavior is often more convenient than the session-level behavior for short-term usage of an advisory lock. Session-level and transaction-level lock requests for the same advisory lock identifier will block each other in the expected way. If a session already holds a given advisory lock, additional requests by it will always succeed, even if other sessions are awaiting the lock; this statement is true regardless of whether the existing lock hold and new request are at session level or transaction level.

Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the [pg\_locks](https://www.postgresql.org/docs/10/view-pg-locks.html) system view.

Both advisory locks and regular locks are stored in a shared memory pool whose size is defined by the configuration variables [**max\_locks\_per\_transaction**](https://www.postgresql.org/docs/10/runtime-config-locks.html#GUC-MAX-LOCKS-PER-TRANSACTION) and [**max\_connections**](https://www.postgresql.org/docs/10/runtime-config-connection.html#GUC-MAX-CONNECTIONS). Care must be taken not to exhaust this memory or the server will be unable to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured.

In certain cases using advisory locking methods, especially in queries involving explicit ordering and LIMIT clauses, care must be taken to control the locks acquired because of the order in which SQL expressions are evaluated. For example:

SELECT pg\_advisory\_lock(id) FROM foo WHERE id = 12345; -- ok

SELECT pg\_advisory\_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!

SELECT pg\_advisory\_lock(q.id) FROM

(

SELECT id FROM foo WHERE id > 12345 LIMIT 100

) q; -- ok

In the above queries, the second form is dangerous because the LIMIT is not guaranteed to be applied before the locking function is executed. This might cause some locks to be acquired that the application was not expecting, and hence would fail to release (until it ends the session). From the point of view of the application, such locks would be dangling, although still viewable in pg\_locks.

The functions provided to manipulate advisory locks are described in [**Section 9.26.10**](https://www.postgresql.org/docs/10/functions-admin.html#FUNCTIONS-ADVISORY-LOCKS).

## 13.4. Data Consistency Checks at the Application Level

It is very difficult to enforce business rules regarding data integrity using Read Committed transactions because the view of the data is shifting with each statement, and even a single statement may not restrict itself to the statement's snapshot if a write conflict occurs.

While a Repeatable Read transaction has a stable view of the data throughout its execution, there is a subtle issue with using MVCC snapshots for data consistency checks, involving something known as read/write conflicts. If one transaction writes data and a concurrent transaction attempts to read the same data (whether before or after the write), it cannot see the work of the other transaction. The reader then appears to have executed first regardless of which started first or which committed first. If that is as far as it goes, there is no problem, but if the reader also writes data which is read by a concurrent transaction there is now a transaction which appears to have run before either of the previously mentioned transactions. If the transaction which appears to have executed last actually commits first, it is very easy for a cycle to appear in a graph of the order of execution of the transactions. When such a cycle appears, integrity checks will not work correctly without some help.

As mentioned in [**Section 13.2.3**](https://www.postgresql.org/docs/10/transaction-iso.html#XACT-SERIALIZABLE), Serializable transactions are just Repeatable Read transactions which add nonblocking monitoring for dangerous patterns of read/write conflicts. When a pattern is detected which could cause a cycle in the apparent order of execution, one of the transactions involved is rolled back to break the cycle.

### 13.4.1. Enforcing Consistency With Serializable Transactions

If the Serializable transaction isolation level is used for all writes and for all reads which need a consistent view of the data, no other effort is required to ensure consistency. Software from other environments which is written to use serializable transactions to ensure consistency should “just work” in this regard in PostgreSQL.

When using this technique, it will avoid creating an unnecessary burden for application programmers if the application software goes through a framework which automatically retries transactions which are rolled back with a serialization failure. It may be a good idea to set default\_transaction\_isolation to serializable. It would also be wise to take some action to ensure that no other transaction isolation level is used, either inadvertently or to subvert integrity checks, through checks of the transaction isolation level in triggers.

See [**Section 13.2.3**](https://www.postgresql.org/docs/10/transaction-iso.html#XACT-SERIALIZABLE) for performance suggestions.

Warning

This level of integrity protection using Serializable transactions does not yet extend to hot standby mode ([**Section 26.5**](https://www.postgresql.org/docs/10/hot-standby.html)). Because of that, those using hot standby may want to use Repeatable Read and explicit locking on the master.

### 13.4.2. Enforcing Consistency With Explicit Blocking Locks

When non-serializable writes are possible, to ensure the current validity of a row and protect it against concurrent updates one must use SELECT FOR UPDATE, SELECT FOR SHARE, or an appropriate LOCK TABLE statement. (SELECT FOR UPDATE and SELECT FOR SHARE lock just the returned rows against concurrent updates, while LOCK TABLE locks the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Also of note to those converting from other environments is the fact that SELECT FOR UPDATE does not ensure that a concurrent transaction will not update or delete a selected row. To do that in PostgreSQL you must actually update the row, even if no values need to be changed. SELECT FOR UPDATE temporarily blocks other transactions from acquiring the same lock or executing an UPDATE or DELETE which would affect the locked row, but once the transaction holding this lock commits or rolls back, a blocked transaction will proceed with the conflicting operation unless an actual UPDATE of the row was performed while the lock was held.

Global validity checks require extra thought under non-serializable MVCC. For example, a banking application might wish to check that the sum of all credits in one table equals the sum of debits in another table, when both tables are being actively updated. Comparing the results of two successive SELECT sum(...) commands will not work reliably in Read Committed mode, since the second query will likely include the results of transactions not counted by the first. Doing the two sums in a single repeatable read transaction will give an accurate picture of only the effects of transactions that committed before the repeatable read transaction started — but one might legitimately wonder whether the answer is still relevant by the time it is delivered. If the repeatable read transaction itself applied some changes before trying to make the consistency check, the usefulness of the check becomes even more debatable, since now it includes some but not all post-transaction-start changes. In such cases a careful person might wish to lock all tables needed for the check, in order to get an indisputable picture of current reality. A SHARE mode (or higher) lock guarantees that there are no uncommitted changes in the locked table, other than those of the current transaction.

Note also that if one is relying on explicit locking to prevent concurrent changes, one should either use Read Committed mode, or in Repeatable Read mode be careful to obtain locks before performing queries. A lock obtained by a repeatable read transaction guarantees that no other transactions modifying the table are still running, but if the snapshot seen by the transaction predates obtaining the lock, it might predate some now-committed changes in the table. A repeatable read transaction's snapshot is actually frozen at the start of its first query or data-modification command (SELECT, INSERT, UPDATE, or DELETE), so it is possible to obtain locks explicitly before the snapshot is frozen.

## 13.5. Caveats

Some DDL commands, currently only [**TRUNCATE**](https://www.postgresql.org/docs/10/sql-truncate.html) and the table-rewriting forms of [**ALTER TABLE**](https://www.postgresql.org/docs/10/sql-altertable.html), are not MVCC-safe. This means that after the truncation or rewrite commits, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the DDL command committed. This will only be an issue for a transaction that did not access the table in question before the DDL command started — any transaction that has done so would hold at least an ACCESS SHARE table lock, which would block the DDL command until that transaction completes. So these commands will not cause any apparent inconsistency in the table contents for successive queries on the target table, but they could cause visible inconsistency between the contents of the target table and other tables in the database.

Support for the Serializable transaction isolation level has not yet been added to Hot Standby replication targets (described in [**Section 26.5**](https://www.postgresql.org/docs/10/hot-standby.html)). The strictest isolation level currently supported in hot standby mode is Repeatable Read. While performing all permanent database writes within Serializable transactions on the master will ensure that all standbys will eventually reach a consistent state, a Repeatable Read transaction run on the standby can sometimes see a transient state that is inconsistent with any serial execution of the transactions on the master.

**13.6. Locking and Indexes**

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL. The various index types are handled as follows:

B-tree, GiST and SP-GiST indexes

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

Hash indexes

Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

GIN indexes

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that insertion of a GIN-indexed value usually produces several index key insertions per row, so GIN might do substantial work for a single value's insertion.

Currently, B-tree indexes offer the best performance for concurrent applications; since they also have more features than hash indexes, they are the recommended index type for concurrent applications that need to index scalar data. When dealing with non-scalar data, B-trees are not useful, and GiST, SP-GiST or GIN indexes should be used instead.

## Chapter 14. Performance Tips 性能优化 ！！！

Query performance can be affected by many things. Some of these can be controlled by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

## 14.1. Using EXPLAIN

PostgreSQL devises a query plan for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to choose good plans. You can use the [**EXPLAIN**](https://www.postgresql.org/docs/10/sql-explain.html) command to see what query plan the planner creates for any query. Plan-reading is an art that requires some experience to master, but this section attempts to cover the basics.

Examples in this section are drawn from the regression test database after doing a VACUUM ANALYZE, using 9.3 development sources. You should be able to get similar results if you try the examples yourself, but your estimated costs and row counts might vary slightly because ANALYZE's statistics are random samples rather than exact, and because costs are inherently somewhat platform-dependent.

The examples use EXPLAIN's default “text” output format, which is compact and convenient for humans to read. If you want to feed EXPLAIN's output to a program for further analysis, you should use one of its machine-readable output formats (XML, JSON, or YAML) instead.

### 14.1.1. EXPLAIN Basics

The structure of a query plan is a tree of plan nodes. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. There are also non-table row sources, such as VALUES clauses and set-returning functions in FROM, which have their own scan node types. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes above the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. Additional lines might appear, indented from the node's summary line, to show additional properties of the node. The very first line (the summary line for the topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like:

EXPLAIN SELECT \* FROM tenk1;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

Since this query has no WHERE clause, it must scan all the rows of the table, so the planner has chosen to use a simple sequential scan plan. The numbers that are quoted in parentheses are (left to right):

* Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.
* Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node's parent node might stop short of reading all available rows (see the LIMIT example below).
* Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.
* Estimated average width of rows output by this plan node (in bytes).

The costs are measured in arbitrary units determined by the planner's cost parameters (see [**Section 19.7.2**](https://www.postgresql.org/docs/10/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS)). Traditional practice is to measure the costs in units of disk page fetches; that is, [**seq\_page\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-SEQ-PAGE-COST) is conventionally set to 1.0 and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.)

The rows value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the number scanned, as a result of filtering by any WHERE-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

EXPLAIN SELECT \* FROM tenk1;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

These numbers are derived very straightforwardly. If you do:

SELECT relpages, reltuples FROM pg\_class WHERE relname = 'tenk1';

you will find that tenk1 has 358 disk pages and 10000 rows. The estimated cost is computed as (disk pages read \* [**seq\_page\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-SEQ-PAGE-COST)) + (rows scanned \* [**cpu\_tuple\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CPU-TUPLE-COST)). By default, seq\_page\_cost is 1.0 and cpu\_tuple\_cost is 0.01, so the estimated cost is (358 \* 1.0) + (10000 \* 0.01) = 458.

Now let's modify the query to add a WHERE condition:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 7000;

QUERY PLAN

------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..483.00 rows=7001 width=244)

Filter: (unique1 < 7000)

Notice that the EXPLAIN output shows the WHERE clause being applied as a “filter” condition attached to the Seq Scan plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has been reduced because of the WHERE clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit (by 10000 \* [**cpu\_operator\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-CPU-OPERATOR-COST), to be exact) to reflect the extra CPU time spent checking the WHERE condition.

The actual number of rows this query would select is 7000, but the rows estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it can change after each ANALYZE command, because the statistics produced by ANALYZE are taken from a randomized sample of the table.

Now, let's make the condition more restrictive:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 100;

QUERY PLAN

------------------------------------------------------------------------------

Bitmap Heap Scan on tenk1 (cost=5.07..229.20 rows=101 width=244)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0)

Index Cond: (unique1 < 100)

Here the planner has decided to use a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two plan levels is that the upper plan node sorts the row locations identified by the index into physical order before reading them, to minimize the cost of separate fetches. The “bitmap” mentioned in the node names is the mechanism that does the sorting.)

Now let's add another condition to the WHERE clause:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

QUERY PLAN

------------------------------------------------------------------------------

Bitmap Heap Scan on tenk1 (cost=5.04..229.43 rows=1 width=244)

Recheck Cond: (unique1 < 100)

Filter: (stringu1 = 'xxx'::name)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0)

Index Cond: (unique1 < 100)

The added condition stringu1 = 'xxx' reduces the output row count estimate, but not the cost because we still have to visit the same set of rows. Notice that the stringu1 clause cannot be applied as an index condition, since this index is only on the unique1 column. Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up slightly to reflect this extra checking.

In some cases the planner will prefer a “simple” index scan plan:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 = 42;

QUERY PLAN

-----------------------------------------------------------------------------

Index Scan using tenk1\_unique1 on tenk1 (cost=0.29..8.30 rows=1 width=244)

Index Cond: (unique1 = 42)

In this type of plan the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You'll most often see this plan type for queries that fetch just a single row. It's also often used for queries that have an ORDER BY condition that matches the index order, because then no extra sorting step is needed to satisfy the ORDER BY.

If there are separate indexes on several of the columns referenced in WHERE, the planner might choose to use an AND or OR combination of the indexes:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

QUERY PLAN

-------------------------------------------------------------------------------------

Bitmap Heap Scan on tenk1 (cost=25.08..60.21 rows=10 width=244)

Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))

-> BitmapAnd (cost=25.08..25.08 rows=10 width=0)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0)

Index Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique2 (cost=0.00..19.78 rows=999 width=0)

Index Cond: (unique2 > 9000)

But this requires visiting both indexes, so it's not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you'll see the plan change accordingly.

Here is an example showing the effects of LIMIT:

EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

QUERY PLAN

-------------------------------------------------------------------------------------

Limit (cost=0.29..14.48 rows=2 width=244)

-> Index Scan using tenk1\_unique2 on tenk1 (cost=0.29..71.27 rows=10 width=244)

Index Cond: (unique2 > 9000)

Filter: (unique1 < 100)

This is the same query as above, but we added a LIMIT so that not all the rows need be retrieved, and the planner changed its mind about what to do. Notice that the total cost and row count of the Index Scan node are shown as if it were run to completion. However, the Limit node is expected to stop after retrieving only a fifth of those rows, so its total cost is only a fifth as much, and that's the actual estimated cost of the query. This plan is preferred over adding a Limit node to the previous plan because the Limit could not avoid paying the startup cost of the bitmap scan, so the total cost would be something over 25 units with that approach.

Let's try joining two tables, using the columns we have been discussing:

EXPLAIN SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

QUERY PLAN

--------------------------------------------------------------------------------------

Nested Loop (cost=4.65..118.62 rows=10 width=488)

-> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)

Recheck Cond: (unique1 < 10)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..4.36 rows=10 width=0)

Index Cond: (unique1 < 10)

-> Index Scan using tenk2\_unique2 on tenk2 t2 (cost=0.29..7.91 rows=1 width=244)

Index Cond: (unique2 = t1.unique2)

In this plan, we have a nested-loop join node with two table scans as inputs, or children. The indentation of the node summary lines reflects the plan tree structure. The join's first, or “outer”, child is a bitmap scan similar to those we saw before. Its cost and row count are the same as we'd get from SELECT ... WHERE unique1 < 10 because we are applying the WHERE clause unique1 < 10 at that node. The t1.unique2 = t2.unique2 clause is not relevant yet, so it doesn't affect the row count of the outer scan. The nested-loop join node will run its second, or “inner” child once for each row obtained from the outer child. Column values from the current outer row can be plugged into the inner scan; here, the t1.unique2 value from the outer row is available, so we get a plan and costs similar to what we saw above for a simple SELECT ... WHERE t2.unique2 = ***constant*** case. (The estimated cost is actually a bit lower than what was seen above, as a result of caching that's expected to occur during the repeated index scans on t2.) The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (10 \* 7.91, here), plus a little CPU time for join processing.

In this example the join's output row count is the same as the product of the two scans' row counts, but that's not true in all cases because there can be additional WHERE clauses that mention both tables and so can only be applied at the join point, not to either input scan. Here's an example:

EXPLAIN SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

QUERY PLAN

---------------------------------------------------------------------------------------------

Nested Loop (cost=4.65..49.46 rows=33 width=488)

Join Filter: (t1.hundred < t2.hundred)

-> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)

Recheck Cond: (unique1 < 10)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..4.36 rows=10 width=0)

Index Cond: (unique1 < 10)

-> Materialize (cost=0.29..8.51 rows=10 width=244)

-> Index Scan using tenk2\_unique2 on tenk2 t2 (cost=0.29..8.46 rows=10 width=244)

Index Cond: (unique2 < 10)

The condition t1.hundred < t2.hundred can't be tested in the tenk2\_unique2 index, so it's applied at the join node. This reduces the estimated output row count of the join node, but does not change either input scan.

Notice that here the planner has chosen to “materialize” the inner relation of the join, by putting a Materialize plan node atop it. This means that the t2 index scan will be done just once, even though the nested-loop join node needs to read that data ten times, once for each row from the outer relation. The Materialize node saves the data in memory as it's read, and then returns the data from memory on each subsequent pass.

When dealing with outer joins, you might see join plan nodes with both “Join Filter” and plain “Filter” conditions attached. Join Filter conditions come from the outer join's ON clause, so a row that fails the Join Filter condition could still get emitted as a null-extended row. But a plain Filter condition is applied after the outer-join rules and so acts to remove rows unconditionally. In an inner join there is no semantic difference between these types of filters.

If we change the query's selectivity a bit, we might get a very different join plan:

EXPLAIN SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN

------------------------------------------------------------------------------------------

Hash Join (cost=230.47..713.98 rows=101 width=488)

Hash Cond: (t2.unique2 = t1.unique2)

-> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244)

-> Hash (cost=229.20..229.20 rows=101 width=244)

-> Bitmap Heap Scan on tenk1 t1 (cost=5.07..229.20 rows=101 width=244)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0)

Index Cond: (unique1 < 100)

Here, the planner has chosen to use a hash join, in which rows of one table are entered into an in-memory hash table, after which the other table is scanned and the hash table is probed for matches to each row. Again note how the indentation reflects the plan structure: the bitmap scan on tenk1 is the input to the Hash node, which constructs the hash table. That's then returned to the Hash Join node, which reads rows from its outer child plan and searches the hash table for each one.

Another possible type of join is a merge join, illustrated here:

EXPLAIN SELECT \*

FROM tenk1 t1, onek t2

WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN

------------------------------------------------------------------------------------------

Merge Join (cost=198.11..268.19 rows=10 width=488)

Merge Cond: (t1.unique2 = t2.unique2)

-> Index Scan using tenk1\_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101 width=244)

Filter: (unique1 < 100)

-> Sort (cost=197.83..200.33 rows=1000 width=244)

Sort Key: t2.unique2

-> Seq Scan on onek t2 (cost=0.00..148.00 rows=1000 width=244)

Merge join requires its input data to be sorted on the join keys. In this plan the tenk1 data is sorted by using an index scan to visit the rows in the correct order, but a sequential scan and sort is preferred for onek, because there are many more rows to be visited in that table. (Sequential-scan-and-sort frequently beats an index scan for sorting many rows, because of the nonsequential disk access required by the index scan.)

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the cheapest, using the enable/disable flags described in [**Section 19.7.1**](https://www.postgresql.org/docs/10/runtime-config-query.html#RUNTIME-CONFIG-QUERY-ENABLE). (This is a crude tool, but useful. See also [**Section 14.3**](https://www.postgresql.org/docs/10/explicit-joins.html).) For example, if we're unconvinced that sequential-scan-and-sort is the best way to deal with table onek in the previous example, we could try

SET enable\_sort = off;

EXPLAIN SELECT \*

FROM tenk1 t1, onek t2

WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN

------------------------------------------------------------------------------------------

Merge Join (cost=0.56..292.65 rows=10 width=488)

Merge Cond: (t1.unique2 = t2.unique2)

-> Index Scan using tenk1\_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101 width=244)

Filter: (unique1 < 100)

-> Index Scan using onek\_unique2 on onek t2 (cost=0.28..224.79 rows=1000 width=244)

which shows that the planner thinks that sorting onek by index-scanning is about 12% more expensive than sequential-scan-and-sort. Of course, the next question is whether it's right about that. We can investigate that using EXPLAIN ANALYZE, as discussed below.

### 14.1.2. EXPLAIN ANALYZE

It is possible to check the accuracy of the planner's estimates by using EXPLAIN's ANALYZE option. With this option, EXPLAIN actually executes the query, and then displays the true row counts and true run time accumulated within each plan node, along with the same estimates that a plain EXPLAIN shows. For example, we might get a result like this:

EXPLAIN ANALYZE SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

QUERY PLAN

---------------------------------------------------------------------------------------------------------------------------------

Nested Loop (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10 loops=1)

-> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244) (actual time=0.057..0.121 rows=10 loops=1)

Recheck Cond: (unique1 < 10)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..4.36 rows=10 width=0) (actual time=0.024..0.024 rows=10 loops=1)

Index Cond: (unique1 < 10)

-> Index Scan using tenk2\_unique2 on tenk2 t2 (cost=0.29..7.91 rows=1 width=244) (actual time=0.021..0.022 rows=1 loops=10)

Index Cond: (unique2 = t1.unique2)

Planning time: 0.181 ms

Execution time: 0.501 ms

Note that the “actual time” values are in milliseconds of real time, whereas the cost estimates are expressed in arbitrary units; so they are unlikely to match up. The thing that's usually most important to look for is whether the estimated row counts are reasonably close to reality. In this example the estimates were all dead-on, but that's quite unusual in practice.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan will be executed once per outer row in the above nested-loop plan. In such cases, the loops value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the loops value to get the total time actually spent in the node. In the above example, we spent a total of 0.220 milliseconds executing the index scans on tenk2.

In some cases EXPLAIN ANALYZE shows additional execution statistics beyond the plan node execution times and row counts. For example, Sort and Hash nodes provide extra information:

EXPLAIN ANALYZE SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------------------

Sort (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)

Sort Key: t1.fivethous

Sort Method: quicksort Memory: 77kB

-> Hash Join (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100 loops=1)

Hash Cond: (t2.unique2 = t1.unique2)

-> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244) (actual time=0.007..2.583 rows=10000 loops=1)

-> Hash (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 28kB

-> Bitmap Heap Scan on tenk1 t1 (cost=5.07..229.20 rows=101 width=244) (actual time=0.080..0.526 rows=100 loops=1)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0) (actual time=0.049..0.049 rows=100 loops=1)

Index Cond: (unique1 < 100)

Planning time: 0.194 ms

Execution time: 8.008 ms

The Sort node shows the sort method used (in particular, whether the sort was in-memory or on-disk) and the amount of memory or disk space needed. The Hash node shows the number of hash buckets and batches as well as the peak amount of memory used for the hash table. (If the number of batches exceeds one, there will also be disk space usage involved, but that is not shown.)

Another type of extra information is the number of rows removed by a filter condition:

EXPLAIN ANALYZE SELECT \* FROM tenk1 WHERE ten < 7;

QUERY PLAN

---------------------------------------------------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 rows=7000 loops=1)

Filter: (ten < 7)

Rows Removed by Filter: 3000

Planning time: 0.083 ms

Execution time: 5.905 ms

These counts can be particularly valuable for filter conditions applied at join nodes. The “Rows Removed” line only appears when at least one scanned row, or potential join pair in the case of a join node, is rejected by the filter condition.

A case similar to filter conditions occurs with “lossy” index scans. For example, consider this search for polygons containing a specific point:

EXPLAIN ANALYZE SELECT \* FROM polygon\_tbl WHERE f1 @> polygon '(0.5,2.0)';

QUERY PLAN

------------------------------------------------------------------------------------------------------

Seq Scan on polygon\_tbl (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 rows=0 loops=1)

Filter: (f1 @> '((0.5,2))'::polygon)

Rows Removed by Filter: 4

Planning time: 0.040 ms

Execution time: 0.083 ms

The planner thinks (quite correctly) that this sample table is too small to bother with an index scan, so we have a plain sequential scan in which all the rows got rejected by the filter condition. But if we force an index scan to be used, we see:

SET enable\_seqscan TO off;

EXPLAIN ANALYZE SELECT \* FROM polygon\_tbl WHERE f1 @> polygon '(0.5,2.0)';

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------

Index Scan using gpolygonind on polygon\_tbl (cost=0.13..8.15 rows=1 width=32) (actual time=0.062..0.062 rows=0 loops=1)

Index Cond: (f1 @> '((0.5,2))'::polygon)

Rows Removed by Index Recheck: 1

Planning time: 0.034 ms

Execution time: 0.144 ms

Here we can see that the index returned one candidate row, which was then rejected by a recheck of the index condition. This happens because a GiST index is “lossy” for polygon containment tests: it actually returns the rows with polygons that overlap the target, and then we have to do the exact containment test on those rows.

EXPLAIN has a BUFFERS option that can be used with ANALYZE to get even more run time statistics:

EXPLAIN (ANALYZE, BUFFERS) SELECT \* FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

QUERY PLAN

---------------------------------------------------------------------------------------------------------------------------------

Bitmap Heap Scan on tenk1 (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.342 rows=10 loops=1)

Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))

Buffers: shared hit=15

-> BitmapAnd (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 loops=1)

Buffers: shared hit=7

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0) (actual time=0.043..0.043 rows=100 loops=1)

Index Cond: (unique1 < 100)

Buffers: shared hit=2

-> Bitmap Index Scan on tenk1\_unique2 (cost=0.00..19.78 rows=999 width=0) (actual time=0.227..0.227 rows=999 loops=1)

Index Cond: (unique2 > 9000)

Buffers: shared hit=5

Planning time: 0.088 ms

Execution time: 0.423 ms

The numbers provided by BUFFERS help to identify which parts of the query are the most I/O-intensive.

Keep in mind that because EXPLAIN ANALYZE actually runs the query, any side-effects will happen as usual, even though whatever results the query might output are discarded in favor of printing the EXPLAIN data. If you want to analyze a data-modifying query without changing your tables, you can roll the command back afterwards, for example:

BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------

Update on tenk1 (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0 loops=1)

-> Bitmap Heap Scan on tenk1 (cost=5.07..229.46 rows=101 width=250) (actual time=0.101..0.439 rows=100 loops=1)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.04 rows=101 width=0) (actual time=0.043..0.043 rows=100 loops=1)

Index Cond: (unique1 < 100)

Planning time: 0.079 ms

Execution time: 14.727 ms

ROLLBACK;

As seen in this example, when the query is an INSERT, UPDATE, or DELETE command, the actual work of applying the table changes is done by a top-level Insert, Update, or Delete plan node. The plan nodes underneath this node perform the work of locating the old rows and/or computing the new data. So above, we see the same sort of bitmap table scan we've seen already, and its output is fed to an Update node that stores the updated rows. It's worth noting that although the data-modifying node can take a considerable amount of run time (here, it's consuming the lion's share of the time), the planner does not currently add anything to the cost estimates to account for that work. That's because the work to be done is the same for every correct query plan, so it doesn't affect planning decisions.

When an UPDATE or DELETE command affects an inheritance hierarchy, the output might look like this:

EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;

QUERY PLAN

-----------------------------------------------------------------------------------

Update on parent (cost=0.00..24.53 rows=4 width=14)

Update on parent

Update on child1

Update on child2

Update on child3

-> Seq Scan on parent (cost=0.00..0.00 rows=1 width=14)

Filter: (f1 = 101)

-> Index Scan using child1\_f1\_key on child1 (cost=0.15..8.17 rows=1 width=14)

Index Cond: (f1 = 101)

-> Index Scan using child2\_f1\_key on child2 (cost=0.15..8.17 rows=1 width=14)

Index Cond: (f1 = 101)

-> Index Scan using child3\_f1\_key on child3 (cost=0.15..8.17 rows=1 width=14)

Index Cond: (f1 = 101)

In this example the Update node needs to consider three child tables as well as the originally-mentioned parent table. So there are four input scanning subplans, one per table. For clarity, the Update node is annotated to show the specific target tables that will be updated, in the same order as the corresponding subplans. (These annotations are new as of PostgreSQL 9.5; in prior versions the reader had to intuit the target tables by inspecting the subplans.)

The Planning time shown by EXPLAIN ANALYZE is the time it took to generate the query plan from the parsed query and optimize it. It does not include parsing or rewriting.

The Execution time shown by EXPLAIN ANALYZE includes executor start-up and shut-down time, as well as the time to run any triggers that are fired, but it does not include parsing, rewriting, or planning time. Time spent executing BEFORE triggers, if any, is included in the time for the related Insert, Update, or Delete node; but time spent executing AFTER triggers is not counted there because AFTER triggers are fired after completion of the whole plan. The total time spent in each trigger (either BEFORE or AFTER) is also shown separately. Note that deferred constraint triggers will not be executed until end of transaction and are thus not considered at all by EXPLAIN ANALYZE.

### 14.1.3. Caveats

There are two significant ways in which run times measured by EXPLAIN ANALYZE can deviate from normal execution of the same query. First, since no output rows are delivered to the client, network transmission costs and I/O conversion costs are not included. Second, the measurement overhead added by EXPLAIN ANALYZE can be significant, especially on machines with slow gettimeofday()operating-system calls. You can use the [**pg\_test\_timing**](https://www.postgresql.org/docs/10/pgtesttiming.html) tool to measure the overhead of timing on your system.

EXPLAIN results should not be extrapolated to situations much different from the one you are actually testing; for example, results on a toy-sized table cannot be assumed to apply to large tables. The planner's cost estimates are not linear and so it might choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index. (We saw this happening in the polygon\_tbl example above.)

There are cases in which the actual and estimated values won't match up well, but nothing is really wrong. One such case occurs when plan node execution is stopped short by a LIMIT or similar effect. For example, in the LIMIT query we used before,

EXPLAIN ANALYZE SELECT \* FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------

Limit (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)

-> Index Scan using tenk1\_unique2 on tenk1 (cost=0.29..72.42 rows=10 width=244) (actual time=0.174..0.244 rows=2 loops=1)

Index Cond: (unique2 > 9000)

Filter: (unique1 < 100)

Rows Removed by Filter: 287

Planning time: 0.096 ms

Execution time: 0.336 ms

the estimated cost and row count for the Index Scan node are shown as though it were run to completion. But in reality the Limit node stopped requesting rows after it got two, so the actual row count is only 2 and the run time is less than the cost estimate would suggest. This is not an estimation error, only a discrepancy in the way the estimates and true values are displayed.

Merge joins also have measurement artifacts that can confuse the unwary. A merge join will stop reading one input if it's exhausted the other input and the next key value in the one input is greater than the last key value of the other input; in such a case there can be no more matches and so no need to scan the rest of the first input. This results in not reading all of one child, with results like those mentioned for LIMIT. Also, if the outer (first) child contains rows with duplicate key values, the inner (second) child is backed up and rescanned for the portion of its rows matching that key value. EXPLAIN ANALYZE counts these repeated emissions of the same inner rows as if they were real additional rows. When there are many outer duplicates, the reported actual row count for the inner child plan node can be significantly larger than the number of rows that are actually in the inner relation.

BitmapAnd and BitmapOr nodes always report their actual row counts as zero, due to implementation limitations.

## 14.2. Statistics Used by the Planner

### 14.2.1. Single-Column Statistics

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table [pg\_class](https://www.postgresql.org/docs/10/catalog-pg-class.html), in the columns reltuples and relpages. We can look at it with queries similar to this one:

SELECT relname, relkind, reltuples, relpages

FROM pg\_class

WHERE relname LIKE 'tenk1%';

relname | relkind | reltuples | relpages

----------------------+---------+-----------+----------

tenk1 | r | 10000 | 358

tenk1\_hundred | i | 10000 | 30

tenk1\_thous\_tenthous | i | 10000 | 30

tenk1\_unique1 | i | 10000 | 30

tenk1\_unique2 | i | 10000 | 30

(5 rows)

Here we can see that tenk1 contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, reltuples and relpages are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX. A VACUUM or ANALYZE operation that does not scan the entire table (which is commonly the case) will incrementally update the reltuples count on the basis of the part of the table it did scan, resulting in an approximate value. In any case, the planner will scale the values it finds in pg\_class to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the selectivity of WHERE clauses, that is, the fraction of rows that match each condition in the WHERE clause. The information used for this task is stored in the [pg\_statistic](https://www.postgresql.org/docs/10/catalog-pg-statistic.html) system catalog. Entries in pg\_statistic are updated by the ANALYZE and VACUUM ANALYZE commands, and are always approximate even when freshly updated.

Rather than look at pg\_statistic directly, it's better to look at its view [pg\_stats](https://www.postgresql.org/docs/10/view-pg-stats.html) when examining the statistics manually. pg\_stats is designed to be more easily readable. Furthermore, pg\_stats is readable by all, whereas pg\_statistic is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The pg\_stats view is restricted to show only rows about tables that the current user can read.) For example, we might do:

SELECT attname, inherited, n\_distinct,

array\_to\_string(most\_common\_vals, E'\n') as most\_common\_vals

FROM pg\_stats

WHERE tablename = 'road';

attname | inherited | n\_distinct | most\_common\_vals

---------+-----------+------------+------------------------------------

name | f | -0.363388 | I- 580 Ramp+

| | | I- 880 Ramp+

| | | Sp Railroad +

| | | I- 580 +

| | | I- 680 Ramp

name | t | -0.284859 | I- 880 Ramp+

| | | I- 580 Ramp+

| | | I- 680 Ramp+

| | | I- 580 +

| | | State Hwy 13 Ramp

(2 rows)

Note that two rows are displayed for the same column, one corresponding to the complete inheritance hierarchy starting at the road table (inherited=t), and another one including only the road table itself (inherited=f).

The amount of information stored in pg\_statistic by ANALYZE, in particular the maximum number of entries in the most\_common\_vals and histogram\_bounds arrays for each column, can be set on a column-by-column basis using the ALTER TABLE SET STATISTICS command, or globally by setting the [**default\_statistics\_target**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-DEFAULT-STATISTICS-TARGET) configuration variable. The default limit is presently 100 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in pg\_statistic and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.

Further details about the planner's use of statistics can be found in [**Chapter 68**](https://www.postgresql.org/docs/10/planner-stats-details.html).

### 14.2.2. Extended Statistics

It is common to see slow queries running bad execution plans because multiple columns used in the query clauses are correlated. The planner normally assumes that multiple conditions are independent of each other, an assumption that does not hold when column values are correlated. Regular statistics, because of their per-individual-column nature, cannot capture any knowledge about cross-column correlation. However, PostgreSQL has the ability to compute multivariate statistics, which can capture such information.

Because the number of possible column combinations is very large, it's impractical to compute multivariate statistics automatically. Instead, extended statistics objects, more often called just statistics objects, can be created to instruct the server to obtain statistics across interesting sets of columns.

Statistics objects are created using the [**CREATE STATISTICS**](https://www.postgresql.org/docs/10/sql-createstatistics.html) command. Creation of such an object merely creates a catalog entry expressing interest in the statistics. Actual data collection is performed by ANALYZE (either a manual command, or background auto-analyze). The collected values can be examined in the [pg\_statistic\_ext](https://www.postgresql.org/docs/10/catalog-pg-statistic-ext.html) catalog.

ANALYZE computes extended statistics based on the same sample of table rows that it takes for computing regular single-column statistics. Since the sample size is increased by increasing the statistics target for the table or any of its columns (as described in the previous section), a larger statistics target will normally result in more accurate extended statistics, as well as more time spent calculating them.

The following subsections describe the kinds of extended statistics that are currently supported.

#### 14.2.2.1. Functional Dependencies

The simplest kind of extended statistics tracks functional dependencies, a concept used in definitions of database normal forms. We say that column b is functionally dependent on column a if knowledge of the value of a is sufficient to determine the value of b, that is there are no two rows having the same value of a but different values of b. In a fully normalized database, functional dependencies should exist only on primary keys and superkeys. However, in practice many data sets are not fully normalized for various reasons; intentional denormalization for performance reasons is a common example. Even in a fully normalized database, there may be partial correlation between some columns, which can be expressed as partial functional dependency.

The existence of functional dependencies directly affects the accuracy of estimates in certain queries. If a query contains conditions on both the independent and the dependent column(s), the conditions on the dependent columns do not further reduce the result size; but without knowledge of the functional dependency, the query planner will assume that the conditions are independent, resulting in underestimating the result size.

To inform the planner about functional dependencies, ANALYZE can collect measurements of cross-column dependency. Assessing the degree of dependency between all sets of columns would be prohibitively expensive, so data collection is limited to those groups of columns appearing together in a statistics object defined with the dependencies option. It is advisable to create dependenciesstatistics only for column groups that are strongly correlated, to avoid unnecessary overhead in both ANALYZE and later query planning.

Here is an example of collecting functional-dependency statistics:

CREATE STATISTICS stts (dependencies) ON zip, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxdependencies

FROM pg\_statistic\_ext

WHERE stxname = 'stts';

stxname | stxkeys | stxdependencies

---------+---------+------------------------------------------

stts | 1 5 | {"1 => 5": 1.000000, "5 => 1": 0.423130}

(1 row)

Here it can be seen that column 1 (zip code) fully determines column 5 (city) so the coefficient is 1.0, while city only determines zip code about 42% of the time, meaning that there are many cities (58%) that are represented by more than a single ZIP code.

When computing the selectivity for a query involving functionally dependent columns, the planner adjusts the per-condition selectivity estimates using the dependency coefficients so as not to produce an underestimate.

##### 14.2.2.1.1. Limitations of Functional Dependencies

Functional dependencies are currently only applied when considering simple equality conditions that compare columns to constant values. They are not used to improve estimates for equality conditions comparing two columns or comparing a column to an expression, nor for range clauses, LIKE or any other type of condition.

When estimating with functional dependencies, the planner assumes that conditions on the involved columns are compatible and hence redundant. If they are incompatible, the correct estimate would be zero rows, but that possibility is not considered. For example, given a query like

SELECT \* FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';

the planner will disregard the city clause as not changing the selectivity, which is correct. However, it will make the same assumption about

SELECT \* FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';

even though there will really be zero rows satisfying this query. Functional dependency statistics do not provide enough information to conclude that, however.

In many practical situations, this assumption is usually satisfied; for example, there might be a GUI in the application that only allows selecting compatible city and ZIP code values to use in a query. But if that's not the case, functional dependencies may not be a viable option.

#### 14.2.2.2. Multivariate N-Distinct Counts

Single-column statistics store the number of distinct values in each column. Estimates of the number of distinct values when combining more than one column (for example, for GROUP BY a, b) are frequently wrong when the planner only has single-column statistical data, causing it to select bad plans.

To improve such estimates, ANALYZE can collect n-distinct statistics for groups of columns. As before, it's impractical to do this for every possible column grouping, so data is collected only for those groups of columns appearing together in a statistics object defined with the ndistinct option. Data will be collected for each possible combination of two or more columns from the set of listed columns.

Continuing the previous example, the n-distinct counts in a table of ZIP codes might look like the following:

CREATE STATISTICS stts2 (ndistinct) ON zip, state, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxndistinct AS nd

FROM pg\_statistic\_ext

WHERE stxname = 'stts2';

-[ RECORD 1 ]--------------------------------------------------------

k | 1 2 5

nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}

(1 row)

This indicates that there are three combinations of columns that have 33178 distinct values: ZIP code and state; ZIP code and city; and ZIP code, city and state (the fact that they are all equal is expected given that ZIP code alone is unique in this table). On the other hand, the combination of city and state has only 27435 distinct values.

It's advisable to create ndistinct statistics objects only on combinations of columns that are actually used for grouping, and for which misestimation of the number of groups is resulting in bad plans. Otherwise, the ANALYZE cycles are just wasted.

## 14.3. Controlling the Planner with Explicit JOIN Clauses

It is possible to control the query planner to some extent by using the explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as:

SELECT \* FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the WHERE condition a.id = b.id, and then joins C to this joined table, using the other WHERE condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the WHERE clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but might have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a genetic probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the [**geqo\_threshold**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-GEQO-THRESHOLD) run-time parameter.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider:

SELECT \* FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner might be able to determine that more than one join order is safe. For example, given:

SELECT \* FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);

it is valid to join A to either B or C first. Currently, only FULL JOIN completely constrains the join order. Most practical cases involving LEFT JOIN or RIGHT JOIN can be rearranged to some extent.

Explicit inner join syntax (INNER JOIN, CROSS JOIN, or unadorned JOIN) is semantically the same as listing the input relations in FROM, so it does not constrain the join order.

Even though most kinds of JOIN don't completely constrain the join order, it is possible to instruct the PostgreSQL query planner to treat all JOIN clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

SELECT \* FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;

SELECT \* FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;

SELECT \* FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

But if we tell the planner to honor the JOIN order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit JOINs, set the [**join\_collapse\_limit**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-JOIN-COLLAPSE-LIMIT) run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators within items of a plain FROM list. For example, consider:

SELECT \* FROM a CROSS JOIN b, c, d, e WHERE ...;

With join\_collapse\_limit = 1, this forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider:

SELECT \*

FROM x, y,

(SELECT \* FROM a, b, c WHERE something) AS ss

WHERE somethingelse;

This situation might arise from use of a view that contains a join; the view's SELECT rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding:

SELECT \* FROM x, y, a, b, c WHERE something AND somethingelse;

This usually results in a better plan than planning the subquery separately. (For example, the outer WHERE conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than from\_collapse\_limit FROM items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

[**from\_collapse\_limit**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-FROM-COLLAPSE-LIMIT) and [**join\_collapse\_limit**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-JOIN-COLLAPSE-LIMIT) are similarly named because they do almost the same thing: one controls when the planner will “flatten out” subqueries, and the other controls when it will flatten out explicit joins. Typically you would either set join\_collapse\_limit equal to from\_collapse\_limit (so that explicit joins and subqueries act similarly) or set join\_collapse\_limit to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

## 14.4. Populating a Database

One might need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

### 14.4.1. Disable Autocommit

When using multiple INSERTs, turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing BEGIN at the start and COMMIT at the end. Some client libraries might do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

### 14.4.2. Use COPY

Use [**COPY**](https://www.postgresql.org/docs/10/sql-copy.html) to load all the rows in one command, instead of using a series of INSERT commands. The COPY command is optimized for loading large numbers of rows; it is less flexible than INSERT, but incurs significantly less overhead for large data loads. Since COPY is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use COPY, it might help to use [**PREPARE**](https://www.postgresql.org/docs/10/sql-prepare.html) to create a prepared INSERT statement, and then use EXECUTE as many times as required. This avoids some of the overhead of repeatedly parsing and planning INSERT. Different interfaces provide this facility in different ways; look for “prepared statements” in the interface documentation.

Note that loading a large number of rows using COPY is almost always faster than using INSERT, even if PREPARE is used and multiple insertions are batched into a single transaction.

COPY is fastest when used within the same transaction as an earlier CREATE TABLE or TRUNCATE command. In such cases no WAL needs to be written, because in case of an error, the files containing the newly loaded data will be removed anyway. However, this consideration only applies when [**wal\_level**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-WAL-LEVEL) is minimal for non-partitioned tables as all commands must write WAL otherwise.

### 14.4.3. Remove Indexes

If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using COPY, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it might be a win to drop the indexes, load the table, and then recreate the indexes. Of course, the database performance for other users might suffer during the time the indexes are missing. One should also think twice before dropping a unique index, since the error checking afforded by the unique constraint will be lost while the index is missing.

### 14.4.4. Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked “in bulk” more efficiently than row-by-row. So it might be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

What's more, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). Loading many millions of rows can cause the trigger event queue to overflow available memory, leading to intolerable swapping or even outright failure of the command. Therefore it may be necessary, not just desirable, to drop and re-apply foreign keys when loading large amounts of data. If temporarily removing the constraint isn't acceptable, the only other recourse may be to split up the load operation into smaller transactions.

### 14.4.5. Increase maintenance\_work\_mem

Temporarily increasing the [**maintenance\_work\_mem**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAINTENANCE-WORK-MEM) configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up CREATE INDEX commands and ALTER TABLE ADD FOREIGN KEY commands. It won't do much for COPY itself, so this advice is only useful when you are using one or both of the above techniques.

### 14.4.6. Increase max\_wal\_size

Temporarily increasing the [**max\_wal\_size**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-MAX-WAL-SIZE) configuration variable can also make large data loads faster. This is because loading a large amount of data into PostgreSQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the checkpoint\_timeout configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing max\_wal\_size temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

### 14.4.7. Disable WAL Archival and Streaming Replication

When loading large amounts of data into an installation that uses WAL archiving or streaming replication, it might be faster to take a new base backup after the load has completed than to process a large amount of incremental WAL data. To prevent incremental WAL logging while loading, disable archiving and streaming replication, by setting [**wal\_level**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-WAL-LEVEL) to minimal, [**archive\_mode**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-ARCHIVE-MODE) to off, and [**max\_wal\_senders**](https://www.postgresql.org/docs/10/runtime-config-replication.html#GUC-MAX-WAL-SENDERS) to zero. But note that changing these settings requires a server restart.

Aside from avoiding the time for the archiver or WAL sender to process the WAL data, doing this will actually make certain commands faster, because they are designed not to write WAL at all if wal\_level is minimal. (They can guarantee crash safety more cheaply by doing an fsync at the end than by writing WAL.) This applies to the following commands:

* CREATE TABLE AS SELECT
* CREATE INDEX (and variants such as ALTER TABLE ADD PRIMARY KEY)
* ALTER TABLE SET TABLESPACE
* CLUSTER
* COPY FROM, when the target table has been created or truncated earlier in the same transaction

### 14.4.8. Run ANALYZE Afterwards

Whenever you have significantly altered the distribution of data within a table, running [**ANALYZE**](https://www.postgresql.org/docs/10/sql-analyze.html) is strongly recommended. This includes bulk loading large amounts of data into the table. Running ANALYZE (or VACUUM ANALYZE) ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics. Note that if the autovacuum daemon is enabled, it might run ANALYZE automatically; see [**Section 24.1.3**](https://www.postgresql.org/docs/10/routine-vacuuming.html#VACUUM-FOR-STATISTICS) and [**Section 24.1.6**](https://www.postgresql.org/docs/10/routine-vacuuming.html#AUTOVACUUM) for more information.

### 14.4.9. Some Notes About pg\_dump

Dump scripts generated by pg\_dump automatically apply several, but not all, of the above guidelines. To reload a pg\_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while restoring a dump, not while creating it. The same points apply whether loading a text dump with psql or using pg\_restore to load from a pg\_dump archive file.)

By default, pg\_dump uses COPY, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case several guidelines are handled automatically. What is left for you to do is to:

* Set appropriate (i.e., larger than normal) values for maintenance\_work\_mem and max\_wal\_size.
* If using WAL archiving or streaming replication, consider disabling them during the restore. To do that, set archive\_mode to off, wal\_level to minimal, and max\_wal\_senders to zero before loading the dump. Afterwards, set them back to the right values and take a fresh base backup.
* Experiment with the parallel dump and restore modes of both pg\_dump and pg\_restore and find the optimal number of concurrent jobs to use. Dumping and restoring in parallel by means of the -j option should give you a significantly higher performance over the serial mode.
* Consider whether the whole dump should be restored as a single transaction. To do that, pass the -1 or --single-transaction command-line option to psql or pg\_restore. When using this mode, even the smallest of errors will rollback the entire restore, possibly discarding many hours of processing. Depending on how interrelated the data is, that might seem preferable to manual cleanup, or not. COPY commands will run fastest if you use a single transaction and have WAL archiving turned off.
* If multiple CPUs are available in the database server, consider using pg\_restore's --jobs option. This allows concurrent data loading and index creation.
* Run ANALYZE afterwards.

A data-only dump will still use COPY, but it does not drop or recreate indexes, and it does not normally touch foreign keys. [**[12]**](https://www.postgresql.org/docs/10/populate.html#ftn.id-1.5.13.7.11.4.2) So when loading a data-only dump, it is up to you to drop and recreate indexes and foreign keys if you wish to use those techniques. It's still useful to increase max\_wal\_size while loading the data, but don't bother increasing maintenance\_work\_mem; rather, you'd do that while manually recreating indexes and foreign keys afterwards. And don't forget to ANALYZE when you're done; see [**Section 24.1.3**](https://www.postgresql.org/docs/10/routine-vacuuming.html#VACUUM-FOR-STATISTICS) and [**Section 24.1.6**](https://www.postgresql.org/docs/10/routine-vacuuming.html#AUTOVACUUM) for more information.

[**[12]**](https://www.postgresql.org/docs/10/populate.html#id-1.5.13.7.11.4.2) You can get the effect of disabling foreign keys by using the --disable-triggers option — but realize that that eliminates, rather than just postpones, foreign key validation, and so it is possible to insert bad data if you use it.

## 14.5. Non-Durable Settings

Durability is a database feature that guarantees the recording of committed transactions even if the server crashes or loses power. However, durability adds significant database overhead, so if your site does not require such a guarantee, PostgreSQL can be configured to run much faster. The following are configuration changes you can make to improve performance in such cases. Except as noted below, durability is still guaranteed in case of a crash of the database software; only abrupt operating system stoppage creates a risk of data loss or corruption when these settings are used.

* Place the database cluster's data directory in a memory-backed file system (i.e. RAM disk). This eliminates all database disk I/O, but limits data storage to the amount of available memory (and perhaps swap).
* Turn off [**fsync**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-FSYNC); there is no need to flush data to disk.
* Turn off [**synchronous\_commit**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-SYNCHRONOUS-COMMIT); there might be no need to force WAL writes to disk on every commit. This setting does risk transaction loss (though not data corruption) in case of a crash of the database.
* Turn off [**full\_page\_writes**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-FULL-PAGE-WRITES); there is no need to guard against partial page writes.
* Increase [**max\_wal\_size**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-MAX-WAL-SIZE) and [**checkpoint\_timeout**](https://www.postgresql.org/docs/10/runtime-config-wal.html#GUC-CHECKPOINT-TIMEOUT); this reduces the frequency of checkpoints, but increases the storage requirements of /pg\_wal.
* Create [**unlogged tables**](https://www.postgresql.org/docs/10/sql-createtable.html#SQL-CREATETABLE-UNLOGGED) to avoid WAL writes, though it makes the tables non-crash-safe.

## Chapter 15. Parallel Query 多个CPU同时处理一个查询计划

PostgreSQL can devise query plans which can leverage multiple CPUs in order to answer queries faster. This feature is known as parallel query. Many queries cannot benefit from parallel query, either due to limitations of the current implementation or because there is no imaginable query plan which is any faster than the serial query plan. However, for queries that can benefit, the speedup from parallel query is often very significant. Many queries can run more than twice as fast when using parallel query, and some queries can run four times faster or even more. Queries that touch a large amount of data but return only a few rows to the user will typically benefit most. This chapter explains some details of how parallel query works and in which situations it can be used so that users who wish to make use of it can understand what to expect.

## 15.1. How Parallel Query Works

When the optimizer determines that parallel query is the fastest execution strategy for a particular query, it will create a query plan which includes a Gather or Gather Merge node. Here is a simple example:

EXPLAIN SELECT \* FROM pgbench\_accounts WHERE filler LIKE '%x%';

QUERY PLAN

-------------------------------------------------------------------------------------

Gather (cost=1000.00..217018.43 rows=1 width=97)

Workers Planned: 2

-> Parallel Seq Scan on pgbench\_accounts (cost=0.00..216018.33 rows=1 width=97)

Filter: (filler ~~ '%x%'::text)

(4 rows)

In all cases, the Gather or Gather Merge node will have exactly one child plan, which is the portion of the plan that will be executed in parallel. If the Gather or Gather Merge node is at the very top of the plan tree, then the entire query will execute in parallel. If it is somewhere else in the plan tree, then only the portion of the plan below it will run in parallel. In the example above, the query accesses only one table, so there is only one plan node other than the Gather node itself; since that plan node is a child of the Gather node, it will run in parallel.

[**Using EXPLAIN**](https://www.postgresql.org/docs/10/using-explain.html), you can see the number of workers chosen by the planner. When the Gather node is reached during query execution, the process which is implementing the user's session will request a number of [**background worker processes**](https://www.postgresql.org/docs/10/bgworker.html) equal to the number of workers chosen by the planner. The number of background workers that the planner will consider using is limited to at most [**max\_parallel\_workers\_per\_gather**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-PARALLEL-WORKERS-PER-GATHER). The total number of background workers that can exist at any one time is limited by both [**max\_worker\_processes**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-WORKER-PROCESSES) and [**max\_parallel\_workers**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-PARALLEL-WORKERS). Therefore, it is possible for a parallel query to run with fewer workers than planned, or even with no workers at all. The optimal plan may depend on the number of workers that are available, so this can result in poor query performance. If this occurrence is frequent, consider increasing max\_worker\_processes and max\_parallel\_workers so that more workers can be run simultaneously or alternatively reducing max\_parallel\_workers\_per\_gather so that the planner requests fewer workers.

Every background worker process which is successfully started for a given parallel query will execute the parallel portion of the plan. The leader will also execute that portion of the plan, but it has an additional responsibility: it must also read all of the tuples generated by the workers. When the parallel portion of the plan generates only a small number of tuples, the leader will often behave very much like an additional worker, speeding up query execution. Conversely, when the parallel portion of the plan generates a large number of tuples, the leader may be almost entirely occupied with reading the tuples generated by the workers and performing any further processing steps which are required by plan nodes above the level of the Gather node or Gather Merge node. In such cases, the leader will do very little of the work of executing the parallel portion of the plan.

When the node at the top of the parallel portion of the plan is Gather Merge rather than Gather, it indicates that each process executing the parallel portion of the plan is producing tuples in sorted order, and that the leader is performing an order-preserving merge. In contrast, Gather reads tuples from the workers in whatever order is convenient, destroying any sort order that may have existed.

## 15.2. When Can Parallel Query Be Used?

There are several settings which can cause the query planner not to generate a parallel query plan under any circumstances. In order for any parallel query plans whatsoever to be generated, the following settings must be configured as indicated.

* [**max\_parallel\_workers\_per\_gather**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-PARALLEL-WORKERS-PER-GATHER) must be set to a value which is greater than zero. This is a special case of the more general principle that no more workers should be used than the number configured via max\_parallel\_workers\_per\_gather.
* [**dynamic\_shared\_memory\_type**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-DYNAMIC-SHARED-MEMORY-TYPE) must be set to a value other than none. Parallel query requires dynamic shared memory in order to pass data between cooperating processes.

In addition, the system must not be running in single-user mode. Since the entire database system is running in single process in this situation, no background workers will be available.

Even when it is in general possible for parallel query plans to be generated, the planner will not generate them for a given query if any of the following are true:

* The query writes any data or locks any database rows. If a query contains a data-modifying operation either at the top level or within a CTE, no parallel plans for that query will be generated. This is a limitation of the current implementation which could be lifted in a future release.
* The query might be suspended during execution. In any situation in which the system thinks that partial or incremental execution might occur, no parallel plan is generated. For example, a cursor created using [**DECLARE CURSOR**](https://www.postgresql.org/docs/10/sql-declare.html) will never use a parallel plan. Similarly, a PL/pgSQL loop of the form FOR x IN query LOOP .. END LOOP will never use a parallel plan, because the parallel query system is unable to verify that the code in the loop is safe to execute while parallel query is active.
* The query uses any function marked PARALLEL UNSAFE. Most system-defined functions are PARALLEL SAFE, but user-defined functions are marked PARALLEL UNSAFE by default. See the discussion of [**Section 15.4**](https://www.postgresql.org/docs/10/parallel-safety.html).
* The query is running inside of another query that is already parallel. For example, if a function called by a parallel query issues an SQL query itself, that query will never use a parallel plan. This is a limitation of the current implementation, but it may not be desirable to remove this limitation, since it could result in a single query using a very large number of processes.
* The transaction isolation level is serializable. This is a limitation of the current implementation.

Even when parallel query plan is generated for a particular query, there are several circumstances under which it will be impossible to execute that plan in parallel at execution time. If this occurs, the leader will execute the portion of the plan below the Gather node entirely by itself, almost as if the Gather node were not present. This will happen if any of the following conditions are met:

* No background workers can be obtained because of the limitation that the total number of background workers cannot exceed [**max\_worker\_processes**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-WORKER-PROCESSES).
* No background workers can be obtained because of the limitation that the total number of background workers launched for purposes of parallel query cannot exceed [**max\_parallel\_workers**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-PARALLEL-WORKERS).
* The client sends an Execute message with a non-zero fetch count. See the discussion of the [**extended query protocol**](https://www.postgresql.org/docs/10/protocol-flow.html#PROTOCOL-FLOW-EXT-QUERY). Since [**libpq**](https://www.postgresql.org/docs/10/libpq.html) currently provides no way to send such a message, this can only occur when using a client that does not rely on libpq. If this is a frequent occurrence, it may be a good idea to set [**max\_parallel\_workers\_per\_gather**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-PARALLEL-WORKERS-PER-GATHER) to zero in sessions where it is likely, so as to avoid generating query plans that may be suboptimal when run serially.
* A prepared statement is executed using a CREATE TABLE .. AS EXECUTE .. statement. This construct converts what otherwise would have been a read-only operation into a read-write operation, making it ineligible for parallel query.
* The transaction isolation level is serializable. This situation does not normally arise, because parallel query plans are not generated when the transaction isolation level is serializable. However, it can happen if the transaction isolation level is changed to serializable after the plan is generated and before it is executed.

## 15.3. Parallel Plans

Because each worker executes the parallel portion of the plan to completion, it is not possible to simply take an ordinary query plan and run it using multiple workers. Each worker would produce a full copy of the output result set, so the query would not run any faster than normal but would produce incorrect results. Instead, the parallel portion of the plan must be what is known internally to the query optimizer as a partial plan; that is, it must be constructed so that each process which executes the plan will generate only a subset of the output rows in such a way that each required output row is guaranteed to be generated by exactly one of the cooperating processes. Generally, this means that the scan on the driving table of the query must be a parallel-aware scan.

### 15.3.1. Parallel Scans

The following types of parallel-aware table scans are currently supported.

* In a parallel sequential scan, the table's blocks will be divided among the cooperating processes. Blocks are handed out one at a time, so that access to the table remains sequential.
* In a parallel bitmap heap scan, one process is chosen as the leader. That process performs a scan of one or more indexes and builds a bitmap indicating which table blocks need to be visited. These blocks are then divided among the cooperating processes as in a parallel sequential scan. In other words, the heap scan is performed in parallel, but the underlying index scan is not.
* In a parallel index scan or parallel index-only scan, the cooperating processes take turns reading data from the index. Currently, parallel index scans are supported only for btree indexes. Each process will claim a single index block and will scan and return all tuples referenced by that block; other process can at the same time be returning tuples from a different index block. The results of a parallel btree scan are returned in sorted order within each worker process.

Other scan types, such as scans of non-btree indexes, may support parallel scans in the future.

### 15.3.2. Parallel Joins

Just as in a non-parallel plan, the driving table may be joined to one or more other tables using a nested loop, hash join, or merge join. The inner side of the join may be any kind of non-parallel plan that is otherwise supported by the planner provided that it is safe to run within a parallel worker. For example, if a nested loop join is chosen, the inner plan may be an index scan which looks up a value taken from the outer side of the join.

Each worker will execute the inner side of the join in full. This is typically not a problem for nested loops, but may be inefficient for cases involving hash or merge joins. For example, for a hash join, this restriction means that an identical hash table is built in each worker process, which works fine for joins against small tables but may not be efficient when the inner table is large. For a merge join, it might mean that each worker performs a separate sort of the inner relation, which could be slow. Of course, in cases where a parallel plan of this type would be inefficient, the query planner will normally choose some other plan (possibly one which does not use parallelism) instead.

### 15.3.3. Parallel Aggregation

PostgreSQL supports parallel aggregation by aggregating in two stages. First, each process participating in the parallel portion of the query performs an aggregation step, producing a partial result for each group of which that process is aware. This is reflected in the plan as a Partial Aggregate node. Second, the partial results are transferred to the leader via Gather or Gather Merge. Finally, the leader re-aggregates the results across all workers in order to produce the final result. This is reflected in the plan as a Finalize Aggregate node.

Because the Finalize Aggregate node runs on the leader process, queries which produce a relatively large number of groups in comparison to the number of input rows will appear less favorable to the query planner. For example, in the worst-case scenario the number of groups seen by the Finalize Aggregate node could be as many as the number of input rows which were seen by all worker processes in the Partial Aggregate stage. For such cases, there is clearly going to be no performance benefit to using parallel aggregation. The query planner takes this into account during the planning process and is unlikely to choose parallel aggregate in this scenario.

Parallel aggregation is not supported in all situations. Each aggregate must be [**safe**](https://www.postgresql.org/docs/10/parallel-safety.html) for parallelism and must have a combine function. If the aggregate has a transition state of type internal, it must have serialization and deserialization functions. See [**CREATE AGGREGATE**](https://www.postgresql.org/docs/10/sql-createaggregate.html) for more details. Parallel aggregation is not supported if any aggregate function call contains DISTINCT or ORDER BY clause and is also not supported for ordered set aggregates or when the query involves GROUPING SETS. It can only be used when all joins involved in the query are also part of the parallel portion of the plan.

### 15.3.4. Parallel Plan Tips

If a query that is expected to do so does not produce a parallel plan, you can try reducing [**parallel\_setup\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-PARALLEL-SETUP-COST) or [**parallel\_tuple\_cost**](https://www.postgresql.org/docs/10/runtime-config-query.html#GUC-PARALLEL-TUPLE-COST). Of course, this plan may turn out to be slower than the serial plan which the planner preferred, but this will not always be the case. If you don't get a parallel plan even with very small values of these settings (e.g. after setting them both to zero), there may be some reason why the query planner is unable to generate a parallel plan for your query. See [**Section 15.2**](https://www.postgresql.org/docs/10/when-can-parallel-query-be-used.html) and [**Section 15.4**](https://www.postgresql.org/docs/10/parallel-safety.html) for information on why this may be the case.

When executing a parallel plan, you can use EXPLAIN (ANALYZE, VERBOSE) to display per-worker statistics for each plan node. This may be useful in determining whether the work is being evenly distributed between all plan nodes and more generally in understanding the performance characteristics of the plan.

## 15.4. Parallel Safety

[**15.4.1. Parallel Labeling for Functions and Aggregates**](https://www.postgresql.org/docs/10/parallel-safety.html#PARALLEL-LABELING)

The planner classifies operations involved in a query as either parallel safe, parallel restricted, or parallel unsafe. A parallel safe operation is one which does not conflict with the use of parallel query. A parallel restricted operation is one which cannot be performed in a parallel worker, but which can be performed in the leader while parallel query is in use. Therefore, parallel restricted operations can never occur below a Gather or Gather Merge node, but can occur elsewhere in a plan which contains such a node. A parallel unsafe operation is one which cannot be performed while parallel query is in use, not even in the leader. When a query contains anything which is parallel unsafe, parallel query is completely disabled for that query.

The following operations are always parallel restricted.

* Scans of common table expressions (CTEs).
* Scans of temporary tables.
* Scans of foreign tables, unless the foreign data wrapper has an IsForeignScanParallelSafe API which indicates otherwise.
* Access to an InitPlan or correlated SubPlan.

### 15.4.1. Parallel Labeling for Functions and Aggregates

The planner cannot automatically determine whether a user-defined function or aggregate is parallel safe, parallel restricted, or parallel unsafe, because this would require predicting every operation which the function could possibly perform. In general, this is equivalent to the Halting Problem and therefore impossible. Even for simple functions where it could conceivably be done, we do not try, since this would be expensive and error-prone. Instead, all user-defined functions are assumed to be parallel unsafe unless otherwise marked. When using [**CREATE FUNCTION**](https://www.postgresql.org/docs/10/sql-createfunction.html) or [**ALTER FUNCTION**](https://www.postgresql.org/docs/10/sql-alterfunction.html), markings can be set by specifying PARALLEL SAFE, PARALLEL RESTRICTED, or PARALLEL UNSAFE as appropriate. When using [**CREATE AGGREGATE**](https://www.postgresql.org/docs/10/sql-createaggregate.html), the PARALLEL option can be specified with SAFE, RESTRICTED, or UNSAFE as the corresponding value.

Functions and aggregates must be marked PARALLEL UNSAFE if they write to the database, access sequences, change the transaction state even temporarily (e.g. a PL/pgSQL function which establishes an EXCEPTION block to catch errors), or make persistent changes to settings. Similarly, functions must be marked PARALLEL RESTRICTED if they access temporary tables, client connection state, cursors, prepared statements, or miscellaneous backend-local state which the system cannot synchronize across workers. For example, setseed and random are parallel restricted for this last reason.

In general, if a function is labeled as being safe when it is restricted or unsafe, or if it is labeled as being restricted when it is in fact unsafe, it may throw errors or produce wrong answers when used in a parallel query. C-language functions could in theory exhibit totally undefined behavior if mislabeled, since there is no way for the system to protect itself against arbitrary C code, but in most likely cases the result will be no worse than for any other function. If in doubt, it is probably best to label functions as UNSAFE.

If a function executed within a parallel worker acquires locks which are not held by the leader, for example by querying a table not referenced in the query, those locks will be released at worker exit, not end of transaction. If you write a function which does this, and this behavior difference is important to you, mark such functions as PARALLEL RESTRICTED to ensure that they execute only in the leader.

Note that the query planner does not consider deferring the evaluation of parallel-restricted functions or aggregates involved in the query in order to obtain a superior plan. So, for example, if a WHEREclause applied to a particular table is parallel restricted, the query planner will not consider performing a scan of that table in the parallel portion of a plan. In some cases, it would be possible (and perhaps even efficient) to include the scan of that table in the parallel portion of the query and defer the evaluation of the WHERE clause so that it happens above the Gather node. However, the planner does not do this.