# Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. This chapter describes most of them, although additional special-purpose functions appear in relevant sections of the manual. Users can also define their own functions and operators, as described in Part V. The psql commands `\df` and `\do` can be used to list all available functions and operators, respectively.

The notation used throughout this chapter to describe the argument and result data types of a function or operator is like this:

`repeat ( text, integer )` → `text`

which says that the function `repeat` takes one text and one integer argument and returns a result of type text. The right arrow is also used to indicate the result of an example, thus:

`repeat('Pg', 4)` → `PgPgPgPg`

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.

## 9.1. Logical Operators

The usual logical operators are available:

`boolean AND boolean` → `boolean`

`boolean OR boolean` → `boolean`

`NOT boolean` → `boolean`

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 operands without affecting the result. (However, it is not guaranteed that the left operand is evaluated before the right operand. See Section 4.2.14 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.

### Table 9.1. Comparison Operators

| Operator | Description |
|---|---|
| *`datatype`* `<` *`datatype`* → `boolean` | Less than |
| *`datatype`* `>` *`datatype`* → `boolean` | Greater than |
| *`datatype`* `<=` *`datatype`* → `boolean` | Less than or equal to |
| *`datatype`* `>=` *`datatype`* → `boolean` | Greater than or equal to |
| *`datatype`* `=` *`datatype`* → `boolean` | Equal |
| *`datatype`* `<>` *`datatype`* → `boolean` | Not equal |
| *`datatype`* `!=` *`datatype`* → `boolean` | Not equal |

> **Note**
>
> `<>` is the standard SQL notation for “not equal”. `!=` is an alias, which is converted to `<>` at a very early stage of parsing. Hence, it is not possible to implement `!=` and `<>` operators that do different things.

These comparison operators are available for all built-in data types that have a natural ordering, including numeric, string, and date/time types. In addition, arrays, composite types, and ranges can be compared if their component data types are comparable.

It is usually possible to compare values of related data types as well; for example `integer` `>` `bigint` will work. Some cases of this sort are implemented directly by “cross-type” comparison operators, but if no such operator is available, the parser will coerce the less-general type to the more-general type and apply the latter's comparison operator.

As shown above, all comparison operators are binary operators that return values of type `boolean`. Thus, expressions like `1 < 2 < 3` are not valid (because there is no `<` operator to compare a Boolean value with `3`). Use the `BETWEEN` predicates shown below to perform range tests.

There are also some comparison predicates, as shown in Table 9.2. These behave much like operators, but have special syntax mandated by the SQL standard.

### Table 9.2. Comparison Predicates

| **Predicate**<br>**Description**<br>**Example(s)** |
|---|
| *`datatype`* `BETWEEN` *`datatype`* `AND` *`datatype`* → `boolean`<br>Between (inclusive of the range endpoints).<br>`2 BETWEEN 1 AND 3` → `t`<br>`2 BETWEEN 3 AND 1` → `f` |
| *`datatype`* `NOT BETWEEN` *`datatype`* `AND` *`datatype`* → `boolean`<br>Not between (the negation of `BETWEEN`).<br>`2 NOT BETWEEN 1 AND 3` → `f` |

| **Predicate**<br>**Description**<br>**Example(s)** |
|---|
| `datatype BETWEEN SYMMETRIC datatype AND datatype` → `boolean`<br>Between, after sorting the two endpoint values.<br>`2 BETWEEN SYMMETRIC 3 AND 1` → `t` |
| `datatype NOT BETWEEN SYMMETRIC datatype AND datatype` → `boolean`<br>Not between, after sorting the two endpoint values.<br>`2 NOT BETWEEN SYMMETRIC 3 AND 1` → `f` |
| `datatype IS DISTINCT FROM datatype` → `boolean`<br>Not equal, treating null as a comparable value.<br>`1 IS DISTINCT FROM NULL` → `t` (rather than `NULL`)<br>`NULL IS DISTINCT FROM NULL` → `f` (rather than `NULL`) |
| `datatype IS NOT DISTINCT FROM datatype` → `boolean`<br>Equal, treating null as a comparable value.<br>`1 IS NOT DISTINCT FROM NULL` → `f` (rather than `NULL`)<br>`NULL IS NOT DISTINCT FROM NULL` → `t` (rather than `NULL`) |
| `datatype IS NULL` → `boolean`<br>Test whether value is null.<br>`1.5 IS NULL` → `f` |
| `datatype IS NOT NULL` → `boolean`<br>Test whether value is not null.<br>`'null' IS NOT NULL` → `t` |
| `datatype ISNULL` → `boolean`<br>Test whether value is null (nonstandard syntax). |
| `datatype NOTNULL` → `boolean`<br>Test whether value is not null (nonstandard syntax). |
| `boolean IS TRUE` → `boolean`<br>Test whether boolean expression yields true.<br>`true IS TRUE` → `t`<br>`NULL::boolean IS TRUE` → `f` (rather than `NULL`) |
| `boolean IS NOT TRUE` → `boolean`<br>Test whether boolean expression yields false or unknown.<br>`true IS NOT TRUE` → `f`<br>`NULL::boolean IS NOT TRUE` → `t` (rather than `NULL`) |
| `boolean IS FALSE` → `boolean`<br>Test whether boolean expression yields false.<br>`true IS FALSE` → `f`<br>`NULL::boolean IS FALSE` → `f` (rather than `NULL`) |
| `boolean IS NOT FALSE` → `boolean`<br>Test whether boolean expression yields true or unknown.<br>`true IS NOT FALSE` → `t`<br>`NULL::boolean IS NOT FALSE` → `t` (rather than `NULL`) |

| Predicate<br>Description<br>Example(s) |
|---|
| `boolean IS UNKNOWN` → `boolean`<br>Test whether boolean expression yields unknown.<br>`true IS UNKNOWN` → `f`<br>`NULL::boolean IS UNKNOWN` → `t` (rather than `NULL`) |
| `boolean IS NOT UNKNOWN` → `boolean`<br>Test whether boolean expression yields true or false.<br>`true IS NOT UNKNOWN` → `t`<br>`NULL::boolean IS NOT UNKNOWN` → `f` (rather than `NULL`) |

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. `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.

The various variants of `BETWEEN` are implemented in terms of the ordinary comparison operators, and therefore will work for any data type(s) that can be compared.

### Note

The use of `AND` in the `BETWEEN` syntax creates an ambiguity with the use of `AND` as a logical operator. To resolve this, only a limited set of expression types are allowed as the second argument of a `BETWEEN` clause. If you need to write a more complex sub-expression in `BETWEEN`, write parentheses around the sub-expression.

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 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. 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

SELECT ROW(table.*) IS NOT NULL FROM table;  -- detect all-non-null rows

SELECT NOT(ROW(table.*) IS NOT NULL) FROM TABLE; -- detect at least one
 null in rows
```

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.

### Table 9.3. Comparison Functions

| Function<br>Description<br>Example(s) |
|---|
| `num_nonnulls ( VARIADIC "any" )` → `integer` |

| **Function**<br>**Description**<br>**Example(s)** |
|---|
| Returns the number of non-null arguments.<br>`num_nonnulls(1, NULL, 2)` → `2` |
| `num_nulls` ( `VARIADIC "any"` ) → `integer`<br>Returns the number of null arguments.<br>`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 shows the mathematical operators that are available for the standard numeric types. Unless otherwise noted, operators shown as accepting *`numeric_type`* are available for all the types `smallint`, `integer`, `bigint`, `numeric`, `real`, and `double precision`. Operators shown as accepting *`integral_type`* are available for the types `smallint`, `integer`, and `bigint`. Except where noted, each form of an operator returns the same data type as its argument(s). Calls involving multiple argument data types, such as `integer + numeric`, are resolved by using the type appearing later in these lists.

### Table 9.4. Mathematical Operators

| **Operator**<br>**Description**<br>**Example(s)** |
|---|
| *`numeric_type`* `+` *`numeric_type`* → *`numeric_type`*<br>Addition<br>`2 + 3` → `5` |
| `+` *`numeric_type`* → *`numeric_type`*<br>Unary plus (no operation)<br>`+ 3.5` → `3.5` |
| *`numeric_type`* `-` *`numeric_type`* → *`numeric_type`*<br>Subtraction<br>`2 - 3` → `-1` |
| `-` *`numeric_type`* → *`numeric_type`*<br>Negation<br>`- (-4)` → `4` |
| *`numeric_type`* `*` *`numeric_type`* → *`numeric_type`*<br>Multiplication<br>`2 * 3` → `6` |
| *`numeric_type`* `/` *`numeric_type`* → *`numeric_type`*<br>Division (for integral types, division truncates the result towards zero)<br>`5.0 / 2` → `2.5000000000000000`<br>`5 / 2` → `2`<br>`(-5) / 2` → `-2` |
| *`numeric_type`* `%` *`numeric_type`* → *`numeric_type`*<br>Modulo (remainder); available for `smallint`, `integer`, `bigint`, and `numeric` |

| Operator<br>Description<br>Example(s) |
|---|
| `5 % 4` → `1` |
| `numeric` ^ `numeric` → `numeric`<br>`double precision` ^ `double precision` → `double precision`<br>Exponentiation<br>`2 ^ 3` → `8`<br>Unlike typical mathematical practice, multiple uses of ^ will associate left to right by default:<br>`2 ^ 3 ^ 3` → `512`<br>`2 ^ (3 ^ 3)` → `134217728` |
| `\|/` `double precision` → `double precision`<br>Square root<br>`\|/ 25.0` → `5` |
| `\|\|/` `double precision` → `double precision`<br>Cube root<br>`\|\|/ 64.0` → `4` |
| @ *`numeric_type`* → *`numeric_type`*<br>Absolute value<br>`@ -5.0` → `5.0` |
| *`integral_type`* & *`integral_type`* → *`integral_type`*<br>Bitwise AND<br>`91 & 15` → `11` |
| *`integral_type`* \| *`integral_type`* → *`integral_type`*<br>Bitwise OR<br>`32 \| 3` → `35` |
| *`integral_type`* # *`integral_type`* → *`integral_type`*<br>Bitwise exclusive OR<br>`17 # 5` → `20` |
| ~ *`integral_type`* → *`integral_type`*<br>Bitwise NOT<br>`~1` → `-2` |
| *`integral_type`* << `integer` → *`integral_type`*<br>Bitwise shift left<br>`1 << 4` → `16` |
| *`integral_type`* >> `integer` → *`integral_type`*<br>Bitwise shift right<br>`8 >> 2` → `2` |

Table 9.5 shows the available mathematical functions. 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(s); cross-type cases are resolved in the same way as explained above for operators. 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<br>Description<br>Example(s) |
|---|
| `abs` ( *`numeric_type`* ) → *`numeric_type`*<br>Absolute value<br>`abs(-17.4)` → `17.4` |
| `cbrt` ( `double precision` ) → `double precision`<br>Cube root<br>`cbrt(64.0)` → `4` |
| `ceil` ( `numeric` ) → `numeric`<br>`ceil` ( `double precision` ) → `double precision`<br>Nearest integer greater than or equal to argument<br>`ceil(42.2)` → `43`<br>`ceil(-42.8)` → `-42` |
| `ceiling` ( `numeric` ) → `numeric`<br>`ceiling` ( `double precision` ) → `double precision`<br>Nearest integer greater than or equal to argument (same as `ceil`)<br>`ceiling(95.3)` → `96` |
| `degrees` ( `double precision` ) → `double precision`<br>Converts radians to degrees<br>`degrees(0.5)` → `28.64788975654116` |
| `div` ( *`y`* `numeric`, *`x`* `numeric` ) → `numeric`<br>Integer quotient of *`y`*/*`x`* (truncates towards zero)<br>`div(9, 4)` → `2` |
| `erf` ( `double precision` ) → `double precision`<br>Error function<br>`erf(1.0)` → `0.8427007929497149` |
| `erfc` ( `double precision` ) → `double precision`<br>Complementary error function (`1 - erf(x)`, without loss of precision for large inputs)<br>`erfc(1.0)` → `0.15729920705028513` |
| `exp` ( `numeric` ) → `numeric`<br>`exp` ( `double precision` ) → `double precision`<br>Exponential (`e` raised to the given power)<br>`exp(1.0)` → `2.7182818284590452` |
| `factorial` ( `bigint` ) → `numeric`<br>Factorial<br>`factorial(5)` → `120` |
| `floor` ( `numeric` ) → `numeric`<br>`floor` ( `double precision` ) → `double precision`<br>Nearest integer less than or equal to argument<br>`floor(42.8)` → `42`<br>`floor(-42.8)` → `-43` |

| Function<br>Description<br>Example(s) |
|---|
| `gamma ( double precision ) → double precision`<br>Gamma function<br>`gamma(0.5)` → `1.772453850905516`<br>`gamma(6)` → `120` |
| `gcd (` *`numeric_type`*`,` *`numeric_type`* `) →` *`numeric_type`*<br>Greatest common divisor (the largest positive number that divides both inputs with no remainder); returns `0` if both inputs are zero; available for `integer`, `bigint`, and `numeric`<br>`gcd(1071, 462)` → `21` |
| `lcm (` *`numeric_type`*`,` *`numeric_type`* `) →` *`numeric_type`*<br>Least common multiple (the smallest strictly positive number that is an integral multiple of both inputs); returns `0` if either input is zero; available for `integer`, `bigint`, and `numeric`<br>`lcm(1071, 462)` → `23562` |
| `lgamma ( double precision ) → double precision`<br>Natural logarithm of the absolute value of the gamma function<br>`lgamma(1000)` → `5905.220423209181` |
| `ln ( numeric ) → numeric`<br>`ln ( double precision ) → double precision`<br>Natural logarithm<br>`ln(2.0)` → `0.6931471805599453` |
| `log ( numeric ) → numeric`<br>`log ( double precision ) → double precision`<br>Base 10 logarithm<br>`log(100)` → `2` |
| `log10 ( numeric ) → numeric`<br>`log10 ( double precision ) → double precision`<br>Base 10 logarithm (same as `log`)<br>`log10(1000)` → `3` |
| `log (` *`b`* `numeric,` *`x`* `numeric ) → numeric`<br>Logarithm of *`x`* to base *`b`*<br>`log(2.0, 64.0)` → `6.0000000000000000` |
| `min_scale ( numeric ) → integer`<br>Minimum scale (number of fractional decimal digits) needed to represent the supplied value precisely<br>`min_scale(8.4100)` → `2` |
| `mod (` *`y numeric_type`*`,` *`x numeric_type`* `) →` *`numeric_type`*<br>Remainder of *`y`*/*`x`*; available for `smallint`, `integer`, `bigint`, and `numeric`<br>`mod(9, 4)` → `1` |
| `pi ( ) → double precision`<br>Approximate value of π<br>`pi()` → `3.141592653589793` |
| `power (` *`a`* `numeric,` *`b`* `numeric ) → numeric` |

| Function<br>Description<br>Example(s) |
|---|
| `power` ( *a* `double precision`, *b* `double precision` ) → `double precision`<br>*a* raised to the power of *b*<br>`power(9, 3)` → `729` |
| `radians` ( `double precision` ) → `double precision`<br>Converts degrees to radians<br>`radians(45.0)` → `0.7853981633974483` |
| `round` ( `numeric` ) → `numeric`<br>`round` ( `double precision` ) → `double precision`<br>Rounds to nearest integer. For `numeric`, ties are broken by rounding away from zero. For `double precision`, the tie-breaking behavior is platform dependent, but “round to nearest even” is the most common rule.<br>`round(42.4)` → `42` |
| `round` ( *v* `numeric`, *s* `integer` ) → `numeric`<br>Rounds *v* to *s* decimal places. Ties are broken by rounding away from zero.<br>`round(42.4382, 2)` → `42.44`<br>`round(1234.56, -1)` → `1230` |
| `scale` ( `numeric` ) → `integer`<br>Scale of the argument (the number of decimal digits in the fractional part)<br>`scale(8.4100)` → `4` |
| `sign` ( `numeric` ) → `numeric`<br>`sign` ( `double precision` ) → `double precision`<br>Sign of the argument (-1, 0, or +1)<br>`sign(-8.4)` → `-1` |
| `sqrt` ( `numeric` ) → `numeric`<br>`sqrt` ( `double precision` ) → `double precision`<br>Square root<br>`sqrt(2)` → `1.4142135623730951` |
| `trim_scale` ( `numeric` ) → `numeric`<br>Reduces the value's scale (number of fractional decimal digits) by removing trailing zeroes<br>`trim_scale(8.4100)` → `8.41` |
| `trunc` ( `numeric` ) → `numeric`<br>`trunc` ( `double precision` ) → `double precision`<br>Truncates to integer (towards zero)<br>`trunc(42.8)` → `42`<br>`trunc(-42.8)` → `-42` |
| `trunc` ( *v* `numeric`, *s* `integer` ) → `numeric`<br>Truncates *v* to *s* decimal places<br>`trunc(42.4382, 2)` → `42.43` |
| `width_bucket` ( *operand* `numeric`, *low* `numeric`, *high* `numeric`, *count* `integer` ) → `integer` |

| Function<br>Description<br>Example(s) |
|---|
| `width_bucket` ( *`operand`* `double precision`, *`low`* `double precision`, *`high`* `double precision`, *`count`* `integer` ) → `integer`<br>Returns the number of the bucket in which *`operand`* falls in a histogram having *`count`* equal-width buckets spanning the range *`low`* to *`high`*. Returns `0` or *`count`*`+1` for an input outside that range.<br>`width_bucket(5.35, 0.024, 10.06, 5)` → `3` |
| `width_bucket` ( *`operand`* `anycompatible`, *`thresholds`* `anycompatiblearray` ) → `integer`<br>Returns the number of the bucket in which *`operand`* falls given an array listing the lower bounds of the buckets. Returns `0` for an input less than the first lower bound. *`operand`* and the array elements can be of any type having standard comparison operators. The *`thresholds`* array *must be sorted*, smallest first, or unexpected results will be obtained.<br>`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])` → `2` |

Table 9.6 shows functions for generating random numbers.

### Table 9.6. Random Functions

| Function<br>Description<br>Example(s) |
|---|
| `random` ( ) → `double precision`<br>Returns a random value in the range 0.0 <= x < 1.0<br>`random()` → `0.897124072839091` |
| `random` ( *`min`* `integer`, *`max`* `integer` ) → `integer`<br>`random` ( *`min`* `bigint`, *`max`* `bigint` ) → `bigint`<br>`random` ( *`min`* `numeric`, *`max`* `numeric` ) → `numeric`<br>Returns a random value in the range *`min`* <= x <= *`max`*. For type `numeric`, the result will have the same number of fractional decimal digits as *`min`* or *`max`*, whichever has more.<br>`random(1, 10)` → `7`<br>`random(-0.499, 0.499)` → `0.347` |
| `random_normal` ( [ *`mean`* `double precision` [, *`stddev`* `double precision` ]] ) → `double precision`<br>Returns a random value from the normal distribution with the given parameters; *`mean`* defaults to 0.0 and *`stddev`* defaults to 1.0<br>`random_normal(0.0, 1.0)` → `0.051285419` |
| `setseed` ( `double precision` ) → `void`<br>Sets the seed for subsequent `random()` and `random_normal()` calls; argument must be between -1.0 and 1.0, inclusive<br>`setseed(0.12345)` |

The `random()` and `random_normal()` functions listed in Table 9.6 use a deterministic pseudo-random number generator. It is fast but not suitable for cryptographic applications; see the pgcrypto module for a more secure alternative. If `setseed()` is called, the series of results of subsequent calls to these functions in the current session can be repeated by re-issuing `setseed()` with the same argument. Without any prior `setseed()` call in the same session, the first call to any of these functions obtains a seed from a platform-dependent source of random bits.

Table 9.7 shows the available trigonometric functions. Each of these functions comes in two variants, one that measures angles in radians and one that measures angles in degrees.

### Table 9.7. Trigonometric Functions

| Function<br>Description<br>Example(s) |
|---|
| `acos` ( `double precision` ) → `double precision`<br>Inverse cosine, result in radians<br>`acos(1)` → `0` |
| `acosd` ( `double precision` ) → `double precision`<br>Inverse cosine, result in degrees<br>`acosd(0.5)` → `60` |
| `asin` ( `double precision` ) → `double precision`<br>Inverse sine, result in radians<br>`asin(1)` → `1.5707963267948966` |
| `asind` ( `double precision` ) → `double precision`<br>Inverse sine, result in degrees<br>`asind(0.5)` → `30` |
| `atan` ( `double precision` ) → `double precision`<br>Inverse tangent, result in radians<br>`atan(1)` → `0.7853981633974483` |
| `atand` ( `double precision` ) → `double precision`<br>Inverse tangent, result in degrees<br>`atand(1)` → `45` |
| `atan2` ( *y* `double precision`, *x* `double precision` ) → `double precision`<br>Inverse tangent of *y*/*x*, result in radians<br>`atan2(1, 0)` → `1.5707963267948966` |
| `atan2d` ( *y* `double precision`, *x* `double precision` ) → `double precision`<br>Inverse tangent of *y*/*x*, result in degrees<br>`atan2d(1, 0)` → `90` |
| `cos` ( `double precision` ) → `double precision`<br>Cosine, argument in radians<br>`cos(0)` → `1` |
| `cosd` ( `double precision` ) → `double precision`<br>Cosine, argument in degrees<br>`cosd(60)` → `0.5` |
| `cot` ( `double precision` ) → `double precision`<br>Cotangent, argument in radians<br>`cot(0.5)` → `1.830487721712452` |
| `cotd` ( `double precision` ) → `double precision`<br>Cotangent, argument in degrees<br>`cotd(45)` → `1` |
| `sin` ( `double precision` ) → `double precision` |

| Function<br>Description<br>Example(s) |
|---|
| Sine, argument in radians<br>`sin(1)` → `0.8414709848078965` |
| `sind ( double precision ) → double precision`<br>Sine, argument in degrees<br>`sind(30)` → `0.5` |
| `tan ( double precision ) → double precision`<br>Tangent, argument in radians<br>`tan(1)` → `1.5574077246549023` |
| `tand ( double precision ) → double precision`<br>Tangent, argument in degrees<br>`tand(45)` → `1` |

> **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)`.

Table 9.8 shows the available hyperbolic functions.

### Table 9.8. Hyperbolic Functions

| Function<br>Description<br>Example(s) |
|---|
| `sinh ( double precision ) → double precision`<br>Hyperbolic sine<br>`sinh(1)` → `1.1752011936438014` |
| `cosh ( double precision ) → double precision`<br>Hyperbolic cosine<br>`cosh(0)` → `1` |
| `tanh ( double precision ) → double precision`<br>Hyperbolic tangent<br>`tanh(1)` → `0.7615941559557649` |
| `asinh ( double precision ) → double precision`<br>Inverse hyperbolic sine<br>`asinh(1)` → `0.881373587019543` |
| `acosh ( double precision ) → double precision`<br>Inverse hyperbolic cosine<br>`acosh(1)` → `0` |
| `atanh ( double precision ) → double precision`<br>Inverse hyperbolic tangent<br>`atanh(0.5)` → `0.5493061443340548` |

# 9.4. String Functions and Operators

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`. Except where noted, these functions and operators are declared to accept and return type `text`. They will interchangeably accept `character varying` arguments. Values of type `character` will be converted to `text` before the function or operator is applied, resulting in stripping any trailing spaces in the `character` value.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9.9. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9.10).

> **Note**
>
> The string concatenation operator (`||`) will accept non-string input, so long as at least one input is of string type, as shown in Table 9.9. For other cases, inserting an explicit coercion to `text` can be used to have non-string input accepted.

### Table 9.9. SQL String Functions and Operators

| Function/Operator<br>Description<br>Example(s) |
|---|
| `text` `\|\|` `text` → `text`<br>Concatenates the two strings.<br>`'Post' \|\| 'greSQL'` → `PostgreSQL` |
| `text` `\|\|` `anynonarray` → `text`<br>`anynonarray` `\|\|` `text` → `text`<br>Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array `\|\|` operators. If you want to concatenate an array's text equivalent, cast it to `text` explicitly.)<br>`'Value: ' \|\| 42` → `Value: 42` |
| `btrim` ( *`string`* `text` [, *`characters`* `text` ] ) → `text`<br>Removes the longest string containing only characters in *`characters`* (a space by default) from the start and end of *`string`*.<br>`btrim('xyxtrimyyx', 'xyz')` → `trim` |
| `text` `IS` [`NOT`] [*`form`*] `NORMALIZED` → `boolean`<br>Checks whether the string is in the specified Unicode normalization form. The optional *`form`* key word specifies the form: `NFC` (the default), `NFD`, `NFKC`, or `NFKD`. This expression can only be used when the server encoding is `UTF8`. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings.<br>`U&'\0061\0308bc' IS NFD NORMALIZED` → `t` |
| `bit_length` ( `text` ) → `integer`<br>Returns number of bits in the string (8 times the `octet_length`).<br>`bit_length('jose')` → `32` |
| `char_length` ( `text` ) → `integer`<br>`character_length` ( `text` ) → `integer`<br>Returns number of characters in the string.<br>`char_length('josé')` → `4` |

| Function/Operator<br>Description<br>Example(s) |
|---|
| `lower` ( `text` ) → `text`<br>Converts the string to all lower case, according to the rules of the database's locale.<br>`lower('TOM')` → `tom` |
| `lpad` ( *`string`* `text`, *`length`* `integer` [, *`fill`* `text` ] ) → `text`<br>Extends 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).<br>`lpad('hi', 5, 'xy')` → `xyxhi` |
| `ltrim` ( *`string`* `text` [, *`characters`* `text` ] ) → `text`<br>Removes the longest string containing only characters in *`characters`* (a space by default) from the start of *`string`*.<br>`ltrim('zzzytest', 'xyz')` → `test` |
| `normalize` ( `text` [, *`form`* ] ) → `text`<br>Converts the string to the specified Unicode normalization form. The optional *`form`* key word specifies the form: `NFC` (the default), `NFD`, `NFKC`, or `NFKD`. This function can only be used when the server encoding is `UTF8`.<br>`normalize(U&'\0061\0308bc', NFC)` → `U&'\00E4bc'` |
| `octet_length` ( `text` ) → `integer`<br>Returns number of bytes in the string.<br>`octet_length('josé')` → `5` (if server encoding is UTF8) |
| `octet_length` ( `character` ) → `integer`<br>Returns number of bytes in the string. Since this version of the function accepts type `character` directly, it will not strip trailing spaces.<br>`octet_length('abc '::character(4))` → `4` |
| `overlay` ( *`string`* `text` `PLACING` *`newsubstring`* `text` `FROM` *`start`* `integer` [ `FOR` *`count`* `integer` ] ) → `text`<br>Replaces the substring of *`string`* that starts at the *`start`*'th character and extends for *`count`* characters with *`newsubstring`*. If *`count`* is omitted, it defaults to the length of *`newsubstring`*.<br>`overlay('Txxxxas' placing 'hom' from 2 for 4)` → `Thomas` |
| `position` ( *`substring`* `text` `IN` *`string`* `text` ) → `integer`<br>Returns first starting index of the specified *`substring`* within *`string`*, or zero if it's not present.<br>`position('om' in 'Thomas')` → `3` |
| `rpad` ( *`string`* `text`, *`length`* `integer` [, *`fill`* `text` ] ) → `text`<br>Extends 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.<br>`rpad('hi', 5, 'xy')` → `hixyx` |
| `rtrim` ( *`string`* `text` [, *`characters`* `text` ] ) → `text`<br>Removes the longest string containing only characters in *`characters`* (a space by default) from the end of *`string`*.<br>`rtrim('testxxzx', 'xyz')` → `test` |
| `substring` ( *`string`* `text` [ `FROM` *`start`* `integer` ] [ `FOR` *`count`* `integer` ] ) → `text`<br>Extracts the substring of *`string`* starting at the *`start`*'th character if that is specified, and stopping after *`count`* characters if that is specified. Provide at least one of *`start`* and *`count`*.<br>`substring('Thomas' from 2 for 3)` → `hom` |

| Function/Operator<br>Description<br>Example(s) |
|---|
| `substring('Thomas' from 3)` → `omas`<br>`substring('Thomas' for 2)` → `Th` |
| `substring` ( *`string`* `text` `FROM` *`pattern`* `text` ) → `text`<br>Extracts the first substring matching POSIX regular expression; see Section 9.7.3.<br>`substring('Thomas' from '...$')` → `mas` |
| `substring` ( *`string`* `text` `SIMILAR` *`pattern`* `text` `ESCAPE` *`escape`* `text` ) → `text`<br>`substring` ( *`string`* `text` `FROM` *`pattern`* `text` `FOR` *`escape`* `text` ) → `text`<br>Extracts the first substring matching SQL regular expression; see Section 9.7.2. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete.<br>`substring('Thomas' similar '%#"o_a#"_' escape '#')` → `oma` |
| `trim` ( [ `LEADING` \| `TRAILING` \| `BOTH` ] [ *`characters`* `text` ] `FROM` *`string`* `text` ) → `text`<br>Removes the longest string containing only characters in *`characters`* (a space by default) from the start, end, or both ends (`BOTH` is the default) of *`string`*.<br>`trim(both 'xyz' from 'yxTomxx')` → `Tom` |
| `trim` ( [ `LEADING` \| `TRAILING` \| `BOTH` ] [ `FROM` ] *`string`* `text` [, *`characters`* `text` ] ) → `text`<br>This is a non-standard syntax for `trim()`.<br>`trim(both from 'yxTomxx', 'xyz')` → `Tom` |
| `unicode_assigned` ( `text` ) → `boolean`<br>Returns `true` if all characters in the string are assigned Unicode codepoints; `false` otherwise. This function can only be used when the server encoding is `UTF8`. |
| `upper` ( `text` ) → `text`<br>Converts the string to all upper case, according to the rules of the database's locale.<br>`upper('tom')` → `TOM` |

Additional string manipulation functions and operators are available and are listed in Table 9.10. (Some of these are used internally to implement the SQL-standard string functions listed in Table 9.9.) There are also pattern-matching operators, which are described in Section 9.7, and operators for full-text search, which are described in Chapter 12.

### Table 9.10. Other String Functions and Operators

| Function/Operator<br>Description<br>Example(s) |
|---|
| `text` `^@` `text` → `boolean`<br>Returns true if the first string starts with the second string (equivalent to the `starts_with()` function).<br>`'alphabet' ^@ 'alph'` → `t` |
| `ascii` ( `text` ) → `integer`<br>Returns the numeric code of the first character of the argument. In UTF8 encoding, returns the Unicode code point of the character. In other multibyte encodings, the argument must be an ASCII character.<br>`ascii('x')` → `120` |
| `chr` ( `integer` ) → `text` |

| Function/Operator<br>Description<br>Example(s) |
|---|
| Returns the character with the given code. In UTF8 encoding the argument is treated as a Unicode code point. In other multibyte encodings the argument must designate an ASCII character. `chr(0)` is disallowed because text data types cannot store that character.<br>`chr(65)` → `A` |
| `concat` ( *val1* "any" [, *val2* "any" [, ...] ] ) → `text`<br>Concatenates the text representations of all the arguments. NULL arguments are ignored.<br>`concat('abcde', 2, NULL, 22)` → `abcde222` |
| `concat_ws` ( *sep* `text`, *val1* "any" [, *val2* "any" [, ...] ] ) → `text`<br>Concatenates all but the first argument, with separators. The first argument is used as the separator string, and should not be NULL. Other NULL arguments are ignored.<br>`concat_ws(',', 'abcde', 2, NULL, 22)` → `abcde,2,22` |
| `format` ( *formatstr* `text` [, *formatarg* "any" [, ...] ] ) → `text`<br>Formats arguments according to a format string; see Section 9.4.1. This function is similar to the C function `sprintf`.<br>`format('Hello %s, %1$s', 'World')` → `Hello World, World` |
| `initcap` ( `text` ) → `text`<br>Converts 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.<br>`initcap('hi THOMAS')` → `Hi Thomas` |
| `casefold` ( `text` ) → `text`<br>Performs case folding of the input string according to the collation. Case folding is similar to case conversion, but the purpose of case folding is to facilitate case-insensitive matching of strings, whereas the purpose of case conversion is to convert to a particular cased form. This function can only be used when the server encoding is `UTF8`.<br>Ordinarily, case folding simply converts to lowercase, but there may be exceptions depending on the collation. For instance, some characters have more than two lowercase variants, or fold to uppercase.<br>Case folding may change the length of the string. For instance, in the `PG_UNICODE_FAST` collation, ß (U+00DF) folds to `ss`.<br>`casefold` can be used for Unicode Default Caseless Matching. It does not always preserve the normalized form of the input string (see normalize).<br>The `libc` provider doesn't support case folding, so `casefold` is identical to lower. |
| `left` ( *string* `text`, *n* `integer` ) → `text`<br>Returns first *n* characters in the string, or when *n* is negative, returns all but last \|*n*\| characters.<br>`left('abcde', 2)` → `ab` |
| `length` ( `text` ) → `integer`<br>Returns the number of characters in the string.<br>`length('jose')` → `4` |
| `md5` ( `text` ) → `text`<br>Computes the MD5 hash of the argument, with the result written in hexadecimal.<br>`md5('abc')` → `900150983cd24fb0d6963f7d28e17f72` |
| `parse_ident` ( *qualified_identifier* `text` [, *strict_mode* `boolean` `DEFAULT` `true` ] ) → `text[]`<br>Splits *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 |

| Function/Operator<br>Description<br>Example(s) |
|---|
| 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[]`.<br>`parse_ident('"SomeSchema".someTable')` → `{SomeSchema,sometable}` |
| `pg_client_encoding` ( ) → `name`<br>Returns current client encoding name.<br>`pg_client_encoding()` → `UTF8` |
| `quote_ident` ( `text` ) → `text`<br>Returns 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 41.1.<br>`quote_ident('Foo bar')` → `"Foo bar"` |
| `quote_literal` ( `text` ) → `text`<br>Returns the given string suitably quoted to be used as a string literal in an SQL statement 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 41.1.<br>`quote_literal(E'O\'Reilly')` → `'O''Reilly'` |
| `quote_literal` ( `anyelement` ) → `text`<br>Converts the given value to text and then quotes it as a literal. Embedded single-quotes and backslashes are properly doubled.<br>`quote_literal(42.5)` → `'42.5'` |
| `quote_nullable` ( `text` ) → `text`<br>Returns the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, returns `NULL`. Embedded single-quotes and backslashes are properly doubled. See also Example 41.1.<br>`quote_nullable(NULL)` → `NULL` |
| `quote_nullable` ( `anyelement` ) → `text`<br>Converts the given value to text and then quotes it as a literal; or, if the argument is null, returns `NULL`. Embedded single-quotes and backslashes are properly doubled.<br>`quote_nullable(42.5)` → `'42.5'` |
| `regexp_count` ( *`string`* `text`, *`pattern`* `text` [, *`start`* `integer` [, *`flags`* `text` ] ] ) → `integer`<br>Returns the number of times the POSIX regular expression *`pattern`* matches in the *`string`*; see Section 9.7.3.<br>`regexp_count('123456789012', '\d\d\d', 2)` → `3` |
| `regexp_instr` ( *`string`* `text`, *`pattern`* `text` [, *`start`* `integer` [, *`N`* `integer` [, *`endoption`* `integer` [, *`flags`* `text` [, *`subexpr`* `integer` ] ] ] ] ] ) → `integer`<br>Returns the position within *`string`* where the *`N`*'th match of the POSIX regular expression *`pattern`* occurs, or zero if there is no such match; see Section 9.7.3.<br>`regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i')` → `3`<br>`regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i', 2)` → `5` |
| `regexp_like` ( *`string`* `text`, *`pattern`* `text` [, *`flags`* `text` ] ) → `boolean`<br>Checks whether a match of the POSIX regular expression *`pattern`* occurs within *`string`*; see Section 9.7.3. |

| Function/Operator<br>Description<br>Example(s) |
|---|
| `regexp_like('Hello World', 'world$', 'i')` → `t` |
| `regexp_match` ( *`string`* `text`, *`pattern`* `text` [, *`flags`* `text` ] ) → `text[]`<br>Returns substrings within the first match of the POSIX regular expression *`pattern`* to the *`string`*; see Section 9.7.3.<br>`regexp_match('foobarbequebaz', '(bar)(beque)')` → `{bar,beque}` |
| `regexp_matches` ( *`string`* `text`, *`pattern`* `text` [, *`flags`* `text` ] ) → `setof text[]`<br>Returns substrings within the first match of the POSIX regular expression *`pattern`* to the *`string`*, or substrings within all such matches if the `g` flag is used; see Section 9.7.3.<br>`regexp_matches('foobarbequebaz', 'ba.', 'g')` →<br>`{bar}`<br>`{baz}` |
| `regexp_replace` ( *`string`* `text`, *`pattern`* `text`, *`replacement`* `text` [, *`flags`* `text` ] ) → `text`<br>Replaces the substring that is the first match to the POSIX regular expression *`pattern`*, or all such matches if the `g` flag is used; see Section 9.7.3.<br>`regexp_replace('Thomas', '.[mN]a.', 'M')` → `ThM` |
| `regexp_replace` ( *`string`* `text`, *`pattern`* `text`, *`replacement`* `text`, *`start`* `integer` [, *`N`* `integer` [, *`flags`* `text` ] ] ) → `text`<br>Replaces the substring that is the *`N`*'th match to the POSIX regular expression *`pattern`*, or all such matches if *`N`* is zero, with the search beginning at the *`start`*'th character of *`string`*. If *`N`* is omitted, it defaults to 1. See Section 9.7.3.<br>`regexp_replace('Thomas', '.', 'X', 3, 2)` → `ThoXas`<br>`regexp_replace(string=>'hello world', pattern=>'l', replacement=>'XX', start=>1, "N"=>2)` → `helXXo world` |
| `regexp_split_to_array` ( *`string`* `text`, *`pattern`* `text` [, *`flags`* `text` ] ) → `text[]`<br>Splits *`string`* using a POSIX regular expression as the delimiter, producing an array of results; see Section 9.7.3.<br>`regexp_split_to_array('hello world', '\s+')` → `{hello,world}` |
| `regexp_split_to_table` ( *`string`* `text`, *`pattern`* `text` [, *`flags`* `text` ] ) → `setof text`<br>Splits *`string`* using a POSIX regular expression as the delimiter, producing a set of results; see Section 9.7.3.<br>`regexp_split_to_table('hello world', '\s+')` →<br>`hello`<br>`world` |
| `regexp_substr` ( *`string`* `text`, *`pattern`* `text` [, *`start`* `integer` [, *`N`* `integer` [, *`flags`* `text` [, *`subexpr`* `integer` ] ] ] ] ) → `text`<br>Returns the substring within *`string`* that matches the *`N`*'th occurrence of the POSIX regular expression *`pattern`*, or `NULL` if there is no such match; see Section 9.7.3.<br>`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i')` → `CDEF`<br>`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i', 2)` → `EF` |
| `repeat` ( *`string`* `text`, *`number`* `integer` ) → `text`<br>Repeats *`string`* the specified *`number`* of times. |

| Function/Operator<br>Description<br>Example(s) |
|---|
| `repeat('Pg', 4)` → `PgPgPgPg` |
| `replace` ( *`string`* `text`, *`from`* `text`, *`to`* `text` ) → `text`<br>Replaces all occurrences in *`string`* of substring *`from`* with substring *`to`*.<br>`replace('abcdefabcdef', 'cd', 'XX')` → `abXXefabXXef` |
| `reverse` ( `text` ) → `text`<br>Reverses the order of the characters in the string.<br>`reverse('abcde')` → `edcba` |
| `right` ( *`string`* `text`, *`n`* `integer` ) → `text`<br>Returns last *`n`* characters in the string, or when *`n`* is negative, returns all but first \|*`n`*\| characters.<br>`right('abcde', 2)` → `de` |
| `split_part` ( *`string`* `text`, *`delimiter`* `text`, *`n`* `integer` ) → `text`<br>Splits *`string`* at occurrences of *`delimiter`* and returns the *`n`*'th field (counting from one), or when *`n`* is negative, returns the \|*`n`*\|'th-from-last field.<br>`split_part('abc~@~def~@~ghi', '~@~', 2)` → `def`<br>`split_part('abc,def,ghi,jkl', ',', -2)` → `ghi` |
| `starts_with` ( *`string`* `text`, *`prefix`* `text` ) → `boolean`<br>Returns true if *`string`* starts with *`prefix`*.<br>`starts_with('alphabet', 'alph')` → `t` |
| `string_to_array` ( *`string`* `text`, *`delimiter`* `text` [, *`null_string`* `text` ] ) → `text[]`<br>Splits the *`string`* at occurrences of *`delimiter`* and forms the resulting fields into a `text` array. If *`delimiter`* is `NULL`, each character in the *`string`* will become a separate element in the array. If *`delimiter`* is an empty string, then the *`string`* is treated as a single field. If *`null_string`* is supplied and is not `NULL`, fields matching that string are replaced by `NULL`. See also `array_to_string`.<br>`string_to_array('xx~~yy~~zz', '~~', 'yy')` → `{xx,NULL,zz}` |
| `string_to_table` ( *`string`* `text`, *`delimiter`* `text` [, *`null_string`* `text` ] ) → `setof text`<br>Splits the *`string`* at occurrences of *`delimiter`* and returns the resulting fields as a set of `text` rows. If *`delimiter`* is `NULL`, each character in the *`string`* will become a separate row of the result. If *`delimiter`* is an empty string, then the *`string`* is treated as a single field. If *`null_string`* is supplied and is not `NULL`, fields matching that string are replaced by `NULL`.<br>`string_to_table('xx~^~yy~^~zz', '~^~', 'yy')` →<br>`xx`<br>`NULL`<br>`zz` |
| `strpos` ( *`string`* `text`, *`substring`* `text` ) → `integer`<br>Returns first starting index of the specified *`substring`* within *`string`*, or zero if it's not present. (Same as `position(`*`substring`*` in `*`string`*`)`, but note the reversed argument order.)<br>`strpos('high', 'ig')` → `2` |
| `substr` ( *`string`* `text`, *`start`* `integer` [, *`count`* `integer` ] ) → `text`<br>Extracts the substring of *`string`* starting at the *`start`*'th character, and extending for *`count`* characters if that is specified. (Same as `substring(`*`string`*` from `*`start`*` for `*`count`*`)`.)<br>`substr('alphabet', 3)` → `phabet` |

| Function/Operator<br>Description<br>Example(s) |
|---|
| substr('alphabet', 3, 2) → ph |
| to_ascii ( *string* text ) → text<br>to_ascii ( *string* text, *encoding* name ) → text<br>to_ascii ( *string* text, *encoding* integer ) → text<br>Converts *string* to ASCII from another encoding, which may be identified by name or number. If *encoding* is omitted the database encoding is assumed (which in practice is the only useful case). The conversion consists primarily of dropping accents. Conversion is only supported from LATIN1, LATIN2, LATIN9, and WIN1250 encodings. (See the unaccent module for another, more flexible solution.)<br>to_ascii('Karél') → Karel |
| to_bin ( integer ) → text<br>to_bin ( bigint ) → text<br>Converts the number to its equivalent two's complement binary representation.<br>to_bin(2147483647) → 1111111111111111111111111111111<br>to_bin(-1234) → 11111111111111111111101100101110 |
| to_hex ( integer ) → text<br>to_hex ( bigint ) → text<br>Converts the number to its equivalent two's complement hexadecimal representation.<br>to_hex(2147483647) → 7fffffff<br>to_hex(-1234) → fffffb2e |
| to_oct ( integer ) → text<br>to_oct ( bigint ) → text<br>Converts the number to its equivalent two's complement octal representation.<br>to_oct(2147483647) → 17777777777<br>to_oct(-1234) → 37777775456 |
| translate ( *string* text, *from* text, *to* text ) → text<br>Replaces each character in *string* that matches a character in the *from* set with the corresponding character in the *to* set. If *from* is longer than *to*, occurrences of the extra characters in *from* are deleted.<br>translate('12345', '143', 'ax') → a2x5 |
| unistr ( text ) → text<br>Evaluate escaped Unicode characters in the argument. Unicode characters can be specified as *XXXX* (4 hexadecimal digits), \\+*XXXXXX* (6 hexadecimal digits), \\u*XXXX* (4 hexadecimal digits), or \\U*XXXXXXXX* (8 hexadecimal digits). To specify a backslash, write two backslashes. All other characters are taken literally.<br>If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.<br>This function provides a (non-standard) alternative to string constants with Unicode escapes (see Section 4.1.2.3).<br>unistr('d\0061t\+000061') → data<br>unistr('d\u0061t\U00000061') → data |

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 36.5.6). 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 9.21, and the functions for converting between strings and the `bytea` type in Table 9.13.

## 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 41.1.

# 9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating binary strings, that is values of type `bytea`. Many of these are equivalent, in purpose and syntax, to the text-string functions described in the previous section.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9.11. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9.12).

### Table 9.11. SQL Binary String Functions and Operators

| Function/Operator<br>Description<br>Example(s) |
|---|
| `bytea` `\|\|` `bytea` → `bytea`<br>Concatenates the two binary strings.<br>`'\x123456'::bytea \|\| '\x789a00bcde'::bytea` → `\x123456789a00bcde` |
| `bit_length` ( `bytea` ) → `integer`<br>Returns number of bits in the binary string (8 times the `octet_length`).<br>`bit_length('\x123456'::bytea)` → `24` |
| `btrim` ( *`bytes`* `bytea`, *`bytesremoved`* `bytea` ) → `bytea`<br>Removes the longest string containing only bytes appearing in *`bytesremoved`* from the start and end of *`bytes`*.<br>`btrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678` |
| `ltrim` ( *`bytes`* `bytea`, *`bytesremoved`* `bytea` ) → `bytea`<br>Removes the longest string containing only bytes appearing in *`bytesremoved`* from the start of *`bytes`*.<br>`ltrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x34567890` |
| `octet_length` ( `bytea` ) → `integer`<br>Returns number of bytes in the binary string.<br>`octet_length('\x123456'::bytea)` → `3` |
| `overlay` ( *`bytes`* `bytea` `PLACING` *`newsubstring`* `bytea` `FROM` *`start`* `integer` [ `FOR` *`count`* `integer` ] ) → `bytea`<br>Replaces the substring of *`bytes`* that starts at the *`start`*'th byte and extends for *`count`* bytes with *`newsubstring`*. If *`count`* is omitted, it defaults to the length of *`newsubstring`*.<br>`overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3)` → `\x12020390` |
| `position` ( *`substring`* `bytea` `IN` *`bytes`* `bytea` ) → `integer`<br>Returns first starting index of the specified *`substring`* within *`bytes`*, or zero if it's not present.<br>`position('\x5678'::bytea in '\x1234567890'::bytea)` → `3` |
| `rtrim` ( *`bytes`* `bytea`, *`bytesremoved`* `bytea` ) → `bytea`<br>Removes the longest string containing only bytes appearing in *`bytesremoved`* from the end of *`bytes`*.<br>`rtrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x12345678` |

| Function/Operator<br>Description<br>Example(s) |
|---|
| `substring` ( *`bytes`* `bytea` [ `FROM` *`start`* `integer` ] [ `FOR` *`count`* `integer` ] ) → `bytea`<br>Extracts the substring of *`bytes`* starting at the *`start`*'th byte if that is specified, and stopping after *`count`* bytes if that is specified. Provide at least one of *`start`* and *`count`*.<br>`substring('\x1234567890'::bytea from 3 for 2)` → `\x5678` |
| `trim` ( [ `LEADING` \| `TRAILING` \| `BOTH` ] *`bytesremoved`* `bytea` `FROM` *`bytes`* `bytea` ) → `bytea`<br>Removes the longest string containing only bytes appearing in *`bytesremoved`* from the start, end, or both ends (`BOTH` is the default) of *`bytes`*.<br>`trim('\x9012'::bytea from '\x1234567890'::bytea)` → `\x345678` |
| `trim` ( [ `LEADING` \| `TRAILING` \| `BOTH` ] [ `FROM` ] *`bytes`* `bytea`, *`bytesremoved`* `bytea` ) → `bytea`<br>This is a non-standard syntax for `trim()`.<br>`trim(both from '\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678` |

Additional binary string manipulation functions are available and are listed in Table 9.12. Some of them are used internally to implement the SQL-standard string functions listed in Table 9.11.

### Table 9.12. Other Binary String Functions

| Function<br>Description<br>Example(s) |
|---|
| `bit_count` ( *`bytes`* `bytea` ) → `bigint`<br>Returns the number of bits set in the binary string (also known as "popcount").<br>`bit_count('\x1234567890'::bytea)` → `15` |
| `crc32` ( `bytea` ) → `bigint`<br>Computes the CRC-32 value of the binary string.<br>`crc32('abc'::bytea)` → `891568578` |
| `crc32c` ( `bytea` ) → `bigint`<br>Computes the CRC-32C value of the binary string.<br>`crc32c('abc'::bytea)` → `910901175` |
| `get_bit` ( *`bytes`* `bytea`, *`n`* `bigint` ) → `integer`<br>Extracts n'th bit from binary string.<br>`get_bit('\x1234567890'::bytea, 30)` → `1` |
| `get_byte` ( *`bytes`* `bytea`, *`n`* `integer` ) → `integer`<br>Extracts n'th byte from binary string.<br>`get_byte('\x1234567890'::bytea, 4)` → `144` |
| `length` ( `bytea` ) → `integer`<br>Returns the number of bytes in the binary string.<br>`length('\x1234567890'::bytea)` → `5` |
| `length` ( *`bytes`* `bytea`, *`encoding`* `name` ) → `integer`<br>Returns the number of characters in the binary string, assuming that it is text in the given *`encoding`*.<br>`length('jose'::bytea, 'UTF8')` → `4` |
| `md5` ( `bytea` ) → `text`<br>Computes the MD5 hash of the binary string, with the result written in hexadecimal. |

| Function<br>Description<br>Example(s) |
|---|
| `md5('Th\000omas'::bytea)` → `8ab2d3c9689aaf18b4958c334c82d8b1` |
| `reverse` ( `bytea` ) → `bytea`<br>Reverses the order of the bytes in the binary string.<br>`reverse('\xabcd'::bytea)` → `\xcdab` |
| `set_bit` ( *bytes* `bytea`, *n* `bigint`, *newvalue* `integer` ) → `bytea`<br>Sets n'th bit in binary string to *newvalue*.<br>`set_bit('\x1234567890'::bytea, 30, 0)` → `\x1234563890` |
| `set_byte` ( *bytes* `bytea`, *n* `integer`, *newvalue* `integer` ) → `bytea`<br>Sets n'th byte in binary string to *newvalue*.<br>`set_byte('\x1234567890'::bytea, 4, 64)` → `\x1234567840` |
| `sha224` ( `bytea` ) → `bytea`<br>Computes the SHA-224 hash of the binary string.<br>`sha224('abc'::bytea)` → `\x23097d223405d8228642a477bda255b32aadbce4b-da0b3f7e36c9da7` |
| `sha256` ( `bytea` ) → `bytea`<br>Computes the SHA-256 hash of the binary string.<br>`sha256('abc'::bytea)` → `\xba7816bf8f01cfea414140de5dae2223 b00361a396177a9cb410ff61f20015ad` |
| `sha384` ( `bytea` ) → `bytea`<br>Computes the SHA-384 hash of the binary string.<br>`sha384('abc'::bytea)` → `\xcb00753f45a35e8bb5a03d699ac65007 272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7` |
| `sha512` ( `bytea` ) → `bytea`<br>Computes the SHA-512 hash of the binary string.<br>`sha512('abc'::bytea)` → `\xddaf35a193617abacc417349ae204131 12e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd 454d4423643ce80e2a9ac94fa54ca49f` |
| `substr` ( *bytes* `bytea`, *start* `integer` [, *count* `integer` ] ) → `bytea`<br>Extracts the substring of *bytes* starting at the *start*'th byte, and extending for *count* bytes if that is specified. (Same as `substring(`*bytes* `from` *start* `for` *count*`)`.)<br>`substr('\x1234567890'::bytea, 3, 2)` → `\x5678` |

Functions `get_byte` and `set_byte` number the first byte of a binary string as byte 0. Functions `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.

For historical reasons, the function `md5` returns a hex-encoded value of type `text` whereas the SHA-2 functions return type `bytea`. Use the functions `encode` and `decode` to convert between the two. For example write `encode(sha256('abc'), 'hex')` to get a hex-encoded text representation, or `decode(md5('abc'), 'hex')` to get a `bytea` value.

Functions for converting strings between different character sets (encodings), and for representing arbitrary binary data in textual form, are shown in Table 9.13. For these functions, an argument or result of type `text` is expressed in the database's default encoding, while arguments or results of type `bytea` are in an encoding named by another argument.

**Table 9.13. Text/Binary String Conversion Functions**

| Function<br>Description<br>Example(s) |
|---|
| `convert` ( *`bytes`* `bytea`, *`src_encoding`* `name`, *`dest_encoding`* `name` ) → `bytea`<br>Converts a binary string representing text in encoding *`src_encoding`* to a binary string in encoding *`dest_encoding`* (see Section 23.3.4 for available conversions).<br>`convert('text_in_utf8', 'UTF8', 'LATIN1')` → `\x746578745f696e5f75746638` |
| `convert_from` ( *`bytes`* `bytea`, *`src_encoding`* `name` ) → `text`<br>Converts a binary string representing text in encoding *`src_encoding`* to `text` in the database encoding (see Section 23.3.4 for available conversions).<br>`convert_from('text_in_utf8', 'UTF8')` → `text_in_utf8` |
| `convert_to` ( *`string`* `text`, *`dest_encoding`* `name` ) → `bytea`<br>Converts a `text` string (in the database encoding) to a binary string encoded in encoding *`dest_encoding`* (see Section 23.3.4 for available conversions).<br>`convert_to('some_text', 'UTF8')` → `\x736f6d655f74657874` |
| `encode` ( *`bytes`* `bytea`, *`format`* `text` ) → `text`<br>Encodes binary data into a textual representation; supported *`format`* values are: `base64`, `escape`, `hex`.<br>`encode('123\000\001', 'base64')` → `MTIzAAE=` |
| `decode` ( *`string`* `text`, *`format`* `text` ) → `bytea`<br>Decodes binary data from a textual representation; supported *`format`* values are the same as for `encode`.<br>`decode('MTIzAAE=', 'base64')` → `\x3132330001` |

The `encode` and `decode` functions support the following textual formats:

base64

The `base64` format is that of RFC 2045 Section 6.8[1]. As per the RFC, encoded lines are broken at 76 characters. However instead of the MIME CRLF end-of-line marker, only a newline is used for end-of-line. The `decode` function ignores carriage-return, newline, space, and tab characters. Otherwise, an error is raised when `decode` is supplied invalid base64 data — including when trailing padding is incorrect.

escape

The `escape` format converts zero bytes and bytes with the high bit set into octal escape sequences (`\`*`nnn`*), and it doubles backslashes. Other byte values are represented literally. The `decode` function will raise an error if a backslash is not followed by either a second backslash or three octal digits; it accepts other byte values unchanged.

hex

The `hex` format represents each 4 bits of data as one hexadecimal digit, `0` through `f`, writing the higher-order digit of each byte first. The `encode` function outputs the `a`-`f` hex digits in lower case. Because the smallest unit of data is 8 bits, there are always an even number of characters returned by `encode`. The `decode` function accepts the `a`-`f` characters in either upper or lower case. An error is raised when `decode` is given invalid hex data — including when given an odd number of characters.

[1] https://datatracker.ietf.org/doc/html/rfc2045#section-6.8

In addition, it is possible to cast integral values to and from type `bytea`. Casting an integer to `bytea` produces 2, 4, or 8 bytes, depending on the width of the integer type. The result is the two's complement representation of the integer, with the most significant byte first. Some examples:

```
1234::smallint::bytea          \x04d2
cast(1234 as bytea)            \x000004d2
cast(-1234 as bytea)           \xfffffb2e
'\x8000'::bytea::smallint      -32768
'\x8000'::bytea::integer       32768
```

Casting a `bytea` to an integer will raise an error if the length of the `bytea` exceeds the width of the integer type.

See also the aggregate function `string_agg` in Section 9.21 and the large object functions in Section 33.4.

# 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`. (While only type `bit` is mentioned in these tables, values of type `bit varying` can be used interchangeably.) Bit strings support the usual comparison operators shown in Table 9.1, as well as the operators shown in Table 9.14.

### Table 9.14. Bit String Operators

| Operator<br>Description<br>Example(s) |
|---|
| `bit || bit` → `bit`<br>Concatenation<br>`B'10001' || B'011'` → `10001011` |
| `bit & bit` → `bit`<br>Bitwise AND (inputs must be of equal length)<br>`B'10001' & B'01101'` → `00001` |
| `bit | bit` → `bit`<br>Bitwise OR (inputs must be of equal length)<br>`B'10001' | B'01101'` → `11101` |
| `bit # bit` → `bit`<br>Bitwise exclusive OR (inputs must be of equal length)<br>`B'10001' # B'01101'` → `11100` |
| `~ bit` → `bit`<br>Bitwise NOT<br>`~ B'10001'` → `01110` |
| `bit << integer` → `bit`<br>Bitwise shift left (string length is preserved)<br>`B'10001' << 3` → `01000` |
| `bit >> integer` → `bit`<br>Bitwise shift right (string length is preserved)<br>`B'10001' >> 2` → `00100` |

Some of the functions available for binary strings are also available for bit strings, as shown in Table 9.15.

### Table 9.15. Bit String Functions

| Function<br>Description<br>Example(s) |
|---|
| `bit_count ( bit ) → bigint`<br>Returns the number of bits set in the bit string (also known as "popcount").<br>`bit_count(B'10111')` → `4` |
| `bit_length ( bit ) → integer`<br>Returns number of bits in the bit string.<br>`bit_length(B'10111')` → `5` |
| `length ( bit ) → integer`<br>Returns number of bits in the bit string.<br>`length(B'10111')` → `5` |
| `octet_length ( bit ) → integer`<br>Returns number of bytes in the bit string.<br>`octet_length(B'1011111011')` → `2` |
| `overlay ( bits bit PLACING newsubstring bit FROM start integer [ FOR count integer ] ) → bit`<br>Replaces the substring of *bits* that starts at the *start*'th bit and extends for *count* bits with *newsubstring*. If *count* is omitted, it defaults to the length of *newsubstring*.<br>`overlay(B'01010101010101010' placing B'11111' from 2 for 3)` → `01111101010101010` |
| `position ( substring bit IN bits bit ) → integer`<br>Returns first starting index of the specified *substring* within *bits*, or zero if it's not present.<br>`position(B'010' in B'000001101011')` → `8` |
| `substring ( bits bit [ FROM start integer ] [ FOR count integer ] ) → bit`<br>Extracts the substring of *bits* starting at the *start*'th bit if that is specified, and stopping after *count* bits if that is specified. Provide at least one of *start* and *count*.<br>`substring(B'110010111111' from 3 for 2)` → `00` |
| `get_bit ( bits bit, n integer ) → integer`<br>Extracts *n*'th bit from bit string; the first (leftmost) bit is bit 0.<br>`get_bit(B'101010101010101010', 6)` → `1` |
| `set_bit ( bits bit, n integer, newvalue integer ) → bit`<br>Sets *n*'th bit in bit string to *newvalue*; the first (leftmost) bit is bit 0.<br>`set_bit(B'101010101010101010', 6, 0)` → `101010001010101010` |

In addition, it is possible to cast integral values to and from type `bit`. 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. 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.

# 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 TO` provides 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.

`SIMILAR TO` and POSIX-style regular expressions do not support nondeterministic collations. If required, use `LIKE` or apply a different collation to the expression to work around this limitation.

## 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 supports nondeterministic collations (see Section 23.2.2.4), such as case-insensitive collations or collations that, say, ignore punctuation. So with a case-insensitive collation, one could have:

```
'AbC' LIKE 'abc' COLLATE case_insensitive    true
'AbC' LIKE 'a%' COLLATE case_insensitive     true
```

With collations that ignore certain characters or in general that consider strings of different lengths equal, the semantics can become a bit more complicated. Consider these examples:

```
'.foo.' LIKE 'foo' COLLATE ign_punct    true
'.foo.' LIKE 'f_o' COLLATE ign_punct    true
'.foo.' LIKE '_oo' COLLATE ign_punct    false
```

The way the matching works is that the pattern is partitioned into sequences of wildcards and non-wildcard strings (wildcards being _ and %). For example, the pattern f_o is partitioned into f, _, o, the pattern _oo is partitioned into _, oo. The input string matches the pattern if it can be partitioned in such a way that the wildcards match one character or any number of characters respectively and the non-wildcard partitions are equal under the applicable collation. So for example, '.foo.' LIKE 'f_o' COLLATE ign_punct is true because one can partition .foo. into .f, o, o., and then '.f' = 'f' COLLATE ign_punct, 'o' matches the _ wildcard, and 'o.' = 'o' COLLATE ign_punct. But '.foo.' LIKE '_oo' COLLATE ign_punct is false because .foo. cannot be partitioned in a way that the first character is any character and the rest of the string compares equal to oo. (Note that the single-character wildcard always matches exactly one character, independent of the collation. So in this example, the _ would match ., but then the rest of the input string won't match the rest of the pattern.)

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 turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 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.

According to the SQL standard, omitting ESCAPE means there is no escape character (rather than defaulting to a backslash), and a zero-length ESCAPE value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. (But this does not support nondeterministic collations.) 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. You may see these operator names in EXPLAIN output and similar places, since the parser actually translates LIKE et al. to these operators.

The phrases LIKE, ILIKE, NOT LIKE, and NOT ILIKE are generally treated as operators in PostgreSQL syntax; for example they can be used in *expression* *operator* ANY (*subquery*) constructs, although an ESCAPE clause cannot be included there. In some obscure cases it may be necessary to use the underlying operator names instead.

Also see the starts-with operator ^@ and the corresponding starts_with() function, which are useful in cases where simply matching the beginning of a string is needed.

## 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 (POSIX) 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. A different escape character can be specified with `ESCAPE`, or the escape capability can be disabled by writing `ESCAPE ''`.

According to the SQL standard, omitting `ESCAPE` means there is no escape character (rather than defaulting to a backslash), and a zero-length `ESCAPE` value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

Another nonstandard extension is that following the escape character with a letter or digit provides access to the escape sequences defined for POSIX regular expressions; see Table 9.20, Table 9.21, and Table 9.22 below.

Some examples:

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

The `substring` function with three parameters provides extraction of a substring that matches an SQL regular expression pattern. The function can be written according to standard SQL syntax:

```
substring(string similar pattern escape escape-character)
```

or using the now obsolete SQL:1999 syntax:

```
substring(string from pattern for escape-character)
```

or as a plain three-argument function:

```
substring(string, pattern, escape-character)
```

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 for which the matching data sub-string is of interest, the pattern should contain two occurrences of the escape character followed by a double quote (`"`). The text matching the portion of the pattern between these separators is returned when the match is successful.

The escape-double-quote separators actually divide `substring`'s pattern into three independent regular expressions; for example, a vertical bar (`|`) in any of the three sections affects only that section. Also, the first and third of these regular expressions are defined to match the smallest possible amount of text, not the largest, when there is any ambiguity about how much of the data string matches which pattern. (In POSIX parlance, the first and third regular expressions are forced to be non-greedy.)

As an extension to the SQL standard, PostgreSQL allows there to be just one escape-double-quote separator, in which case the third regular expression is taken as empty; or no separators, in which case the first and third regular expressions are taken as empty.

Some examples, with `#"` delimiting the return string:

```
substring('foobar' similar '%#"o_b#"%' escape '#')   oob
substring('foobar' similar '#"o_b#"%' escape '#')    NULL
```

## 9.7.3. POSIX Regular Expressions

Table 9.16 lists the available operators for pattern matching using POSIX regular expressions.

### Table 9.16. Regular Expression Match Operators

| Operator<br>Description<br>Example(s) |
|---|
| `text ~ text → boolean`<br>String matches regular expression, case sensitively<br>`'thomas' ~ 't.*ma' → t` |
| `text ~* text → boolean`<br>String matches regular expression, case-insensitively<br>`'thomas' ~* 'T.*ma' → t` |
| `text !~ text → boolean`<br>String does not match regular expression, case sensitively<br>`'thomas' !~ 't.*max' → t` |
| `text !~* text → boolean`<br>String does not match regular expression, case-insensitively<br>`'thomas' !~* 'T.*ma' → f` |

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:

```
'abcd' ~ 'bc'     true
'abcd' ~ 'a.c'    true — dot matches any character
'abcd' ~ 'a.*d'   true — * repeats the preceding pattern item
'abcd' ~ '(b|x)'  true — | means OR, parentheses group
'abcd' ~ '^a'     true — ^ anchors to start of string
'abcd' ~ '^(b|c)' false — would match except for anchoring
```

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 first 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_count` function counts the number of places where a POSIX regular expression pattern matches a string. It has the syntax `regexp_count`(*`string`*, *`pattern`* [, *`start`* [, *`flags`* ]]). *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. For example, including `i` in *`flags`* specifies case-insensitive matching. Supported flags are described in Table 9.24.

Some examples:

```
regexp_count('ABCABCAXYaxy', 'A.')          3
regexp_count('ABCABCAXYaxy', 'A.', 1, 'i')  4
```

The `regexp_instr` function returns the starting or ending position of the *`N`*'th match of a POSIX regular expression pattern to a string, or zero if there is no such match. It has the syntax `regexp_instr`(*`string`*, *`pattern`* [, *`start`* [, *`N`* [, *`endoption`* [, *`flags`* [, *`subexpr`* ]]]]]). *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. If *`N`* is specified then the *`N`*'th match of the pattern is located, otherwise the first match is located. If the *`endoption`* parameter is omitted or specified as zero, the function returns the position of the first character of the match. Otherwise, *`endoption`* must be one, and the function returns the position of the character following the match. 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 9.24. For a pattern containing parenthesized subexpressions, *`subexpr`* is an integer indicating which subexpression is of interest: the result identifies the position of the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When *`subexpr`* is omitted or zero, the result identifies the position of the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_instr('number of your street, town zip, FR', '[^,]+', 1, 2)
```

```
                                    23
regexp_instr(string=>'ABCDEFGHI', pattern=>'(c..)(...)', start=>1, "N"=>1,
 endoption=>0, flags=>'i', subexpr=>2)
                                    6
```

The `regexp_like` function checks whether a match of a POSIX regular expression pattern occurs within a string, returning boolean true or false. It has the syntax `regexp_like(`*`string`*, *`pattern`* [, *`flags`* ]). 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 9.24. This function has the same results as the ~ operator if no flags are specified. If only the `i` flag is specified, it has the same results as the `~*` operator.

Some examples:

```
regexp_like('Hello World', 'world')       false
regexp_like('Hello World', 'world', 'i')  true
```

The `regexp_match` function returns a text array of matching substring(s) within 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 9.24.

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)
```

## Tip

In the common case where you just want the whole matching substring or `NULL` for no match, the best solution is to use `regexp_substr()`. However, `regexp_substr()` only exists in PostgreSQL version 15 and up. When working in older versions, you can extract the first element of `regexp_match()`'s result, for example:

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

The `regexp_matches` function returns a set of text arrays of matching substring(s) within matches of 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 9.24, 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_replace` function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax `regexp_replace`(*string*, *pattern*, *replacement* [, *flags* ]) or `regexp_replace`(*string*, *pattern*, *replacement*, *start* [, *N* [, *flags* ]]). The source *string* is returned unchanged if there is no match to the *pattern*. If there is a match, the *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. *pattern* is searched for in *string*, normally from the beginning of the string, but if the *start* parameter is provided then beginning from that character index. By default, only the first match of the pattern is replaced. If *N* is specified and is greater than zero, then the *N*'th match of the pattern is replaced. If the `g` flag is given, or if *N* is specified and is zero, then all matches at or after the *start* position are replaced. (The `g` flag is ignored when *N* is specified.) The *flags* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags (though not `g`) are described in Table 9.24.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

```
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 0, 'i')
                                   X PXstgrXSQL fXnctXXn
regexp_replace(string=>'A PostgreSQL function', pattern=>'a|e|i|o|u',
 replacement=>'X', start=>1, "N"=>3, flags=>'i')
                                   A PostgrXSQL function
```

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 9.24.

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 the other regexp functions, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

The `regexp_substr` function returns the substring that matches a POSIX regular expression pattern, or `NULL` if there is no match. It has the syntax `regexp_substr(`*`string`*, *`pattern`* [, *`start`* [, *`N`* [, *`flags`* [, *`subexpr`* ]]]]). *`pattern`* is searched for in *`string`*, normally from the beginning of the string, but if the *`start`* parameter is provided then beginning from that character index. If *`N`* is specified then the *`N`*'th match of the pattern is returned, otherwise the first match is returned. 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 9.24. For a pattern containing parenthesized subexpressions, *`subexpr`* is an integer indicating which subexpression is of interest: the result is the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When *`subexpr`* is omitted or zero, the result is the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_substr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    town zip
regexp_substr('ABCDEFGHI', '(c..)(...)', 1, 1, 'i', 2)
                                   FGH
```

## 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. 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 9.17. The possible quantifiers and their meanings are shown in Table 9.18.

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 9.19; some more constraints are described later.

### Table 9.17. 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 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 (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 turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

### Table 9.18. 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,}` |

| Quantifier | Matches |
|---|---|
| `{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 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 9.19. 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), 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. A character class cannot be used as an endpoint of a range. The POSIX standard defines these character class names: `alnum` (letters and numeric digits), `alpha` (letters), `blank` (space and tab), `cntrl` (control characters), `digit` (numeric digits), `graph` (printable characters except space), `lower` (lower-case letters), `print` (printable characters including space), `punct` (punctuation), `space` (any white space), `upper` (upper-case letters), and `xdigit` (hexadecimal digits). The behavior of these standard character classes is generally consistent across platforms for characters in the 7-bit ASCII set. Whether a given non-ASCII character is considered to belong to one of these classes depends on the *collation* that is used for the regular-expression function or operator (see Section 23.2), or by default on the database's `LC_CTYPE` locale setting (see Section 23.1). The classification of non-ASCII characters can vary across platforms even in similarly-named locales. (But the `C` locale never considers any non-ASCII characters to belong to any of these classes.) In addition to these standard character classes, PostgreSQL defines the `word` character class, which is the same as `alnum` plus the underscore (_) character, and the `ascii` character class, which contains exactly the 7-bit ASCII set.

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 any character belonging to the `word` character class, that is, any letter, digit, or 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 9.20.

*Class-shorthand escapes* provide shorthands for certain commonly-used character classes. They are shown in Table 9.21.

A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9.22.

A *back reference* (`\n`) matches the same string matched by the previous parenthesized subexpression specified by the number *n* (see Table 9.23). 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. The back reference considers only the string characters matched by the referenced subexpression, not any constraints contained in it. For example, `(^\d)\1` will match `22`.

### Table 9.20. 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 9.21. Regular Expression Class-Shorthand Escapes

| Escape | Description |
|---|---|
| `\d` | matches any digit, like `[[:digit:]]` |
| `\s` | matches any whitespace character, like `[[:space:]]` |
| `\w` | matches any word character, like `[[:word:]]` |

| Escape | Description |
|---|---|
| \D | matches any non-digit, like [^[:digit:]] |
| \S | matches any non-whitespace character, like [^[:space:]] |
| \W | matches any non-word character, like [^[:word:]] |

The class-shorthand escapes also work within bracket expressions, although the definitions shown above are not quite syntactically valid in that context. For example, [a-c\d] is equivalent to [a-c[:digit:]].

**Table 9.22. Regular Expression Constraint Escapes**

| Escape | Description |
|---|---|
| \A | matches only at the beginning of the string (see Section 9.7.3.5 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 for how this differs from $) |

A word is defined as in the specification of [[:<:]] and [[:>:]] above. Constraint escapes are illegal within bracket expressions.

**Table 9.23. 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 9.24. Note that these same option letters are used in the *`flags`* parameters of regex functions.

**Table 9.24. 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) (overrides operator type) |
| `m` | historical synonym for `n` |
| `n` | newline-sensitive matching (see Section 9.7.3.5) |
| `p` | partial newline-sensitive matching (see Section 9.7.3.5) |
| `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) |
| `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 not cross lines unless the RE explicitly includes a newline) 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*. Also, the character class shorthands `\D` and `\W` will match a newline regardless of this mode. (Before PostgreSQL 14, they did not match newlines when in newline-sensitive mode. Write `[^[:digit:]]` or `[^[:word:]]` to get the old behavior.)

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.

## 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.

## 9.7.3.8. Differences from SQL Standard and XQuery

Since SQL:2008, the SQL standard includes regular expression operators and functions that performs pattern matching according to the XQuery regular expression standard:

- `LIKE_REGEX`
- `OCCURRENCES_REGEX`
- `POSITION_REGEX`
- `SUBSTRING_REGEX`
- `TRANSLATE_REGEX`

PostgreSQL does not currently implement these operators and functions. You can get approximately equivalent functionality in each case as shown in Table 9.25. (Various optional clauses on both sides have been omitted in this table.)

### Table 9.25. Regular Expression Functions Equivalencies

| SQL standard | PostgreSQL |
|---|---|
| *string* LIKE_REGEX *pattern* | regexp_like(*string*, *pattern*) or *string* ~ *pattern* |
| OCCURRENCES_REGEX(*pattern* IN *string*) | regexp_count(*string*, *pattern*) |
| POSITION_REGEX(*pattern* IN *string*) | regexp_instr(*string*, *pattern*) |
| SUBSTRING_REGEX(*pattern* IN *string*) | regexp_substr(*string*, *pattern*) |
| TRANSLATE_REGEX(*pattern* IN *string* WITH *replacement*) | regexp_replace(*string*, *pattern*, *replacement*) |

Regular expression functions similar to those provided by PostgreSQL are also available in a number of other SQL implementations, whereas the SQL-standard functions are not as widely implemented. Some of the details of the regular expression syntax will likely differ in each implementation.

The SQL-standard operators and functions use XQuery regular expressions, which are quite close to the ARE syntax described above. Notable differences between the existing POSIX-based regular-expression feature and XQuery regular expressions include:

- XQuery character class subtraction is not supported. An example of this feature is using the following to match only English consonants: `[a-z-[aeiou]]`.
- XQuery character class shorthands `\c`, `\C`, `\i`, and `\I` are not supported.
- XQuery character class elements using `\p{UnicodeProperty}` or the inverse `\P{UnicodeProperty}` are not supported.
- POSIX interprets character classes such as `\w` (see Table 9.21) according to the prevailing locale (which you can control by attaching a `COLLATE` clause to the operator or function). XQuery specifies these classes by reference to Unicode character properties, so equivalent behavior is obtained only with a locale that follows the Unicode rules.
- The SQL standard (not XQuery itself) attempts to cater for more variants of "newline" than POSIX does. The newline-sensitive matching options described above consider only ASCII NL (`\n`) to be a newline, but SQL would have us treat CR (`\r`), CRLF (`\r\n`) (a Windows-style newline), and some Unicode-only characters like LINE SEPARATOR (U+2028) as newlines as well. Notably, `.` and `\s` should count `\r\n` as one character not two according to SQL.
- Of the character-entry escapes described in Table 9.20, XQuery supports only `\n`, `\r`, and `\t`.

- XQuery does not support the `[:name:]` syntax for character classes within bracket expressions.
- XQuery does not have lookahead or lookbehind constraints, nor any of the constraint escapes described in Table 9.22.
- The metasyntax forms described in Section 9.7.3.4 do not exist in XQuery.
- The regular expression flag letters defined by XQuery are related to but not the same as the option letters for POSIX (Table 9.24). While the `i` and `q` options behave the same, others do not:
  - XQuery's `s` (allow dot to match newline) and `m` (allow `^` and `$` to match at newlines) flags provide access to the same behaviors as POSIX's `n`, `p` and `w` flags, but they do *not* match the behavior of POSIX's `s` and `m` flags. Note in particular that dot-matches-newline is the default behavior in POSIX but not XQuery.
  - XQuery's `x` (ignore whitespace in pattern) flag is noticeably different from POSIX's expanded-mode flag. POSIX's `x` flag also allows `#` to begin a comment in the pattern, and POSIX will not ignore a whitespace character after a backslash.

# 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 9.26 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 9.26. Formatting Functions

| Function<br>Description<br>Example(s) |
|---|
| `to_char` ( `timestamp`, `text` ) → `text`<br>`to_char` ( `timestamp with time zone`, `text` ) → `text`<br>Converts time stamp to string according to the given format.<br>`to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS')` → `05:31:12` |
| `to_char` ( `interval`, `text` ) → `text`<br>Converts interval to string according to the given format.<br>`to_char(interval '15h 2m 12s', 'HH24:MI:SS')` → `15:02:12` |
| `to_char` ( *`numeric_type`*, `text` ) → `text`<br>Converts number to string according to the given format; available for `integer`, `bigint`, `numeric`, `real`, `double precision`.<br>`to_char(125, '999')` → `125`<br>`to_char(125.8::real, '999D9')` → `125.8`<br>`to_char(-125.8, '999D99S')` → `125.80-` |
| `to_date` ( `text`, `text` ) → `date`<br>Converts string to date according to the given format.<br>`to_date('05 Dec 2000', 'DD Mon YYYY')` → `2000-12-05` |
| `to_number` ( `text`, `text` ) → `numeric`<br>Converts string to numeric according to the given format.<br>`to_number('12,454.8-', '99G999D9S')` → `-12454.8` |
| `to_timestamp` ( `text`, `text` ) → `timestamp with time zone`<br>Converts string to time stamp according to the given format. (See also `to_timestamp(double precision)` in Table 9.33.) |

| Function<br>Description<br>Example(s) |
|---|
| `to_timestamp('05 Dec 2000', 'DD Mon YYYY')` → `2000-12-05 00:00:00-05` |

## 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. If there are characters in the template string that are not template patterns, the corresponding characters in the input data string are simply skipped over (whether or not they are equal to the template string characters).

Table 9.27 shows the template patterns available for formatting date and time values.

### Table 9.27. 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) |
| `FF1` | tenth of second (0–9) |
| `FF2` | hundredth of second (00–99) |
| `FF3` | millisecond (000–999) |
| `FF4` | tenth of a millisecond (0000–9999) |
| `FF5` | hundredth of a millisecond (00000–99999) |
| `FF6` | microsecond (000000–999999) |
| `SSSS`, `SSSSS` | 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 |

| **Pattern** | **Description** |
|---|---|
| `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 Date (integer days since November 24, 4714 BC at local midnight; see Section B.7) |
| `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 |

| Pattern | Description |
|---|---|
| `tz` | lower case time-zone abbreviation |
| `TZH` | time-zone hours |
| `TZM` | time-zone minutes |
| `OF` | time-zone offset from UTC (*`HH`* or *`HH`*`:`*`MM`*) |

Modifiers can be applied to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` modifier. Table 9.28 shows the modifier patterns for date/time formatting.

### Table 9.28. 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 (use localized day and month names based on 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` suppresses trailing blanks whether or not `FM` is specified.

- `to_timestamp` and `to_date` ignore letter case in the input; so for example `MON`, `Mon`, and `mon` all accept the same strings. When using the `TM` modifier, case-folding is done according to the rules of the function's input collation (see Section 23.2).

- `to_timestamp` and `to_date` skip multiple blank spaces at the beginning of the input string and around date and time values unless the `FX` option is used. For example, `to_timestamp(' 2000    JUN', 'YYYY MON')` and `to_timestamp('2000 - JUN', 'YYYY-MON')` work, but `to_timestamp('2000    JUN', 'FXYYYY MON')` returns an error because `to_timestamp` expects only a single space. `FX` must be specified as the first item in the template.

- A separator (a space or non-letter/non-digit character) in the template string of `to_timestamp` and `to_date` matches any single separator in the input string or is skipped, unless the `FX` option is used. For example, `to_timestamp('2000JUN', 'YYYY///MON')` and `to_timestamp('2000/JUN', 'YYYY MON')` work, but `to_timestamp('2000//JUN', 'YYYY/MON')` returns an error because the number of separators in the input string exceeds the number of separators in the template.

  If `FX` is specified, a separator in the template string matches exactly one character in the input string. But note that the input string character is not required to be the same as the separator from the template string. For example, `to_timestamp('2000/JUN', 'FXYYYY MON')` works, but `to_timestamp('2000/JUN', 'FXYYYY  MON')` returns an error because the second space in the template string consumes the letter `J` from the input string.

- A `TZH` template pattern can match a signed number. Without the `FX` option, minus signs may be ambiguous, and could be interpreted as a separator. This ambiguity is resolved as follows: If the number of separators before `TZH` in the template string is less than the number of separators before the minus sign in the input string, the minus sign is interpreted as part of `TZH`. Otherwise, the minus sign is considered to be a separator between values. For example, `to_timestamp('2000 -10', 'YYYY TZH')` matches `-10` to `TZH`, but `to_timestamp('2000 -10', 'YYYY  TZH')` matches `10` to `TZH`.

- 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 template patterns. 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`, literal text and double-quoted strings result in skipping the number of characters contained in the string; for example `"XX"` skips two input characters (whether or not they are `XX`).

> **Tip**
>
> Prior to PostgreSQL 12, it was possible to skip arbitrary text in the input string using non-letter or non-digit characters. For example, `to_timestamp('2000y6m1d', 'yyyy-MM-DD')` used to work. Now you can only use letter characters for this purpose. For example, `to_timestamp('2000y6m1d',  'yyyytMMtDDt')` and `to_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d"')` skip `y`, `m`, and `d`.

- If you want to have a double quote in the output you must precede it with a backslash, for example `'\"YYYY Month\"'`. Backslashes are not otherwise special outside of double-quoted strings. Within a double-quoted string, a backslash causes the next character to be taken literally, whatever it is (but this has no special effect unless the next character is a double quote or another backslash).

- 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`, negative years are treated as signifying BC. If you write both a negative year and an explicit `BC` field, you get AD again. An input of year zero is treated as 1 BC.

- 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('200001130', 'YYYYMMDD')` will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like `to_date('20000-1130', 'YYYY-MMDD')` or `to_date('20000Nov30', '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 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 9.29 shows the template patterns available for formatting numeric values.

### Table 9.29. 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` or `rn` | Roman numeral (values 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.)

- If the format provides fewer fractional digits than the number being formatted, `to_char()` will round the number to the specified number of fractional digits.
- 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 and 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.
- In `to_number`, if non-data template patterns such as `L` or `TH` are used, the corresponding number of input characters are skipped, whether or not they match the template pattern, unless they are data characters (that is, digits, sign, decimal point, or comma). For example, `TH` would skip two non-data characters.
- `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. The `V` can be thought of as marking the position of an implicit decimal point in the input or output string. `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).
- In `to_number()`, the `RN` pattern converts Roman numerals (in standard form) to numbers. Input is case-insensitive, so `RN` and `rn` are equivalent. `RN` cannot be used in combination with any other formatting patterns or modifiers except `FM`, which is applicable only in `to_char()` and is ignored in `to_number()`.

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 9.30 shows the modifier patterns for numeric formatting.

### Table 9.30. 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 9.31 shows some examples of the use of the `to_char` function.

### Table 9.31. `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(current_timestamp AT TIME ZONE 'UTC', 'YYYY-MM-DD"T"HH24:MI:SS"Z"')` | `'2022-12-06T05:39:18Z'`, ISO 8601 extended format |

| Expression | Result |
|---|---|
| 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 9.33 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9.32 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

In addition, the usual comparison operators shown in Table 9.1 are available for the date/time types. Dates and timestamps (with or without time zone) are all comparable, while times (with or without time zone) and intervals can only be compared to other values of the same data type. When comparing a timestamp without time zone to a timestamp with time zone, the former value is assumed to be given in the time zone specified by the TimeZone configuration parameter, and is rotated to UTC for comparison to the latter value (which is already in UTC internally). Similarly, a date value is assumed to represent midnight in the `TimeZone` zone when comparing it to a timestamp.

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 9.32. Date/Time Operators

| Operator<br>Description<br>Example(s) |
|---|
| `date` + `integer` → `date`<br>Add a number of days to a date<br>`date '2001-09-28' + 7` → `2001-10-05` |
| `date` + `interval` → `timestamp`<br>Add an interval to a date<br>`date '2001-09-28' + interval '1 hour'` → `2001-09-28 01:00:00` |
| `date` + `time` → `timestamp`<br>Add a time-of-day to a date<br>`date '2001-09-28' + time '03:00'` → `2001-09-28 03:00:00` |
| `interval` + `interval` → `interval`<br>Add intervals<br>`interval '1 day' + interval '1 hour'` → `1 day 01:00:00` |
| `timestamp` + `interval` → `timestamp`<br>Add an interval to a timestamp<br>`timestamp '2001-09-28 01:00' + interval '23 hours'` → `2001-09-29 00:00:00` |
| `time` + `interval` → `time`<br>Add an interval to a time<br>`time '01:00' + interval '3 hours'` → `04:00:00` |
| - `interval` → `interval`<br>Negate an interval<br>`- interval '23 hours'` → `-23:00:00` |
| `date` - `date` → `integer`<br>Subtract dates, producing the number of days elapsed<br>`date '2001-10-01' - date '2001-09-28'` → `3` |
| `date` - `integer` → `date`<br>Subtract a number of days from a date |

| Operator<br>Description<br>Example(s) |
|---|
| `date '2001-10-01' - 7` → `2001-09-24` |
| `date` - `interval` → `timestamp`<br>Subtract an interval from a date<br>`date '2001-09-28' - interval '1 hour'` → `2001-09-27 23:00:00` |
| `time` - `time` → `interval`<br>Subtract times<br>`time '05:00' - time '03:00'` → `02:00:00` |
| `time` - `interval` → `time`<br>Subtract an interval from a time<br>`time '05:00' - interval '2 hours'` → `03:00:00` |
| `timestamp` - `interval` → `timestamp`<br>Subtract an interval from a timestamp<br>`timestamp '2001-09-28 23:00' - interval '23 hours'` → `2001-09-28 00:00:00` |
| `interval` - `interval` → `interval`<br>Subtract intervals<br>`interval '1 day' - interval '1 hour'` → `1 day -01:00:00` |
| `timestamp` - `timestamp` → `interval`<br>Subtract timestamps (converting 24-hour intervals into days, similarly to `justify_hours()`)<br>`timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00'` → `63 days 15:00:00` |
| `interval` * `double precision` → `interval`<br>Multiply an interval by a scalar<br>`interval '1 second' * 900` → `00:15:00`<br>`interval '1 day' * 21` → `21 days`<br>`interval '1 hour' * 3.5` → `03:30:00` |
| `interval` / `double precision` → `interval`<br>Divide an interval by a scalar<br>`interval '1 hour' / 1.5` → `00:40:00` |

### Table 9.33. Date/Time Functions

| Function<br>Description<br>Example(s) |
|---|
| `age` ( `timestamp`, `timestamp` ) → `interval`<br>Subtract arguments, producing a “symbolic” result that uses years and months, rather than just days<br>`age(timestamp '2001-04-10', timestamp '1957-06-13')` → `43 years 9 mons 27 days` |
| `age` ( `timestamp` ) → `interval`<br>Subtract argument from `current_date` (at midnight)<br>`age(timestamp '1957-06-13')` → `62 years 6 mons 10 days` |

| Function<br>Description<br>Example(s) |
|---|
| `clock_timestamp` ( ) → `timestamp with time zone`<br>Current date and time (changes during statement execution); see Section 9.9.5<br>`clock_timestamp()` → `2019-12-23 14:39:53.662522-05` |
| `current_date` → `date`<br>Current date; see Section 9.9.5<br>`current_date` → `2019-12-23` |
| `current_time` → `time with time zone`<br>Current time of day; see Section 9.9.5<br>`current_time` → `14:39:53.662522-05` |
| `current_time` ( `integer` ) → `time with time zone`<br>Current time of day, with limited precision; see Section 9.9.5<br>`current_time(2)` → `14:39:53.66-05` |
| `current_timestamp` → `timestamp with time zone`<br>Current date and time (start of current transaction); see Section 9.9.5<br>`current_timestamp` → `2019-12-23 14:39:53.662522-05` |
| `current_timestamp` ( `integer` ) → `timestamp with time zone`<br>Current date and time (start of current transaction), with limited precision; see Section 9.9.5<br>`current_timestamp(0)` → `2019-12-23 14:39:53-05` |
| `date_add` ( `timestamp with time zone`, `interval` [, `text` ] ) → `timestamp with time zone`<br>Add an `interval` to a `timestamp with time zone`, computing times of day and daylight-savings adjustments according to the time zone named by the third argument, or the current TimeZone setting if that is omitted. The form with two arguments is equivalent to the `timestamp with time zone` + `interval` operator.<br>`date_add('2021-10-31 00:00:00+02'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-31 23:00:00+00` |
| `date_bin` ( `interval`, `timestamp`, `timestamp` ) → `timestamp`<br>Bin input into specified interval aligned with specified origin; see Section 9.9.3<br>`date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00')` → `2001-02-16 20:35:00` |
| `date_part` ( `text`, `timestamp` ) → `double precision`<br>Get timestamp subfield (equivalent to `extract`); see Section 9.9.1<br>`date_part('hour', timestamp '2001-02-16 20:38:40')` → `20` |
| `date_part` ( `text`, `interval` ) → `double precision`<br>Get interval subfield (equivalent to `extract`); see Section 9.9.1<br>`date_part('month', interval '2 years 3 months')` → `3` |
| `date_subtract` ( `timestamp with time zone`, `interval` [, `text` ] ) → `timestamp with time zone`<br>Subtract an `interval` from a `timestamp with time zone`, computing times of day and daylight-savings adjustments according to the time zone named by the third argument, or the current TimeZone setting if that is omitted. The form with two arguments is equivalent to the `timestamp with time zone` - `interval` operator. |

| Function<br>Description<br>Example(s) |
|---|
| `date_subtract('2021-11-01 00:00:00+01'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-30 22:00:00+00` |
| `date_trunc` ( `text`, `timestamp` ) → `timestamp`<br>Truncate to specified precision; see Section 9.9.2<br>`date_trunc('hour', timestamp '2001-02-16 20:38:40')` → `2001-02-16 20:00:00` |
| `date_trunc` ( `text`, `timestamp with time zone`, `text` ) → `timestamp with time zone`<br>Truncate to specified precision in the specified time zone; see Section 9.9.2<br>`date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')` → `2001-02-16 13:00:00+00` |
| `date_trunc` ( `text`, `interval` ) → `interval`<br>Truncate to specified precision; see Section 9.9.2<br>`date_trunc('hour', interval '2 days 3 hours 40 minutes')` → `2 days 03:00:00` |
| `extract` ( *`field`* `from` `timestamp` ) → `numeric`<br>Get timestamp subfield; see Section 9.9.1<br>`extract(hour from timestamp '2001-02-16 20:38:40')` → `20` |
| `extract` ( *`field`* `from` `interval` ) → `numeric`<br>Get interval subfield; see Section 9.9.1<br>`extract(month from interval '2 years 3 months')` → `3` |
| `isfinite` ( `date` ) → `boolean`<br>Test for finite date (not +/-infinity)<br>`isfinite(date '2001-02-16')` → `true` |
| `isfinite` ( `timestamp` ) → `boolean`<br>Test for finite timestamp (not +/-infinity)<br>`isfinite(timestamp 'infinity')` → `false` |
| `isfinite` ( `interval` ) → `boolean`<br>Test for finite interval (not +/-infinity)<br>`isfinite(interval '4 hours')` → `true` |
| `justify_days` ( `interval` ) → `interval`<br>Adjust interval, converting 30-day time periods to months<br>`justify_days(interval '1 year 65 days')` → `1 year 2 mons 5 days` |
| `justify_hours` ( `interval` ) → `interval`<br>Adjust interval, converting 24-hour time periods to days<br>`justify_hours(interval '50 hours 10 minutes')` → `2 days 02:10:00` |
| `justify_interval` ( `interval` ) → `interval`<br>Adjust interval using `justify_days` and `justify_hours`, with additional sign adjustments<br>`justify_interval(interval '1 mon -1 hour')` → `29 days 23:00:00` |
| `localtime` → `time`<br>Current time of day; see Section 9.9.5<br>`localtime` → `14:39:53.662522` |

| Function<br>Description<br>Example(s) |
|---|
| `localtime` ( `integer` ) → `time`<br>Current time of day, with limited precision; see Section 9.9.5<br>`localtime(0)` → `14:39:53` |
| `localtimestamp` → `timestamp`<br>Current date and time (start of current transaction); see Section 9.9.5<br>`localtimestamp` → `2019-12-23 14:39:53.662522` |
| `localtimestamp` ( `integer` ) → `timestamp`<br>Current date and time (start of current transaction), with limited precision; see Section 9.9.5<br>`localtimestamp(2)` → `2019-12-23 14:39:53.66` |
| `make_date` ( *`year`* `int`, *`month`* `int`, *`day`* `int` ) → `date`<br>Create date from year, month and day fields (negative years signify BC)<br>`make_date(2013, 7, 15)` → `2013-07-15` |
| `make_interval` ( [ *`years`* `int` [, *`months`* `int` [, *`weeks`* `int` [, *`days`* `int` [, *`hours`* `int` [, *`mins`* `int` [, *`secs`* `double precision` ]]]]]]] ) → `interval`<br>Create interval from years, months, weeks, days, hours, minutes and seconds fields, each of which can default to zero<br>`make_interval(days => 10)` → `10 days` |
| `make_time` ( *`hour`* `int`, *`min`* `int`, *`sec`* `double precision` ) → `time`<br>Create time from hour, minute and seconds fields<br>`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`<br>Create timestamp from year, month, day, hour, minute and seconds fields (negative years signify BC)<br>`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`<br>Create timestamp with time zone from year, month, day, hour, minute and seconds fields (negative years signify BC). If *`timezone`* is not specified, the current time zone is used; the examples assume the session time zone is `Europe/London`<br>`make_timestamptz(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5+01`<br>`make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York')` → `2013-07-15 13:15:23.5+01` |
| `now` ( ) → `timestamp with time zone`<br>Current date and time (start of current transaction); see Section 9.9.5<br>`now()` → `2019-12-23 14:39:53.662522-05` |
| `statement_timestamp` ( ) → `timestamp with time zone`<br>Current date and time (start of current statement); see Section 9.9.5<br>`statement_timestamp()` → `2019-12-23 14:39:53.662522-05` |
| `timeofday` ( ) → `text`<br>Current date and time (like `clock_timestamp`, but as a `text` string); see Section 9.9.5<br>`timeofday()` → `Mon Dec 23 14:39:53.662522 2019 EST` |

| Function<br>Description<br>Example(s) |
|---|
| `transaction_timestamp` ( ) → `timestamp with time zone`<br>Current date and time (start of current transaction); see Section 9.9.5<br>`transaction_timestamp()` → `2019-12-23 14:39:53.662522-05` |
| `to_timestamp` ( `double precision` ) → `timestamp with time zone`<br>Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp with time zone<br>`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` or `timestamp with time zone` value, the months, days, and microseconds fields of the `interval` value are handled in turn. First, a nonzero months field advances or decrements the date of the timestamp by the indicated number of months, keeping the day of month the same unless it would be past the end of the new month, in which case the last day of that month is used. (For example, March 31 plus 1 month becomes April 30, but March 31 plus 2 months becomes May 31.) Then the days field advances or decrements the date of the timestamp by the indicated number of days. In both these steps the local time of day is kept the same. Finally, if there is a nonzero microseconds field, it is added or subtracted literally. When doing arithmetic on a `timestamp with time zone` value in a time zone that recognizes DST, this means that adding or subtracting (say) `interval '1 day'` does not necessarily have the same result as adding or subtracting `interval '24 hours'`. For example, with the session time zone set to `America/Denver`:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1
 day';
Result: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24
 hours';
Result: 2005-04-03 13:00:00-06
```

This happens because an hour was skipped due to a change in daylight saving time at `2005-04-03 02:00:00` in time zone `America/Denver`.

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: 10537200.000000
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: 121.9583333333333333
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`, `date`, `time`, or `interval`. (Timestamps and times can be with or without time zone.) *`field`* is an identifier or string that selects what field to extract from the source value. Not all fields are valid for every input data type; for example, fields smaller than a day cannot be extracted from a `date`, while fields of a day or more cannot be extracted from a `time`. The `extract` function returns values of type `numeric`.

The following are valid field names:

`century`

The century; for `interval` values, the year field divided by 100

```
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
SELECT EXTRACT(CENTURY FROM DATE '0001-01-01 AD');
Result: 1
SELECT EXTRACT(CENTURY FROM DATE '0001-12-31 BC');
Result: -1
SELECT EXTRACT(CENTURY FROM INTERVAL '2001 years');
```

```
Result: 20
```

`day`

The day of the month (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 (negative for timestamps before that); for `date` and `timestamp` values, the nominal number of seconds since 1970-01-01 00:00:00, without regard to timezone or daylight-savings rules; 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: 982384720.120000
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Result: 982355920.120000
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800.000000
```

You can convert an epoch value back to a `timestamp with time zone` with `to_timestamp`:

```
SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00
```

Beware that applying `to_timestamp` to an epoch extracted from a `date` or `timestamp` value could produce a misleading result: the result will effectively assume that the original value had been given in UTC, which might not be the case.

`hour`

The hour field (0–23 in timestamps, unrestricted in intervals)

```
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

```
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.

`julian`

The *Julian Date* corresponding to the date or timestamp. Timestamps that are not local midnight result in a fractional value. See Section B.7 for more information.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Result: 2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Result: 2453737.50000000000000000000
```

`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; for `interval` values, the year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
SELECT EXTRACT(MILLENNIUM FROM INTERVAL '2001 years');
Result: 2
```

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: 28500.000
```

`minute`

The minutes field (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

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; for `interval` values, the month field divided by 3 plus 1

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
SELECT EXTRACT(QUARTER FROM INTERVAL '1 year 6 months');
Result: 3
```

`second`

The seconds field, including any fractional seconds

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40.000000
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.500000
```

`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.

For `interval` values, the week field is simply the number of integral days divided by 7.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
SELECT EXTRACT(WEEK FROM INTERVAL '13 days 24 hours');
Result: 1
```

`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
```

When processing an `interval` value, the `extract` function produces field values that match the interpretation used by the interval output function. This can produce surprising results if one starts with a non-normalized interval representation, for example:

```
SELECT INTERVAL '80 minutes';
Result: 01:20:00
SELECT EXTRACT(MINUTES FROM INTERVAL '80 minutes');
Result: 20
```

### Note

When the input value is +/-Infinity, `extract` returns +/-Infinity for monotonically-increasing fields (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century`, and `millennium` for `timestamp` inputs; `epoch`, `hour`, `day`, `year`, `decade`, `century`, and `millennium` for `interval` inputs). 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.

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`. For historical reasons, the `date_part` function returns values of type `double precision`. This can result in a loss of precision in certain uses. Using `extract` is recommended instead.

```
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 [, time_zone ])
```

*source* is a value expression of type `timestamp`, `timestamp with time zone`, 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 likewise of type `timestamp`, `timestamp with time zone`, or `interval`, and it has 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
```

When the input value is of type `timestamp with time zone`, the truncation is performed with respect to a particular time zone; for example, truncation to `day` produces a value that is midnight in that zone. By default, truncation is done with respect to the current TimeZone setting, but the optional *time_zone* argument can be provided to specify a different time zone. The time zone name can be specified in any of the ways described in Section 8.5.3.

A time zone cannot be specified when processing `timestamp without time zone` or `interval` inputs. These are always taken at face value.

Examples (assuming the local time zone is `America/New_York`):

```
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
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40+00');
Result: 2001-02-16 00:00:00-05
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00',
 'Australia/Sydney');
Result: 2001-02-16 08:00:00-05
SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00
```

## 9.9.3. `date_bin`

The function `date_bin` “bins” the input timestamp into the specified interval (the *stride*) aligned with a specified origin.

```
date_bin(stride, source, origin)
```

*`source`* is a value expression of type `timestamp` or `timestamp with time zone`. (Values of type `date` are cast automatically to `timestamp`.) *`stride`* is a value expression of type `interval`. The return value is likewise of type `timestamp` or `timestamp with time zone`, and it marks the beginning of the bin into which the *`source`* is placed.

Examples:

```
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP
 '2001-01-01');
Result: 2020-02-11 15:30:00
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP
 '2001-01-01 00:02:30');
Result: 2020-02-11 15:32:30
```

In the case of full units (1 minute, 1 hour, etc.), it gives the same result as the analogous `date_trunc` call, but the difference is that `date_bin` can truncate to an arbitrary interval.

The *`stride`* interval must be greater than zero and cannot contain units of month or larger.

## 9.9.4. `AT TIME ZONE` and `AT LOCAL`

The `AT TIME ZONE` operator converts time stamp *without* time zone to/from time stamp *with* time zone, and `time with time zone` values to different time zones. Table 9.34 shows its variants.

### Table 9.34. `AT TIME ZONE` and `AT LOCAL` Variants

| Operator<br>Description<br>Example(s) |
|---|
| `timestamp without time zone` AT TIME ZONE *`zone`* → `timestamp with time zone`<br>Converts given time stamp *without* time zone to time stamp *with* time zone, assuming the given value is in the named time zone.<br>`timestamp '2001-02-16 20:38:40' at time zone 'America/Denver'` → `2001-02-17 03:38:40+00` |
| `timestamp without time zone` AT LOCAL → `timestamp with time zone`<br>Converts given time stamp *without* time zone to time stamp *with* the session's `TimeZone` value as time zone.<br>`timestamp '2001-02-16 20:38:40' at local` → `2001-02-17 03:38:40+00` |
| `timestamp with time zone` AT TIME ZONE *`zone`* → `timestamp without time zone`<br>Converts given time stamp *with* time zone to time stamp *without* time zone, as the time would appear in that zone.<br>`timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver'` → `2001-02-16 18:38:40` |
| `timestamp with time zone` AT LOCAL → `timestamp without time zone`<br>Converts given time stamp *with* time zone to time stamp *without* time zone, as the time would appear with the session's `TimeZone` value as time zone. |

| Operator<br>Description<br>Example(s) |
|---|
| `timestamp with time zone '2001-02-16 20:38:40-05' at local` → `2001-02-16 18:38:40` |
| `time with time zone` `AT TIME ZONE` *`zone`* → `time with time zone`<br>Converts given time *with* time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the named destination zone.<br>`time with time zone '05:34:17-05' at time zone 'UTC'` → `10:34:17+00` |
| `time with time zone` `AT LOCAL` → `time with time zone`<br>Converts given time *with* time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the session's `TimeZone` value.<br>Assuming the session's `TimeZone` is set to `UTC`:<br>`time with time zone '05:34:17-05' at local` → `10:34:17+00` |

In these expressions, the desired time zone *`zone`* can be specified either as a text value (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. The interval case is only useful for zones that have fixed offsets from UTC, so it is not very common in practice.

The syntax `AT LOCAL` may be used as shorthand for `AT TIME ZONE` *`local`*, where *`local`* is the session's `TimeZone` value.

Examples (assuming the current TimeZone setting 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' AT TIME ZONE 'Asia/Tokyo' AT TIME
 ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT LOCAL;
Result: 2001-02-16 17:38:40
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE
 '+05';
Result: 2001-02-16 20:38:40
SELECT TIME WITH TIME ZONE '20:38:40-05' AT LOCAL;
Result: 17:38:40
```

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. The fourth example shifts the time stamp with time zone value to the time zone currently specified by the `TimeZone` setting and returns the value without a time zone. The fifth example demonstrates that the sign in a POSIX-style time zone specification has the opposite meaning of the sign in an ISO-8601 datetime literal, as described in Section 8.5.3 and Appendix B.

The sixth example is a cautionary tale. Due to the fact that there is no date associated with the input value, the conversion is made using the current date of the session. Therefore, this static example may show a wrong result depending on the time of the year it is viewed because `'America/Los_Angeles'` observes Daylight Savings Time.

The function `timezone(`*`zone`*`, `*`timestamp`*`)` is equivalent to the SQL-conforming construct *`timestamp`* `AT TIME ZONE` *`zone`*.

The function `timezone(zone, time)` is equivalent to the SQL-conforming construct `time AT TIME ZONE zone`.

The function `timezone(timestamp)` is equivalent to the SQL-conforming construct `timestamp AT LOCAL`.

The function `timezone(time)` is equivalent to the SQL-conforming construct `time AT LOCAL`.

## 9.9.5. 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: 2019-12-23
SELECT CURRENT_TIMESTAMP;
Result: 2019-12-23 14:39:53.662522-05
SELECT CURRENT_TIMESTAMP(2);
Result: 2019-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
Result: 2019-12-23 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';  -- but see tip below
```

**Tip**

Do not use the third form when specifying a value to be evaluated later, for example in a `DEFAULT` clause for a table column. 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. (See also Section 8.5.1.4.)

## 9.9.6. Delaying Execution

The following functions are available to delay execution of the server process:

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

`pg_sleep` makes the current session's process sleep until the given number of seconds have elapsed. Fractional-second delays can be specified. `pg_sleep_for` is a convenience function to allow the sleep time to be 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.

# 9.10. Enum Support Functions

For enum types (described in Section 8.7), there are several functions that allow cleaner programming without hard-coding particular values of an enum type. These are listed in Table 9.35. The examples assume an enum type created as:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue',
 'purple');
```

### Table 9.35. Enum Support Functions

| Function<br>Description<br>Example(s) |
|---|
| `enum_first` ( `anyenum` ) → `anyenum`<br>Returns the first value of the input enum type.<br>`enum_first(null::rainbow)` → `red` |
| `enum_last` ( `anyenum` ) → `anyenum`<br>Returns the last value of the input enum type.<br>`enum_last(null::rainbow)` → `purple` |
| `enum_range` ( `anyenum` ) → `anyarray`<br>Returns all values of the input enum type in an ordered array.<br>`enum_range(null::rainbow)` → `{red,orange,yellow,green,blue,purple}` |
| `enum_range` ( `anyenum`, `anyenum` ) → `anyarray`<br>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.<br>`enum_range('orange'::rainbow, 'green'::rainbow)` → `{orange,yellow,green}`<br>`enum_range(NULL, 'green'::rainbow)` → `{red,orange,yellow,green}`<br>`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 used in 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 9.36, Table 9.37, and Table 9.38.

### Table 9.36. Geometric Operators

| **Operator**<br>**Description**<br>**Example(s)** |
|---|
| *geometric_type* + point → *geometric_type*<br>Adds the coordinates of the second point to those of each point of the first argument, thus performing translation. Available for point, box, path, circle.<br>box '(1,1),(0,0)' + point '(2,0)' → (3,1),(2,0) |
| path + path → path<br>Concatenates two open paths (returns NULL if either path is closed).<br>path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]' → [(0,0),(1,1),(2,2),(3,3),(4,4)] |
| *geometric_type* - point → *geometric_type*<br>Subtracts the coordinates of the second point from those of each point of the first argument, thus performing translation. Available for point, box, path, circle.<br>box '(1,1),(0,0)' - point '(2,0)' → (-1,1),(-2,0) |
| *geometric_type* * point → *geometric_type*<br>Multiplies each point of the first argument by the second point (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex multiplication). If one interprets the second point as a vector, this is equivalent to scaling the object's size and distance from the origin by the length of the vector, and rotating it counterclockwise around the origin by the vector's angle from the *x* axis. Available for point, box,[a] path, circle.<br>path '((0,0),(1,0),(1,1))' * point '(3.0,0)' → ((0,0),(3,0),(3,3))<br>path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45)) → ((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095)) |
| *geometric_type* / point → *geometric_type*<br>Divides each point of the first argument by the second point (treating a point as being a complex number represented by real and imaginary parts, and performing standard complex division). If one interprets the second point as a vector, this is equivalent to scaling the object's size and distance from the origin down by the length of the vector, and rotating it clockwise around the origin by the vector's angle from the *x* axis. Available for point, box,[a] path, circle.<br>path '((0,0),(1,0),(1,1))' / point '(2.0,0)' → ((0,0),(0.5,0),(0.5,0.5))<br>path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45)) → ((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0)) |
| @-@ *geometric_type* → double precision<br>Computes the total length. Available for lseg, path.<br>@-@ path '[(0,0),(1,0),(1,1)]' → 2 |
| @@ *geometric_type* → point<br>Computes the center point. Available for box, lseg, polygon, circle.<br>@@ box '(2,2),(0,0)' → (1,1) |
| # *geometric_type* → integer<br>Returns the number of points. Available for path, polygon.<br># path '((1,0),(0,1),(-1,0))' → 3 |
| *geometric_type* # *geometric_type* → point<br>Computes the point of intersection, or NULL if there is none. Available for lseg, line.<br>lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]' → (0.5,0.5) |

<table>
<tr><th>Operator</th><th>Description</th><th>Example(s)</th></tr>
<tr><td>box # box → box</td><td>Computes the intersection of two boxes, or NULL if there is none.</td><td>box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)' → (1,1),(-1,-1)</td></tr>
<tr><td>geometric_type ## geometric_type → point</td><td>Computes the closest point to the first object on the second object. Available for these pairs of types: (point, box), (point, lseg), (point, line), (lseg, box), (lseg, lseg), (line, lseg).</td><td>point '(0,0)' ## lseg '[(2,0),(0,2)]' → (1,1)</td></tr>
<tr><td>geometric_type <-> geometric_type → double precision</td><td>Computes the distance between the objects. Available for all seven geometric types, for all combinations of point with another geometric type, and for these additional pairs of types: (box, lseg), (lseg, line), (polygon, circle) (and the commutator cases).</td><td>circle '<(0,0),1>' <-> circle '<(5,0),1>' → 3</td></tr>
<tr><td>geometric_type @> geometric_type → boolean</td><td>Does first object contain second? Available for these pairs of types: (box, point), (box, box), (path, point), (polygon, point), (polygon, polygon), (circle, point), (circle, circle).</td><td>circle '<(0,0),2>' @> point '(1,1)' → t</td></tr>
<tr><td>geometric_type <@ geometric_type → boolean</td><td>Is first object contained in or on second? Available for these pairs of types: (point, box), (point, lseg), (point, line), (point, path), (point, polygon), (point, circle), (box, box), (lseg, box), (lseg, line), (polygon, polygon), (circle, circle).</td><td>point '(1,1)' <@ circle '<(0,0),2>' → t</td></tr>
<tr><td>geometric_type && geometric_type → boolean</td><td>Do these objects overlap? (One point in common makes this true.) Available for box, polygon, circle.</td><td>box '(1,1),(0,0)' && box '(2,2),(0,0)' → t</td></tr>
<tr><td>geometric_type << geometric_type → boolean</td><td>Is first object strictly left of second? Available for point, box, polygon, circle.</td><td>circle '<(0,0),1>' << circle '<(5,0),1>' → t</td></tr>
<tr><td>geometric_type >> geometric_type → boolean</td><td>Is first object strictly right of second? Available for point, box, polygon, circle.</td><td>circle '<(5,0),1>' >> circle '<(0,0),1>' → t</td></tr>
<tr><td>geometric_type &< geometric_type → boolean</td><td>Does first object not extend to the right of second? Available for box, polygon, circle.</td><td>box '(1,1),(0,0)' &< box '(2,2),(0,0)' → t</td></tr>
<tr><td>geometric_type &> geometric_type → boolean</td><td>Does first object not extend to the left of second? Available for box, polygon, circle.</td><td>box '(3,3),(0,0)' &> box '(2,2),(0,0)' → t</td></tr>
<tr><td>geometric_type <<| geometric_type → boolean</td><td>Is first object strictly below second? Available for point, box, polygon, circle.</td><td>box '(3,3),(0,0)' <<| box '(5,5),(3,4)' → t</td></tr>
<tr><td>geometric_type |>> geometric_type → boolean</td><td></td><td></td></tr>
</table>

| Operator<br>Description<br>Example(s) |
|---|
| Is first object strictly above second? Available for `point`, `box`, `polygon`, `circle`.<br>`box '(5,5),(3,4)' \|>> box '(3,3),(0,0)'` → `t` |
| *`geometric_type`* `&<\|` *`geometric_type`* → `boolean`<br>Does first object not extend above second? Available for `box`, `polygon`, `circle`.<br>`box '(1,1),(0,0)' &<\| box '(2,2),(0,0)'` → `t` |
| *`geometric_type`* `\|&>` *`geometric_type`* → `boolean`<br>Does first object not extend below second? Available for `box`, `polygon`, `circle`.<br>`box '(3,3),(0,0)' \|&> box '(2,2),(0,0)'` → `t` |
| `box <^ box` → `boolean`<br>Is first object below second (allows edges to touch)?<br>`box '((1,1),(0,0))' <^ box '((2,2),(1,1))'` → `t` |
| `box >^ box` → `boolean`<br>Is first object above second (allows edges to touch)?<br>`box '((2,2),(1,1))' >^ box '((1,1),(0,0))'` → `t` |
| *`geometric_type`* `?#` *`geometric_type`* → `boolean`<br>Do these objects intersect? Available for these pairs of types: (`box`, `box`), (`lseg`, `box`), (`lseg`, `lseg`), (`lseg`, `line`), (`line`, `box`), (`line`, `line`), (`path`, `path`).<br>`lseg '[(-1,0),(1,0)]' ?# box '(2,2),(-2,-2)'` → `t` |
| `?- line` → `boolean`<br>`?- lseg` → `boolean`<br>Is line horizontal?<br>`?- lseg '[(-1,0),(1,0)]'` → `t` |
| `point ?- point` → `boolean`<br>Are points horizontally aligned (that is, have same y coordinate)?<br>`point '(1,0)' ?- point '(0,0)'` → `t` |
| `?\| line` → `boolean`<br>`?\| lseg` → `boolean`<br>Is line vertical?<br>`?\| lseg '[(-1,0),(1,0)]'` → `f` |
| `point ?\| point` → `boolean`<br>Are points vertically aligned (that is, have same x coordinate)?<br>`point '(0,1)' ?\| point '(0,0)'` → `t` |
| `line ?-\| line` → `boolean`<br>`lseg ?-\| lseg` → `boolean`<br>Are lines perpendicular?<br>`lseg '[(0,0),(0,1)]' ?-\| lseg '[(0,0),(1,0)]'` → `t` |
| `line ?\|\| line` → `boolean`<br>`lseg ?\|\| lseg` → `boolean`<br>Are lines parallel?<br>`lseg '[(-1,0),(1,0)]' ?\|\| lseg '[(-1,2),(1,2)]'` → `t` |

| **Operator**<br>**Description**<br>**Example(s)** |
|---|
| *`geometric_type`* `~=` *`geometric_type`* → `boolean`<br>Are these objects the same? Available for `point`, `box`, `polygon`, `circle`.<br>`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'` → `t` |

[a]"Rotating" a box with these operators only moves its corner points: the box is still considered to have sides parallel to the axes. Hence the box's size is not preserved, as a true rotation would do.

### Caution

Note that the "same as" operator, `~=`, represents the usual notion of equality for the `point`, `box`, `polygon`, and `circle` types. Some of the geometric types also have an `=` operator, but `=` compares for equal *areas* only. The other scalar comparison operators (`<=` and so on), where available for these types, likewise compare areas.

### Note

Before PostgreSQL 14, the point is strictly below/above comparison operators `point <<| point` and `point |>> point` were respectively called `<^` and `>^`. These names are still available, but are deprecated and will eventually be removed.

**Table 9.37. Geometric Functions**

| **Function**<br>**Description**<br>**Example(s)** |
|---|
| `area` ( *`geometric_type`* ) → `double precision`<br>Computes area. Available for `box`, `path`, `circle`. A `path` input must be closed, else NULL is returned. Also, if the `path` is self-intersecting, the result may be meaningless.<br>`area(box '(2,2),(0,0)')` → `4` |
| `center` ( *`geometric_type`* ) → `point`<br>Computes center point. Available for `box`, `circle`.<br>`center(box '(1,2),(0,0)')` → `(0.5,1)` |
| `diagonal` ( `box` ) → `lseg`<br>Extracts box's diagonal as a line segment (same as `lseg(box)`).<br>`diagonal(box '(1,2),(0,0)')` → `[(1,2),(0,0)]` |
| `diameter` ( `circle` ) → `double precision`<br>Computes diameter of circle.<br>`diameter(circle '<(0,0),2>')` → `4` |
| `height` ( `box` ) → `double precision`<br>Computes vertical size of box.<br>`height(box '(1,2),(0,0)')` → `2` |
| `isclosed` ( `path` ) → `boolean`<br>Is path closed?<br>`isclosed(path '((0,0),(1,1),(2,0))')` → `t` |
| `isopen` ( `path` ) → `boolean` |

| Function<br>Description<br>Example(s) |
|---|
| Is path open?<br>`isopen(path '[(0,0),(1,1),(2,0)]')` → `t` |
| `length` ( *`geometric_type`* ) → `double precision`<br>Computes the total length. Available for `lseg`, `path`.<br>`length(path '((-1,0),(1,0))')` → `4` |
| `npoints` ( *`geometric_type`* ) → `integer`<br>Returns the number of points. Available for `path`, `polygon`.<br>`npoints(path '[(0,0),(1,1),(2,0)]')` → `3` |
| `pclose` ( `path` ) → `path`<br>Converts path to closed form.<br>`pclose(path '[(0,0),(1,1),(2,0)]')` → `((0,0),(1,1),(2,0))` |
| `popen` ( `path` ) → `path`<br>Converts path to open form.<br>`popen(path '((0,0),(1,1),(2,0))')` → `[(0,0),(1,1),(2,0)]` |
| `radius` ( `circle` ) → `double precision`<br>Computes radius of circle.<br>`radius(circle '<(0,0),2>')` → `2` |
| `slope` ( `point`, `point` ) → `double precision`<br>Computes slope of a line drawn through the two points.<br>`slope(point '(0,0)', point '(2,1)')` → `0.5` |
| `width` ( `box` ) → `double precision`<br>Computes horizontal size of box.<br>`width(box '(1,2),(0,0)')` → `1` |

## Table 9.38. Geometric Type Conversion Functions

| Function<br>Description<br>Example(s) |
|---|
| `box` ( `circle` ) → `box`<br>Computes box inscribed within the circle.<br>`box(circle '<(0,0),2>')` → `(1.414213562373095,1.414213562373095),(-1.414213562373095,-1.414213562373095)` |
| `box` ( `point` ) → `box`<br>Converts point to empty box.<br>`box(point '(1,0)')` → `(1,0),(1,0)` |
| `box` ( `point`, `point` ) → `box`<br>Converts any two corner points to box.<br>`box(point '(0,1)', point '(1,0)')` → `(1,1),(0,0)` |
| `box` ( `polygon` ) → `box`<br>Computes bounding box of polygon.<br>`box(polygon '((0,0),(1,1),(2,0))')` → `(2,1),(0,0)` |

| **Function**<br>**Description**<br>**Example(s)** |
|---|
| `bound_box` ( `box`, `box` ) → `box`<br>Computes bounding box of two boxes.<br>`bound_box(box '(1,1),(0,0)', box '(4,4),(3,3)')` → `(4,4),(0,0)` |
| `circle` ( `box` ) → `circle`<br>Computes smallest circle enclosing box.<br>`circle(box '(1,1),(0,0)')` → `<(0.5,0.5),0.7071067811865476>` |
| `circle` ( `point`, `double precision` ) → `circle`<br>Constructs circle from center and radius.<br>`circle(point '(0,0)', 2.0)` → `<(0,0),2>` |
| `circle` ( `polygon` ) → `circle`<br>Converts polygon to circle. The circle's center is the mean of the positions of the polygon's points, and the radius is the average distance of the polygon's points from that center.<br>`circle(polygon '((0,0),(1,3),(2,0))')` → `<(1,1),1.6094757082487299>` |
| `line` ( `point`, `point` ) → `line`<br>Converts two points to the line through them.<br>`line(point '(-1,0)', point '(1,0)')` → `{0,-1,0}` |
| `lseg` ( `box` ) → `lseg`<br>Extracts box's diagonal as a line segment.<br>`lseg(box '(1,0),(-1,0)')` → `[(1,0),(-1,0)]` |
| `lseg` ( `point`, `point` ) → `lseg`<br>Constructs line segment from two endpoints.<br>`lseg(point '(-1,0)', point '(1,0)')` → `[(-1,0),(1,0)]` |
| `path` ( `polygon` ) → `path`<br>Converts polygon to a closed path with the same list of points.<br>`path(polygon '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))` |
| `point` ( `double precision`, `double precision` ) → `point`<br>Constructs point from its coordinates.<br>`point(23.4, -44.5)` → `(23.4,-44.5)` |
| `point` ( `box` ) → `point`<br>Computes center of box.<br>`point(box '(1,0),(-1,0)')` → `(0,0)` |
| `point` ( `circle` ) → `point`<br>Computes center of circle.<br>`point(circle '<(0,0),2>')` → `(0,0)` |
| `point` ( `lseg` ) → `point`<br>Computes center of line segment.<br>`point(lseg '[(-1,0),(1,0)]')` → `(0,0)` |
| `point` ( `polygon` ) → `point`<br>Computes center of polygon (the mean of the positions of the polygon's points).<br>`point(polygon '((0,0),(1,1),(2,0))')` → `(1,0.3333333333333333)` |

| Function<br>Description<br>Example(s) |
|---|
| `polygon` ( `box` ) → `polygon`<br>Converts box to a 4-point polygon.<br>`polygon(box '(1,1),(0,0)')` → `((0,0),(0,1),(1,1),(1,0))` |
| `polygon` ( `circle` ) → `polygon`<br>Converts circle to a 12-point polygon.<br>`polygon(circle '<(0,0),2>')` → `((-2,0), (-1.7320508075688774,0.9999999999999999), (-1.0000000000000002,1.7320508075688772), (-1.2246063538223773e-16,2), (0.9999999999999996,1.7320508075688774), (1.732050807568877,1.0000000000000007),(2,2.4492127076447545e-16), (1.7320508075688776,-0.9999999999999994), (1.0000000000000009,-1.7320508075688767), (3.673819061467132e-16,-2), (-0.9999999999999987,-1.732050807568878), (-1.7320508075688767,-1.0000000000000009))` |
| `polygon` ( `integer`, `circle` ) → `polygon`<br>Converts circle to an *n*-point polygon.<br>`polygon(4, circle '<(3,0),1>')` → `((2,0),(3,1), (4,1.2246063538223773e-16),(3,-1))` |
| `polygon` ( `path` ) → `polygon`<br>Converts closed path to a polygon with the same list of points.<br>`polygon(path '((0,0),(1,1),(2,0))')` → `((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.

# 9.12. Network Address Functions and Operators

The IP network address types, `cidr` and `inet`, support the usual comparison operators shown in Table 9.1 as well as the specialized operators and functions shown in Table 9.39 and Table 9.40.

Any `cidr` value can be cast to `inet` implicitly; therefore, the operators and functions shown below 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.

### Table 9.39. IP Address Operators

| Operator<br>Description<br>Example(s) |
|---|
| `inet` << `inet` → `boolean`<br>Is subnet strictly contained by subnet? This operator, and the next four, test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any bits to the right of the netmasks) and determine whether one network is identical to or a subnet of the other. |

<table>
<tr><th>Operator<br>Description<br>Example(s)</th></tr>
<tr><td>inet '192.168.1.5' << inet '192.168.1/24' → t<br>inet '192.168.0.5' << inet '192.168.1/24' → f<br>inet '192.168.1/24' << inet '192.168.1/24' → f</td></tr>
<tr><td>inet <<= inet → boolean<br>Is subnet contained by or equal to subnet?<br>inet '192.168.1/24' <<= inet '192.168.1/24' → t</td></tr>
<tr><td>inet >> inet → boolean<br>Does subnet strictly contain subnet?<br>inet '192.168.1/24' >> inet '192.168.1.5' → t</td></tr>
<tr><td>inet >>= inet → boolean<br>Does subnet contain or equal subnet?<br>inet '192.168.1/24' >>= inet '192.168.1/24' → t</td></tr>
<tr><td>inet && inet → boolean<br>Does either subnet contain or equal the other?<br>inet '192.168.1/24' && inet '192.168.1.80/28' → t<br>inet '192.168.1/24' && inet '192.168.2.0/28' → f</td></tr>
<tr><td>~ inet → inet<br>Computes bitwise NOT.<br>~ inet '192.168.1.6' → 63.87.254.249</td></tr>
<tr><td>inet & inet → inet<br>Computes bitwise AND.<br>inet '192.168.1.6' & inet '0.0.0.255' → 0.0.0.6</td></tr>
<tr><td>inet | inet → inet<br>Computes bitwise OR.<br>inet '192.168.1.6' | inet '0.0.0.255' → 192.168.1.255</td></tr>
<tr><td>inet + bigint → inet<br>Adds an offset to an address.<br>inet '192.168.1.6' + 25 → 192.168.1.31</td></tr>
<tr><td>bigint + inet → inet<br>Adds an offset to an address.<br>200 + inet '::ffff:fff0:1' → ::ffff:255.240.0.201</td></tr>
<tr><td>inet - bigint → inet<br>Subtracts an offset from an address.<br>inet '192.168.1.43' - 36 → 192.168.1.7</td></tr>
<tr><td>inet - inet → bigint<br>Computes the difference of two addresses.<br>inet '192.168.1.43' - inet '192.168.1.19' → 24<br>inet '::1' - inet '::ffff:1' → -4294901760</td></tr>
</table>

### Table 9.40. IP Address Functions

| Function<br>Description<br>Example(s) |
|---|
| `abbrev` ( `inet` ) → `text`<br>Creates an abbreviated display format as text. (The result is the same as the `inet` output function produces; it is "abbreviated" only in comparison to the result of an explicit cast to `text`, which for historical reasons will never suppress the netmask part.)<br>`abbrev(inet '10.1.0.0/32')` → `10.1.0.0` |
| `abbrev` ( `cidr` ) → `text`<br>Creates an abbreviated display format as text. (The abbreviation consists of dropping all-zero octets to the right of the netmask; more examples are in Table 8.22.)<br>`abbrev(cidr '10.1.0.0/16')` → `10.1/16` |
| `broadcast` ( `inet` ) → `inet`<br>Computes the broadcast address for the address's network.<br>`broadcast(inet '192.168.1.5/24')` → `192.168.1.255/24` |
| `family` ( `inet` ) → `integer`<br>Returns the address's family: `4` for IPv4, `6` for IPv6.<br>`family(inet '::1')` → `6` |
| `host` ( `inet` ) → `text`<br>Returns the IP address as text, ignoring the netmask.<br>`host(inet '192.168.1.0/24')` → `192.168.1.0` |
| `hostmask` ( `inet` ) → `inet`<br>Computes the host mask for the address's network.<br>`hostmask(inet '192.168.23.20/30')` → `0.0.0.3` |
| `inet_merge` ( `inet`, `inet` ) → `cidr`<br>Computes the smallest network that includes both of the given networks.<br>`inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24')` → `192.168.0.0/22` |
| `inet_same_family` ( `inet`, `inet` ) → `boolean`<br>Tests whether the addresses belong to the same IP family.<br>`inet_same_family(inet '192.168.1.5/24', inet '::1')` → `f` |
| `masklen` ( `inet` ) → `integer`<br>Returns the netmask length in bits.<br>`masklen(inet '192.168.1.5/24')` → `24` |
| `netmask` ( `inet` ) → `inet`<br>Computes the network mask for the address's network.<br>`netmask(inet '192.168.1.5/24')` → `255.255.255.0` |
| `network` ( `inet` ) → `cidr`<br>Returns the network part of the address, zeroing out whatever is to the right of the netmask. (This is equivalent to casting the value to `cidr`.)<br>`network(inet '192.168.1.5/24')` → `192.168.1.0/24` |
| `set_masklen` ( `inet`, `integer` ) → `inet`<br>Sets the netmask length for an `inet` value. The address part does not change. |

| Function<br>Description<br>Example(s) |
|---|
| `set_masklen(inet '192.168.1.5/24', 16)` → `192.168.1.5/16` |
| `set_masklen` ( `cidr`, `integer` ) → `cidr`<br>Sets the netmask length for a `cidr` value. Address bits to the right of the new netmask are set to zero.<br>`set_masklen(cidr '192.168.1.0/24', 16)` → `192.168.0.0/16` |
| `text` ( `inet` ) → `text`<br>Returns the unabbreviated IP address and netmask length as text. (This has the same result as an explicit cast to `text`.)<br>`text(inet '192.168.1.5')` → `192.168.1.5/32` |

> **Tip**
>
> The `abbrev`, `host`, and `text` functions are primarily intended to offer alternative display formats for IP addresses.

The MAC address types, `macaddr` and `macaddr8`, support the usual comparison operators shown in Table 9.1 as well as the specialized functions shown in Table 9.41. In addition, they support the bitwise logical operators `~`, `&` and `|` (NOT, AND and OR), just as shown above for IP addresses.

### Table 9.41. MAC Address Functions

| Function<br>Description<br>Example(s) |
|---|
| `trunc` ( `macaddr` ) → `macaddr`<br>Sets the last 3 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).<br>`trunc(macaddr '12:34:56:78:90:ab')` → `12:34:56:00:00:00` |
| `trunc` ( `macaddr8` ) → `macaddr8`<br>Sets the last 5 bytes of the address to zero. The remaining prefix can be associated with a particular manufacturer (using data not included in PostgreSQL).<br>`trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')` → `12:34:56:00:00:00:00:00` |
| `macaddr8_set7bit` ( `macaddr8` ) → `macaddr8`<br>Sets the 7th bit of the address to one, creating what is known as modified EUI-64, for inclusion in an IPv6 address.<br>`macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef')` → `02:34:56:ff:fe:ab:cd:ef` |

# 9.13. Text Search Functions and Operators

Table 9.42, Table 9.43 and Table 9.44 summarize the functions and operators that are provided for full text searching. See Chapter 12 for a detailed explanation of PostgreSQL's text search facility.

### Table 9.42. Text Search Operators

<table>
<tr><th>Operator<br>Description<br>Example(s)</th></tr>
<tr><td>tsvector @@ tsquery → boolean<br>tsquery @@ tsvector → boolean<br>Does tsvector match tsquery? (The arguments can be given in either order.)<br>to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') → t</td></tr>
<tr><td>text @@ tsquery → boolean<br>Does text string, after implicit invocation of to_tsvector(), match tsquery?<br>'fat cats ate rats' @@ to_tsquery('cat & rat') → t</td></tr>
<tr><td>tsvector || tsvector → tsvector<br>Concatenates two tsvectors. If both inputs contain lexeme positions, the second input's positions are adjusted accordingly.<br>'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector → 'a':1 'b':2,5 'c':3 'd':4</td></tr>
<tr><td>tsquery && tsquery → tsquery<br>ANDs two tsquerys together, producing a query that matches documents that match both input queries.<br>'fat | rat'::tsquery && 'cat'::tsquery → ( 'fat' | 'rat' ) & 'cat'</td></tr>
<tr><td>tsquery || tsquery → tsquery<br>ORs two tsquerys together, producing a query that matches documents that match either input query.<br>'fat | rat'::tsquery || 'cat'::tsquery → 'fat' | 'rat' | 'cat'</td></tr>
<tr><td>!! tsquery → tsquery<br>Negates a tsquery, producing a query that matches documents that do not match the input query.<br>!! 'cat'::tsquery → !'cat'</td></tr>
<tr><td>tsquery <-> tsquery → tsquery<br>Constructs a phrase query, which matches if the two input queries match at successive lexemes.<br>to_tsquery('fat') <-> to_tsquery('rat') → 'fat' <-> 'rat'</td></tr>
<tr><td>tsquery @> tsquery → boolean<br>Does first tsquery contain the second? (This considers only whether all the lexemes appearing in one query appear in the other, ignoring the combining operators.)<br>'cat'::tsquery @> 'cat & rat'::tsquery → f</td></tr>
<tr><td>tsquery <@ tsquery → boolean<br>Is first tsquery contained in the second? (This considers only whether all the lexemes appearing in one query appear in the other, ignoring the combining operators.)<br>'cat'::tsquery <@ 'cat & rat'::tsquery → t<br>'cat'::tsquery <@ '!cat & rat'::tsquery → t</td></tr>
</table>

In addition to these specialized operators, the usual comparison operators shown in Table 9.1 are available 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 9.43. Text Search Functions

<table>
<tr><th>Function<br>Description<br>Example(s)</th></tr>
<tr><td>array_to_tsvector ( text[] ) → tsvector<br>Converts an array of text strings to a tsvector. The given strings are used as lexemes as-is, without further processing. Array elements must not be empty strings or NULL.<br>array_to_tsvector('{fat,cat,rat}'::text[]) → 'cat' 'fat' 'rat'</td></tr>
<tr><td>get_current_ts_config ( ) → regconfig<br>Returns the OID of the current default text search configuration (as set by default_text_search_config).<br>get_current_ts_config() → english</td></tr>
<tr><td>length ( tsvector ) → integer<br>Returns the number of lexemes in the tsvector.<br>length('fat:2,4 cat:3 rat:5A'::tsvector) → 3</td></tr>
<tr><td>numnode ( tsquery ) → integer<br>Returns the number of lexemes plus operators in the tsquery.<br>numnode('(fat & rat) | cat'::tsquery) → 5</td></tr>
<tr><td>plainto_tsquery ( [ config regconfig, ] query text ) → tsquery<br>Converts text to a tsquery, normalizing words according to the specified or default configuration. Any punctuation in the string is ignored (it does not determine query operators). The resulting query matches documents containing all non-stopwords in the text.<br>plainto_tsquery('english', 'The Fat Rats') → 'fat' & 'rat'</td></tr>
<tr><td>phraseto_tsquery ( [ config regconfig, ] query text ) → tsquery<br>Converts text to a tsquery, normalizing words according to the specified or default configuration. Any punctuation in the string is ignored (it does not determine query operators). The resulting query matches phrases containing all non-stopwords in the text.<br>phraseto_tsquery('english', 'The Fat Rats') → 'fat' <-> 'rat'<br>phraseto_tsquery('english', 'The Cat and Rats') → 'cat' <2> 'rat'</td></tr>
<tr><td>websearch_to_tsquery ( [ config regconfig, ] query text ) → tsquery<br>Converts text to a tsquery, normalizing words according to the specified or default configuration. Quoted word sequences are converted to phrase tests. The word “or” is understood as producing an OR operator, and a dash produces a NOT operator; other punctuation is ignored. This approximates the behavior of some common web search tools.<br>websearch_to_tsquery('english', '"fat rat" or cat dog') → 'fat' <-> 'rat' | 'cat' & 'dog'</td></tr>
<tr><td>querytree ( tsquery ) → text<br>Produces a representation of the indexable portion of a tsquery. A result that is empty or just T indicates a non-indexable query.<br>querytree('foo & ! bar'::tsquery) → 'foo'</td></tr>
<tr><td>setweight ( vector tsvector, weight "char" ) → tsvector<br>Assigns the specified weight to each element of the vector.<br>setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A') → 'cat':3A 'fat':2A,4A 'rat':5A</td></tr>
<tr><td>setweight ( vector tsvector, weight "char", lexemes text[] ) → tsvector</td></tr>
</table>

| Function<br>Description<br>Example(s) |
|---|
| Assigns the specified *weight* to elements of the *vector* that are listed in *lexemes*. The strings in *lexemes* are taken as lexemes as-is, without further processing. Strings that do not match any lexeme in *vector* are ignored.<br>`setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}')` → `'cat':3A 'fat':2,4 'rat':5A,6A` |
| `strip` ( `tsvector` ) → `tsvector`<br>Removes positions and weights from the `tsvector`.<br>`strip('fat:2,4 cat:3 rat:5A'::tsvector)` → `'cat' 'fat' 'rat'` |
| `to_tsquery` ( [ *config* `regconfig`, ] *query* `text` ) → `tsquery`<br>Converts text to a `tsquery`, normalizing words according to the specified or default configuration. The words must be combined by valid `tsquery` operators.<br>`to_tsquery('english', 'The & Fat & Rats')` → `'fat' & 'rat'` |
| `to_tsvector` ( [ *config* `regconfig`, ] *document* `text` ) → `tsvector`<br>Converts text to a `tsvector`, normalizing words according to the specified or default configuration. Position information is included in the result.<br>`to_tsvector('english', 'The Fat Rats')` → `'fat':2 'rat':3` |
| `to_tsvector` ( [ *config* `regconfig`, ] *document* `json` ) → `tsvector`<br>`to_tsvector` ( [ *config* `regconfig`, ] *document* `jsonb` ) → `tsvector`<br>Converts each string value in the JSON document to a `tsvector`, normalizing words according to the specified or default configuration. The results are then concatenated in document order to produce the output. Position information is generated as though one stopword exists between each pair of string values. (Beware that “document order” of the fields of a JSON object is implementation-dependent when the input is `jsonb`; observe the difference in the examples.)<br>`to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json)` → `'dog':5 'fat':2 'rat':3`<br>`to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb)` → `'dog':1 'fat':4 'rat':5` |
| `json_to_tsvector` ( [ *config* `regconfig`, ] *document* `json`, *filter* `jsonb` ) → `tsvector`<br>`jsonb_to_tsvector` ( [ *config* `regconfig`, ] *document* `jsonb`, *filter* `jsonb` ) → `tsvector`<br>Selects each item in the JSON document that is requested by the *filter* and converts each one to a `tsvector`, normalizing words according to the specified or default configuration. The results are then concatenated in document order to produce the output. Position information is generated as though one stopword exists between each pair of selected items. (Beware that “document order” of the fields of a JSON object is implementation-dependent when the input is `jsonb`.) The *filter* must be a `jsonb` array containing zero or more of these keywords: `"string"` (to include all string values), `"numeric"` (to include all numeric values), `"boolean"` (to include all boolean values), `"key"` (to include all keys), or `"all"` (to include all the above). As a special case, the *filter* can also be a simple JSON value that is one of these keywords.<br>`json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')` → `'123':5 'fat':2 'rat':3`<br>`json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123}'::json, '"all"')` → `'123':9 'cat':1 'dog':7 'fat':4 'rat':5` |
| `ts_delete` ( *vector* `tsvector`, *lexeme* `text` ) → `tsvector`<br>Removes any occurrence of the given *lexeme* from the *vector*. The *lexeme* string is treated as a lexeme as-is, without further processing. |

| Function<br>Description<br>Example(s) |
|---|
| `ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')` → `'cat':3 'rat':5A` |
| `ts_delete` ( *`vector`* `tsvector`, *`lexemes`* `text[]` ) → `tsvector`<br>Removes any occurrences of the lexemes in *`lexemes`* from the *`vector`*. The strings in *`lexemes`* are taken as lexemes as-is, without further processing. Strings that do not match any lexeme in *`vector`* are ignored.<br>`ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])` → `'cat':3` |
| `ts_filter` ( *`vector`* `tsvector`, *`weights`* `"char"[]` ) → `tsvector`<br>Selects only elements with the given *`weights`* from the *`vector`*.<br>`ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}')` → `'cat':3B 'rat':5A` |
| `ts_headline` ( [ *`config`* `regconfig`, ] *`document`* `text`, *`query`* `tsquery` [, *`options`* `text` ] ) → `text`<br>Displays, in an abbreviated form, the match(es) for the *`query`* in the *`document`*, which must be raw text not a `tsvector`. Words in the document are normalized according to the specified or default configuration before matching to the query. Use of this function is discussed in Section 12.3.4, which also describes the available *`options`*.<br>`ts_headline('The fat cat ate the rat.', 'cat')` → `The fat <b>cat</b> ate the rat.` |
| `ts_headline` ( [ *`config`* `regconfig`, ] *`document`* `json`, *`query`* `tsquery` [, *`options`* `text` ] ) → `text`<br>`ts_headline` ( [ *`config`* `regconfig`, ] *`document`* `jsonb`, *`query`* `tsquery` [, *`options`* `text` ] ) → `text`<br>Displays, in an abbreviated form, match(es) for the *`query`* that occur in string values within the JSON *`document`*. See Section 12.3.4 for more details.<br>`ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat')` → `{"cat": "raining <b>cats</b> and dogs"}` |
| `ts_rank` ( [ *`weights`* `real[]`, ] *`vector`* `tsvector`, *`query`* `tsquery` [, *`normalization`* `integer` ] ) → `real`<br>Computes a score showing how well the *`vector`* matches the *`query`*. See Section 12.3.3 for details.<br>`ts_rank(to_tsvector('raining cats and dogs'), 'cat')` → `0.06079271` |
| `ts_rank_cd` ( [ *`weights`* `real[]`, ] *`vector`* `tsvector`, *`query`* `tsquery` [, *`normalization`* `integer` ] ) → `real`<br>Computes a score showing how well the *`vector`* matches the *`query`*, using a cover density algorithm. See Section 12.3.3 for details.<br>`ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat')` → `0.1` |
| `ts_rewrite` ( *`query`* `tsquery`, *`target`* `tsquery`, *`substitute`* `tsquery` ) → `tsquery`<br>Replaces occurrences of *`target`* with *`substitute`* within the *`query`*. See Section 12.4.2.1 for details.<br>`ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo\|bar'::tsquery)` → `'b' & ( 'foo' \| 'bar' )` |
| `ts_rewrite` ( *`query`* `tsquery`, *`select`* `text` ) → `tsquery`<br>Replaces portions of the *`query`* according to target(s) and substitute(s) obtained by executing a `SELECT` command. See Section 12.4.2.1 for details. |

<table>
<tr><th>Function<br>Description<br>Example(s)</th></tr>
<tr><td><code>SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')</code> → <code>'b' & ( 'foo' | 'bar' )</code></td></tr>
<tr><td><code>tsquery_phrase</code> ( <code>query1 tsquery</code>, <code>query2 tsquery</code> ) → <code>tsquery</code><br>Constructs a phrase query that searches for matches of <code>query1</code> and <code>query2</code> at successive lexemes (same as <code><-></code> operator).<br><code>tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))</code> → <code>'fat' <-> 'cat'</code></td></tr>
<tr><td><code>tsquery_phrase</code> ( <code>query1 tsquery</code>, <code>query2 tsquery</code>, <code>distance integer</code> ) → <code>tsquery</code><br>Constructs a phrase query that searches for matches of <code>query1</code> and <code>query2</code> that occur exactly <code>distance</code> lexemes apart.<br><code>tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)</code> → <code>'fat' <10> 'cat'</code></td></tr>
<tr><td><code>tsvector_to_array</code> ( <code>tsvector</code> ) → <code>text[]</code><br>Converts a <code>tsvector</code> to an array of lexemes.<br><code>tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)</code> → <code>{cat,fat,rat}</code></td></tr>
<tr><td><code>unnest</code> ( <code>tsvector</code> ) → <code>setof record</code> ( <code>lexeme text</code>, <code>positions smallint[]</code>, <code>weights text</code> )<br>Expands a <code>tsvector</code> into a set of rows, one per lexeme.<br><code>select * from unnest('cat:3 fat:2,4 rat:5A'::tsvector)</code> →<pre>
 lexeme | positions | weights
--------+-----------+---------
 cat    | {3}       | {D}
 fat    | {2,4}     | {D,D}
 rat    | {5}       | {A}
</pre></td></tr>
</table>

### Note

All the text search functions that accept an optional `regconfig` argument will use the configuration specified by default_text_search_config when that argument is omitted.

The functions in Table 9.44 are listed separately because they are not usually used in everyday text searching operations. They are primarily helpful for development and debugging of new text search configurations.

**Table 9.44. Text Search Debugging Functions**

<table>
<tr><th>Function<br>Description<br>Example(s)</th></tr>
<tr><td><code>ts_debug</code> ( [ <code>config regconfig</code>, ] <code>document text</code> ) → <code>setof record</code> ( <code>alias text</code>, <code>description text</code>, <code>token text</code>, <code>dictionaries regdictionary[]</code>, <code>dictionary regdictionary</code>, <code>lexemes text[]</code> )<br>Extracts and normalizes tokens from the <code>document</code> according to the specified or default text search configuration, and returns information about how each token was processed. See Section 12.8.1 for details.<br><code>ts_debug('english', 'The Brightest supernovaes')</code> → <code>(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...</code></td></tr>
</table>

| Function<br>Description<br>Example(s) |
|---|
| `ts_lexize` ( *dict* `regdictionary`, *token* `text` ) → `text[]`<br>Returns an array of replacement 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 not a known word. See Section 12.8.3 for details.<br>`ts_lexize('english_stem', 'stars')` → `{star}` |
| `ts_parse` ( *parser_name* `text`, *document* `text` ) → `setof record` ( *tokid* `integer`, *token* `text` )<br>Extracts tokens from the *document* using the named parser. See Section 12.8.2 for details.<br>`ts_parse('default', 'foo - bar')` → `(1,foo) ...` |
| `ts_parse` ( *parser_oid* `oid`, *document* `text` ) → `setof record` ( *tokid* `integer`, *token* `text` )<br>Extracts tokens from the *document* using a parser specified by OID. See Section 12.8.2 for details.<br>`ts_parse(3722, 'foo - bar')` → `(1,foo) ...` |
| `ts_token_type` ( *parser_name* `text` ) → `setof record` ( *tokid* `integer`, *alias* `text`, *description* `text` )<br>Returns a table that describes each type of token the named parser can recognize. See Section 12.8.2 for details.<br>`ts_token_type('default')` → `(1,asciiword,"Word, all ASCII") ...` |
| `ts_token_type` ( *parser_oid* `oid` ) → `setof record` ( *tokid* `integer`, *alias* `text`, *description* `text` )<br>Returns a table that describes each type of token a parser specified by OID can recognize. See Section 12.8.2 for details.<br>`ts_token_type(3722)` → `(1,asciiword,"Word, all ASCII") ...` |
| `ts_stat` ( *sqlquery* `text` [, *weights* `text` ] ) → `setof record` ( *word* `text`, *ndoc* `integer`, *nentry* `integer` )<br>Executes the *sqlquery*, which must return a single `tsvector` column, and returns statistics about each distinct lexeme contained in the data. See Section 12.4.4 for details.<br>`ts_stat('SELECT vector FROM apod')` → `(foo,10,15) ...` |

# 9.14. UUID Functions

Table 9.45 shows the PostgreSQL functions that can be used to generate UUIDs.

### Table 9.45. UUID Generation Functions

| Function<br>Description<br>Example(s) |
|---|
| `gen_random_uuid` → `uuid`<br>`uuidv4` → `uuid`<br>Generate a version 4 (random) UUID.<br>`gen_random_uuid()` → `5b30857f-0bfa-48b5-ac0b-5c64e28078d1`<br>`uuidv4()` → `b42410ee-132f-42ee-9e4f-09a6485c95b8` |
| `uuidv7` ( [ *shift* `interval` ] ) → `uuid` |

| Function<br>Description<br>Example(s) |
|---|
| Generate a version 7 (time-ordered) UUID. The timestamp is computed using UNIX timestamp with millisecond precision + sub-millisecond timestamp + random. The optional parameter *shift* will shift the computed timestamp by the given `interval`.<br>`uuidv7()` → `019535d9-3df7-79fb-b466-fa907fa17f9e` |

> **Note**
>
> The uuid-ossp module provides additional functions that implement other standard algorithms for generating UUIDs.

Table 9.46 shows the PostgreSQL functions that can be used to extract information from UUIDs.

**Table 9.46. UUID Extraction Functions**

| Function<br>Description<br>Example(s) |
|---|
| `uuid_extract_timestamp ( uuid )` → `timestamp with time zone`<br>Extracts a `timestamp with time zone` from UUID version 1 and 7. For other versions, this function returns null. Note that the extracted timestamp is not necessarily exactly equal to the time the UUID was generated; this depends on the implementation that generated the UUID.<br>`uuid_extract_timestamp('019535d9-3df7-79fb-b466-fa907fa17f9e'::uuid)` → `2025-02-23 21:46:24.503-05` |
| `uuid_extract_version ( uuid )` → `smallint`<br>Extracts the version from a UUID of the variant described by RFC 9562[2]. For other variants, this function returns null. For example, for a UUID generated by `gen_random_uuid`, this function will return 4.<br>`uuid_extract_version('41db1265-8bc1-4ab3-992f-885799a4af1d'::uuid)` → `4`<br>`uuid_extract_version('019535d9-3df7-79fb-b466-fa907fa17f9e'::uuid)` → `7` |

PostgreSQL also provides the usual comparison operators shown in Table 9.1 for UUIDs.

See Section 8.12 for details on the data type `uuid` in PostgreSQL.

# 9.15. XML Functions

The functions and function-like expressions described in this section operate on values of type `xml`. See Section 8.13 for information about the `xml` type. The function-like expressions `xmlparse` and `xmlserialize` for converting to and from type `xml` are documented there, not in this section.

Use of most of these functions requires PostgreSQL to have been built with `configure --with-libxml`.

## 9.15.1. Producing XML Content

A set of functions and function-like expressions is 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.

[2] https://datatracker.ietf.org/doc/html/rfc9562

### 9.15.1.1. `xmltext`

`xmltext ( text )` → `xml`

The function `xmltext` returns an XML value with a single text node containing the input argument as its content. Predefined entities like ampersand (&), left and right angle brackets (`< >`), and quotation marks (`""`) are escaped.

Example:

```
SELECT xmltext('< foo & bar >');
         xmltext
-------------------------
 &lt; foo &amp; bar &gt;
```

### 9.15.1.2. `xmlcomment`

`xmlcomment ( text )` → `xml`

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 “`-`”, otherwise the resulting construct would not be a valid XML comment. If the argument is null, the result is null.

Example:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

### 9.15.1.3. `xmlconcat`

`xmlconcat ( xml [, ...] )` → `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/>
```

### 9.15.1.4. xmlelement

```
xmlelement ( NAME name [, XMLATTRIBUTES ( attvalue [ AS attname ]
 [, ...] ) ] [, content [, ...]] ) → xml
```

The `xmlelement` expression produces an XML element with the given name, attributes, and content. The *name* and *attname* items shown in the syntax are simple identifiers, not values. The *attvalue* and *content* items are expressions, which can yield any PostgreSQL data type. The argument(s) within `XMLATTRIBUTES` generate attributes of the XML element; the *content* value(s) are concatenated to form its 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. The particular behavior for individual data types is expected to evolve in order to align the PostgreSQL mappings with those specified in SQL:2006 and later, as discussed in Section D.3.1.3.

## 9.15.1.5. `xmlforest`

`xmlforest ( content [ AS name ] [, ...] )` → `xml`

The `xmlforest` expression produces an XML forest (sequence) of elements using the given names and content. As for `xmlelement`, each *`name`* must be a simple identifier, while the *`content`* expressions can have any data type.

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`.

## 9.15.1.6. `xmlpi`

```
xmlpi ( NAME name [, content ] ) → xml
```

The `xmlpi` expression creates an XML processing instruction. As for `xmlelement`, the *`name`* must be a simple identifier, while the *`content`* expression can have any data type. The *`content`*, if present, must not contain the character sequence `?>`.

Example:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

## 9.15.1.7. `xmlroot`

```
xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO
 VALUE} ] ) → xml
```

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>
```

## 9.15.1.8. `xmlagg`

```
xmlagg ( xml ) → 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 9.21 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. 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>
```

## 9.15.2. XML Predicates

The expressions described in this section check properties of `xml` values.

### 9.15.2.1. IS DOCUMENT

`xml IS DOCUMENT → boolean`

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 about the difference between documents and content fragments.

### 9.15.2.2. IS NOT DOCUMENT

`xml IS NOT DOCUMENT → boolean`

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.

### 9.15.2.3. XMLEXISTS

`XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}] ) → boolean`

The function `xmlexists` evaluates an XPath 1.0 expression (the first argument), with the passed XML value as its context item. The function returns false if the result of that evaluation yields an empty node-set, true if it yields any other value. The function returns null if any argument is null. A nonnull value passed as the context item must be an XML document, not a content fragment or any non-XML value.

Example:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

The BY REF and BY VALUE clauses are accepted in PostgreSQL, but are ignored, as discussed in Section D.3.2.

In the SQL standard, the xmlexists function evaluates an expression in the XML Query language, but PostgreSQL allows only an XPath 1.0 expression, as discussed in Section D.3.1.

### 9.15.2.4. xml_is_well_formed

xml_is_well_formed ( text ) → boolean

xml_is_well_formed_document ( text ) → boolean

xml_is_well_formed_content ( text ) → boolean

These functions check whether a text string represents well-formed XML, returning a Boolean result. xml_is_well_formed_document checks for a well-formed document, while xml_is_well_formed_content checks for well-formed content. xml_is_well_formed does the former if the xmloption configuration parameter is set to DOCUMENT, or the latter if it is set to CONTENT. This means that xml_is_well_formed is 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.

## 9.15.3. 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.

### 9.15.3.1. xpath

`xpath ( xpath text, xml xml [, nsarray text[] ] ) → xml[]`

The function `xpath` evaluates the XPath 1.0 expression *`xpath`* (given as text) 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)
```

### 9.15.3.2. xpath_exists

`xpath_exists ( xpath text, xml xml [, nsarray text[] ] ) → boolean`

The function `xpath_exists` is a specialized form of the `xpath` function. Instead of returning the individual XML values that satisfy the XPath 1.0 expression, this function returns a Boolean indicating whether the query was satisfied or not (specifically, whether it produced any value other than an empty node-set). This function is equivalent to the `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)
```

### 9.15.3.3. xmltable

```
XMLTABLE (
    [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
    row_expression PASSING [BY {REF|VALUE}] document_expression [BY
 {REF|VALUE}]
    COLUMNS name { type [PATH column_expression]
 [DEFAULT default_expression] [NOT NULL | NULL]
                  | FOR ORDINALITY }
            [, ...]
) → setof record
```

The `xmltable` expression produces a table based on an XML value, an XPath filter to extract rows, and a set of column definitions. Although it syntactically resembles a function, it can only appear as a table in a query's `FROM` clause.

The optional `XMLNAMESPACES` clause gives a comma-separated list of namespace definitions, where each *namespace_uri* is a `text` expression and each *namespace_name* is a simple identifier. 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 1.0 expression (given as `text`) that is evaluated, passing the XML value *document_expression* as its context item, to obtain a set of XML nodes. These nodes are what `xmltable` transforms into output rows. No rows will be produced if the *document_expression* is null, nor if the *row_expression* produces an empty node-set or any value other than a node-set.

*document_expression* provides the context item for the *row_expression*. It must be a well-formed XML document; fragments/forests are not accepted. The `BY REF` and `BY VALUE` clauses are accepted but ignored, as discussed in Section D.3.2.

In the SQL standard, the `xmltable` function evaluates expressions in the XML Query language, but PostgreSQL allows only XPath 1.0 expressions, as discussed in Section D.3.1.

The required `COLUMNS` clause specifies the column(s) that will be produced in the output table. See the syntax summary above for the format. A name is required for each column, as is a data type (unless `FOR ORDINALITY` is specified, in which case type `integer` is implicit). The path, default and nullability clauses are optional.

A column marked `FOR ORDINALITY` will be populated with row numbers, starting with 1, in the order of nodes retrieved from the *row_expression*'s result node-set. At most one column may be marked `FOR ORDINALITY`.

> **Note**
>
> XPath 1.0 does not specify an order for nodes in a node-set, so code that relies on a particular order of the results will be implementation-dependent. Details can be found in Section D.3.1.2.

The *column_expression* for a column is an XPath 1.0 expression that is evaluated for each row, with the current node from the *row_expression* result as its context item, 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 a non-XML value (which is limited to string, boolean, or double in XPath 1.0) and the column has a PostgreSQL type other than `xml`, the column will be set as if by assigning the value's string representation to the PostgreSQL type. (If the value is a boolean, its string representation is taken to be `1` or `0` if the output column's type category is numeric, otherwise `true` or `false`.)

If a column's XPath expression returns a non-empty set of XML nodes and the column's PostgreSQL type is `xml`, the column will be assigned the expression result exactly, if it is of document or content form. [3]

A non-XML result assigned to an `xml` output column produces content, a single text node with the string value of the result. An XML result assigned to a column of any other type may not have more than one node, or an error is raised. If there is exactly one node, the column will be set as if by assigning the node's string value (as defined for the XPath 1.0 `string` function) to the PostgreSQL type.

The string value of an XML element is the concatenation, in document order, of all text nodes contained in that element and its descendants. The string value of an element with no descendant text nodes is an empty string (not `NULL`). Any `xsi:nil` attributes are ignored. 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. The XPath 1.0 `string` function may be consulted for the rules defining the string value of other XML node types and non-XML values.

The conversion rules presented here are not exactly those of the SQL standard, as discussed in Section D.3.1.3.

If the path expression returns an empty node-set (typically, when it does not match) for a given row, the column will be set to `NULL`, unless a *`default_expression`* is specified; then the value resulting from evaluating that expression is used.

A *`default_expression`*, rather than being evaluated immediately when `xmltable` is called, is evaluated each time a default is needed for the column. If the expression qualifies as stable or immutable, the repeat evaluation may be skipped. This means that you can usefully use volatile functions like `nextval` in *`default_expression`*.

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.

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;
```

[3] A result containing more than one element node at the top level, or non-whitespace text outside of an element, is an example of content form. An XPath result can be of neither form, for example if it returns an attribute node selected from the element that contains it. Such a result will be put into content form with each such disallowed node replaced by its string value, as defined for the XPath 1.0 `string` function.

```
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   bbbxxxCC
```

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)
```

## 9.15.4. 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 ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
                tableforest boolean, targetns text ) → xml
```

`table_to_xml` maps the content of the named table, passed as parameter *`table`*. The `regclass` type accepts strings identifying tables using the usual notation, including optional schema qualification and double quotes (see Section 8.19 for details). `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 ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

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 ( table regclass, nulls boolean,
                             tableforest boolean, targetns text ) → xml
query_to_xml_and_xmlschema ( query text, nulls boolean,
                             tableforest boolean, targetns text ) → xml
```

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 ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
                      tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
                              tableforest boolean, targetns text ) → xml

database_to_xml ( nulls boolean,
```

```
                  tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
                        tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
                                tableforest boolean, targetns text ) → xml
```

These functions ignore tables that are not readable by the current user. The database-wide functions additionally ignore schemas that the current user does not have `USAGE` (lookup) privilege for.

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, Example 9.1 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.

### Example 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>
```

# 9.16. JSON Functions and Operators

This section describes:

- functions and operators for processing and creating JSON data
- the SQL/JSON path language
- the SQL/JSON query functions

To provide native support for JSON data types within the SQL environment, PostgreSQL implements the *SQL/JSON data model*. This model comprises sequences of items. Each item can hold SQL scalar values, with an additional SQL/JSON null value, and composite data structures that use JSON arrays and objects. The model is a formalization of the implied data model in the JSON specification RFC 7159[4].

[4] https://datatracker.ietf.org/doc/html/rfc7159

SQL/JSON allows you to handle JSON data alongside regular SQL data, with transaction support, including:

- Uploading JSON data into the database and storing it in regular SQL columns as character or binary strings.
- Generating JSON objects and arrays from relational data.
- Querying JSON data using SQL/JSON query functions and SQL/JSON path language expressions.

To learn more about the SQL/JSON standard, see [sqltr-19075-6]. For details on JSON types supported in PostgreSQL, see Section 8.14.

## 9.16.1. Processing and Creating JSON Data

Table 9.47 shows the operators that are available for use with JSON data types (see Section 8.14). In addition, the usual comparison operators shown in Table 9.1 are available for `jsonb`, though not for `json`. The comparison operators follow the ordering rules for B-tree operations outlined in Section 8.14.4. See also Section 9.21 for the aggregate function `json_agg` which aggregates record values as JSON, 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`.

### Table 9.47. `json` and `jsonb` Operators

| Operator<br>Description<br>Example(s) |
|---|
| `json -> integer → json`<br>`jsonb -> integer → jsonb`<br>Extracts *n*'th element of JSON array (array elements are indexed from zero, but negative integers count from the end).<br>`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2` → `{"c":"baz"}`<br>`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3` → `{"a":"foo"}` |
| `json -> text → json`<br>`jsonb -> text → jsonb`<br>Extracts JSON object field with the given key.<br>`'{"a": {"b":"foo"}}'::json -> 'a'` → `{"b":"foo"}` |
| `json ->> integer → text`<br>`jsonb ->> integer → text`<br>Extracts *n*'th element of JSON array, as `text`.<br>`'[1,2,3]'::json ->> 2` → `3` |
| `json ->> text → text`<br>`jsonb ->> text → text`<br>Extracts JSON object field with the given key, as `text`.<br>`'{"a":1,"b":2}'::json ->> 'b'` → `2` |
| `json #> text[] → json`<br>`jsonb #> text[] → jsonb`<br>Extracts JSON sub-object at the specified path, where path elements can be either field keys or array indexes.<br>`'{"a": {"b": ["foo","bar"]}}'::json #> '{a,b,1}'` → `"bar"` |
| `json #>> text[] → text`<br>`jsonb #>> text[] → text` |

| Operator<br>Description<br>Example(s) |
|---|
| Extracts JSON sub-object at the specified path as `text`.<br>`'{"a": {"b": ["foo","bar"]}}'::json #>> '{a,b,1}'` → `bar` |

> **Note**
>
> 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 key or array element exists.

Some further operators exist only for `jsonb`, as shown in Table 9.48. Section 8.14.4 describes how these operators can be used to effectively search indexed `jsonb` data.

### Table 9.48. Additional `jsonb` Operators

| Operator<br>Description<br>Example(s) |
|---|
| `jsonb @> jsonb` → `boolean`<br>Does the first JSON value contain the second? (See Section 8.14.3 for details about containment.)<br>`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb` → `t` |
| `jsonb <@ jsonb` → `boolean`<br>Is the first JSON value contained in the second?<br>`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb` → `t` |
| `jsonb ? text` → `boolean`<br>Does the text string exist as a top-level key or array element within the JSON value?<br>`'{"a":1, "b":2}'::jsonb ? 'b'` → `t`<br>`'["a", "b", "c"]'::jsonb ? 'b'` → `t` |
| `jsonb ?\| text[]` → `boolean`<br>Do any of the strings in the text array exist as top-level keys or array elements?<br>`'{"a":1, "b":2, "c":3}'::jsonb ?\| array['b', 'd']` → `t` |
| `jsonb ?& text[]` → `boolean`<br>Do all of the strings in the text array exist as top-level keys or array elements?<br>`'["a", "b", "c"]'::jsonb ?& array['a', 'b']` → `t` |
| `jsonb \|\| jsonb` → `jsonb`<br>Concatenates two `jsonb` values. Concatenating two arrays generates an array containing all the elements of each input. Concatenating two objects generates an object containing the union of their keys, taking the second object's value when there are duplicate keys. All other cases are treated by converting a non-array input into a single-element array, and then proceeding as for two arrays. Does not operate recursively: only the top-level array or object structure is merged.<br>`'["a", "b"]'::jsonb \|\| '["a", "d"]'::jsonb` → `["a", "b", "a", "d"]`<br>`'{"a": "b"}'::jsonb \|\| '{"c": "d"}'::jsonb` → `{"a": "b", "c": "d"}`<br>`'[1, 2]'::jsonb \|\| '3'::jsonb` → `[1, 2, 3]`<br>`'{"a": "b"}'::jsonb \|\| '42'::jsonb` → `[{"a": "b"}, 42]`<br>To append an array to another array as a single entry, wrap it in an additional layer of array, for example: |

| Operator<br>Description<br>Example(s) |
|---|
| `'[1, 2]'::jsonb \|\| jsonb_build_array('[3, 4]'::jsonb)` → `[1, 2, [3, 4]]` |
| `jsonb - text` → `jsonb`<br>Deletes a key (and its value) from a JSON object, or matching string value(s) from a JSON array.<br>`'{"a": "b", "c": "d"}'::jsonb - 'a'` → `{"c": "d"}`<br>`'["a", "b", "c", "b"]'::jsonb - 'b'` → `["a", "c"]` |
| `jsonb - text[]` → `jsonb`<br>Deletes all matching keys or array elements from the left operand.<br>`'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]` → `{}` |
| `jsonb - integer` → `jsonb`<br>Deletes the array element with specified index (negative integers count from the end). Throws an error if JSON value is not an array.<br>`'["a", "b"]'::jsonb - 1` → `["a"]` |
| `jsonb #- text[]` → `jsonb`<br>Deletes the field or array element at the specified path, where path elements can be either field keys or array indexes.<br>`'["a", {"b":1}]'::jsonb #- '{1,b}'` → `["a", {}]` |
| `jsonb @? jsonpath` → `boolean`<br>Does JSON path return any item for the specified JSON value? (This is useful only with SQL-standard JSON path expressions, not predicate check expressions, since those always return a value.)<br>`'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)'` → `t` |
| `jsonb @@ jsonpath` → `boolean`<br>Returns the result of a JSON path predicate check for the specified JSON value. (This is useful only with predicate check expressions, not SQL-standard JSON path expressions, since it will return `NULL` if the path result is not a single boolean value.)<br>`'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2'` → `t` |

## Note

The `jsonpath` operators `@?` and `@@` suppress the following errors: missing object field or array element, unexpected JSON item type, datetime and numeric errors. The `jsonpath`-related functions described below can also be told to suppress these types of errors. This behavior might be helpful when searching JSON document collections of varying structure.

Table 9.49 shows the functions that are available for constructing `json` and `jsonb` values. Some functions in this table have a `RETURNING` clause, which specifies the data type returned. It must be one of `json`, `jsonb`, `bytea`, a character string type (`text`, `char`, or `varchar`), or a type that can be cast to `json`. By default, the `json` type is returned.

### Table 9.49. JSON Creation Functions

| Function<br>Description<br>Example(s) |
|---|
| `to_json ( anyelement )` → `json` |

| Function<br>Description<br>Example(s) |
|---|
| `to_jsonb` ( `anyelement` ) → `jsonb`<br>Converts any SQL value to `json` or `jsonb`. Arrays and composites are converted recursively to arrays and objects (multidimensional arrays become arrays of arrays in JSON). Otherwise, if there is a cast from the SQL data type to `json`, the cast function will be used to perform the conversion;[a] otherwise, a scalar JSON value is produced. For any scalar other than a number, a Boolean, or a null value, the text representation will be used, with escaping as necessary to make it a valid JSON string value.<br>`to_json('Fred said "Hi."'::text)` → `"Fred said \"Hi.\""`<br>`to_jsonb(row(42, 'Fred said "Hi."'::text))` → `{"f1": 42, "f2": "Fred said \"Hi.\""}` |
| `array_to_json` ( `anyarray` [, `boolean` ] ) → `json`<br>Converts an SQL array to a JSON array. The behavior is the same as `to_json` except that line feeds will be added between top-level array elements if the optional boolean parameter is true.<br>`array_to_json('{{1,5},{99,100}}'::int[])` → `[[1,5],[99,100]]` |
| `json_array` ( [ { *value_expression* [ `FORMAT JSON` ] } [, ...] ] [ { `NULL` \| `ABSENT` } `ON NULL` ] [ `RETURNING` *data_type* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ])<br>`json_array` ( [ *query_expression* ] [ `RETURNING` *data_type* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ])<br>Constructs a JSON array from either a series of *value_expression* parameters or from the results of *query_expression*, which must be a SELECT query returning a single column. If `ABSENT ON NULL` is specified, NULL values are ignored. This is always the case if a *query_expression* is used.<br>`json_array(1,true,json '{"a":null}')` → `[1, true, {"a":null}]`<br>`json_array(SELECT * FROM (VALUES(1),(2)) t)` → `[1, 2]` |
| `row_to_json` ( `record` [, `boolean` ] ) → `json`<br>Converts an SQL composite value to a JSON object. The behavior is the same as `to_json` except that line feeds will be added between top-level elements if the optional boolean parameter is true.<br>`row_to_json(row(1,'foo'))` → `{"f1":1,"f2":"foo"}` |
| `json_build_array` ( `VARIADIC "any"` ) → `json`<br>`jsonb_build_array` ( `VARIADIC "any"` ) → `jsonb`<br>Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list. Each argument is converted as per `to_json` or `to_jsonb`.<br>`json_build_array(1, 2, 'foo', 4, 5)` → `[1, 2, "foo", 4, 5]` |
| `json_build_object` ( `VARIADIC "any"` ) → `json`<br>`jsonb_build_object` ( `VARIADIC "any"` ) → `jsonb`<br>Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating keys and values. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`.<br>`json_build_object('foo', 1, 2, row(3,'bar'))` → `{"foo" : 1, "2" : {"f1":3,"f2":"bar"}}` |
| `json_object` ( [ { *key_expression* { `VALUE` \| ':' } *value_expression* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] }[, ...] ] [ { `NULL` \| `ABSENT` } `ON NULL` ] [ { `WITH` \| `WITHOUT` } `UNIQUE` [ `KEYS` ] ] [ `RETURNING` *data_type* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ])<br>Constructs a JSON object of all the key/value pairs given, or an empty object if none are given. *key_expression* is a scalar expression defining the JSON key, which is converted to the `text` type. It cannot be `NULL` nor can it belong to a type that has a cast to the `json` type. If `WITH UNIQUE KEYS` is specified, there must not be any duplicate *key_expression*. Any pair for |

| Function<br>Description<br>Example(s) |
|---|
| which the *value_expression* evaluates to `NULL` is omitted from the output if `ABSENT ON NULL` is specified; if `NULL ON NULL` is specified or the clause omitted, the key is included with value `NULL`.<br>`json_object('code' VALUE 'P123', 'title': 'Jaws')` → `{"code" : "P123", "title" : "Jaws"}` |
| `json_object` ( `text[]` ) → `json`<br>`jsonb_object` ( `text[]` ) → `jsonb`<br>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. All values are converted to JSON strings.<br>`json_object('{a, 1, b, "def", c, 3.5}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}`<br>`json_object('{{a, 1}, {b, "def"}, {c, 3.5}}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}` |
| `json_object` ( *keys* `text[]`, *values* `text[]` ) → `json`<br>`jsonb_object` ( *keys* `text[]`, *values* `text[]` ) → `jsonb`<br>This form of `json_object` takes keys and values pairwise from separate text arrays. Otherwise it is identical to the one-argument form.<br>`json_object('{a,b}', '{1,2}')` → `{"a": "1", "b": "2"}` |
| `json` ( *expression* [ `FORMAT JSON` [ `ENCODING UTF8` ]] [ { `WITH` \| `WITHOUT` } `UNIQUE` [ `KEYS` ]] ) → `json`<br>Converts a given expression specified as `text` or `bytea` string (in UTF8 encoding) into a JSON value. If *expression* is NULL, an SQL null value is returned. If `WITH UNIQUE` is specified, the *expression* must not contain any duplicate object keys.<br>`json('{"a":123, "b":[true,"foo"], "a":"bar"}')` → `{"a":123, "b":[true,"foo"], "a":"bar"}` |
| `json_scalar` ( *expression* )<br>Converts a given SQL scalar value into a JSON scalar value. If the input is NULL, an SQL null is returned. If the input is number or a boolean value, a corresponding JSON number or boolean value is returned. For any other value, a JSON string is returned.<br>`json_scalar(123.45)` → `123.45`<br>`json_scalar(CURRENT_TIMESTAMP)` → `"2022-05-10T10:51:04.62128-04:00"` |
| `json_serialize` ( *expression* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] [ `RETURNING` *data_type* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ] )<br>Converts an SQL/JSON expression into a character or binary string. The *expression* can be of any JSON type, any character string type, or `bytea` in UTF8 encoding. The returned type used in `RETURNING` can be any character string type or `bytea`. The default is `text`.<br>`json_serialize('{ "a" : 1 } ' RETURNING bytea)` → `\x7b20226122203a2031207d20` |

[a] For example, the hstore 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 9.50 details SQL/JSON facilities for testing JSON.

## Table 9.50. SQL/JSON Testing Functions

**Function signature**
**Description**
**Example(s)**

*expression* IS [ NOT ] JSON [ { VALUE | SCALAR | ARRAY | OBJECT } ] [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]

This predicate tests whether *expression* can be parsed as JSON, possibly of a specified type. If SCALAR or ARRAY or OBJECT is specified, the test is whether or not the JSON is of that particular type. If WITH UNIQUE KEYS is specified, then any object in the *expression* is also tested to see if it has duplicate keys.

```
SELECT js,
  js IS JSON "json?",
  js IS JSON SCALAR "scalar?",
  js IS JSON OBJECT "object?",
  js IS JSON ARRAY "array?"
FROM (VALUES
      ('123'), ('"abc"'), ('{"a": "b"}'), ('[1,2]'),('abc'))
 foo(js);
     js     | json? | scalar? | object? | array?
------------+-------+---------+---------+--------
 123        | t     | t       | f       | f
 "abc"      | t     | t       | f       | f
 {"a": "b"} | t     | f       | t       | f
 [1,2]      | t     | f       | f       | t
 abc        | f     | f       | f       | f
```

```
SELECT js,
  js IS JSON OBJECT "object?",
  js IS JSON ARRAY "array?",
  js IS JSON ARRAY WITH UNIQUE KEYS "array w. UK?",
  js IS JSON ARRAY WITHOUT UNIQUE KEYS "array w/o UK?"
FROM (VALUES ('[{"a":"1"},
 {"b":"2","b":"3"}]')) foo(js);
-[ RECORD 1 ]-+--------------------
js            | [{"a":"1"},        +
              |  {"b":"2","b":"3"}]
object?       | f
array?        | t
array w. UK?  | f
array w/o UK? | t
```

Table 9.51 shows the functions that are available for processing `json` and `jsonb` values.

## Table 9.51. JSON Processing Functions

**Function**
**Description**
**Example(s)**

`json_array_elements ( json ) → setof json`

`jsonb_array_elements ( jsonb ) → setof jsonb`

Expands the top-level JSON array into a set of JSON values.

`select * from json_array_elements('[1,true, [2,false]]') →`

| Function<br>Description<br>Example(s) |
|---|
| `   value`<br>`-----------`<br>` 1`<br>` true`<br>` [2,false]` |
| `json_array_elements_text` ( `json` ) → `setof text`<br>`jsonb_array_elements_text` ( `jsonb` ) → `setof text`<br>Expands the top-level JSON array into a set of `text` values.<br>`select * from json_array_elements_text('["foo", "bar"]')` →<br><br>`   value`<br>`-----------`<br>` foo`<br>` bar` |
| `json_array_length` ( `json` ) → `integer`<br>`jsonb_array_length` ( `jsonb` ) → `integer`<br>Returns the number of elements in the top-level JSON array.<br>`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')` → `5`<br>`jsonb_array_length('[]')` → `0` |
| `json_each` ( `json` ) → `setof record` ( *`key`* `text`, *`value`* `json` )<br>`jsonb_each` ( `jsonb` ) → `setof record` ( *`key`* `text`, *`value`* `jsonb` )<br>Expands the top-level JSON object into a set of key/value pairs.<br>`select * from json_each('{"a":"foo", "b":"bar"}')` →<br><br>` key \| value`<br>`-----+-------`<br>` a   \| "foo"`<br>` b   \| "bar"` |
| `json_each_text` ( `json` ) → `setof record` ( *`key`* `text`, *`value`* `text` )<br>`jsonb_each_text` ( `jsonb` ) → `setof record` ( *`key`* `text`, *`value`* `text` )<br>Expands the top-level JSON object into a set of key/value pairs. The returned *`value`*s will be of type `text`.<br>`select * from json_each_text('{"a":"foo", "b":"bar"}')` →<br><br>` key \| value`<br>`-----+-------`<br>` a   \| foo`<br>` b   \| bar` |
| `json_extract_path` ( *`from_json`* `json`, `VARIADIC` *`path_elems`* `text[]` ) → `json`<br>`jsonb_extract_path` ( *`from_json`* `jsonb`, `VARIADIC` *`path_elems`* `text[]` ) → `jsonb`<br>Extracts JSON sub-object at the specified path. (This is functionally equivalent to the `#>` operator, but writing the path out as a variadic list can be more convenient in some cases.) |

| Function<br>Description<br>Example(s) |
|---|
| `json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `"foo"` |
| `json_extract_path_text` ( *from_json* json, VARIADIC *path_elems* `text[]` ) → `text`<br>`jsonb_extract_path_text` ( *from_json* jsonb, VARIADIC *path_elems* `text[]` ) → `text`<br>Extracts JSON sub-object at the specified path as `text`. (This is functionally equivalent to the `#>>` operator.)<br>`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `foo` |
| `json_object_keys` ( `json` ) → `setof text`<br>`jsonb_object_keys` ( `jsonb` ) → `setof text`<br>Returns the set of keys in the top-level JSON object.<br>`select * from json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}')` →<br><br>`json_object_keys`<br>`------------------`<br>`f1`<br>`f2` |
| `json_populate_record` ( *base* `anyelement`, *from_json* `json` ) → `anyelement`<br>`jsonb_populate_record` ( *base* `anyelement`, *from_json* `jsonb` ) → `anyelement`<br>Expands the top-level JSON object to a row having the composite type of the *base* argument. The JSON object is scanned for fields whose names match column names of the output row type, and their values are inserted into those columns of the output. (Fields that do not correspond to any output column name are ignored.) In typical use, the value of *base* is just `NULL`, which means that any output columns that do not match any object field will be filled with nulls. However, if *base* isn't `NULL` then the values it contains will be used for unmatched columns.<br>To convert a JSON value to the SQL type of an output column, the following rules are applied in sequence:<br><br>• A JSON null value is converted to an SQL null in all cases.<br>• If the output column is of type `json` or `jsonb`, the JSON value is just reproduced exactly.<br>• 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.<br>• 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.<br>• Otherwise, if the JSON value is a string, the contents of the string are fed to the input conversion function for the column's data type.<br>• Otherwise, the ordinary text representation of the JSON value is fed to the input conversion function for the column's data type.<br><br>While the example below uses a constant JSON value, typical use would be to reference a `json` or `jsonb` column laterally from another table in the query's `FROM` clause. Writing `json_populate_record` in the `FROM` clause is good practice, since all of the extracted columns are available for use without duplicate function calls.<br>`create type subrowtype as (d int, e text); create type myrowtype as (a int, b text[], c subrowtype);`<br>`select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}')` → |

**Function**
**Description**
**Example(s)**

```
 a |   b       |      c
---+-----------+-------------
 1 | {2,"a b"} | (4,"a b c")
```

`jsonb_populate_record_valid` ( *base* `anyelement`, *from_json* `json` ) → `boolean`

Function for testing `jsonb_populate_record`. Returns `true` if the input `jsonb_populate_record` would finish without an error for the given input JSON object; that is, it's valid input, `false` otherwise.

```
create type jsb_char2 as (a char(2));
select jsonb_populate_record_valid(NULL::jsb_char2, '{"a": "aaa"}'); →

 jsonb_populate_record_valid
-----------------------------
 f
(1 row)
```

```
select * from jsonb_populate_record(NULL::jsb_char2, '{"a": "aaa"}') q; →

ERROR:  value too long for type character(2)
```

```
select jsonb_populate_record_valid(NULL::jsb_char2, '{"a": "aa"}'); →

 jsonb_populate_record_valid
-----------------------------
 t
(1 row)
```

```
select * from jsonb_populate_record(NULL::jsb_char2, '{"a": "aa"}') q; →

 a
----
 aa
(1 row)
```

`json_populate_recordset` ( *base* `anyelement`, *from_json* `json` ) → `setof anyelement`

`jsonb_populate_recordset` ( *base* `anyelement`, *from_json* `jsonb` ) → `setof anyelement`

Expands the top-level JSON array of objects to a set of rows having the composite type of the *base* argument. Each element of the JSON array is processed as described above for `json[b]_populate_record`.

```
create type twoints as (a int, b int);
select * from json_populate_recordset(null::twoints, '[{"a":1,"b":2}, {"a":3,"b":4}]') →
```

| Function<br>Description<br>Example(s) |
|---|
| ` a \| b`<br>`---+---`<br>` 1 \| 2`<br>` 3 \| 4` |
| `json_to_record` ( `json` ) → `record`<br>`jsonb_to_record` ( `jsonb` ) → `record`<br>Expands the top-level JSON object to a row having the composite type defined by an AS clause. (As with all functions returning `record`, the calling query must explicitly define the structure of the record with an AS clause.) The output record is filled from fields of the JSON object, in the same way as described above for `json[b]_populate_record`. Since there is no input record value, unmatched columns are always filled with nulls.<br>`create type myrowtype as (a int, b text);`<br>`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)` →<br><br>` a \|    b    \|    c    \| d \|       r`<br>`---+---------+---------+---+---------------`<br>` 1 \| [1,2,3] \| {1,2,3} \|   \| (123,"a b c")` |
| `json_to_recordset` ( `json` ) → `setof record`<br>`jsonb_to_recordset` ( `jsonb` ) → `setof record`<br>Expands the top-level JSON array of objects to a set of rows having the composite type defined by an AS clause. (As with all functions returning `record`, the calling query must explicitly define the structure of the record with an AS clause.) Each element of the JSON array is processed as described above for `json[b]_populate_record`.<br>`select * from json_to_recordset('[{"a":1,"b":"foo"}, {"a":"2","c":"bar"}]') as x(a int, b text)` →<br><br>` a \|  b`<br>`---+-----`<br>` 1 \| foo`<br>` 2 \|` |
| `jsonb_set` ( *`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean` ] ) → `jsonb`<br>Returns *`target`* with the item designated by *`path`* replaced by *`new_value`*, or with *`new_value`* added if *`create_if_missing`* is true (which is the default) and the item designated by *`path`* does not exist. All earlier steps in the path must exist, or the *`target`* is returned unchanged. As with the path oriented operators, negative integers that appear in the *`path`* count from the end of JSON arrays. If the last path step is an array index that is out of range, and *`create_if_missing`* is true, the new value is added at the beginning of the array if the index is negative, or at the end of the array if it is positive.<br>`jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', '[2,3,4]', false)` → `[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]`<br>`jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]')` → `[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]` |
| `jsonb_set_lax` ( *`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean` [, *`null_value_treatment`* `text` ]] ) → `jsonb` |

| Function<br>Description<br>Example(s) |
|---|
| If *new_value* is not NULL, behaves identically to jsonb_set. Otherwise behaves according to the value of *null_value_treatment* which must be one of 'raise_exception', 'use_json_null', 'delete_key', or 'return_target'. The default is 'use_json_null'.<br>jsonb_set_lax('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', null) → [{"f1": null, "f2": null}, 2, null, 3]<br>jsonb_set_lax('[{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target') → [{"f1": 99, "f2": null}, 2] |
| jsonb_insert ( *target* jsonb, *path* text[], *new_value* jsonb [, *insert_after* boolean ] ) → jsonb<br>Returns *target* with *new_value* inserted. If the item designated by the *path* is an array element, *new_value* will be inserted before that item if *insert_after* is false (which is the default), or after it if *insert_after* is true. If the item designated by the *path* is an object field, *new_value* will be inserted only if the object does not already contain that key. All earlier steps in the path must exist, or the *target* is returned unchanged. As with the path oriented operators, negative integers that appear in the *path* count from the end of JSON arrays. If the last path step is an array index that is out of range, the new value is added at the beginning of the array if the index is negative, or at the end of the array if it is positive.<br>jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"') → {"a": [0, "new_value", 1, 2]}<br>jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true) → {"a": [0, 1, "new_value", 2]} |
| json_strip_nulls ( *target* json [,*strip_in_arrays* boolean ] ) → json<br>jsonb_strip_nulls ( *target* jsonb [,*strip_in_arrays* boolean ] ) → jsonb<br>Deletes all object fields that have null values from the given JSON value, recursively. If *strip_in_arrays* is true (the default is false), null array elements are also stripped. Otherwise they are not stripped. Bare null values are never stripped.<br>json_strip_nulls('[{"f1":1, "f2":null}, 2, null, 3]') → [{"f1":1},2,null,3]<br>jsonb_strip_nulls('[1,2,null,3,4]', true); → [1,2,3,4] |
| jsonb_path_exists ( *target* jsonb, *path* jsonpath [, *vars* jsonb [, *silent* boolean ]] ) → boolean<br>Checks whether the JSON path returns any item for the specified JSON value. (This is useful only with SQL-standard JSON path expressions, not predicate check expressions, since those always return a value.) If the *vars* argument is specified, it must be a JSON object, and its fields provide named values to be substituted into the jsonpath expression. If the *silent* argument is specified and is true, the function suppresses the same errors as the @? and @@ operators do.<br>jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}') → t |
| jsonb_path_match ( *target* jsonb, *path* jsonpath [, *vars* jsonb [, *silent* boolean ]] ) → boolean<br>Returns the SQL boolean result of a JSON path predicate check for the specified JSON value. (This is useful only with predicate check expressions, not SQL-standard JSON path expressions, since it will either fail or return NULL if the path result is not a single boolean value.) The optional *vars* and *silent* arguments act the same as for jsonb_path_exists.<br>jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2, "max":4}') → t |

| Function<br>Description<br>Example(s) |
|---|
| `jsonb_path_query` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `setof jsonb`<br>Returns all JSON items returned by the JSON path for the specified JSON value. For SQL-standard JSON path expressions it returns the JSON values selected from *`target`*. For predicate check expressions it returns the result of the predicate check: `true`, `false`, or `null`. The optional *`vars`* and *`silent`* arguments act the same as for `jsonb_path_exists`.<br>`select * from jsonb_path_query('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` →<br>` jsonb_path_query`<br>`------------------`<br>` 2`<br>` 3`<br>` 4` |
| `jsonb_path_query_array` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `jsonb`<br>Returns all JSON items returned by the JSON path for the specified JSON value, as a JSON array. The parameters are the same as for `jsonb_path_query`.<br>`jsonb_path_query_array('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `[2, 3, 4]` |
| `jsonb_path_query_first` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `jsonb`<br>Returns the first JSON item returned by the JSON path for the specified JSON value, or `NULL` if there are no results. The parameters are the same as for `jsonb_path_query`.<br>`jsonb_path_query_first('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `2` |
| `jsonb_path_exists_tz` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `boolean`<br>`jsonb_path_match_tz` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `boolean`<br>`jsonb_path_query_tz` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `setof jsonb`<br>`jsonb_path_query_array_tz` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `jsonb`<br>`jsonb_path_query_first_tz` ( *`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean` ]] ) → `jsonb`<br>These functions act like their counterparts described above without the `_tz` suffix, except that these functions support comparisons of date/time values that require timezone-aware conversions. The example below requires interpretation of the date-only value `2015-08-02` as a timestamp with time zone, so the result depends on the current TimeZone setting. Due to this dependency, these functions are marked as stable, which means these functions cannot be used in indexes. Their counterparts are immutable, and so can be used in indexes; but they will throw errors if asked to make such comparisons.<br>`jsonb_path_exists_tz('["2015-08-01 12:00:00-05"]', '$[*] ? (@.datetime() < "2015-08-02".datetime())')` → `t` |
| `jsonb_pretty` ( `jsonb` ) → `text`<br>Converts the given JSON value to pretty-printed, indented text. |

| Function<br>Description<br>Example(s) |
|---|
| `jsonb_pretty('[{"f1":1,"f2":null}, 2]')` → <br><pre>[<br>    {<br>        "f1": 1,<br>        "f2": null<br>    },<br>    2<br>]</pre> |
| `json_typeof ( json )` → `text`<br>`jsonb_typeof ( jsonb )` → `text`<br>Returns the type of the top-level JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`. (The `null` result should not be confused with an SQL NULL; see the examples.)<br>`json_typeof('-123.4')` → `number`<br>`json_typeof('null'::json)` → `null`<br>`json_typeof(NULL::json) IS NULL` → `t` |

## 9.16.2. The SQL/JSON Path Language

SQL/JSON path expressions specify item(s) to be retrieved from a JSON value, similarly to XPath expressions used for access to XML content. In PostgreSQL, path expressions are implemented as the `jsonpath` data type and can use any elements described in Section 8.14.7.

JSON query functions and operators pass the provided path expression to the *path engine* for evaluation. If the expression matches the queried JSON data, the corresponding JSON item, or set of items, is returned. If there is no match, the result will be `NULL`, `false`, or an error, depending on the function. Path expressions are written in the SQL/JSON path language and can include arithmetic expressions and functions.

A path expression consists of a sequence of elements allowed by the `jsonpath` data type. The path expression is normally evaluated from left to right, but you can use parentheses to change the order of operations. If the evaluation is successful, a sequence of JSON items is produced, and the evaluation result is returned to the JSON query function that completes the specified computation.

To refer to the JSON value being queried (the *context item*), use the `$` variable in the path expression. The first element of a path must always be `$`. It can be followed by one or more accessor operators, which go down the JSON structure level by level to retrieve sub-items of the context item. Each accessor operator acts on the result(s) of the previous evaluation step, producing zero, one, or more output items from each input item.

For example, suppose you have some JSON data from a GPS tracker that you would like to parse, such as:

```
SELECT '{
  "track": {
    "segments": [
      {
        "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      {
        "location":   [ 47.706, 13.2635 ],
```

```
        "start time": "2018-10-14 10:39:21",
        "HR": 135
      }
    ]
  }
}' AS json \gset
```

(The above example can be copied-and-pasted into psql to set things up for the following examples. Then psql will expand `:'json'` into a suitably-quoted string constant containing the JSON value.)

To retrieve the available track segments, you need to use the `.key` accessor operator to descend through surrounding JSON objects, for example:

```
=> select jsonb_path_query(:'json', '$.track.segments');

 jsonb_path_query
-----------------------------------------------------------
-----------------------------------------------------------
---------------------------------------------
 [{"HR": 73, "location": [47.763, 13.4034], "start time": "2018-10-14
 10:05:14"}, {"HR": 135, "location": [47.706, 13.2635], "start time":
 "2018-10-14 10:39:21"}]
```

To retrieve the contents of an array, you typically use the `[*]` operator. The following example will return the location coordinates for all the available track segments:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

Here we started with the whole JSON input value (`$`), then the `.track` accessor selected the JSON object associated with the `"track"` object key, then the `.segments` accessor selected the JSON array associated with the `"segments"` key within that object, then the `[*]` accessor selected each element of that array (producing a series of items), then the `.location` accessor selected the JSON array associated with the `"location"` key within each of those objects. In this example, each of those objects had a `"location"` key; but if any of them did not, the `.location` accessor would have simply produced no output for that input item.

To return the coordinates of the first segment only, you can specify the corresponding subscript in the `[]` accessor operator. Recall that JSON array indexes are 0-relative:

```
=> select jsonb_path_query(:'json', '$.track.segments[0].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
```

The result of each path evaluation step can be processed by one or more of the `jsonpath` operators and methods listed in Section 9.16.2.3. Each method name must be preceded by a dot. For example, you can get the size of an array:

```
=> select jsonb_path_query(:'json', '$.track.segments.size()');
 jsonb_path_query
------------------
 2
```

More examples of using `jsonpath` operators and methods within path expressions appear below in Section 9.16.2.3.

A path can also contain *filter expressions* that work similarly to the `WHERE` clause in SQL. A filter expression begins with a question mark and provides a condition in parentheses:

```
? (condition)
```

Filter expressions must be written just after the path evaluation step to which they should apply. The result of that step is filtered to include only those items that satisfy the provided condition. SQL/JSON defines three-valued logic, so the condition can produce `true`, `false`, or `unknown`. The `unknown` value plays the same role as SQL `NULL` and can be tested for with the `is unknown` predicate. Further path evaluation steps use only those items for which the filter expression returned `true`.

The functions and operators that can be used in filter expressions are listed in Table 9.53. Within a filter expression, the `@` variable denotes the value being considered (i.e., one result of the preceding path step). You can write accessor operators after `@` to retrieve component items.

For example, suppose you would like to retrieve all heart rate values higher than 130. You can achieve this as follows:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR ? (@ > 130)');
 jsonb_path_query
------------------
 135
```

To get the start times of segments with such values, you have to filter out irrelevant segments before selecting the start times, so the filter expression is applied to the previous step, and the path used in the condition is different:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.HR >
 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

You can use several filter expressions in sequence, if required. The following example selects start times of all segments that contain locations with relevant coordinates and high heart rate values:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.location[1] <
 13.4) ? (@.HR > 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

Using filter expressions at different nesting levels is also allowed. The following example first filters all segments by location, and then returns high heart rate values for these segments, if available:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.location[1] <
 13.4).HR ? (@ > 130)');
 jsonb_path_query
------------------
 135
```

You can also nest filter expressions within each other. This example returns the size of the track if it contains any segments with high heart rate values, or an empty sequence otherwise:

```
=> select jsonb_path_query(:'json', '$.track ? (exists(@.segments[*] ?
 (@.HR > 130))).segments.size()');
 jsonb_path_query
------------------
 2
```

### 9.16.2.1. Deviations from the SQL Standard

PostgreSQL's implementation of the SQL/JSON path language has the following deviations from the SQL/JSON standard.

#### 9.16.2.1.1. Boolean Predicate Check Expressions

As an extension to the SQL standard, a PostgreSQL path expression can be a Boolean predicate, whereas the SQL standard allows predicates only within filters. While SQL-standard path expressions return the relevant element(s) of the queried JSON value, predicate check expressions return the single three-valued `jsonb` result of the predicate: `true`, `false`, or `null`. For example, we could write this SQL-standard filter expression:

```
=> select jsonb_path_query(:'json', '$.track.segments ?(@[*].HR > 130)');
                                jsonb_path_query
-----------------------------------------------------------
---------------------
 {"HR": 135, "location": [47.706, 13.2635], "start time": "2018-10-14
 10:39:21"}
```

The similar predicate check expression simply returns `true`, indicating that a match exists:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR > 130');
 jsonb_path_query
------------------
 true
```

> **Note**
>
> Predicate check expressions are required in the `@@` operator (and the `jsonb_path_match` function), and should not be used with the `@?` operator (or the `jsonb_path_exists` function).

#### 9.16.2.1.2. Regular Expression Interpretation

There are minor differences in the interpretation of regular expression patterns used in `like_regex` filters, as described in Section 9.16.2.4.

### 9.16.2.2. Strict and Lax Modes

When you query JSON data, the path expression may not match the actual JSON data structure. An attempt to access a non-existent member of an object or element of an array is defined as a structural error. SQL/JSON path expressions have two modes of handling structural errors:

- lax (default) — the path engine implicitly adapts the queried data to the specified path. Any structural errors that cannot be fixed as described below are suppressed, producing no match.
- strict — if a structural error occurs, an error is raised.

Lax mode facilitates matching of a JSON document and path expression when the JSON data does not conform to the expected schema. If an operand does not match the requirements of a particular operation, it can be automatically wrapped as an SQL/JSON array, or unwrapped by converting its elements into an SQL/JSON sequence before performing the operation. Also, comparison operators automatically unwrap their operands in lax mode, so you can compare SQL/JSON arrays out-of-the-box. An array of size 1 is considered equal to its sole element. Automatic unwrapping is not performed when:

- The path expression contains `type()` or `size()` methods that return the type and the number of elements in the array, respectively.
- The queried JSON data contain nested arrays. In this case, only the outermost array is unwrapped, while all the inner arrays remain unchanged. Thus, implicit unwrapping can only go one level down within each path evaluation step.

For example, when querying the GPS data listed above, you can abstract from the fact that it stores an array of segments when using lax mode:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments.location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

In strict mode, the specified path must exactly match the structure of the queried JSON document, so using this path expression will cause an error:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments.location');
ERROR:  jsonpath member accessor can only be applied to an object
```

To get the same result as in lax mode, you have to explicitly unwrap the `segments` array:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

The unwrapping behavior of lax mode can lead to surprising results. For instance, the following query using the `.**` accessor selects every `HR` value twice:

```
=> select jsonb_path_query(:'json', 'lax $.**.HR');
 jsonb_path_query
------------------
 73
 135
 73
 135
```

This happens because the `.**` accessor selects both the `segments` array and each of its elements, while the `.HR` accessor automatically unwraps arrays when using lax mode. To avoid surprising results, we recommend using the `.**` accessor only in strict mode. The following query selects each `HR` value just once:

```
=> select jsonb_path_query(:'json', 'strict $.**.HR');
 jsonb_path_query
------------------
 73
 135
```

The unwrapping of arrays can also lead to unexpected results. Consider this example, which selects all the `location` arrays:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
(2 rows)
```

As expected it returns the full arrays. But applying a filter expression causes the arrays to be unwrapped to evaluate each item, returning only the items that match the expression:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments[*].location ?
(@[*] > 15)');
 jsonb_path_query
------------------
 47.763
 47.706
(2 rows)
```

This despite the fact that the full arrays are selected by the path expression. Use strict mode to restore selecting the arrays:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments[*].location ?
(@[*] > 15)');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
(2 rows)
```

### 9.16.2.3. SQL/JSON Path Operators and Methods

Table 9.52 shows the operators and methods available in `jsonpath`. Note that while the unary operators and methods can be applied to multiple values resulting from a preceding path step, the binary operators (addition etc.) can only be applied to single values. In lax mode, methods applied to an array will be executed for each value in the array. The exceptions are `.type()` and `.size()`, which apply to the array itself.

**Table 9.52. `jsonpath` Operators and Methods**

| Operator/Method<br>Description<br>Example(s) |
|---|
| *number* + *number* → *number*<br>Addition<br>`jsonb_path_query('[2]', '$[0] + 3')` → `5` |
| + *number* → *number*<br>Unary plus (no operation); unlike addition, this can iterate over multiple values<br>`jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x')` → `[2, 3, 4]` |
| *number* - *number* → *number*<br>Subtraction<br>`jsonb_path_query('[2]', '7 - $[0]')` → `5` |
| - *number* → *number*<br>Negation; unlike subtraction, this can iterate over multiple values |

| Operator/Method<br>Description<br>Example(s) |
|---|
| `jsonb_path_query_array('{"x": [2,3,4]}', '- $.x')` → `[-2, -3, -4]` |
| *`number`* `*` *`number`* → *`number`*<br>Multiplication<br>`jsonb_path_query('[4]', '2 * $[0]')` → `8` |
| *`number`* `/` *`number`* → *`number`*<br>Division<br>`jsonb_path_query('[8.5]', '$[0] / 2')` → `4.2500000000000000` |
| *`number`* `%` *`number`* → *`number`*<br>Modulo (remainder)<br>`jsonb_path_query('[32]', '$[0] % 10')` → `2` |
| *`value`* `. type()` → *`string`*<br>Type of the JSON item (see `json_typeof`)<br>`jsonb_path_query_array('[1, "2", {}]', '$[*].type()')` → `["number", "string", "object"]` |
| *`value`* `. size()` → *`number`*<br>Size of the JSON item (number of array elements, or 1 if not an array)<br>`jsonb_path_query('{"m": [11, 15]}', '$.m.size()')` → `2` |
| *`value`* `. boolean()` → *`boolean`*<br>Boolean value converted from a JSON boolean, number, or string<br>`jsonb_path_query_array('[1, "yes", false]', '$[*].boolean()')` → `[true, true, false]` |
| *`value`* `. string()` → *`string`*<br>String value converted from a JSON boolean, number, string, or datetime<br>`jsonb_path_query_array('[1.23, "xyz", false]', '$[*].string()')` → `["1.23", "xyz", "false"]`<br>`jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp().string()')` → `"2023-08-15T12:34:56"` |
| *`value`* `. double()` → *`number`*<br>Approximate floating-point number converted from a JSON number or string<br>`jsonb_path_query('{"len": "1.9"}', '$.len.double() * 2')` → `3.8` |
| *`number`* `. ceiling()` → *`number`*<br>Nearest integer greater than or equal to the given number<br>`jsonb_path_query('{"h": 1.3}', '$.h.ceiling()')` → `2` |
| *`number`* `. floor()` → *`number`*<br>Nearest integer less than or equal to the given number<br>`jsonb_path_query('{"h": 1.7}', '$.h.floor()')` → `1` |
| *`number`* `. abs()` → *`number`*<br>Absolute value of the given number<br>`jsonb_path_query('{"z": -0.3}', '$.z.abs()')` → `0.3` |
| *`value`* `. bigint()` → *`bigint`*<br>Big integer value converted from a JSON number or string |

| Operator/Method<br>Description<br>Example(s) |
|---|
| `jsonb_path_query('{"len": "9876543219"}', '$.len.bigint()')` → `9876543219` |
| *`value`* `. decimal( [ ` *`precision`* ` [ , ` *`scale`* ` ] ] )` → *`decimal`*<br>Rounded decimal value converted from a JSON number or string (`precision` and `scale` must be integer values)<br>`jsonb_path_query('1234.5678', '$.decimal(6, 2)')` → `1234.57` |
| *`value`* `. integer()` → *`integer`*<br>Integer value converted from a JSON number or string<br>`jsonb_path_query('{"len": "12345"}', '$.len.integer()')` → `12345` |
| *`value`* `. number()` → *`numeric`*<br>Numeric value converted from a JSON number or string<br>`jsonb_path_query('{"len": "123.45"}', '$.len.number()')` → `123.45` |
| *`string`* `. datetime()` → *`datetime_type`* (see note)<br>Date/time value converted from a string<br>`jsonb_path_query('["2015-8-1", "2015-08-12"]', '$[*] ? (@.datetime() < "2015-08-2".datetime())')` → `"2015-8-1"` |
| *`string`* `. datetime(`*`template`*`)` → *`datetime_type`* (see note)<br>Date/time value converted from a string using the specified `to_timestamp` template<br>`jsonb_path_query_array('["12:30", "18:40"]', '$[*].datetime("HH24:MI")')` → `["12:30:00", "18:40:00"]` |
| *`string`* `. date()` → *`date`*<br>Date value converted from a string<br>`jsonb_path_query('"2023-08-15"', '$.date()')` → `"2023-08-15"` |
| *`string`* `. time()` → *`time without time zone`*<br>Time without time zone value converted from a string<br>`jsonb_path_query('"12:34:56"', '$.time()')` → `"12:34:56"` |
| *`string`* `. time(`*`precision`*`)` → *`time without time zone`*<br>Time without time zone value converted from a string, with fractional seconds adjusted to the given precision<br>`jsonb_path_query('"12:34:56.789"', '$.time(2)')` → `"12:34:56.79"` |
| *`string`* `. time_tz()` → *`time with time zone`*<br>Time with time zone value converted from a string<br>`jsonb_path_query('"12:34:56 +05:30"', '$.time_tz()')` → `"12:34:56+05:30"` |
| *`string`* `. time_tz(`*`precision`*`)` → *`time with time zone`*<br>Time with time zone value converted from a string, with fractional seconds adjusted to the given precision<br>`jsonb_path_query('"12:34:56.789 +05:30"', '$.time_tz(2)')` → `"12:34:56.79+05:30"` |
| *`string`* `. timestamp()` → *`timestamp without time zone`*<br>Timestamp without time zone value converted from a string |

| Operator/Method<br>Description<br>Example(s) |
|---|
| `jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp()')` → `"2023-08-15T12:34:56"` |
| *`string`* `. timestamp(`*`precision`*`)` → *`timestamp without time zone`*<br>Timestamp without time zone value converted from a string, with fractional seconds adjusted to the given precision<br>`jsonb_path_query('"2023-08-15 12:34:56.789"', '$.timestamp(2)')` → `"2023-08-15T12:34:56.79"` |
| *`string`* `. timestamp_tz()` → *`timestamp with time zone`*<br>Timestamp with time zone value converted from a string<br>`jsonb_path_query('"2023-08-15 12:34:56 +05:30"', '$.timestamp_tz()')` → `"2023-08-15T12:34:56+05:30"` |
| *`string`* `. timestamp_tz(`*`precision`*`)` → *`timestamp with time zone`*<br>Timestamp with time zone value converted from a string, with fractional seconds adjusted to the given precision<br>`jsonb_path_query('"2023-08-15 12:34:56.789 +05:30"', '$.timestamp_tz(2)')` → `"2023-08-15T12:34:56.79+05:30"` |
| *`object`* `. keyvalue()` → *`array`*<br>The object's key-value pairs, represented as an array of objects containing three fields: `"key"`, `"value"`, and `"id"`; `"id"` is a unique identifier of the object the key-value pair belongs to<br>`jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()')` → `[{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]` |

## Note

The result type of the `datetime()` and `datetime(`*`template`*`)` methods can be `date`, `timetz`, `time`, `timestamptz`, or `timestamp`. Both methods determine their result type dynamically.

The `datetime()` method sequentially tries to match its input string to the ISO formats for `date`, `timetz`, `time`, `timestamptz`, and `timestamp`. It stops on the first matching format and emits the corresponding data type.

The `datetime(`*`template`*`)` method determines the result type according to the fields used in the provided template string.

The `datetime()` and `datetime(`*`template`*`)` methods use the same parsing rules as the `to_timestamp` SQL function does (see Section 9.8), with three exceptions. First, these methods don't allow unmatched template patterns. Second, only the following separators are allowed in the template string: minus sign, period, solidus (slash), comma, apostrophe, semicolon, colon and space. Third, separators in the template string must exactly match the input string.

If different date/time types need to be compared, an implicit cast is applied. A `date` value can be cast to `timestamp` or `timestamptz`, `timestamp` can be cast to `timestamptz`, and `time` to `timetz`. However, all but the first of these conversions depend on the current TimeZone setting, and thus can only be performed within timezone-aware `jsonpath` functions. Similarly, other date/time-related methods that convert strings to date/time types also do this casting, which may involve the current TimeZone setting. Therefore, these conversions can also only be performed within timezone-aware `jsonpath` functions.

Table 9.53 shows the available filter expression elements.

### Table 9.53. `jsonpath` Filter Expression Elements

| Predicate/Value | Description | Example(s) |
|---|---|---|
| *value* == *value* → boolean | Equality comparison (this, and the other comparison operators, work on all JSON scalar values) | `jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)')` → `[1, 1]`<br>`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")')` → `["a"]` |
| *value* != *value* → boolean<br>*value* <> *value* → boolean | Non-equality comparison | `jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)')` → `[2, 3]`<br>`jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")')` → `["a", "c"]` |
| *value* < *value* → boolean | Less-than comparison | `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)')` → `[1]` |
| *value* <= *value* → boolean | Less-than-or-equal-to comparison | `jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <= "b")')` → `["a", "b"]` |
| *value* > *value* → boolean | Greater-than comparison | `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ > 2)')` → `[3]` |
| *value* >= *value* → boolean | Greater-than-or-equal-to comparison | `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ >= 2)')` → `[2, 3]` |
| true → boolean | JSON constant `true` | `jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == true)')` → `{"name": "Chris", "parent": true}` |
| false → boolean | JSON constant `false` | `jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)')` → `{"name": "John", "parent": false}` |
| null → *value* | JSON constant `null` (note that, unlike in SQL, comparison to `null` works normally) | `jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name')` → `"Mary"` |
| *boolean* && *boolean* → boolean | | |

| Predicate/Value<br>Description<br>Example(s) |
|---|
| Boolean AND<br>`jsonb_path_query('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)')` → `3` |
| *`boolean`* `\|\|` *`boolean`* → `boolean`<br>Boolean OR<br>`jsonb_path_query('[1, 3, 7]', '$[*] ? (@ < 1 \|\| @ > 5)')` → `7` |
| `!` *`boolean`* → `boolean`<br>Boolean NOT<br>`jsonb_path_query('[1, 3, 7]', '$[*] ? (!(@ < 5))')` → `7` |
| *`boolean`* `is unknown` → `boolean`<br>Tests whether a Boolean condition is `unknown`.<br>`jsonb_path_query('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)')` → `"foo"` |
| *`string`* `like_regex` *`string`* [ `flag` *`string`* ] → `boolean`<br>Tests whether the first operand matches the regular expression given by the second operand, optionally with modifications described by a string of `flag` characters (see Section 9.16.2.4).<br>`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")')` → `["abc", "abdacb"]`<br>`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")')` → `["abc", "aBdC", "abdacb"]` |
| *`string`* `starts with` *`string`* → `boolean`<br>Tests whether the second operand is an initial substring of the first operand.<br>`jsonb_path_query('["John Smith", "Mary Stone", "Bob Johnson"]', '$[*] ? (@ starts with "John")')` → `"John Smith"` |
| `exists (` *`path_expression`* `)` → `boolean`<br>Tests whether a path expression matches at least one SQL/JSON item. Returns `unknown` if the path expression would result in an error; the second example uses this to avoid a no-such-key error in strict mode.<br>`jsonb_path_query('{"x": [1, 2], "y": [2, 4]}', 'strict $.* ? (exists (@ ? (@[*] > 2)))')` → `[2, 4]`<br>`jsonb_path_query_array('{"value": 41}', 'strict $ ? (exists (@.name)) .name')` → `[]` |

### 9.16.2.4. SQL/JSON Regular Expressions

SQL/JSON path expressions allow matching text to a regular expression with the `like_regex` filter. For example, the following SQL/JSON path query would case-insensitively match all strings in an array that start with an English vowel:

```
$[*] ? (@ like_regex "^[aeiou]" flag "i")
```

The optional `flag` string may include one or more of the characters `i` for case-insensitive match, `m` to allow `^` and `$` to match at newlines, `s` to allow `.` to match a newline, and `q` to quote the whole pattern (reducing the behavior to a simple substring match).

The SQL/JSON standard borrows its definition for regular expressions from the `LIKE_REGEX` operator, which in turn uses the XQuery standard. PostgreSQL does not currently support the `LIKE_REGEX` operator. Therefore, the `like_regex` filter is implemented using the POSIX regular expression engine described in Section 9.7.3. This leads to various minor discrepancies from standard SQL/JSON behavior, which are cataloged in Section 9.7.3.8. Note, however, that the flag-letter incompatibilities described there do not apply to SQL/JSON, as it translates the XQuery flag letters to match what the POSIX engine expects.

Keep in mind that the pattern argument of `like_regex` is a JSON path string literal, written according to the rules given in Section 8.14.7. This means in particular that any backslashes you want to use in the regular expression must be doubled. For example, to match string values of the root document that contain only digits:

```
$.* ? (@ like_regex "^\\d+$")
```

## 9.16.3. SQL/JSON Query Functions

SQL/JSON functions `JSON_EXISTS()`, `JSON_QUERY()`, and `JSON_VALUE()` described in Table 9.54 can be used to query JSON documents. Each of these functions apply a *`path_expression`* (an SQL/JSON path query) to a *`context_item`* (the document). See Section 9.16.2 for more details on what the *`path_expression`* can contain. The *`path_expression`* can also reference variables, whose values are specified with their respective names in the `PASSING` clause that is supported by each function. *`context_item`* can be a `jsonb` value or a character string that can be successfully cast to `jsonb`.

### Table 9.54. SQL/JSON Query Functions

| Function signature<br>Description<br>Example(s) |
|---|
| `JSON_EXISTS (`<br>*`context_item`*`,` *`path_expression`*<br>`[ PASSING {` *`value`* `AS` *`varname`* `} [, ...]]`<br>`[{ TRUE \| FALSE \| UNKNOWN \| ERROR } ON ERROR ])` → `boolean`<br>• Returns true if the SQL/JSON *`path_expression`* applied to the *`context_item`* yields any items, false otherwise.<br>• The `ON ERROR` clause specifies the behavior if an error occurs during *`path_expression`* evaluation. Specifying `ERROR` will cause an error to be thrown with the appropriate message. Other options include returning `boolean` values `FALSE` or `TRUE` or the value `UNKNOWN` which is actually an SQL NULL. The default when no `ON ERROR` clause is specified is to return the `boolean` value `FALSE`.<br>Examples:<br>`JSON_EXISTS(jsonb '{"key1": [1,2,3]}', 'strict $.key1[*] ? (@ > $x)' PASSING 2 AS x)` → `t`<br>`JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'lax $.a[5]' ERROR ON ERROR)` → `f`<br>`JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'strict $.a[5]' ERROR ON ERROR)` →<br>`ERROR:  jsonpath array subscript is out of bounds` |
| `JSON_QUERY (`<br>*`context_item`*`,` *`path_expression`*<br>`[ PASSING {` *`value`* `AS` *`varname`* `} [, ...]]`<br>`[ RETURNING` *`data_type`* `[ FORMAT JSON [ ENCODING UTF8 ] ] ]` |

| Function signature<br>Description<br>Example(s) |
|---|
| `[ { WITHOUT \| WITH { CONDITIONAL \| [UNCONDITIONAL] } } [ ARRAY ] WRAPPER ]`<br>`[ { KEEP \| OMIT } QUOTES [ ON SCALAR STRING ] ]`<br>`[ { ERROR \| NULL \| EMPTY { [ ARRAY ] \| OBJECT } \| DEFAULT` *`expression`* `} ON EMPTY ]`<br>`[ { ERROR \| NULL \| EMPTY { [ ARRAY ] \| OBJECT } \| DEFAULT` *`expression`* `} ON ERROR ])` → `jsonb`<br>• Returns the result of applying the SQL/JSON *`path_expression`* to the *`context_item`*.<br>• By default, the result is returned as a value of type `jsonb`, though the `RETURNING` clause can be used to return as some other type to which it can be successfully coerced.<br>• If the path expression may return multiple values, it might be necessary to wrap those values using the `WITH WRAPPER` clause to make it a valid JSON string, because the default behavior is to not wrap them, as if `WITHOUT WRAPPER` were specified. The `WITH WRAPPER` clause is by default taken to mean `WITH UNCONDITIONAL WRAPPER`, which means that even a single result value will be wrapped. To apply the wrapper only when multiple values are present, specify `WITH CONDITIONAL WRAPPER`. Getting multiple values in result will be treated as an error if `WITHOUT WRAPPER` is specified.<br>• If the result is a scalar string, by default, the returned value will be surrounded by quotes, making it a valid JSON value. It can be made explicit by specifying `KEEP QUOTES`. Conversely, quotes can be omitted by specifying `OMIT QUOTES`. To ensure that the result is a valid JSON value, `OMIT QUOTES` cannot be specified when `WITH WRAPPER` is also specified.<br>• The `ON EMPTY` clause specifies the behavior if evaluating *`path_expression`* yields an empty set. The `ON ERROR` clause specifies the behavior if an error occurs when evaluating *`path_expression`*, when coercing the result value to the `RETURNING` type, or when evaluating the `ON EMPTY` expression if the *`path_expression`* evaluation returns an empty set.<br>• For both `ON EMPTY` and `ON ERROR`, specifying `ERROR` will cause an error to be thrown with the appropriate message. Other options include returning an SQL NULL, an empty array (`EMPTY [ARRAY]`), an empty object (`EMPTY OBJECT`), or a user-specified expression (`DEFAULT` *`expression`*) that can be coerced to jsonb or the type specified in `RETURNING`. The default when `ON EMPTY` or `ON ERROR` is not specified is to return an SQL NULL value.<br>Examples:<br>`JSON_QUERY(jsonb '[1,[2,3],null]', 'lax $[*][$off]' PASSING 1 AS off WITH CONDITIONAL WRAPPER)` → `3`<br>`JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' OMIT QUOTES)` → `[1, 2]`<br>`JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' RETURNING int[] OMIT QUOTES ERROR ON ERROR)` →<br><br>`ERROR:  malformed array literal: "[1, 2]"`<br>`DETAIL:  Missing "]" after array dimensions.` |
| `JSON_VALUE (`<br>*`context_item`*`,` *`path_expression`*<br>`[ PASSING {` *`value`* `AS` *`varname`* `} [, ...]]`<br>`[ RETURNING` *`data_type`* `]`<br>`[ { ERROR \| NULL \| DEFAULT` *`expression`* `} ON EMPTY ]`<br>`[ { ERROR \| NULL \| DEFAULT` *`expression`* `} ON ERROR ])` → `text` |

| Function signature<br>Description<br>Example(s) |
|---|
| • Returns the result of applying the SQL/JSON *path_expression* to the *context_item*.<br>• Only use JSON_VALUE() if the extracted value is expected to be a single SQL/JSON scalar item; getting multiple values will be treated as an error. If you expect that extracted value might be an object or an array, use the JSON_QUERY function instead.<br>• By default, the result, which must be a single scalar value, is returned as a value of type text, though the RETURNING clause can be used to return as some other type to which it can be successfully coerced.<br>• The ON ERROR and ON EMPTY clauses have similar semantics as mentioned in the description of JSON_QUERY, except the set of values returned in lieu of throwing an error is different.<br>• Note that scalar strings returned by JSON_VALUE always have their quotes removed, equivalent to specifying OMIT QUOTES in JSON_QUERY.<br>Examples:<br>JSON_VALUE(jsonb '"123.45"', '$' RETURNING float) → 123.45<br>JSON_VALUE(jsonb '"03:04 2015-02-01"', '$.datetime("HH24:MI YYYY-MM-DD")' RETURNING date) → 2015-02-01<br>JSON_VALUE(jsonb '[1,2]', 'strict $[$off]' PASSING 1 as off) → 2<br>JSON_VALUE(jsonb '[1,2]', 'strict $[*]' DEFAULT 9 ON ERROR) → 9 |

### Note

The *context_item* expression is converted to jsonb by an implicit cast if the expression is not already of type jsonb. Note, however, that any parsing errors that occur during that conversion are thrown unconditionally, that is, are not handled according to the (specified or implicit) ON ERROR clause.

### Note

JSON_VALUE() returns an SQL NULL if *path_expression* returns a JSON null, whereas JSON_QUERY() returns the JSON null as is.

## 9.16.4. JSON_TABLE

JSON_TABLE is an SQL/JSON function which queries JSON data and presents the results as a relational view, which can be accessed as a regular SQL table. You can use JSON_TABLE inside the FROM clause of a SELECT, UPDATE, or DELETE and as data source in a MERGE statement.

Taking JSON data as input, JSON_TABLE uses a JSON path expression to extract a part of the provided data to use as a *row pattern* for the constructed view. Each SQL/JSON value given by the row pattern serves as source for a separate row in the constructed view.

To split the row pattern into columns, JSON_TABLE provides the COLUMNS clause that defines the schema of the created view. For each column, a separate JSON path expression can be specified to be evaluated against the row pattern to get an SQL/JSON value that will become the value for the specified column in a given output row.

JSON data stored at a nested level of the row pattern can be extracted using the NESTED PATH clause. Each NESTED PATH clause can be used to generate one or more columns using the data from a nested level of the row pattern. Those columns can be specified using a `COLUMNS` clause that looks similar to the top-level COLUMNS clause. Rows constructed from NESTED COLUMNS are called *child rows* and are joined against the row constructed from the columns specified in the parent `COLUMNS` clause to get the row in the final view. Child columns themselves may contain a `NESTED PATH` specification thus allowing to extract data located at arbitrary nesting levels. Columns produced by multiple `NESTED PATH`s at the same level are considered to be *siblings* of each other and their rows after joining with the parent row are combined using UNION.

The rows produced by `JSON_TABLE` are laterally joined to the row that generated them, so you do not have to explicitly join the constructed view with the original table holding JSON data.

The syntax is:

```
JSON_TABLE (
    context_item, path_expression [ AS json_path_name ] [ PASSING { value
 AS varname } [, ...] ]
    COLUMNS ( json_table_column [, ...] )
    [ { ERROR | EMPTY [ARRAY]} ON ERROR ]
)


where json_table_column is:

  name FOR ORDINALITY
  | name type
        [ FORMAT JSON [ENCODING UTF8]]
        [ PATH path_expression ]
        [ { WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } } [ ARRAY ]
 WRAPPER ]
        [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
        [ { ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT expression
 } ON EMPTY ]
        [ { ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT expression
 } ON ERROR ]
  | name type EXISTS [ PATH path_expression ]
        [ { ERROR | TRUE | FALSE | UNKNOWN } ON ERROR ]
  | NESTED [ PATH ] path_expression [ AS json_path_name ] COLUMNS
 ( json_table_column [, ...] )
```

Each syntax element is described below in more detail.

*`context_item`*, *`path_expression`* `[ AS` *`json_path_name`* `] [ PASSING {` *`value`* `AS` *`varname`* `} [, ...]]`

The *`context_item`* specifies the input document to query, the *`path_expression`* is an SQL/JSON path expression defining the query, and *`json_path_name`* is an optional name for the *`path_expression`*. The optional `PASSING` clause provides data values for the variables mentioned in the *`path_expression`*. The result of the input data evaluation using the aforementioned elements is called the *row pattern*, which is used as the source for row values in the constructed view.

`COLUMNS` ( *`json_table_column`* [, ...] )

The `COLUMNS` clause defining the schema of the constructed view. In this clause, you can specify each column to be filled with an SQL/JSON value obtained by applying a JSON path expression against the row pattern. *`json_table_column`* has the following variants:

*`name`* `FOR ORDINALITY`

Adds an ordinality column that provides sequential row numbering starting from 1. Each `NESTED PATH` (see below) gets its own counter for any nested ordinality columns.

`name type [FORMAT JSON [ENCODING UTF8]] [ PATH path_expression ]`

Inserts an SQL/JSON value obtained by applying *`path_expression`* against the row pattern into the view's output row after coercing it to specified *`type`*.

Specifying `FORMAT JSON` makes it explicit that you expect the value to be a valid `json` object. It only makes sense to specify `FORMAT JSON` if *`type`* is one of `bpchar`, `bytea`, `character varying`, `name`, `json`, `jsonb`, `text`, or a domain over these types.

Optionally, you can specify `WRAPPER` and `QUOTES` clauses to format the output. Note that specifying `OMIT QUOTES` overrides `FORMAT JSON` if also specified, because unquoted literals do not constitute valid `json` values.

Optionally, you can use `ON EMPTY` and `ON ERROR` clauses to specify whether to throw the error or return the specified value when the result of JSON path evaluation is empty and when an error occurs during JSON path evaluation or when coercing the SQL/JSON value to the specified type, respectively. The default for both is to return a `NULL` value.

> **Note**
>
> This clause is internally turned into and has the same semantics as `JSON_VALUE` or `JSON_QUERY`. The latter if the specified type is not a scalar type or if either of `FORMAT JSON`, `WRAPPER`, or `QUOTES` clause is present.

`name type EXISTS [ PATH path_expression ]`

Inserts a boolean value obtained by applying *`path_expression`* against the row pattern into the view's output row after coercing it to specified *`type`*.

The value corresponds to whether applying the `PATH` expression to the row pattern yields any values.

The specified *`type`* should have a cast from the `boolean` type.

Optionally, you can use `ON ERROR` to specify whether to throw the error or return the specified value when an error occurs during JSON path evaluation or when coercing SQL/JSON value to the specified type. The default is to return a boolean value `FALSE`.

> **Note**
>
> This clause is internally turned into and has the same semantics as `JSON_EXISTS`.

`NESTED [ PATH ] path_expression [ AS json_path_name ] COLUMNS ( json_table_column [, ...] )`

Extracts SQL/JSON values from nested levels of the row pattern, generates one or more columns as defined by the `COLUMNS` subclause, and inserts the extracted SQL/JSON values into those columns. The *`json_table_column`* expression in the `COLUMNS` subclause uses the same syntax as in the parent `COLUMNS` clause.

The `NESTED PATH` syntax is recursive, so you can go down multiple nested levels by specifying several `NESTED PATH` subclauses within each other. It allows to unnest the hierarchy of JSON objects and arrays in a single function invocation rather than chaining several `JSON_TABLE` expressions in an SQL statement.

> **Note**
>
> In each variant of *json_table_column* described above, if the PATH clause is omitted, path expression `$.name` is used, where *name* is the provided column name.

AS *json_path_name*

The optional *json_path_name* serves as an identifier of the provided *path_expression*. The name must be unique and distinct from the column names.

{ ERROR | EMPTY } ON ERROR

The optional ON ERROR can be used to specify how to handle errors when evaluating the top-level *path_expression*. Use ERROR if you want the errors to be thrown and EMPTY to return an empty table, that is, a table containing 0 rows. Note that this clause does not affect the errors that occur when evaluating columns, for which the behavior depends on whether the ON ERROR clause is specified against a given column.

Examples

In the examples that follow, the following table containing JSON data will be used:

```
CREATE TABLE my_films ( js jsonb );

INSERT INTO my_films VALUES (
'{ "favorites" : [
   { "kind" : "comedy", "films" : [
     { "title" : "Bananas",
       "director" : "Woody Allen"},
     { "title" : "The Dinner Game",
       "director" : "Francis Veber" } ] },
   { "kind" : "horror", "films" : [
     { "title" : "Psycho",
       "director" : "Alfred Hitchcock" } ] },
   { "kind" : "thriller", "films" : [
     { "title" : "Vertigo",
       "director" : "Alfred Hitchcock" } ] },
   { "kind" : "drama", "films" : [
     { "title" : "Yojimbo",
       "director" : "Akira Kurosawa" } ] }
  ] }');
```

The following query shows how to use JSON_TABLE to turn the JSON objects in the my_films table to a view containing columns for the keys kind, title, and director contained in the original JSON along with an ordinality column:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE (js, '$.favorites[*]' COLUMNS (
   id FOR ORDINALITY,
   kind text PATH '$.kind',
   title text PATH '$.films[*].title' WITH WRAPPER,
   director text PATH '$.films[*].director' WITH WRAPPER)) AS jt;
```

```
 id |   kind   |             title              |             director
```

```
----+----------+-------------------------------
+--------------------------------
  1 | comedy   | ["Bananas", "The Dinner Game"] | ["Woody Allen", "Francis
 Veber"]
  2 | horror   | ["Psycho"]                     | ["Alfred Hitchcock"]
  3 | thriller | ["Vertigo"]                    | ["Alfred Hitchcock"]
  4 | drama    | ["Yojimbo"]                    | ["Akira Kurosawa"]
(4 rows)
```

The following is a modified version of the above query to show the usage of `PASSING` arguments in the filter specified in the top-level JSON path expression and the various options for the individual columns:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE (js, '$.favorites[*] ? (@.films[*].director == $filter)'
   PASSING 'Alfred Hitchcock' AS filter
     COLUMNS (
     id FOR ORDINALITY,
     kind text PATH '$.kind',
     title text FORMAT JSON PATH '$.films[*].title' OMIT QUOTES,
     director text PATH '$.films[*].director' KEEP QUOTES)) AS jt;


 id |   kind   |  title  |      director
----+----------+---------+--------------------
  1 | horror   | Psycho  | "Alfred Hitchcock"
  2 | thriller | Vertigo | "Alfred Hitchcock"
(2 rows)
```

The following is a modified version of the above query to show the usage of `NESTED PATH` for populating title and director columns, illustrating how they are joined to the parent columns id and kind:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*] ? (@.films[*].director == $filter)'
   PASSING 'Alfred Hitchcock' AS filter
   COLUMNS (
    id FOR ORDINALITY,
    kind text PATH '$.kind',
    NESTED PATH '$.films[*]' COLUMNS (
      title text FORMAT JSON PATH '$.title' OMIT QUOTES,
      director text PATH '$.director' KEEP QUOTES))) AS jt;


 id |   kind   |  title  |      director
----+----------+---------+--------------------
  1 | horror   | Psycho  | "Alfred Hitchcock"
  2 | thriller | Vertigo | "Alfred Hitchcock"
(2 rows)
```

The following is the same query but without the filter in the root path:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*]'
   COLUMNS (
    id FOR ORDINALITY,
```

```
    kind text PATH '$.kind',
    NESTED PATH '$.films[*]' COLUMNS (
      title text FORMAT JSON PATH '$.title' OMIT QUOTES,
      director text PATH '$.director' KEEP QUOTES))) AS jt;
```

```
 id |   kind   |      title       |      director
----+----------+------------------+--------------------
  1 | comedy   | Bananas          | "Woody Allen"
  1 | comedy   | The Dinner Game  | "Francis Veber"
  2 | horror   | Psycho           | "Alfred Hitchcock"
  3 | thriller | Vertigo          | "Alfred Hitchcock"
  4 | drama    | Yojimbo          | "Akira Kurosawa"
(5 rows)
```

The following shows another query using a different `JSON` object as input. It shows the UNION "sibling join" between `NESTED` paths `$.movies[*]` and `$.books[*]` and also the usage of `FOR ORDINALITY` column at `NESTED` levels (columns `movie_id`, `book_id`, and `author_id`):

```
SELECT * FROM JSON_TABLE (
'{"favorites":
    [{"movies":
      [{"name": "One", "director": "John Doe"},
       {"name": "Two", "director": "Don Joe"}],
     "books":
      [{"name": "Mystery", "authors": [{"name": "Brown Dan"}]},
       {"name": "Wonder", "authors": [{"name": "Jun Murakami"},
 {"name":"Craig Doe"}]}]
}]}'::json, '$.favorites[*]'
COLUMNS (
  user_id FOR ORDINALITY,
  NESTED '$.movies[*]'
    COLUMNS (
    movie_id FOR ORDINALITY,
    mname text PATH '$.name',
    director text),
  NESTED '$.books[*]'
    COLUMNS (
      book_id FOR ORDINALITY,
      bname text PATH '$.name',
      NESTED '$.authors[*]'
        COLUMNS (
          author_id FOR ORDINALITY,
          author_name text PATH '$.name'))));
```

```
 user_id | movie_id | mname | director | book_id |  bname  | author_id |
 author_name
---------+----------+-------+----------+---------+---------+-----------
+--------------
       1 |        1 | One   | John Doe |         |         |           |
       1 |        2 | Two   | Don Joe  |         |         |           |
       1 |          |       |          |       1 | Mystery |         1 |
 Brown Dan
       1 |          |       |          |       2 | Wonder  |         1 |
 Jun Murakami
       1 |          |       |          |       2 | Wonder  |         2 |
 Craig Doe
```

```
(5 rows)
```

# 9.17. 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. Sequence objects are commonly used to generate unique identifiers for rows of a table. The sequence functions, listed in Table 9.55, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

### Table 9.55. Sequence Functions

| Function<br>Description |
|---|
| `nextval` ( `regclass` ) → `bigint`<br>Advances the sequence object to its next value and returns that value. This is done atomically: even if multiple sessions execute `nextval` concurrently, each will safely receive a distinct sequence value. If the 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 appropriate parameters in the CREATE SEQUENCE command.<br>This function requires `USAGE` or `UPDATE` privilege on the sequence. |
| `setval` ( `regclass`, `bigint` [, `boolean` ] ) → `bigint`<br>Sets the sequence object's current value, and optionally its `is_called` flag. 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 that will be reported by `currval` is also set to the specified value. In the three-parameter form, `is_called` can be set to either `true` or `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,<br><br>`SELECT setval('myseq', 42);           Next nextval will return 43`<br>`SELECT setval('myseq', 42, true);     Same as above`<br>`SELECT setval('myseq', 42, false);    Next nextval will return 42`<br><br>The result returned by `setval` is just the value of its second argument.<br>This function requires `UPDATE` privilege on the sequence. |
| `currval` ( `regclass` ) → `bigint`<br>Returns 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.<br>This function requires `USAGE` or `SELECT` privilege on the sequence. |
| `lastval` () → `bigint`<br>Returns 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.<br>This function requires `USAGE` or `SELECT` privilege on the last used sequence. |

> **Caution**
>
> To avoid blocking concurrent transactions that obtain numbers from the same sequence, the value obtained by `nextval` is not reclaimed for re-use if the calling transaction later aborts. This

means that transaction aborts or database crashes can result in gaps in the sequence of assigned values. That can happen without a transaction abort, too. For example an INSERT with an ON CONFLICT clause 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. Thus, PostgreSQL sequence objects *cannot be used to obtain "gapless" sequences*.

Likewise, sequence state changes made by setval are immediately visible to other transactions, and are not undone if the calling transaction rolls back.

If the database cluster crashes before committing a transaction containing a nextval or setval call, the sequence state change might not have made its way to persistent storage, so that it is uncertain whether the sequence will have its original or updated state after the cluster restarts. This is harmless for usage of the sequence within the database, since other effects of uncommitted transactions will not be visible either. However, if you wish to use a sequence value for persistent outside-the-database purposes, make sure that the nextval call has been committed before doing so.

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. See Section 8.19 for details.

# 9.18. 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 server-side function in a more expressive programming language.

### Note

Although COALESCE, GREATEST, and LEAST are syntactically similar to functions, they are not ordinary functions, and thus cannot be used with explicit VARIADIC array arguments.

## 9.18.1. 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 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, 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.

## 9.18.2. 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)`.

The arguments must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details).

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.

## 9.18.3. 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.

The two arguments must be of comparable types. To be specific, they are compared exactly as if you had written *`value1`*` = `*`value2`*, so there must be a suitable = operator available.

The result has the same type as the first argument — but there is a subtlety. What is actually returned is the first argument of the implied = operator, and in some cases that will have been promoted to match the second argument's type. For example, `NULLIF(1, 2.2)` yields `numeric`, because there is no `integer` = `numeric` operator, only `numeric` = `numeric`.

## 9.18.4. 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 for details).

NULL values in the argument list are ignored. The result will be NULL only if all the expressions evaluate to NULL. (This is a deviation from the SQL standard. According to the standard, the return value is NULL if any argument is NULL. Some other databases behave this way.)

# 9.19. Array Functions and Operators

Table 9.56 shows the specialized operators available for array types. In addition to those, the usual comparison operators shown in Table 9.1 are available for arrays. The comparison operators compare the array contents element-by-element, using the default B-tree comparison function for the element data type, and sort based on the first difference. 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.

### Table 9.56. Array Operators

| Operator<br>Description<br>Example(s) |
|---|
| `anyarray @> anyarray` → `boolean`<br>Does the first array contain the second, that is, does each element appearing in the second array equal some element of the first array? (Duplicates are not treated specially, thus `ARRAY[1]` and `ARRAY[1,1]` are each considered to contain the other.)<br>`ARRAY[1,4,3] @> ARRAY[3,1,3]` → `t` |
| `anyarray <@ anyarray` → `boolean`<br>Is the first array contained by the second?<br>`ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]` → `t` |
| `anyarray && anyarray` → `boolean`<br>Do the arrays overlap, that is, have any elements in common?<br>`ARRAY[1,4,3] && ARRAY[2,1]` → `t` |
| `anycompatiblearray \|\| anycompatiblearray` → `anycompatiblearray`<br>Concatenates the two arrays. Concatenating a null or empty array is a no-op; otherwise the arrays must have the same number of dimensions (as illustrated by the first example) or differ in number of dimensions by one (as illustrated by the second). If the arrays are not of identical element types, they will be coerced to a common type (see Section 10.5).<br>`ARRAY[1,2,3] \|\| ARRAY[4,5,6,7]` → `{1,2,3,4,5,6,7}`<br>`ARRAY[1,2,3] \|\| ARRAY[[4,5,6],[7,8,9.9]]` → `{{1,2,3},{4,5,6},{7,8,9.9}}` |
| `anycompatible \|\| anycompatiblearray` → `anycompatiblearray`<br>Concatenates an element onto the front of an array (which must be empty or one-dimensional).<br>`3 \|\| ARRAY[4,5,6]` → `{3,4,5,6}` |
| `anycompatiblearray \|\| anycompatible` → `anycompatiblearray`<br>Concatenates an element onto the end of an array (which must be empty or one-dimensional).<br>`ARRAY[4,5,6] \|\| 7` → `{4,5,6,7}` |

See Section 8.15 for more details about array operator behavior. See Section 11.2 for more details about which operators support indexed operations.

Table 9.57 shows the functions available for use with array types. See Section 8.15 for more information and examples of the use of these functions.

### Table 9.57. Array Functions

| Function<br>Description<br>Example(s) |
|---|
| `array_append` ( `anycompatiblearray`, `anycompatible` ) → `anycompatiblearray`<br>Appends an element to the end of an array (same as the `anycompatiblearray \|\| anycompatible` operator).<br>`array_append(ARRAY[1,2], 3)` → `{1,2,3}` |
| `array_cat` ( `anycompatiblearray`, `anycompatiblearray` ) → `anycompatiblearray`<br>Concatenates two arrays (same as the `anycompatiblearray \|\| anycompatiblearray` operator).<br>`array_cat(ARRAY[1,2,3], ARRAY[4,5])` → `{1,2,3,4,5}` |
| `array_dims` ( `anyarray` ) → `text`<br>Returns a text representation of the array's dimensions.<br>`array_dims(ARRAY[[1,2,3], [4,5,6]])` → `[1:2][1:3]` |
| `array_fill` ( `anyelement`, `integer[]` [, `integer[]` ] ) → `anyarray`<br>Returns an array filled with copies of the given value, having dimensions of the lengths specified by the second argument. The optional third argument supplies lower-bound values for each dimension (which default to all `1`).<br>`array_fill(11, ARRAY[2,3])` → `{{11,11,11},{11,11,11}}`<br>`array_fill(7, ARRAY[3], ARRAY[2])` → `[2:4]={7,7,7}` |
| `array_length` ( `anyarray`, `integer` ) → `integer`<br>Returns the length of the requested array dimension. (Produces NULL instead of 0 for empty or missing array dimensions.)<br>`array_length(array[1,2,3], 1)` → `3`<br>`array_length(array[]::int[], 1)` → `NULL`<br>`array_length(array['text'], 2)` → `NULL` |
| `array_lower` ( `anyarray`, `integer` ) → `integer`<br>Returns the lower bound of the requested array dimension.<br>`array_lower('[0:2]={1,2,3}'::integer[], 1)` → `0` |
| `array_ndims` ( `anyarray` ) → `integer`<br>Returns the number of dimensions of the array.<br>`array_ndims(ARRAY[[1,2,3], [4,5,6]])` → `2` |
| `array_position` ( `anycompatiblearray`, `anycompatible` [, `integer` ] ) → `integer`<br>Returns the subscript of the first occurrence of the second argument in the array, or `NULL` if it's not present. If the third argument is given, the search begins at that subscript. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to search for `NULL`.<br>`array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon')` → `2` |
| `array_positions` ( `anycompatiblearray`, `anycompatible` ) → `integer[]` |

| Function<br>Description<br>Example(s) |
|---|
| Returns an array of the subscripts of all occurrences of the second argument in the array given as first argument. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to search for `NULL`. `NULL` is returned only if the array is `NULL`; if the value is not found in the array, an empty array is returned.<br>`array_positions(ARRAY['A','A','B','A'], 'A')` → `{1,2,4}` |
| `array_prepend` ( `anycompatible`, `anycompatiblearray` ) → `anycompatiblearray`<br>Prepends an element to the beginning of an array (same as the `anycompatible` `\|\|` `anycompatiblearray` operator).<br>`array_prepend(1, ARRAY[2,3])` → `{1,2,3}` |
| `array_remove` ( `anycompatiblearray`, `anycompatible` ) → `anycompatiblearray`<br>Removes all elements equal to the given value from the array. The array must be one-dimensional. Comparisons are done using `IS NOT DISTINCT FROM` semantics, so it is possible to remove `NULL`s.<br>`array_remove(ARRAY[1,2,3,2], 2)` → `{1,3}` |
| `array_replace` ( `anycompatiblearray`, `anycompatible`, `anycompatible` ) → `anycompatiblearray`<br>Replaces each array element equal to the second argument with the third argument.<br>`array_replace(ARRAY[1,2,5,4], 5, 3)` → `{1,2,3,4}` |
| `array_reverse` ( `anyarray` ) → `anyarray`<br>Reverses the first dimension of the array.<br>`array_reverse(ARRAY[[1,2],[3,4],[5,6]])` → `{{5,6},{3,4},{1,2}}` |
| `array_sample` ( *`array`* `anyarray`, *`n`* `integer` ) → `anyarray`<br>Returns an array of *`n`* items randomly selected from *`array`*. *`n`* may not exceed the length of *`array`*'s first dimension. If *`array`* is multi-dimensional, an “item” is a slice having a given first subscript.<br>`array_sample(ARRAY[1,2,3,4,5,6], 3)` → `{2,6,1}`<br>`array_sample(ARRAY[[1,2],[3,4],[5,6]], 2)` → `{{5,6},{1,2}}` |
| `array_shuffle` ( `anyarray` ) → `anyarray`<br>Randomly shuffles the first dimension of the array.<br>`array_shuffle(ARRAY[[1,2],[3,4],[5,6]])` → `{{5,6},{1,2},{3,4}}` |
| `array_sort` ( *`array`* `anyarray` [, *`descending`* `boolean` [, *`nulls_first`* `boolean` ]] ) → `anyarray`<br>Sorts the first dimension of the array. The sort order is determined by the default sort ordering of the array's element type; however, if the element type is collatable, the collation to use can be specified by adding a `COLLATE` clause to the *`array`* argument.<br>If *`descending`* is true then sort in descending order, otherwise ascending order. If omitted, the default is ascending order. If *`nulls_first`* is true then nulls appear before non-null values, otherwise nulls appear after non-null values. If omitted, *`nulls_first`* is taken to have the same value as *`descending`*.<br>`array_sort(ARRAY[[2,4],[2,1],[6,5]])` → `{{2,1},{2,4},{6,5}}` |
| `array_to_string` ( *`array`* `anyarray`, *`delimiter`* `text` [, *`null_string`* `text` ] ) → `text`<br>Converts each array element to its text representation, and concatenates those separated by the *`delimiter`* string. If *`null_string`* is given and is not `NULL`, then `NULL` array entries are represented by that string; otherwise, they are omitted. See also `string_to_array`.<br>`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')` → `1,2,3,*,5` |

| Function<br>Description<br>Example(s) |
|---|
| `array_upper` ( `anyarray`, `integer` ) → `integer`<br>Returns the upper bound of the requested array dimension.<br>`array_upper(ARRAY[1,8,3,7], 1)` → `4` |
| `cardinality` ( `anyarray` ) → `integer`<br>Returns the total number of elements in the array, or 0 if the array is empty.<br>`cardinality(ARRAY[[1,2],[3,4]])` → `4` |
| `trim_array` ( *array* `anyarray`, *n* `integer` ) → `anyarray`<br>Trims an array by removing the last *n* elements. If the array is multidimensional, only the first dimension is trimmed.<br>`trim_array(ARRAY[1,2,3,4,5,6], 2)` → `{1,2,3,4}` |
| `unnest` ( `anyarray` ) → `setof anyelement`<br>Expands an array into a set of rows. The array's elements are read out in storage order.<br>`unnest(ARRAY[1,2])` →<br>`1`<br>`2`<br>`unnest(ARRAY[['foo','bar'],['baz','quux']])` →<br>`foo`<br>`bar`<br>`baz`<br>`quux` |
| `unnest` ( `anyarray`, `anyarray` [, ... ] ) → `setof anyelement, anyelement [, ... ]`<br>Expands multiple arrays (possibly of different data types) into a set of rows. If the arrays are not all the same length then the shorter ones are padded with `NULL`s. This form is only allowed in a query's FROM clause; see Section 7.2.1.4.<br>`select * from unnest(ARRAY[1,2], ARRAY['foo','bar','baz']) as x(a,b)` →<br>` a \|  b`<br>`---+-----`<br>` 1 \| foo`<br>` 2 \| bar`<br>`   \| baz` |

See also Section 9.21 about the aggregate function `array_agg` for use with arrays.

# 9.20. Range/Multirange Functions and Operators

See Section 8.17 for an overview of range types.

Table 9.58 shows the specialized operators available for range types. Table 9.59 shows the specialized operators available for multirange types. In addition to those, the usual comparison operators shown in Table 9.1 are available for range and multirange types. The comparison operators order first by the range lower bounds, and only if those are equal do they compare the upper bounds. The multirange operators compare each range until one is unequal. This does not usually result in a useful overall ordering, but the operators are provided to allow unique indexes to be constructed on ranges.

**Table 9.58. Range Operators**

<table>
<tr><th>Operator<br>Description<br>Example(s)</th></tr>
<tr><td><code>anyrange @> anyrange → boolean</code><br>Does the first range contain the second?<br><code>int4range(2,4) @> int4range(2,3) → t</code></td></tr>
<tr><td><code>anyrange @> anyelement → boolean</code><br>Does the range contain the element?<br><code>'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp → t</code></td></tr>
<tr><td><code>anyrange <@ anyrange → boolean</code><br>Is the first range contained by the second?<br><code>int4range(2,4) <@ int4range(1,7) → t</code></td></tr>
<tr><td><code>anyelement <@ anyrange → boolean</code><br>Is the element contained in the range?<br><code>42 <@ int4range(1,7) → f</code></td></tr>
<tr><td><code>anyrange && anyrange → boolean</code><br>Do the ranges overlap, that is, have any elements in common?<br><code>int8range(3,7) && int8range(4,12) → t</code></td></tr>
<tr><td><code>anyrange << anyrange → boolean</code><br>Is the first range strictly left of the second?<br><code>int8range(1,10) << int8range(100,110) → t</code></td></tr>
<tr><td><code>anyrange >> anyrange → boolean</code><br>Is the first range strictly right of the second?<br><code>int8range(50,60) >> int8range(20,30) → t</code></td></tr>
<tr><td><code>anyrange &< anyrange → boolean</code><br>Does the first range not extend to the right of the second?<br><code>int8range(1,20) &< int8range(18,20) → t</code></td></tr>
<tr><td><code>anyrange &> anyrange → boolean</code><br>Does the first range not extend to the left of the second?<br><code>int8range(7,20) &> int8range(5,10) → t</code></td></tr>
<tr><td><code>anyrange -|- anyrange → boolean</code><br>Are the ranges adjacent?<br><code>numrange(1.1,2.2) -|- numrange(2.2,3.3) → t</code></td></tr>
<tr><td><code>anyrange + anyrange → anyrange</code><br>Computes the union of the ranges. The ranges must overlap or be adjacent, so that the union is a single range (but see <code>range_merge()</code>).<br><code>numrange(5,15) + numrange(10,20) → [5,20)</code></td></tr>
<tr><td><code>anyrange * anyrange → anyrange</code><br>Computes the intersection of the ranges.</td></tr>
</table>

| Operator | Description | Example(s) |
|---|---|---|
| | | `int8range(5,15) * int8range(10,20)` → `[10,15)` |
| `anyrange - anyrange` → `anyrange` | Computes the difference of the ranges. The second range must not be contained in the first in such a way that the difference would not be a single range. | `int8range(5,15) - int8range(10,20)` → `[5,10)` |

### Table 9.59. Multirange Operators

| Operator | Description | Example(s) |
|---|---|---|
| `anymultirange @> anymultirange` → `boolean` | Does the first multirange contain the second? | `'{[2,4)}'::int4multirange @> '{[2,3)}'::int4multirange` → `t` |
| `anymultirange @> anyrange` → `boolean` | Does the multirange contain the range? | `'{[2,4)}'::int4multirange @> int4range(2,3)` → `t` |
| `anymultirange @> anyelement` → `boolean` | Does the multirange contain the element? | `'{[2011-01-01,2011-03-01)}'::tsmultirange @> '2011-01-10'::timestamp` → `t` |
| `anyrange @> anymultirange` → `boolean` | Does the range contain the multirange? | `'[2,4)'::int4range @> '{[2,3)}'::int4multirange` → `t` |
| `anymultirange <@ anymultirange` → `boolean` | Is the first multirange contained by the second? | `'{[2,4)}'::int4multirange <@ '{[1,7)}'::int4multirange` → `t` |
| `anymultirange <@ anyrange` → `boolean` | Is the multirange contained by the range? | `'{[2,4)}'::int4multirange <@ int4range(1,7)` → `t` |
| `anyrange <@ anymultirange` → `boolean` | Is the range contained by the multirange? | `int4range(2,4) <@ '{[1,7)}'::int4multirange` → `t` |
| `anyelement <@ anymultirange` → `boolean` | Is the element contained by the multirange? | `4 <@ '{[1,7)}'::int4multirange` → `t` |
| `anymultirange && anymultirange` → `boolean` | Do the multiranges overlap, that is, have any elements in common? | `'{[3,7)}'::int8multirange && '{[4,12)}'::int8multirange` → `t` |
| `anymultirange && anyrange` → `boolean` | Does the multirange overlap the range? | `'{[3,7)}'::int8multirange && int8range(4,12)` → `t` |

**Operator**
**Description**
**Example(s)**

`anyrange && anymultirange → boolean`
Does the range overlap the multirange?
`int8range(3,7) && '{[4,12)}'::int8multirange → t`

`anymultirange << anymultirange → boolean`
Is the first multirange strictly left of the second?
`'{[1,10)}'::int8multirange << '{[100,110)}'::int8multirange → t`

`anymultirange << anyrange → boolean`
Is the multirange strictly left of the range?
`'{[1,10)}'::int8multirange << int8range(100,110) → t`

`anyrange << anymultirange → boolean`
Is the range strictly left of the multirange?
`int8range(1,10) << '{[100,110)}'::int8multirange → t`

`anymultirange >> anymultirange → boolean`
Is the first multirange strictly right of the second?
`'{[50,60)}'::int8multirange >> '{[20,30)}'::int8multirange → t`

`anymultirange >> anyrange → boolean`
Is the multirange strictly right of the range?
`'{[50,60)}'::int8multirange >> int8range(20,30) → t`

`anyrange >> anymultirange → boolean`
Is the range strictly right of the multirange?
`int8range(50,60) >> '{[20,30)}'::int8multirange → t`

`anymultirange &< anymultirange → boolean`
Does the first multirange not extend to the right of the second?
`'{[1,20)}'::int8multirange &< '{[18,20)}'::int8multirange → t`

`anymultirange &< anyrange → boolean`
Does the multirange not extend to the right of the range?
`'{[1,20)}'::int8multirange &< int8range(18,20) → t`

`anyrange &< anymultirange → boolean`
Does the range not extend to the right of the multirange?
`int8range(1,20) &< '{[18,20)}'::int8multirange → t`

`anymultirange &> anymultirange → boolean`
Does the first multirange not extend to the left of the second?
`'{[7,20)}'::int8multirange &> '{[5,10)}'::int8multirange → t`

`anymultirange &> anyrange → boolean`
Does the multirange not extend to the left of the range?
`'{[7,20)}'::int8multirange &> int8range(5,10) → t`

`anyrange &> anymultirange → boolean`
Does the range not extend to the left of the multirange?
`int8range(7,20) &> '{[5,10)}'::int8multirange → t`

`anymultirange -|- anymultirange → boolean`

| Operator<br>Description<br>Example(s) |
|---|
| Are the multiranges adjacent?<br>`'{[1.1,2.2)}'::nummultirange -\|- '{[2.2,3.3)}'::nummultirange` → `t` |
| `anymultirange -\|- anyrange` → `boolean`<br>Is the multirange adjacent to the range?<br>`'{[1.1,2.2)}'::nummultirange -\|- numrange(2.2,3.3)` → `t` |
| `anyrange -\|- anymultirange` → `boolean`<br>Is the range adjacent to the multirange?<br>`numrange(1.1,2.2) -\|- '{[2.2,3.3)}'::nummultirange` → `t` |
| `anymultirange + anymultirange` → `anymultirange`<br>Computes the union of the multiranges. The multiranges need not overlap or be adjacent.<br>`'{[5,10)}'::nummultirange + '{[15,20)}'::nummultirange` → `{[5,10), [15,20)}` |
| `anymultirange * anymultirange` → `anymultirange`<br>Computes the intersection of the multiranges.<br>`'{[5,15)}'::int8multirange * '{[10,20)}'::int8multirange` → `{[10,15)}` |
| `anymultirange - anymultirange` → `anymultirange`<br>Computes the difference of the multiranges.<br>`'{[5,20)}'::int8multirange - '{[10,15)}'::int8multirange` → `{[5,10), [15,20)}` |

The left-of/right-of/adjacent operators always return false when an empty range or multirange is involved; that is, an empty range is not considered to be either before or after any other range.

Elsewhere empty ranges and multiranges are treated as the additive identity: anything unioned with an empty value is itself. Anything minus an empty value is itself. An empty multirange has exactly the same points as an empty range. Every range contains the empty range. Every multirange contains as many empty ranges as you like.

The range 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. There are separate operators for union and difference that take multirange parameters and return a multirange, and they do not fail even if their arguments are disjoint. So if you need a union or difference operation for ranges that may be disjoint, you can avoid errors by first casting your ranges to multiranges.

Table 9.60 shows the functions available for use with range types. Table 9.61 shows the functions available for use with multirange types.

### Table 9.60. Range Functions

| Function<br>Description<br>Example(s) |
|---|
| `lower ( anyrange )` → `anyelement`<br>Extracts the lower bound of the range (`NULL` if the range is empty or has no lower bound).<br>`lower(numrange(1.1,2.2))` → `1.1` |
| `upper ( anyrange )` → `anyelement`<br>Extracts the upper bound of the range (`NULL` if the range is empty or has no upper bound).<br>`upper(numrange(1.1,2.2))` → `2.2` |

| Function<br>Description<br>Example(s) |
|---|
| `isempty` ( `anyrange` ) → `boolean`<br>Is the range empty?<br>`isempty(numrange(1.1,2.2))` → `f` |
| `lower_inc` ( `anyrange` ) → `boolean`<br>Is the range's lower bound inclusive?<br>`lower_inc(numrange(1.1,2.2))` → `t` |
| `upper_inc` ( `anyrange` ) → `boolean`<br>Is the range's upper bound inclusive?<br>`upper_inc(numrange(1.1,2.2))` → `f` |
| `lower_inf` ( `anyrange` ) → `boolean`<br>Does the range have no lower bound? (A lower bound of `-Infinity` returns false.)<br>`lower_inf('(,)'::daterange)` → `t` |
| `upper_inf` ( `anyrange` ) → `boolean`<br>Does the range have no upper bound? (An upper bound of `Infinity` returns false.)<br>`upper_inf('(,)'::daterange)` → `t` |
| `range_merge` ( `anyrange`, `anyrange` ) → `anyrange`<br>Computes the smallest range that includes both of the given ranges.<br>`range_merge('[1,2)'::int4range, '[3,4)'::int4range)` → `[1,4)` |

### Table 9.61. Multirange Functions

| Function<br>Description<br>Example(s) |
|---|
| `lower` ( `anymultirange` ) → `anyelement`<br>Extracts the lower bound of the multirange (`NULL` if the multirange is empty or has no lower bound).<br>`lower('{[1.1,2.2)}'::nummultirange)` → `1.1` |
| `upper` ( `anymultirange` ) → `anyelement`<br>Extracts the upper bound of the multirange (`NULL` if the multirange is empty or has no upper bound).<br>`upper('{[1.1,2.2)}'::nummultirange)` → `2.2` |
| `isempty` ( `anymultirange` ) → `boolean`<br>Is the multirange empty?<br>`isempty('{[1.1,2.2)}'::nummultirange)` → `f` |
| `lower_inc` ( `anymultirange` ) → `boolean`<br>Is the multirange's lower bound inclusive?<br>`lower_inc('{[1.1,2.2)}'::nummultirange)` → `t` |
| `upper_inc` ( `anymultirange` ) → `boolean`<br>Is the multirange's upper bound inclusive?<br>`upper_inc('{[1.1,2.2)}'::nummultirange)` → `f` |
| `lower_inf` ( `anymultirange` ) → `boolean`<br>Does the multirange have no lower bound? (A lower bound of `-Infinity` returns false.)<br>`lower_inf('{(,)}'::datemultirange)` → `t` |

| Function<br>Description<br>Example(s) |
|---|
| `upper_inf` ( `anymultirange` ) → `boolean`<br>Does the multirange have no upper bound? (An upper bound of `Infinity` returns false.)<br>`upper_inf('{(,)}'::datemultirange)` → `t` |
| `range_merge` ( `anymultirange` ) → `anyrange`<br>Computes the smallest range that includes the entire multirange.<br>`range_merge('{[1,2), [3,4)}'::int4multirange)` → `[1,4)` |
| `multirange` ( `anyrange` ) → `anymultirange`<br>Returns a multirange containing just the given range.<br>`multirange('[1,2)'::int4range)` → `{[1,2)}` |
| `unnest` ( `anymultirange` ) → `setof anyrange`<br>Expands a multirange into a set of ranges in ascending order.<br>`unnest('{[1,2), [3,4)}'::int4multirange)` →<br><br>`[1,2)`<br>`[3,4)` |

The `lower_inc`, `upper_inc`, `lower_inf`, and `upper_inf` functions all return false for an empty range or multirange.

# 9.21. 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 9.62 while statistical aggregates are in Table 9.63. The built-in within-group ordered-set aggregate functions are listed in Table 9.64 while the built-in within-group hypothetical-set ones are in Table 9.65. Grouping operations, which are closely related to aggregate functions, are listed in Table 9.66. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

Aggregate functions that support *Partial Mode* are eligible to participate in various optimizations, such as parallel aggregation.

While all aggregates below accept an optional `ORDER BY` clause (as outlined in Section 4.2.7), the clause has only been added to aggregates whose output is affected by ordering.

### Table 9.62. General-Purpose Aggregate Functions

| Function<br>Description | Partial Mode |
|---|---|
| `any_value` ( `anyelement` ) → *`same as input type`*<br>Returns an arbitrary value from the non-null input values. | Yes |
| `array_agg` ( `anynonarray` `ORDER BY` `input_sort_columns` ) → `anyarray`<br>Collects all the input values, including nulls, into an array. | Yes |
| `array_agg` ( `anyarray` `ORDER BY` `input_sort_columns` ) → `anyarray`<br>Concatenates all the input arrays into an array of one higher dimension. (The inputs must all have the same dimensionality, and cannot be empty or null.) | Yes |
| `avg` ( `smallint` ) → `numeric`<br>`avg` ( `integer` ) → `numeric` | Yes |

| Function<br>Description | Partial Mode |
|---|---|
| `avg ( bigint ) → numeric`<br>`avg ( numeric ) → numeric`<br>`avg ( real ) → double precision`<br>`avg ( double precision ) → double precision`<br>`avg ( interval ) → interval`<br>Computes the average (arithmetic mean) of all the non-null input values. | |
| `bit_and ( smallint ) → smallint`<br>`bit_and ( integer ) → integer`<br>`bit_and ( bigint ) → bigint`<br>`bit_and ( bit ) → bit`<br>Computes the bitwise AND of all non-null input values. | Yes |
| `bit_or ( smallint ) → smallint`<br>`bit_or ( integer ) → integer`<br>`bit_or ( bigint ) → bigint`<br>`bit_or ( bit ) → bit`<br>Computes the bitwise OR of all non-null input values. | Yes |
| `bit_xor ( smallint ) → smallint`<br>`bit_xor ( integer ) → integer`<br>`bit_xor ( bigint ) → bigint`<br>`bit_xor ( bit ) → bit`<br>Computes the bitwise exclusive OR of all non-null input values. Can be useful as a checksum for an unordered set of values. | Yes |
| `bool_and ( boolean ) → boolean`<br>Returns true if all non-null input values are true, otherwise false. | Yes |
| `bool_or ( boolean ) → boolean`<br>Returns true if any non-null input value is true, otherwise false. | Yes |
| `count ( * ) → bigint`<br>Computes the number of input rows. | Yes |
| `count ( "any" ) → bigint`<br>Computes the number of input rows in which the input value is not null. | Yes |
| `every ( boolean ) → boolean`<br>This is the SQL standard's equivalent to `bool_and`. | Yes |
| `json_agg ( anyelement ORDER BY input_sort_columns ) → json`<br>`jsonb_agg ( anyelement ORDER BY input_sort_columns ) → jsonb`<br>Collects all the input values, including nulls, into a JSON array. Values are converted to JSON as per `to_json` or `to_jsonb`. | No |
| `json_agg_strict ( anyelement ) → json`<br>`jsonb_agg_strict ( anyelement ) → jsonb`<br>Collects all the input values, skipping nulls, into a JSON array. Values are converted to JSON as per `to_json` or `to_jsonb`. | No |

| Function<br>Description | Partial Mode |
|---|---|
| `json_arrayagg` ( [ *`value_expression`* ] [ `ORDER BY` *`sort_expression`* ] [ { `NULL` \| `ABSENT` } `ON NULL` ] [ `RETURNING` *`data_type`* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ])<br>Behaves in the same way as `json_array` but as an aggregate function so it only takes one *`value_expression`* parameter. If `ABSENT ON NULL` is specified, any NULL values are omitted. If `ORDER BY` is specified, the elements will appear in the array in that order rather than in the input order.<br>`SELECT json_arrayagg(v) FROM (VALUES(2),(1)) t(v)` → `[2, 1]` | No |
| `json_objectagg` ( [ { *`key_expression`* { `VALUE` \| ':' } *`value_expression`* } ] [ { `NULL` \| `ABSENT` } `ON NULL` ] [ { `WITH` \| `WITHOUT` } `UNIQUE` [ `KEYS` ] ] [ `RETURNING` *`data_type`* [ `FORMAT JSON` [ `ENCODING UTF8` ] ] ])<br>Behaves like `json_object`, but as an aggregate function, so it only takes one *`key_expression`* and one *`value_expression`* parameter.<br>`SELECT json_objectagg(k:v) FROM (VALUES ('a'::text,current_date),('b',current_date + 1)) AS t(k,v)` → `{ "a" : "2022-05-10", "b" : "2022-05-11" }` | No |
| `json_object_agg` ( *`key`* `"any"`, *`value`* `"any"` `ORDER BY` `input_sort_columns` ) → `json`<br>`jsonb_object_agg` ( *`key`* `"any"`, *`value`* `"any"` `ORDER BY` `input_sort_columns` ) → `jsonb`<br>Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. Values can be null, but keys cannot. | No |
| `json_object_agg_strict` ( *`key`* `"any"`, *`value`* `"any"` ) → `json`<br>`jsonb_object_agg_strict` ( *`key`* `"any"`, *`value`* `"any"` ) → `jsonb`<br>Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. The *`key`* can not be null. If the *`value`* is null then the entry is skipped, | No |
| `json_object_agg_unique` ( *`key`* `"any"`, *`value`* `"any"` ) → `json`<br>`jsonb_object_agg_unique` ( *`key`* `"any"`, *`value`* `"any"` ) → `jsonb`<br>Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. Values can be null, but keys cannot. If there is a duplicate key an error is thrown. | No |
| `json_object_agg_unique_strict` ( *`key`* `"any"`, *`value`* `"any"` ) → `json`<br>`jsonb_object_agg_unique_strict` ( *`key`* `"any"`, *`value`* `"any"` ) → `jsonb`<br>Collects all the key/value pairs into a JSON object. Key arguments are coerced to text; value arguments are converted as per `to_json` or `to_jsonb`. The *`key`* can not be null. If the *`value`* is null then the entry is skipped. If there is a duplicate key an error is thrown. | No |
| `max` ( *`see text`* ) → *`same as input type`*<br>Computes the maximum of the non-null input values. Available for any numeric, string, date/time, or enum type, as well as `bytea`, `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8`, and also arrays and composite types containing sortable data types. | Yes |
| `min` ( *`see text`* ) → *`same as input type`*<br>Computes the minimum of the non-null input values. Available for any numeric, string, date/time, or enum type, as well as `bytea`, `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8`, and also arrays and composite types containing sortable data types. | Yes |
| `range_agg` ( *`value`* `anyrange` ) → `anymultirange` | No |

| Function<br>Description | Partial Mode |
|---|---|
| `range_agg` ( *`value`* `anymultirange` ) → `anymultirange`<br>Computes the union of the non-null input values. | |
| `range_intersect_agg` ( *`value`* `anyrange` ) → `anyrange`<br>`range_intersect_agg` ( *`value`* `anymultirange` ) → `anymultirange`<br>Computes the intersection of the non-null input values. | No |
| `string_agg` ( *`value`* `text`, *`delimiter`* `text` ) → `text`<br>`string_agg` ( *`value`* `bytea`, *`delimiter`* `bytea` `ORDER BY` `input_sort_columns` ) → `bytea`<br>Concatenates the non-null input values into a string. Each value after the first is preceded by the corresponding *`delimiter`* (if it's not null). | Yes |
| `sum` ( `smallint` ) → `bigint`<br>`sum` ( `integer` ) → `bigint`<br>`sum` ( `bigint` ) → `numeric`<br>`sum` ( `numeric` ) → `numeric`<br>`sum` ( `real` ) → `real`<br>`sum` ( `double precision` ) → `double precision`<br>`sum` ( `interval` ) → `interval`<br>`sum` ( `money` ) → `money`<br>Computes the sum of the non-null input values. | Yes |
| `xmlagg` ( `xml` `ORDER BY` `input_sort_columns` ) → `xml`<br>Concatenates the non-null XML input values (see Section 9.15.1.8). | No |

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.

The aggregate functions `array_agg`, `json_agg`, `jsonb_agg`, `json_agg_strict`, `jsonb_agg_strict`, `json_object_agg`, `jsonb_object_agg`, `json_object_agg_strict`, `jsonb_object_agg_strict`, `json_object_agg_unique`, `jsonb_object_agg_unique`, `json_object_agg_unique_strict`, `jsonb_object_agg_unique_strict`, `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. 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.

## Note

The boolean aggregates `bool_and` and `bool_or` correspond to the standard SQL aggregates `every` and `any` or `some`. PostgreSQL supports `every`, but not `any` or `some`, because 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 that includes all rows in the table.

Table 9.63 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Functions shown as accepting *`numeric_type`* are available for all the types `smallint`, `integer`, `bigint`, `numeric`, `real`, and `double precision`. 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 9.63. Aggregate Functions for Statistics

| Function<br>Description | Partial Mode |
|---|---|
| `corr` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the correlation coefficient. | Yes |
| `covar_pop` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the population covariance. | Yes |
| `covar_samp` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the sample covariance. | Yes |
| `regr_avgx` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the average of the independent variable, `sum(`*X*`)/`*N*. | Yes |
| `regr_avgy` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the average of the dependent variable, `sum(`*Y*`)/`*N*. | Yes |
| `regr_count` ( *Y* `double precision`, *X* `double precision` ) → `bigint`<br>Computes the number of rows in which both inputs are non-null. | Yes |
| `regr_intercept` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs. | Yes |
| `regr_r2` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the square of the correlation coefficient. | Yes |
| `regr_slope` ( *Y* `double precision`, *X* `double precision` ) → `double precision`<br>Computes the slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs. | Yes |

| Function<br>Description | Partial Mode |
|---|---|
| `regr_sxx` ( *`Y`* `double precision`, *`X`* `double precision` ) → `double precision`<br>Computes the "sum of squares" of the independent variable, `sum(`*`X`*`^2) - sum(`*`X`*`)^2/`*`N`*. | Yes |
| `regr_sxy` ( *`Y`* `double precision`, *`X`* `double precision` ) → `double precision`<br>Computes the "sum of products" of independent times dependent variables, `sum(`*`X`*`*`*`Y`*`) - sum(`*`X`*`) * sum(`*`Y`*`)/`*`N`*. | Yes |
| `regr_syy` ( *`Y`* `double precision`, *`X`* `double precision` ) → `double precision`<br>Computes the "sum of squares" of the dependent variable, `sum(`*`Y`*`^2) - sum(`*`Y`*`)^2/`*`N`*. | Yes |
| `stddev` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>This is a historical alias for `stddev_samp`. | Yes |
| `stddev_pop` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>Computes the population standard deviation of the input values. | Yes |
| `stddev_samp` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>Computes the sample standard deviation of the input values. | Yes |
| `variance` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>This is a historical alias for `var_samp`. | Yes |
| `var_pop` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>Computes the population variance of the input values (square of the population standard deviation). | Yes |
| `var_samp` ( *`numeric_type`* ) → `double precision` for `real` or `double precision`, otherwise `numeric`<br>Computes the sample variance of the input values (square of the sample standard deviation). | Yes |

Table 9.64 shows some aggregate functions that use the *ordered-set aggregate* syntax. These functions are sometimes referred to as "inverse distribution" functions. Their aggregated input is introduced by `ORDER BY`, and they may also take a *direct argument* that is not aggregated, but is computed only once. All these functions ignore null values in their aggregated 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.

### Table 9.64. Ordered-Set Aggregate Functions

| Function<br>Description | Partial Mode |
|---|---|
| `mode` () `WITHIN GROUP` ( `ORDER BY` `anyelement` ) → `anyelement`<br>Computes the *mode*, the most frequent value of the aggregated argument (arbitrarily choosing the first one if there are multiple equally-frequent values). The aggregated argument must be of a sortable type. | No |
| `percentile_cont` ( *`fraction`* `double precision` ) `WITHIN GROUP` ( `ORDER BY` `double precision` ) → `double precision`<br>`percentile_cont` ( *`fraction`* `double precision` ) `WITHIN GROUP` ( `ORDER BY` `interval` ) → `interval` | No |

| Function<br>Description | Partial Mode |
|---|---|
| Computes the *continuous percentile*, a value corresponding to the specified *fraction* within the ordered set of aggregated argument values. This will interpolate between adjacent input items if needed. | |
| percentile_cont ( *fractions* double precision[] ) WITHIN GROUP ( ORDER BY double precision ) → double precision[]<br>percentile_cont ( *fractions* double precision[] ) WITHIN GROUP ( ORDER BY interval ) → interval[]<br>Computes multiple continuous percentiles. The result is an array of the same dimensions as the *fractions* parameter, with each non-null element replaced by the (possibly interpolated) value corresponding to that percentile. | No |
| percentile_disc ( *fraction* double precision ) WITHIN GROUP ( ORDER BY anyelement ) → anyelement<br>Computes the *discrete percentile*, the first value within the ordered set of aggregated argument values whose position in the ordering equals or exceeds the specified *fraction*. The aggregated argument must be of a sortable type. | No |
| percentile_disc ( *fractions* double precision[] ) WITHIN GROUP ( ORDER BY anyelement ) → anyarray<br>Computes multiple discrete percentiles. The result is an array of the same dimensions as the *fractions* parameter, with each non-null element replaced by the input value corresponding to that percentile. The aggregated argument must be of a sortable type. | No |

Each of the "hypothetical-set" aggregates listed in Table 9.65 is associated with a window function of the same name defined in Section 9.22. In each case, the aggregate's 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 represented by the *sorted_args*. For each of these functions, 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 9.65. Hypothetical-Set Aggregate Functions

| Function<br>Description | Partial Mode |
|---|---|
| rank ( *args* ) WITHIN GROUP ( ORDER BY *sorted_args* ) → bigint<br>Computes the rank of the hypothetical row, with gaps; that is, the row number of the first row in its peer group. | No |
| dense_rank ( *args* ) WITHIN GROUP ( ORDER BY *sorted_args* ) → bigint<br>Computes the rank of the hypothetical row, without gaps; this function effectively counts peer groups. | No |
| percent_rank ( *args* ) WITHIN GROUP ( ORDER BY *sorted_args* ) → double precision<br>Computes the relative rank of the hypothetical row, that is (rank - 1) / (total rows - 1). The value thus ranges from 0 to 1 inclusive. | No |
| cume_dist ( *args* ) WITHIN GROUP ( ORDER BY *sorted_args* ) → double precision<br>Computes the cumulative distribution, that is (number of rows preceding or peers with hypothetical row) / (total rows). The value thus ranges from 1/*N* to 1. | No |

**Table 9.66. Grouping Operations**

| Function<br>Description |
|---|
| `GROUPING ( group_by_expression(s) ) → integer`<br>Returns a bit mask indicating which `GROUP BY` expressions are not included in the current grouping set. Bits are assigned with the rightmost argument corresponding to the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the current result row, and 1 if it is not included. |

The grouping operations shown in Table 9.66 are used in conjunction with grouping sets (see Section 7.2.4) to distinguish result rows. The arguments to the `GROUPING` function are not actually evaluated, but they must exactly match expressions given in the `GROUP BY` clause of the associated query level. 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)
```

Here, the `grouping` value `0` in the first four rows shows that those have been grouped normally, over both the grouping columns. The value `1` indicates that `model` was not grouped by in the next-to-last two rows, and the value `3` indicates that neither `make` nor `model` was grouped by in the last row (which therefore is an aggregate over all the input rows).

# 9.22. 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 for an introduction to this feature, and Section 4.2.8 for syntax details.

The built-in window functions are listed in Table 9.67. 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 ordinary aggregate (i.e., not ordered-set or hypothetical-set aggregates) can be used as a window function; see Section 9.21 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 plain aggregates and return a single row for the entire set.

### Table 9.67. General-Purpose Window Functions

| Function<br>Description |
|---|
| `row_number` () → `bigint`<br>Returns the number of the current row within its partition, counting from 1. |
| `rank` () → `bigint`<br>Returns the rank of the current row, with gaps; that is, the `row_number` of the first row in its peer group. |
| `dense_rank` () → `bigint`<br>Returns the rank of the current row, without gaps; this function effectively counts peer groups. |
| `percent_rank` () → `double precision`<br>Returns the relative rank of the current row, that is (`rank` - 1) / (total partition rows - 1). The value thus ranges from 0 to 1 inclusive. |
| `cume_dist` () → `double precision`<br>Returns the cumulative distribution, that is (number of partition rows preceding or peers with current row) / (total partition rows). The value thus ranges from 1/*N* to 1. |
| `ntile` ( *num_buckets* `integer` ) → `integer`<br>Returns an integer ranging from 1 to the argument value, dividing the partition as equally as possible. |
| `lag` ( *value* `anycompatible` [, *offset* `integer` [, *default* `anycompatible` ]] ) → `anycompatible`<br>Returns *value* evaluated at the row that is *offset* rows before the current row within the partition; if there is no such row, instead returns *default* (which must be of a type compatible with *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* `anycompatible` [, *offset* `integer` [, *default* `anycompatible` ]] ) → `anycompatible`<br>Returns *value* evaluated at the row that is *offset* rows after the current row within the partition; if there is no such row, instead returns *default* (which must be of a type compatible with *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* `anyelement` ) → `anyelement`<br>Returns *value* evaluated at the row that is the first row of the window frame. |
| `last_value` ( *value* `anyelement` ) → `anyelement`<br>Returns *value* evaluated at the row that is the last row of the window frame. |
| `nth_value` ( *value* `anyelement`, *n* `integer` ) → `anyelement`<br>Returns *value* evaluated at the row that is the *n*'th row of the window frame (counting from 1); returns `NULL` if there is no such row. |

All of the functions listed in Table 9.67 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 BY` columns are said to be *peers*. The four ranking functions (including `cume_dist`) are defined so that they give the same answer for all rows of a peer group.

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`, `ROWS` or `GROUPS`) to the `OVER` clause. See Section 4.2.8 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.)

# 9.23. Merge Support Functions

PostgreSQL includes one merge support function that may be used in the `RETURNING` list of a MERGE command to identify the action taken for each row; see Table 9.68.

### Table 9.68. Merge Support Functions

| Function<br>Description |
|---|
| `merge_action` ( ) → `text`<br>Returns the merge action command executed for the current row. This will be `'INSERT'`, `'UPDATE'`, or `'DELETE'`. |

Example:

```
MERGE INTO products p
  USING stock s ON p.product_id = s.product_id
  WHEN MATCHED AND s.quantity > 0 THEN
    UPDATE SET in_stock = true, quantity = s.quantity
  WHEN MATCHED THEN
    UPDATE SET in_stock = false, quantity = 0
  WHEN NOT MATCHED THEN
    INSERT (product_id, in_stock, quantity)
      VALUES (s.product_id, true, s.quantity)
  RETURNING merge_action(), p.*;

 merge_action | product_id | in_stock | quantity
--------------+------------+----------+----------
 UPDATE       |       1001 | t        |       50
 UPDATE       |       1002 | f        |        0
 INSERT       |       1003 | t        |       10
```

Note that this function can only be used in the `RETURNING` list of a `MERGE` command. It is an error to use it in any other part of a query.

# 9.24. 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.

## 9.24.1. 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);
```

## 9.24.2. 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. 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.

## 9.24.3. 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 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.

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. 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.

## 9.24.4. 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. 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 9.25.5 for details about the meaning of a row constructor comparison.

## 9.24.5. 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. 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 9.25.5 for details about the meaning of a row constructor comparison.

## 9.24.6. Single-Row Comparison

*row_constructor operator* (*subquery*)

The left-hand side is a row constructor, as described in Section 4.2.13. 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 9.25.5 for details about the meaning of a row constructor comparison.

# 9.25. 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.

## 9.25.1. IN

*expression* IN (*value* [, ...])

The right-hand side is a parenthesized list of 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.

### 9.25.2. NOT IN

```
expression NOT IN (value [, ...])
```

The right-hand side is a parenthesized list of 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.

### 9.25.3. 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`.

### 9.25.4. 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.

## 9.25.5. Row Constructor Comparison

*row_constructor operator row_constructor*

Each side is a row constructor, as described in Section 4.2.13. The two row constructors must have the same number of fields. The given *operator* is applied to each pair of corresponding fields. (Since the fields could be of different types, this means that a different specific operator could be selected for each pair.) All the selected operators must be members of some B-tree operator class, or be the negator of an = member of a B-tree operator class, meaning that row constructor comparison is only possible when the *operator* is =, <>, <, <=, >, or >=, or has semantics similar to one of these.

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.

*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.

## 9.25.6. 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 9.25.5) or comparing a row constructor to the output of a subquery (as in Section 9.24). 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 9.25.5).

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 and B-Tree deduplication (see Section 65.1.4.3). They are not intended to be generally useful for writing queries, though.

# 9.26. 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 9.69 and Table 9.70. Other, more specialized set-returning functions are described elsewhere in this manual. See Section 7.2.1.4 for ways to combine multiple set-returning functions.

### Table 9.69. Series Generating Functions

| Function<br>Description |
|---|
| `generate_series` ( *`start`* `integer`, *`stop`* `integer` [, *`step`* `integer` ] ) → `setof integer`<br>`generate_series` ( *`start`* `bigint`, *`stop`* `bigint` [, *`step`* `bigint` ] ) → `setof bigint`<br>`generate_series` ( *`start`* `numeric`, *`stop`* `numeric` [, *`step`* `numeric` ] ) → `setof numeric`<br>Generates a series of values from *`start`* to *`stop`*, with a step size of *`step`*. *`step`* defaults to 1. |
| `generate_series` ( *`start`* `timestamp`, *`stop`* `timestamp`, *`step`* `interval` ) → `setof timestamp`<br>`generate_series` ( *`start`* `timestamp with time zone`, *`stop`* `timestamp with time zone`, *`step`* `interval` [, *`timezone`* `text` ] ) → `setof timestamp with time zone`<br>Generates a series of values from *`start`* to *`stop`*, with a step size of *`step`*. In the timezone-aware form, times of day and daylight-savings adjustments are computed according to the time zone named by the *`timezone`* argument, or the current TimeZone setting if that is omitted. |

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 if any input is `NULL`. 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)

-- this example assumes that TimeZone is set to UTC; note the DST
 transition:
SELECT * FROM generate_series('2001-10-22 00:00 -04:00'::timestamptz,
                              '2001-11-01 00:00 -05:00'::timestamptz,
                              '1 day'::interval, 'America/New_York');
    generate_series
------------------------
 2001-10-22 04:00:00+00
 2001-10-23 04:00:00+00
 2001-10-24 04:00:00+00
 2001-10-25 04:00:00+00
 2001-10-26 04:00:00+00
 2001-10-27 04:00:00+00
 2001-10-28 04:00:00+00
 2001-10-29 05:00:00+00
 2001-10-30 05:00:00+00
 2001-10-31 05:00:00+00
 2001-11-01 05:00:00+00
(11 rows)
```

### Table 9.70. Subscript Generating Functions

| Function<br>Description |
|---|
| `generate_subscripts` ( *`array`* `anyarray`, *`dim`* `integer` ) → `setof integer`<br>Generates a series comprising the valid subscripts of the *`dim`*'th dimension of the given array. |

| Function<br>Description |
|---|
| `generate_subscripts` ( *`array`* `anyarray`, *`dim`* `integer`, *`reverse`* `boolean` ) → `setof integer`<br>Generates a series comprising the valid subscripts of the *`dim`*'th dimension of the given array. When *`reverse`* is true, returns the series 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 if any input is `NULL`. 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 function's output column(s), 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)
```

# 9.27. System Information Functions and Operators

The functions described in this section are used to obtain various information about a PostgreSQL installation.

## 9.27.1. Session Information Functions

Table 9.71 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 27.2.26 for more information.

### Table 9.71. Session Information Functions

| Function<br>Description |
|---|
| current_catalog → name<br>current_database () → name<br>Returns the name of the current database. (Databases are called “catalogs” in the SQL standard, so current_catalog is the standard's spelling.) |
| current_query () → text<br>Returns the text of the currently executing query, as submitted by the client (which might contain more than one statement). |

| Function<br>Description |
|---|
| `current_role` → `name`<br>This is equivalent to `current_user`. |
| `current_schema` → `name`<br>`current_schema` () → `name`<br>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` ( *`include_implicit`* `boolean` ) → `name[]`<br>Returns an array of the names of all schemas presently in the effective search path, in their priority order. (Items in the current search_path setting that do not correspond to existing, searchable schemas are omitted.) If the Boolean argument is `true`, then implicitly-searched system schemas such as `pg_catalog` are included in the result. |
| `current_user` → `name`<br>Returns the user name of the current execution context. |
| `inet_client_addr` () → `inet`<br>Returns the IP address of the current client, or `NULL` if the current connection is via a Unix-domain socket. |
| `inet_client_port` () → `integer`<br>Returns the IP port number of the current client, or `NULL` if the current connection is via a Unix-domain socket. |
| `inet_server_addr` () → `inet`<br>Returns the IP address on which the server accepted the current connection, or `NULL` if the current connection is via a Unix-domain socket. |
| `inet_server_port` () → `integer`<br>Returns the IP port number on which the server accepted the current connection, or `NULL` if the current connection is via a Unix-domain socket. |
| `pg_backend_pid` () → `integer`<br>Returns the process ID of the server process attached to the current session. |
| `pg_blocking_pids` ( `integer` ) → `integer[]`<br>Returns an array of the process ID(s) of the sessions that are blocking the server process with the specified process ID from acquiring a lock, or an empty array if there is no such server process or it is not blocked.<br>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.<br>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` () → `timestamp with time zone`<br>Returns the time 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` ( [ `text` ] ) → `text` |

| Function<br>Description |
|---|
| Returns the path name of the log file currently in use by the logging collector. The path includes the log_directory directory and the individual log file name. The result is `NULL` if the logging collector is disabled. When multiple log files exist, each in a different format, `pg_current_logfile` without an argument returns the path of the file having the first format found in the ordered list: `stderr`, `csvlog`, `jsonlog`. `NULL` is returned if no log file has any of these formats. To request information about a specific log file format, supply either `csvlog`, `jsonlog` or `stderr` as the value of the optional parameter. The result is `NULL` if the log format requested is not configured in log_destination. The result reflects the contents of the `current_logfiles` file.<br>This function is restricted to superusers and roles with privileges of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_get_loaded_modules` () → `setof record` ( *`module_name`* `text`, *`version`* `text`, *`file_name`* `text` )<br>Returns a list of the loadable modules that are loaded into the current server session. The *`module_name`* and *`version`* fields are NULL unless the module author supplied values for them using the `PG_MODULE_MAGIC_EXT` macro. The *`file_name`* field gives the file name of the module (shared library). |
| `pg_my_temp_schema` () → `oid`<br>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` ( `oid` ) → `boolean`<br>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_jit_available` () → `boolean`<br>Returns true if a JIT compiler extension is available (see Chapter 30) and the jit configuration parameter is set to `on`. |
| `pg_numa_available` () → `boolean`<br>Returns true if the server has been compiled with NUMA support. |
| `pg_listening_channels` () → `setof text`<br>Returns the set of names of asynchronous notification channels that the current session is listening to. |
| `pg_notification_queue_usage` () → `double precision`<br>Returns the fraction (0–1) of the asynchronous notification queue's maximum size that is currently occupied by notifications that are waiting to be processed. See LISTEN and NOTIFY for more information. |
| `pg_postmaster_start_time` () → `timestamp with time zone`<br>Returns the time when the server started. |
| `pg_safe_snapshot_blocking_pids` ( `integer` ) → `integer[]`<br>Returns an array of the process ID(s) 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.<br>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 for more information about serializable and deferrable transactions.<br>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. |
| `pg_trigger_depth` () → `integer`<br>Returns the current nesting level of PostgreSQL triggers (0 if not called, directly or indirectly, from inside a trigger). |

| Function<br>Description |
|---|
| `session_user` → `name`<br>Returns the session user's name. |
| `system_user` → `text`<br>Returns the authentication method and the identity (if any) that the user presented during the authentication cycle before they were assigned a database role. It is represented as `auth_method:identity` or `NULL` if the user has not been authenticated (for example if Trust authentication has been used). |
| `user` → `name`<br>This is equivalent to `current_user`. |

> **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. 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. 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.)

## 9.27.2. Access Privilege Inquiry Functions

Table 9.72 lists functions that allow querying object access privileges programmatically. (See Section 5.8 for more information about privileges.) In these functions, the user whose privileges are being inquired about can be specified by name or by OID (`pg_authid.oid`), or if the name is given as `public` then the privileges of the PUBLIC pseudo-role are checked. Also, the *`user`* argument can be omitted entirely, in which case the `current_user` is assumed. The object that is being inquired about can be specified either by name or by OID, too. When specifying by name, a schema name can be included if relevant. The access privilege of interest is specified by a text string, which must evaluate to one of the appropriate privilege keywords for the object's type (e.g., `SELECT`). 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');
```

### Table 9.72. Access Privilege Inquiry Functions

| Function<br>Description |
|---|
| `has_any_column_privilege` ( [ *`user`* `name` or `oid`, ] *`table`* `text` or `oid`, *`privilege`* `text` )<br>→ `boolean` |

| **Function**<br>**Description** |
|---|
| Does user have privilege for any column of table? This succeeds either if the privilege is held for the whole table, or if there is a column-level grant of the privilege for at least one column. Allowable privilege types are `SELECT`, `INSERT`, `UPDATE`, and `REFERENCES`. |
| `has_column_privilege` ( [ *`user`* `name` or `oid`, ] *`table`* `text` or `oid`, *`column`* `text` or `smallint`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for the specified table column? This succeeds either if the privilege is held for the whole table, or if there is a column-level grant of the privilege for the column. The column can be specified by name or by attribute number (`pg_attribute`.`attnum`). Allowable privilege types are `SELECT`, `INSERT`, `UPDATE`, and `REFERENCES`. |
| `has_database_privilege` ( [ *`user`* `name` or `oid`, ] *`database`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for database? Allowable privilege types are `CREATE`, `CONNECT`, `TEMPORARY`, and `TEMP` (which is equivalent to `TEMPORARY`). |
| `has_foreign_data_wrapper_privilege` ( [ *`user`* `name` or `oid`, ] *`fdw`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for foreign-data wrapper? The only allowable privilege type is `USAGE`. |
| `has_function_privilege` ( [ *`user`* `name` or `oid`, ] *`function`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for function? The only allowable privilege type is `EXECUTE`.<br>When specifying a function by name rather than by OID, the allowed input is the same as for the `regprocedure` data type (see Section 8.19). An example is:<br><br>`SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');` |
| `has_language_privilege` ( [ *`user`* `name` or `oid`, ] *`language`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for language? The only allowable privilege type is `USAGE`. |
| `has_largeobject_privilege` ( [ *`user`* `name` or `oid`, ] *`largeobject`* `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for large object? Allowable privilege types are `SELECT` and `UPDATE`. |
| `has_parameter_privilege` ( [ *`user`* `name` or `oid`, ] *`parameter`* `text`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for configuration parameter? The parameter name is case-insensitive. Allowable privilege types are `SET` and `ALTER SYSTEM`. |
| `has_schema_privilege` ( [ *`user`* `name` or `oid`, ] *`schema`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for schema? Allowable privilege types are `CREATE` and `USAGE`. |
| `has_sequence_privilege` ( [ *`user`* `name` or `oid`, ] *`sequence`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for sequence? Allowable privilege types are `USAGE`, `SELECT`, and `UPDATE`. |
| `has_server_privilege` ( [ *`user`* `name` or `oid`, ] *`server`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for foreign server? The only allowable privilege type is `USAGE`. |
| `has_table_privilege` ( [ *`user`* `name` or `oid`, ] *`table`* `text` or `oid`, *`privilege`* `text` ) → `boolean` |

| Function<br>Description |
|---|
| Does user have privilege for table? Allowable privilege types are `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, and `MAINTAIN`. |
| `has_tablespace_privilege` ( [ *`user`* `name` or `oid`, ] *`tablespace`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for tablespace? The only allowable privilege type is `CREATE`. |
| `has_type_privilege` ( [ *`user`* `name` or `oid`, ] *`type`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for data type? The only allowable privilege type is `USAGE`. When specifying a type by name rather than by OID, the allowed input is the same as for the `regtype` data type (see Section 8.19). |
| `pg_has_role` ( [ *`user`* `name` or `oid`, ] *`role`* `text` or `oid`, *`privilege`* `text` ) → `boolean`<br>Does user have privilege for role? Allowable privilege types are `MEMBER`, `USAGE`, and `SET`. `MEMBER` denotes direct or indirect membership in the role without regard to what specific privileges may be conferred. `USAGE` denotes whether the privileges of the role are immediately available without doing `SET ROLE`, while `SET` denotes whether it is possible to change to the role using the `SET ROLE` command. `WITH ADMIN OPTION` or `WITH GRANT OPTION` can be added to any of these privilege types to test whether the `ADMIN` privilege is held (all six spellings test the same thing). This function does not allow the special case of setting *`user`* to `public`, because the PUBLIC pseudo-role can never be a member of real roles. |
| `row_security_active` ( *`table`* `text` or `oid` ) → `boolean`<br>Is row-level security active for the specified table in the context of the current user and current environment? |

Table 9.73 shows the operators available for the `aclitem` type, which is the catalog representation of access privileges. See Section 5.8 for information about how to read access privilege values.

### Table 9.73. `aclitem` Operators

| Operator<br>Description<br>Example(s) |
|---|
| `aclitem` = `aclitem` → `boolean`<br>Are `aclitem`s equal? (Notice that type `aclitem` lacks the usual set of comparison operators; it has only equality. In turn, `aclitem` arrays can only be compared for equality.)<br>`'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem` → `f` |
| `aclitem[]` @> `aclitem` → `boolean`<br>Does array contain the specified privileges? (This is true if there is an array entry that matches the `aclitem`'s grantee and grantor, and has at least the specified set of privileges.)<br>`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem` → `t` |
| `aclitem[]` ~ `aclitem` → `boolean`<br>This is a deprecated alias for `@>`.<br>`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] ~ 'calvin=r*/hobbes'::aclitem` → `t` |

Table 9.74 shows some additional functions to manage the `aclitem` type.

**Table 9.74. `aclitem` Functions**

| Function<br>Description |
|---|
| `acldefault` ( *`type`* `"char"`, *`ownerId`* `oid` ) → `aclitem[]`<br>Constructs an `aclitem` array holding the default access privileges for an object of type *`type`* belonging to the role with OID *`ownerId`*. This represents the access privileges that will be assumed when an object's ACL entry is null. (The default access privileges are described in Section 5.8.) The *`type`* parameter must be one of 'c' for `COLUMN`, 'r' for `TABLE` and table-like objects, 's' for `SEQUENCE`, 'd' for `DATABASE`, 'f' for `FUNCTION` or `PROCEDURE`, 'l' for `LANGUAGE`, 'L' for `LARGE OBJECT`, 'n' for `SCHEMA`, 'p' for `PARAMETER`, 't' for `TABLESPACE`, 'F' for `FOREIGN DATA WRAPPER`, 'S' for `FOREIGN SERVER`, or 'T' for `TYPE` or `DOMAIN`. |
| `aclexplode` ( `aclitem[]` ) → `setof record` ( *`grantor`* `oid`, *`grantee`* `oid`, *`privilege_type`* `text`, *`is_grantable`* `boolean` )<br>Returns the `aclitem` array as a set of rows. If the grantee is the pseudo-role PUBLIC, it is represented by zero in the *`grantee`* column. Each granted privilege is represented as `SELECT`, `INSERT`, etc (see Table 5.1 for a full list). Note that each privilege is broken out as a separate row, so only one keyword appears in the *`privilege_type`* column. |
| `makeaclitem` ( *`grantee`* `oid`, *`grantor`* `oid`, *`privileges`* `text`, *`is_grantable`* `boolean` ) → `aclitem`<br>Constructs an `aclitem` with the given properties. *`privileges`* is a comma-separated list of privilege names such as `SELECT`, `INSERT`, etc, all of which are set in the result. (Case of the privilege string is not significant, and extra whitespace is allowed between but not within privilege names.) |

## 9.27.3. Schema Visibility Inquiry Functions

Table 9.75 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. Thus, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

For functions and operators, an object in the search path is said to be visible if there is no object of the same name *and argument data type(s)* earlier in the path. For operator classes and families, both the name and the associated index access method are considered.

**Table 9.75. Schema Visibility Inquiry Functions**

| Function<br>Description |
|---|
| `pg_collation_is_visible` ( *`collation`* `oid` ) → `boolean`<br>Is collation visible in search path? |
| `pg_conversion_is_visible` ( *`conversion`* `oid` ) → `boolean`<br>Is conversion visible in search path? |
| `pg_function_is_visible` ( *`function`* `oid` ) → `boolean`<br>Is function visible in search path? (This also works for procedures and aggregates.) |
| `pg_opclass_is_visible` ( *`opclass`* `oid` ) → `boolean`<br>Is operator class visible in search path? |
| `pg_operator_is_visible` ( *`operator`* `oid` ) → `boolean`<br>Is operator visible in search path? |

| Function<br>Description |
|---|
| `pg_opfamily_is_visible` ( *opclass* `oid` ) → `boolean`<br>Is operator family visible in search path? |
| `pg_statistics_obj_is_visible` ( *stat* `oid` ) → `boolean`<br>Is statistics object visible in search path? |
| `pg_table_is_visible` ( *table* `oid` ) → `boolean`<br>Is table visible in search path? (This works for all types of relations, including views, materialized views, indexes, sequences and foreign tables.) |
| `pg_ts_config_is_visible` ( *config* `oid` ) → `boolean`<br>Is text search configuration visible in search path? |
| `pg_ts_dict_is_visible` ( *dict* `oid` ) → `boolean`<br>Is text search dictionary visible in search path? |
| `pg_ts_parser_is_visible` ( *parser* `oid` ) → `boolean`<br>Is text search parser visible in search path? |
| `pg_ts_template_is_visible` ( *template* `oid` ) → `boolean`<br>Is text search template visible in search path? |
| `pg_type_is_visible` ( *type* `oid` ) → `boolean`<br>Is type (or domain) visible in search path? |

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.

## 9.27.4. System Catalog Information Functions

Table 9.76 lists functions that extract information from the system catalogs.

### Table 9.76. System Catalog Information Functions

| Function<br>Description |
|---|
| `format_type` ( *type* `oid`, *typemod* `integer` ) → `text`<br>Returns the SQL name for 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_basetype` ( `regtype` ) → `regtype`<br>Returns the OID of the base type of a domain identified by its type OID. If the argument is the OID of a non-domain type, returns the argument as-is. Returns NULL if the argument is not a valid type OID. If there's a chain of domain dependencies, it will recurse until finding the base type.<br>Assuming `CREATE DOMAIN mytext AS text`:<br>`pg_basetype('mytext'::regtype)` → `text` |
| `pg_char_to_encoding` ( *encoding* `name` ) → `integer`<br>Converts the supplied encoding name into an integer representing the internal identifier used in some system catalog tables. Returns `-1` if an unknown encoding name is provided. |

| Function<br>Description |
|---|
| `pg_encoding_to_char` ( *encoding* `integer` ) → `name`<br>Converts the integer used as the internal identifier of an encoding in some system catalog tables into a human-readable string. Returns an empty string if an invalid encoding number is provided. |
| `pg_get_catalog_foreign_keys` () → `setof record` ( *fktable* `regclass`, *fkcols* `text[]`, *pktable* `regclass`, *pkcols* `text[]`, *is_array* `boolean`, *is_opt* `boolean` )<br>Returns a set of records describing the foreign key relationships that exist within the PostgreSQL system catalogs. The *fktable* column contains the name of the referencing catalog, and the *fkcols* column contains the name(s) of the referencing column(s). Similarly, the *pktable* column contains the name of the referenced catalog, and the *pkcols* column contains the name(s) of the referenced column(s). If *is_array* is true, the last referencing column is an array, each of whose elements should match some entry in the referenced catalog. If *is_opt* is true, the referencing column(s) are allowed to contain zeroes instead of a valid reference. |
| `pg_get_constraintdef` ( *constraint* `oid` [, *pretty* `boolean` ] ) → `text`<br>Reconstructs the creating command for a constraint. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_expr` ( *expr* `pg_node_tree`, *relation* `oid` [, *pretty* `boolean` ] ) → `text`<br>Decompiles the internal form of an expression stored in the system catalogs, such as the default value for a column. If the expression might contain Vars, specify the OID of the relation they refer to as the second parameter; if no Vars are expected, passing zero is sufficient. |
| `pg_get_functiondef` ( *func* `oid` ) → `text`<br>Reconstructs the creating command for a function or procedure. (This is a decompiled reconstruction, not the original text of the command.) The result is a complete `CREATE OR REPLACE FUNCTION` or `CREATE OR REPLACE PROCEDURE` statement. |
| `pg_get_function_arguments` ( *func* `oid` ) → `text`<br>Reconstructs the argument list of a function or procedure, in the form it would need to appear in within `CREATE FUNCTION` (including default values). |
| `pg_get_function_identity_arguments` ( *func* `oid` ) → `text`<br>Reconstructs the argument list necessary to identify a function or procedure, in the form it would need to appear in within commands such as `ALTER FUNCTION`. This form omits default values. |
| `pg_get_function_result` ( *func* `oid` ) → `text`<br>Reconstructs the `RETURNS` clause of a function, in the form it would need to appear in within `CREATE FUNCTION`. Returns `NULL` for a procedure. |
| `pg_get_indexdef` ( *index* `oid` [, *column* `integer`, *pretty* `boolean` ] ) → `text`<br>Reconstructs the creating command for an index. (This is a decompiled reconstruction, not the original text of the command.) If *column* is supplied and is not zero, only the definition of that column is reconstructed. |
| `pg_get_keywords` () → `setof record` ( *word* `text`, *catcode* `"char"`, *barelabel* `boolean`, *catdesc* `text`, *baredesc* `text` )<br>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 an unreserved keyword, `C` for a keyword that can be a column name, `T` for a keyword that can be a type or function name, or `R` for a fully reserved keyword. The *barelabel* column contains `true` if the keyword can be used as a “bare” column label in `SELECT` lists, or `false` if it can only be used after `AS`. The *catdesc* column contains a possibly-localized string describing the keyword's category. The *baredesc* column contains a possibly-localized string describing the keyword's column label status. |
| `pg_get_partkeydef` ( *table* `oid` ) → `text` |

| Function<br>Description |
|---|
| Reconstructs the definition of a partitioned table's partition key, in the form it would have in the `PARTITION BY` clause of `CREATE TABLE`. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_ruledef` ( *rule* `oid` [, *pretty* `boolean` ] ) → `text`<br>Reconstructs the creating command for a rule. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_serial_sequence` ( *table* `text`, *column* `text` ) → `text`<br>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 that 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, the association can be modified or removed with `ALTER SEQUENCE OWNED BY`. (This function probably should have been called `pg_get_owned_sequence`; its current name reflects the fact that it has historically been used with serial-type columns.) The first parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter potentially contains both schema and table names, it is parsed per usual SQL rules, meaning it is lower-cased by default. The second parameter, being just a column name, is treated literally and so has its case preserved. The result is suitably formatted for passing to the sequence functions (see Section 9.17).<br>A typical use is in reading the current value of the sequence for an identity or serial column, for example:<br>`SELECT currval(pg_get_serial_sequence('sometable', 'id'));` |
| `pg_get_statisticsobjdef` ( *statobj* `oid` ) → `text`<br>Reconstructs the creating command for an extended statistics object. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_triggerdef` ( *trigger* `oid` [, *pretty* `boolean` ] ) → `text`<br>Reconstructs the creating command for a trigger. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_userbyid` ( *role* `oid` ) → `name`<br>Returns a role's name given its OID. |
| `pg_get_viewdef` ( *view* `oid` [, *pretty* `boolean` ] ) → `text`<br>Reconstructs the underlying `SELECT` command for a view or materialized view. (This is a decompiled reconstruction, not the original text of the command.) |
| `pg_get_viewdef` ( *view* `oid`, *wrap_column* `integer` ) → `text`<br>Reconstructs the underlying `SELECT` command for a view or materialized view. (This is a decompiled reconstruction, not the original text of the command.) In this form of the function, pretty-printing is always enabled, and long lines are wrapped to try to keep them shorter than the specified number of columns. |
| `pg_get_viewdef` ( *view* `text` [, *pretty* `boolean` ] ) → `text`<br>Reconstructs the underlying `SELECT` command for a view or materialized view, working from a textual name for the view rather than its OID. (This is deprecated; use the OID variant instead.) |
| `pg_index_column_has_property` ( *index* `regclass`, *column* `integer`, *property* `text` ) → `boolean`<br>Tests whether an index column has the named property. Common index column properties are listed in Table 9.77. (Note that extension access methods can define additional property names for their indexes.) `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. |

| Function<br>Description |
|---|
| `pg_index_has_property` ( *index* `regclass`, *property* `text` ) → `boolean`<br>Tests whether an index has the named property. Common index properties are listed in Table 9.78. (Note that extension access methods can define additional property names for their indexes.) `NULL` is returned if the property name is not known or does not apply to the particular object, or if the OID does not identify a valid object. |
| `pg_indexam_has_property` ( *am* `oid`, *property* `text` ) → `boolean`<br>Tests whether an index access method has the named property. Access method properties are listed in Table 9.79. `NULL` is returned if the property name is not known or does not apply to the particular object, or if the OID does not identify a valid object. |
| `pg_options_to_table` ( *options_array* `text[]` ) → `setof record` ( *option_name* `text`, *option_value* `text` )<br>Returns the set of storage options represented by a value from `pg_class`.`reloptions` or `pg_attribute`.`attoptions`. |
| `pg_settings_get_flags` ( *guc* `text` ) → `text[]`<br>Returns an array of the flags associated with the given GUC, or `NULL` if it does not exist. The result is an empty array if the GUC exists but there are no flags to show. Only the most useful flags listed in Table 9.80 are exposed. |
| `pg_tablespace_databases` ( *tablespace* `oid` ) → `setof oid`<br>Returns the set of OIDs of databases that have objects stored in the specified tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To identify the specific objects populating the tablespace, you will need to connect to the database(s) identified by `pg_tablespace_databases` and query their `pg_class` catalogs. |
| `pg_tablespace_location` ( *tablespace* `oid` ) → `text`<br>Returns the file system path that this tablespace is located in. |
| `pg_typeof` ( `"any"` ) → `regtype`<br>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 8.19); this means that it is the same as an OID for comparison purposes but displays as a type name.<br>`pg_typeof(33)` → `integer` |
| `COLLATION FOR` ( `"any"` ) → `text`<br>Returns the name of the collation of the value that is passed to it. The value is quoted and schema-qualified if necessary. If no collation was derived for the argument expression, then `NULL` is returned. If the argument is not of a collatable data type, then an error is raised.<br>`collation for ('foo'::text)` → `"default"`<br>`collation for ('foo' COLLATE "de_DE")` → `"de_DE"` |
| `to_regclass` ( `text` ) → `regclass`<br>Translates a textual relation name to its OID. A similar result is obtained by casting the string to type `regclass` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regcollation` ( `text` ) → `regcollation`<br>Translates a textual collation name to its OID. A similar result is obtained by casting the string to type `regcollation` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regnamespace` ( `text` ) → `regnamespace` |

| Function<br>Description |
|---|
| Translates a textual schema name to its OID. A similar result is obtained by casting the string to type `regnamespace` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regoper` ( `text` ) → `regoper`<br>Translates a textual operator name to its OID. A similar result is obtained by casting the string to type `regoper` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found or is ambiguous. |
| `to_regoperator` ( `text` ) → `regoperator`<br>Translates a textual operator name (with parameter types) to its OID. A similar result is obtained by casting the string to type `regoperator` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regproc` ( `text` ) → `regproc`<br>Translates a textual function or procedure name to its OID. A similar result is obtained by casting the string to type `regproc` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found or is ambiguous. |
| `to_regprocedure` ( `text` ) → `regprocedure`<br>Translates a textual function or procedure name (with argument types) to its OID. A similar result is obtained by casting the string to type `regprocedure` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regrole` ( `text` ) → `regrole`<br>Translates a textual role name to its OID. A similar result is obtained by casting the string to type `regrole` (see Section 8.19); however, this function will return `NULL` rather than throwing an error if the name is not found. |
| `to_regtype` ( `text` ) → `regtype`<br>Parses a string of text, extracts a potential type name from it, and translates that name into a type OID. A syntax error in the string will result in an error; but if the string is a syntactically valid type name that happens not to be found in the catalogs, the result is `NULL`. A similar result is obtained by casting the string to type `regtype` (see Section 8.19), except that that will throw error for name not found. |
| `to_regtypemod` ( `text` ) → `integer`<br>Parses a string of text, extracts a potential type name from it, and translates its type modifier, if any. A syntax error in the string will result in an error; but if the string is a syntactically valid type name that happens not to be found in the catalogs, the result is `NULL`. The result is `-1` if no type modifier is present.<br>`to_regtypemod` can be combined with to_regtype to produce appropriate inputs for format_type, allowing a string representing a type name to be canonicalized.<br>`format_type(to_regtype('varchar(32)'), to_regtypemod('varchar(32)'))` → `character varying(32)` |

Most of the functions that reconstruct (decompile) database objects have an optional *`pretty`* flag, which if `true` causes the result to be "pretty-printed". Pretty-printing suppresses unnecessary parentheses and adds whitespace for legibility. 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; so avoid using pretty-printed output for dump purposes. Passing `false` for the *`pretty`* parameter yields the same result as omitting the parameter.

## Table 9.77. Index Column Properties

| Name | Description |
|---|---|
| `asc` | Does the column sort in ascending order on a forward scan? |

| Name | Description |
|---|---|
| `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 9.78. 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 9.79. 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? |
| `can_include` | Does the access method support the `INCLUDE` clause of `CREATE INDEX`? |

### Table 9.80. GUC Flags

| Flag | Description |
|---|---|
| `EXPLAIN` | Parameters with this flag are included in `EXPLAIN (SETTINGS)` commands. |
| `NO_SHOW_ALL` | Parameters with this flag are excluded from `SHOW ALL` commands. |
| `NO_RESET` | Parameters with this flag do not support `RESET` commands. |

| Flag | Description |
|---|---|
| NO_RESET_ALL | Parameters with this flag are excluded from RESET ALL commands. |
| NOT_IN_SAMPLE | Parameters with this flag are not included in postgresql.conf by default. |
| RUNTIME_COMPUTED | Parameters with this flag are runtime-computed ones. |

## 9.27.5. Object Information and Addressing Functions

Table 9.81 lists functions related to database object identification and addressing.

### Table 9.81. Object Information and Addressing Functions

| Function<br>Description |
|---|
| pg_get_acl ( *classid* oid, *objid* oid, *objsubid* integer ) → aclitem[]<br>Returns the ACL for a database object, specified by catalog OID, object OID and sub-object ID. This function returns NULL values for undefined objects. |
| pg_describe_object ( *classid* oid, *objid* oid, *objsubid* integer ) → text<br>Returns a textual description of a database object identified 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 especially useful to determine the identity of an object referenced in the pg_depend catalog. This function returns NULL values for undefined objects. |
| pg_identify_object ( *classid* oid, *objid* oid, *objsubid* integer ) → record ( *type* text, *schema* text, *name* text, *identity* text )<br>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. Undefined objects are identified with NULL values. |
| pg_identify_object_as_address ( *classid* oid, *objid* oid, *objsubid* integer ) → record ( *type* text, *object_names* text[], *object_args* text[] )<br>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. |
| pg_get_object_address ( *type* text, *object_names* text[], *object_args* text[] ) → record ( *classid* oid, *objid* oid, *objsubid* integer )<br>Returns a row containing enough information to uniquely identify the database object specified by a type code and object name and argument arrays. The returned values are the ones that would be used in system catalogs such as pg_depend; they can be passed to other system functions such as pg_describe_object or pg_identify_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. Undefined objects are identified with NULL values. |

`pg_get_acl` is useful for retrieving and inspecting the privileges associated with database objects without looking at specific catalogs. For example, to retrieve all the granted privileges on objects in the current database:

```
postgres=# SELECT
    (pg_identify_object(s.classid,s.objid,s.objsubid)).*,
    pg_catalog.pg_get_acl(s.classid,s.objid,s.objsubid) AS acl
FROM pg_catalog.pg_shdepend AS s
JOIN pg_catalog.pg_database AS d
    ON d.datname = current_database() AND
       d.oid = s.dbid
JOIN pg_catalog.pg_authid AS a
    ON a.oid = s.refobjid AND
       s.refclassid = 'pg_authid'::regclass
WHERE s.deptype = 'a';
-[ RECORD 1 ]-----------------------------------------
type     | table
schema   | public
name     | testtab
identity | public.testtab
acl      | {postgres=arwdDxtm/postgres,foo=r/postgres}
```

## 9.27.6. Comment Information Functions

The functions shown in Table 9.82 extract comments previously stored with the COMMENT command. A null value is returned if no comment could be found for the specified parameters.

**Table 9.82. Comment Information Functions**

| Function<br>Description |
|---|
| `col_description` ( *`table`* `oid`, *`column`* `integer` ) → `text`<br>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.) |
| `obj_description` ( *`object`* `oid`, *`catalog`* `name` ) → `text`<br>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. |
| `obj_description` ( *`object`* `oid` ) → `text`<br>Returns the comment for a database object specified by its OID alone. This is *deprecated* since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment might be returned. |
| `shobj_description` ( *`object`* `oid`, *`catalog`* `name` ) → `text`<br>Returns the comment for a shared database object specified by its OID and the name of the containing system catalog. This is just like `obj_description` except that it is used for retrieving comments on shared objects (that is, databases, roles, and tablespaces). Some system catalogs are global to all databases within each cluster, and the descriptions for objects in them are stored globally as well. |

## 9.27.7. Data Validity Checking Functions

The functions shown in Table 9.83 can be helpful for checking validity of proposed input data.

### Table 9.83. Data Validity Checking Functions

<table>
<tr><th>Function<br>Description<br>Example(s)</th></tr>
<tr><td><code>pg_input_is_valid</code> ( string <code>text</code>, type <code>text</code> ) → <code>boolean</code><br>Tests whether the given string is valid input for the specified data type, returning true or false.<br>This function will only work as desired if the data type's input function has been updated to report invalid input as a “soft” error. Otherwise, invalid input will abort the transaction, just as if the string had been cast to the type directly.<br><code>pg_input_is_valid('42', 'integer')</code> → <code>t</code><br><code>pg_input_is_valid('42000000000', 'integer')</code> → <code>f</code><br><code>pg_input_is_valid('1234.567', 'numeric(7,4)')</code> → <code>f</code></td></tr>
<tr><td><code>pg_input_error_info</code> ( string <code>text</code>, type <code>text</code> ) → <code>record</code> ( message <code>text</code>, detail <code>text</code>, hint <code>text</code>, sql_error_code <code>text</code> )<br>Tests whether the given string is valid input for the specified data type; if not, return the details of the error that would have been thrown. If the input is valid, the results are NULL. The inputs are the same as for <code>pg_input_is_valid</code>.<br>This function will only work as desired if the data type's input function has been updated to report invalid input as a “soft” error. Otherwise, invalid input will abort the transaction, just as if the string had been cast to the type directly.<br><code>SELECT * FROM pg_input_error_info('42000000000', 'integer')</code> →<br><pre>
                       message                        | detail | hint | sql_error_code
------------------------------------------------------+--------+------+----------------
 value "42000000000" is out of range for type integer |        |      | 22003
</pre></td></tr>
</table>

## 9.27.8. Transaction ID and Snapshot Information Functions

The functions shown in Table 9.84 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 9.84. Transaction ID and Snapshot Information Functions

| Function<br>Description |
|---|
| `age` ( `xid` ) → `integer`<br>Returns the number of transactions between the supplied transaction id and the current transaction counter. |
| `mxid_age` ( `xid` ) → `integer`<br>Returns the number of multixacts IDs between the supplied multixact ID and the current multixacts counter. |
| `pg_current_xact_id` () → `xid8`<br>Returns the current transaction's ID. It will assign a new one if the current transaction does not have one already (because it has not performed any database updates); see Section 67.1 for details. If executed in a subtransaction, this will return the top-level transaction ID; see Section 67.3 for details. |
| `pg_current_xact_id_if_assigned` () → `xid8` |

| Function<br>Description |
|---|
| Returns the current transaction's ID, or `NULL` if no ID is assigned yet. (It's best to use this variant if the transaction might otherwise be read-only, to avoid unnecessary consumption of an XID.) If executed in a subtransaction, this will return the top-level transaction ID. |
| `pg_xact_status` ( `xid8` ) → `text`<br>Reports the commit status of a recent transaction. The result is one of `in progress`, `committed`, or `aborted`, provided that the transaction is recent enough that the system retains the commit status of that transaction. If it is old enough that no references to the transaction survive in the system and the commit status information has been discarded, the result is `NULL`. Applications might use this function, for example, to determine whether their transaction committed or aborted after the application and database server become disconnected while a `COMMIT` is in progress. Note that prepared transactions are reported as `in progress`; applications must check `pg_prepared_xacts` if they need to determine whether a transaction ID belongs to a prepared transaction. |
| `pg_current_snapshot` () → `pg_snapshot`<br>Returns a current *snapshot*, a data structure showing which transaction IDs are now in-progress. Only top-level transaction IDs are included in the snapshot; subtransaction IDs are not shown; see Section 67.3 for details. |
| `pg_snapshot_xip` ( `pg_snapshot` ) → `setof xid8`<br>Returns the set of in-progress transaction IDs contained in a snapshot. |
| `pg_snapshot_xmax` ( `pg_snapshot` ) → `xid8`<br>Returns the `xmax` of a snapshot. |
| `pg_snapshot_xmin` ( `pg_snapshot` ) → `xid8`<br>Returns the `xmin` of a snapshot. |
| `pg_visible_in_snapshot` ( `xid8`, `pg_snapshot` ) → `boolean`<br>Is the given transaction ID *visible* according to this snapshot (that is, was it completed before the snapshot was taken)? Note that this function will not give the correct answer for a subtransaction ID (subxid); see Section 67.3 for details. |

The internal transaction ID type `xid` is 32 bits wide and wraps around every 4 billion transactions. However, the functions shown in Table 9.84, except `age` and `mxid_age`, use a 64-bit type `xid8` that does not wrap around during the life of an installation and can be converted to `xid` by casting if required; see Section 67.1 for details. The data type `pg_snapshot` stores information about transaction ID visibility at a particular moment in time. Its components are described in Table 9.85. `pg_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`.

### Table 9.85. Snapshot Components

| Name | Description |
|---|---|
| `xmin` | Lowest transaction ID that was still active. All transaction IDs less than `xmin` are either committed and visible, or rolled back and dead. |
| `xmax` | One past the highest completed transaction ID. All transaction IDs greater than or equal to `xmax` had not yet completed as of the time of the snapshot, and thus are invisible. |
| `xip_list` | Transactions in progress at the time of the snapshot. A transaction ID that is `xmin <= X < xmax` and not in this list was already completed at the time of the snapshot, and thus is either visible or dead according to its commit status. This list does not include the transaction IDs of subtransactions (subxids). |

In releases of PostgreSQL before 13 there was no `xid8` type, so variants of these functions were provided that used `bigint` to represent a 64-bit XID, with a correspondingly distinct snapshot data type `txid_snapshot`. These older functions have `txid` in their names. They are still supported for backward compatibility, but may be removed from a future release. See Table 9.86.

**Table 9.86. Deprecated Transaction ID and Snapshot Information Functions**

| Function<br>Description |
|---|
| `txid_current` () → `bigint`<br>See `pg_current_xact_id()`. |
| `txid_current_if_assigned` () → `bigint`<br>See `pg_current_xact_id_if_assigned()`. |
| `txid_current_snapshot` () → `txid_snapshot`<br>See `pg_current_snapshot()`. |
| `txid_snapshot_xip` ( `txid_snapshot` ) → `setof bigint`<br>See `pg_snapshot_xip()`. |
| `txid_snapshot_xmax` ( `txid_snapshot` ) → `bigint`<br>See `pg_snapshot_xmax()`. |
| `txid_snapshot_xmin` ( `txid_snapshot` ) → `bigint`<br>See `pg_snapshot_xmin()`. |
| `txid_visible_in_snapshot` ( `bigint`, `txid_snapshot` ) → `boolean`<br>See `pg_visible_in_snapshot()`. |
| `txid_status` ( `bigint` ) → `text`<br>See `pg_xact_status()`. |

## 9.27.9. Committed Transaction Information Functions

The functions shown in Table 9.87 provide information about when past transactions were committed. They only provide useful data when the track_commit_timestamp configuration option is enabled, and only for transactions that were committed after it was enabled. Commit timestamp information is routinely removed during vacuum.

**Table 9.87. Committed Transaction Information Functions**

| Function<br>Description |
|---|
| `pg_xact_commit_timestamp` ( `xid` ) → `timestamp with time zone`<br>Returns the commit timestamp of a transaction. |
| `pg_xact_commit_timestamp_origin` ( `xid` ) → `record` ( *`timestamp`* `timestamp with time zone`, *`roident`* `oid`)<br>Returns the commit timestamp and replication origin of a transaction. |
| `pg_last_committed_xact` () → `record` ( *`xid`* `xid`, *`timestamp`* `timestamp with time zone`, *`roident`* `oid` )<br>Returns the transaction ID, commit timestamp and replication origin of the latest committed transaction. |

## 9.27.10. Control Data Functions

The functions shown in Table 9.88 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, not specific to any one database. These functions provide most of the same information, from the same source, as the pg_controldata application.

### Table 9.88. Control Data Functions

| Function<br>Description |
|---|
| pg_control_checkpoint () → record<br>Returns information about current checkpoint state, as shown in Table 9.89. |
| pg_control_system () → record<br>Returns information about current control file state, as shown in Table 9.90. |
| pg_control_init () → record<br>Returns information about cluster initialization state, as shown in Table 9.91. |
| pg_control_recovery () → record<br>Returns information about recovery state, as shown in Table 9.92. |

### Table 9.89. pg_control_checkpoint Output Columns

| Column Name | Data Type |
|---|---|
| checkpoint_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 |

### Table 9.90. pg_control_system Output Columns

| Column Name | Data Type |
|---|---|
| pg_control_version | integer |
| catalog_version_no | integer |
| system_identifier | bigint |
| pg_control_last_modified | timestamp with time zone |

**Table 9.91. pg_control_init Output 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 |
| float8_pass_by_value | boolean |
| data_page_checksum_version | integer |
| default_char_signedness | boolean |

**Table 9.92. pg_control_recovery Output 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 |

## 9.27.11. Version Information Functions

The functions shown in Table 9.93 print version information.

**Table 9.93. Version Information Functions**

| Function<br>Description |
|---|
| version () → text<br>Returns a string describing the PostgreSQL server's version. You can also get this information from server_version, or for a machine-readable version use server_version_num. Software developers should use server_version_num (available since 8.2) or PQserverVersion instead of parsing the text version. |
| unicode_version () → text<br>Returns a string representing the version of Unicode used by PostgreSQL. |
| icu_unicode_version () → text<br>Returns a string representing the version of Unicode used by ICU, if the server was built with ICU support; otherwise returns NULL |

## 9.27.12. WAL Summarization Information Functions

The functions shown in Table 9.94 print information about the status of WAL summarization. See summarize_wal.

**Table 9.94. WAL Summarization Information Functions**

| Function<br>Description |
|---|
| `pg_available_wal_summaries` () → `setof record` ( *`tli`* `bigint`, *`start_lsn`* `pg_lsn`, *`end_lsn`* `pg_lsn` )<br>Returns information about the WAL summary files present in the data directory, under `pg_wal/summaries`. One row will be returned per WAL summary file. Each file summarizes WAL on the indicated TLI within the indicated LSN range. This function might be useful to determine whether enough WAL summaries are present on the server to take an incremental backup based on some prior backup whose start LSN is known. |
| `pg_wal_summary_contents` ( *`tli`* `bigint`, *`start_lsn`* `pg_lsn`, *`end_lsn`* `pg_lsn` ) → `setof record` ( *`relfilenode`* `oid`, *`reltablespace`* `oid`, *`reldatabase`* `oid`, *`relforknumber`* `smallint`, *`relblocknumber`* `bigint`, *`is_limit_block`* `boolean` )<br>Returns one information about the contents of a single WAL summary file identified by TLI and starting and ending LSNs. Each row with `is_limit_block` false indicates that the block identified by the remaining output columns was modified by at least one WAL record within the range of records summarized by this file. Each row with `is_limit_block` true indicates either that (a) the relation fork was truncated to the length given by `relblocknumber` within the relevant range of WAL records or (b) that the relation fork was created or dropped within the relevant range of WAL records; in such cases, `relblocknumber` will be zero. |
| `pg_get_wal_summarizer_state` () → `record` ( *`summarized_tli`* `bigint`, *`summarized_lsn`* `pg_lsn`, *`pending_lsn`* `pg_lsn`, *`summarizer_pid`* `int` )<br>Returns information about the progress of the WAL summarizer. If the WAL summarizer has never run since the instance was started, then `summarized_tli` and `summarized_lsn` will be `0` and `0/0` respectively; otherwise, they will be the TLI and ending LSN of the last WAL summary file written to disk. If the WAL summarizer is currently running, `pending_lsn` will be the ending LSN of the last record that it has consumed, which must always be greater than or equal to `summarized_lsn`; if the WAL summarizer is not running, it will be equal to `summarized_lsn`. `summarizer_pid` is the PID of the WAL summarizer process, if it is running, and otherwise NULL.<br>As a special exception, the WAL summarizer will refuse to generate WAL summary files if run on WAL generated under `wal_level=minimal`, since such summaries would be unsafe to use as the basis for an incremental backup. In this case, the fields above will continue to advance as if summaries were being generated, but nothing will be written to disk. Once the summarizer reaches WAL generated while `wal_level` was set to `replica` or higher, it will resume writing summaries to disk. |

# 9.28. System Administration Functions

The functions described in this section are used to control and monitor a PostgreSQL installation.

## 9.28.1. Configuration Settings Functions

Table 9.95 shows the functions available to query and alter run-time configuration parameters.

**Table 9.95. Configuration Settings Functions**

| Function<br>Description<br>Example(s) |
|---|
| `current_setting` ( *`setting_name`* `text` [, *`missing_ok`* `boolean` ] ) → `text`<br>Returns the current value of the setting *`setting_name`*. If there is no such setting, `current_setting` throws an error unless *`missing_ok`* is supplied and is `true` (in which case NULL is returned). This function corresponds to the SQL command SHOW. |

| Function<br>Description<br>Example(s) |
|---|
| `current_setting('datestyle')` → `ISO, MDY` |
| `set_config` ( *`setting_name`* `text`, *`new_value`* `text`, *`is_local`* `boolean` ) → `text`<br>Sets the parameter *`setting_name`* to *`new_value`*, and returns that value. If *`is_local`* is `true`, the new value will only apply during the current transaction. If you want the new value to apply for the rest of the current session, use `false` instead. This function corresponds to the SQL command SET.<br>`set_config` accepts the NULL value for *`new_value`*, but as settings cannot be null, it is interpreted as a request to reset the setting to its default value.<br>`set_config('log_statement_stats', 'off', false)` → `off` |

## 9.28.2. Server Signaling Functions

The functions shown in Table 9.96 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.

Each of these functions returns `true` if the signal was successfully sent and `false` if sending the signal failed.

### Table 9.96. Server Signaling Functions

| Function<br>Description |
|---|
| `pg_cancel_backend` ( *`pid`* `integer` ) → `boolean`<br>Cancels the current query of the session whose backend process has the specified process ID. This is also allowed if the calling role is a member of the role whose backend is being canceled or the calling role has privileges of `pg_signal_backend`, however only superusers can cancel superuser backends. As an exception, roles with privileges of `pg_signal_autovacuum_worker` are permitted to cancel autovacuum worker processes, which are otherwise considered superuser backends. |
| `pg_log_backend_memory_contexts` ( *`pid`* `integer` ) → `boolean`<br>Requests to log the memory contexts of the backend with the specified process ID. This function can send the request to backends and auxiliary processes except logger. These memory contexts will be logged at `LOG` message level. They will appear in the server log based on the log configuration set (see Section 19.8 for more information), but will not be sent to the client regardless of client_min_messages. |
| `pg_reload_conf` () → `boolean`<br>Causes all processes of the PostgreSQL server to reload their configuration files. (This is initiated by sending a SIGHUP signal to the postmaster process, which in turn sends SIGHUP to each of its children.) You can use the `pg_file_settings`, `pg_hba_file_rules` and `pg_ident_file_mappings` views to check the configuration files for possible errors, before reloading. |
| `pg_rotate_logfile` () → `boolean`<br>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. |
| `pg_terminate_backend` ( *`pid`* `integer`, *`timeout`* `bigint` DEFAULT `0` ) → `boolean`<br>Terminates the session whose backend process has the specified process ID. This is also allowed if the calling role is a member of the role whose backend is being terminated or the calling role has privileges of `pg_signal_backend`, however only superusers can terminate superuser backends. As an exception, roles with privileges of `pg_signal_autovacuum_worker` are permitted to terminate autovacuum worker processes, which are otherwise considered superuser backends.<br>If *`timeout`* is not specified or zero, this function returns `true` whether the process actually terminates or not, indicating only that the sending of the signal was successful. If the *`timeout`* is speci- |

| Function<br>Description |
|---|
| fied (in milliseconds) and greater than zero, the function waits until the process is actually terminated or until the given time has passed. If the process is terminated, the function returns `true`. On time-out, a warning is emitted and `false` is returned. |

`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_log_backend_memory_contexts` can be used to log the memory contexts of a backend process. For example:

```
postgres=# SELECT pg_log_backend_memory_contexts(pg_backend_pid());
 pg_log_backend_memory_contexts
--------------------------------
 t
(1 row)
```

One message for each memory context will be logged. For example:

```
LOG:  logging memory contexts of PID 10377
STATEMENT:  SELECT pg_log_backend_memory_contexts(pg_backend_pid());
LOG:  level: 1; TopMemoryContext: 80800 total in 6 blocks; 14432 free (5
 chunks); 66368 used
LOG:  level: 2; pgstat TabStatusArray lookup hash table: 8192 total in 1
 blocks; 1408 free (0 chunks); 6784 used
LOG:  level: 2; TopTransactionContext: 8192 total in 1 blocks; 7720 free (1
 chunks); 472 used
LOG:  level: 2; RowDescriptionContext: 8192 total in 1 blocks; 6880 free (0
 chunks); 1312 used
LOG:  level: 2; MessageContext: 16384 total in 2 blocks; 5152 free (0
 chunks); 11232 used
LOG:  level: 2; Operator class cache: 8192 total in 1 blocks; 512 free (0
 chunks); 7680 used
LOG:  level: 2; smgr relation table: 16384 total in 2 blocks; 4544 free (3
 chunks); 11840 used
LOG:  level: 2; TransactionAbortContext: 32768 total in 1 blocks; 32504
 free (0 chunks); 264 used
...
LOG:  level: 2; ErrorContext: 8192 total in 1 blocks; 7928 free (3 chunks);
 264 used
LOG:  Grand total: 1651920 bytes in 201 blocks; 622360 free (88 chunks);
 1029560 used
```

If there are more than 100 child contexts under the same parent, the first 100 child contexts are logged, along with a summary of the remaining contexts. Note that frequent calls to this function could incur significant overhead, because it may generate a large number of log messages.

## 9.28.3. Backup Control Functions

The functions shown in Table 9.97 assist in making on-line backups. These functions cannot be executed during recovery (except `pg_backup_start`, `pg_backup_stop`, and `pg_wal_lsn_diff`).

For details about proper usage of these functions, see Section 25.3.

### Table 9.97. Backup Control Functions

| Function<br>Description |
|---|
| `pg_create_restore_point` ( *name* `text` ) → `pg_lsn`<br>Creates a named marker record in the write-ahead log that can later be used as a recovery target, and returns the corresponding write-ahead log location. The given name can then be used with 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.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_current_wal_flush_lsn` () → `pg_lsn`<br>Returns the current write-ahead log flush location (see notes below). |
| `pg_current_wal_insert_lsn` () → `pg_lsn`<br>Returns the current write-ahead log insert location (see notes below). |
| `pg_current_wal_lsn` () → `pg_lsn`<br>Returns the current write-ahead log write location (see notes below). |
| `pg_backup_start` ( *label* `text` [, *fast* `boolean` ] ) → `pg_lsn`<br>Prepares the server to begin an on-line backup. The only required parameter is an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) If the optional second parameter is given as `true`, it specifies executing `pg_backup_start` as quickly as possible. This forces an immediate checkpoint which will cause a spike in I/O operations, slowing any concurrently executing queries.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_backup_stop` ( [*wait_for_archive* `boolean` ] ) → `record` ( *lsn* `pg_lsn`, *labelfile* `text`, *spcmapfile* `text` )<br>Finishes performing an on-line backup. The desired contents of the backup label file and the tablespace map file are returned as part of the result of the function and must be written to files in the backup area. These files must not be written to the live data directory (doing so will cause PostgreSQL to fail to restart in the event of a crash).<br>There is an optional parameter of type `boolean`. If false, the function will return immediately after the backup is completed, without waiting for WAL to be archived. This behavior is only useful with backup software that independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. By default or when this parameter is true, `pg_backup_stop` will wait for WAL to be archived when archiving is enabled. (On a 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.)<br>When executed on a primary, this function also creates a backup history file in the write-ahead log archive area. The history file includes the label given to `pg_backup_start`, the starting and ending write-ahead log locations for the backup, and the starting and ending times of the backup. 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.<br>The result of the function is a single record. The *lsn* column holds the backup's ending write-ahead log location (which again can be ignored). The second column returns the contents of the backup label file, and the third column returns the contents of the tablespace map file. These must be stored as part of the backup and are required as part of the restore process.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_switch_wal` () → `pg_lsn` |

| Function<br>Description |
|---|
| Forces the server to switch to a new write-ahead log file, which allows the current file to be archived (assuming you are using continuous archiving). The result is the ending write-ahead log location plus 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.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_walfile_name` ( *`lsn`* `pg_lsn` ) → `text`<br>Converts a write-ahead log location to the name of the WAL file holding that location. |
| `pg_walfile_name_offset` ( *`lsn`* `pg_lsn` ) → `record` ( *`file_name`* `text`, *`file_offset`* `integer` )<br>Converts a write-ahead log location to a WAL file name and byte offset within that file. |
| `pg_split_walfile_name` ( *`file_name`* `text` ) → `record` ( *`segment_number`* `numeric`, *`timeline_id`* `bigint` )<br>Extracts the sequence number and timeline ID from a WAL file name. |
| `pg_wal_lsn_diff` ( *`lsn1`* `pg_lsn`, *`lsn2`* `pg_lsn` ) → `numeric`<br>Calculates the difference in bytes (*`lsn1`* - *`lsn2`*) between two write-ahead log locations. This can be used with `pg_stat_replication` or some of the functions shown in Table 9.97 to get the replication lag. |

`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 the flush location is the last location known 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 all 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 a `pg_lsn` value. For example:

```
postgres=# SELECT * FROM pg_walfile_name_offset((pg_backup_stop()).lsn);
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Similarly, `pg_walfile_name` extracts just the write-ahead log file name.

`pg_split_walfile_name` is useful to compute a LSN from a file offset and WAL file name, for example:

```
postgres=# \set file_name '000000010000000100C000AB'
postgres=# \set offset 256
postgres=# SELECT '0/0'::pg_lsn + pd.segment_number * ps.setting::int
 + :offset AS lsn
  FROM pg_split_walfile_name(:'file_name') pd,
       pg_show_all_settings() ps
  WHERE ps.name = 'wal_segment_size';
      lsn
---------------
```

```
 C001/AB000100
(1 row)
```

## 9.28.4. Recovery Control Functions

The functions shown in Table 9.98 provide information about the current status of a standby server. These functions may be executed both during recovery and in normal running.

### Table 9.98. Recovery Information Functions

| Function<br>Description |
|---|
| `pg_is_in_recovery` () → `boolean`<br>Returns true if recovery is still in progress. |
| `pg_last_wal_receive_lsn` () → `pg_lsn`<br>Returns the last write-ahead log location that has been received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed then this will remain static at the location 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`<br>Returns the last write-ahead log location that has been replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this will remain static at the location of the last WAL record applied during recovery. When the server has been started normally without recovery, the function returns `NULL`. |
| `pg_last_xact_replay_timestamp` () → `timestamp with time zone`<br>Returns the time stamp of the 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, the function returns `NULL`. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this will remain static at the time of the last transaction applied during recovery. When the server has been started normally without recovery, the function returns `NULL`. |
| `pg_get_wal_resource_managers` () → `setof record` ( *`rm_id`* `integer`, *`rm_name`* `text`, *`rm_builtin`* `boolean` )<br>Returns the currently-loaded WAL resource managers in the system. The column *`rm_builtin`* indicates whether it's a built-in resource manager, or a custom resource manager loaded by an extension. |

The functions shown in Table 9.99 control the progress of recovery. These functions may be executed only during recovery.

### Table 9.99. Recovery Control Functions

| Function<br>Description |
|---|
| `pg_is_wal_replay_paused` () → `boolean`<br>Returns true if recovery pause is requested. |
| `pg_get_wal_replay_pause_state` () → `text`<br>Returns recovery pause state. The return values are `not paused` if pause is not requested, `pause requested` if pause is requested but recovery is not yet paused, and `paused` if the recovery is actually paused. |
| `pg_promote` ( *`wait`* `boolean` `DEFAULT` `true`, *`wait_seconds`* `integer` `DEFAULT` `60` ) → `boolean` |

| Function<br>Description |
|---|
| Promotes a standby server to primary status. With *wait* set to true (the default), the function waits until promotion is completed or *wait_seconds* seconds have passed, and returns true if promotion is successful and false otherwise. If *wait* is set to false, the function returns true immediately after sending a SIGUSR1 signal to the postmaster to trigger promotion.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| pg_wal_replay_pause () → void<br>Request to pause recovery. A request doesn't mean that recovery stops right away. If you want a guarantee that recovery is actually paused, you need to check for the recovery pause state returned by pg_get_wal_replay_pause_state(). Note that pg_is_wal_replay_paused() returns whether a request is made. While recovery is paused, no further database changes are applied. If hot standby is active, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until recovery is resumed.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| pg_wal_replay_resume () → void<br>Restarts recovery if it was paused.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |

pg_wal_replay_pause and pg_wal_replay_resume cannot be executed while a promotion is ongoing. If a promotion is triggered while recovery is paused, the paused state ends and promotion continues.

If streaming replication is disabled, the paused state may continue indefinitely without a problem. If streaming replication is in progress then 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.

## 9.28.5. 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 9.100, and imported with the SET TRANSACTION command.

### Table 9.100. Snapshot Synchronization Functions

| Function<br>Description |
|---|
| pg_export_snapshot () → text<br>Saves the transaction's 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.<br>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, trans- |

| Function<br>Description |
|---|
| actions use the same snapshot throughout their lifetime. Once a transaction has exported any snapshots, it cannot be prepared with PREPARE TRANSACTION. |
| `pg_log_standby_snapshot` () → `pg_lsn`<br>Take a snapshot of running transactions and write it to WAL, without having to wait for bgwriter or checkpointer to log one. This is useful for logical decoding on standby, as logical slot creation has to wait until such a record is replayed on the standby. |

# 9.28.6. Replication Management Functions

The functions shown in Table 9.101 are for controlling and interacting with replication features. See Section 26.2.5, Section 26.2.6, and Chapter 48 for information about the underlying features. Use of functions for replication origin is only allowed to the superuser by default, but may be allowed to other users by using the `GRANT` command. Use of functions for replication slots is restricted to superusers and users having `REPLICATION` privilege.

Many of these functions have equivalent commands in the replication protocol; see Section 54.4.

The functions described in Section 9.28.3, Section 9.28.4, and Section 9.28.5 are also relevant for replication.

### Table 9.101. Replication Management Functions

| Function<br>Description |
|---|
| `pg_create_physical_replication_slot` ( *`slot_name`* `name` [, *`immediately_reserve`* `boolean`, *`temporary`* `boolean` ] ) → `record` ( *`slot_name`* `name`, *`lsn`* `pg_lsn` )<br>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 54.4. 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 the 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`<br>Drops the physical or logical replication slot named *`slot_name`*. Same as replication protocol command `DROP_REPLICATION_SLOT`. |
| `pg_create_logical_replication_slot` ( *`slot_name`* `name`, *`plugin`* `name` [, *`temporary`* `boolean`, *`twophase`* `boolean`, *`failover`* `boolean` ] ) → `record` ( *`slot_name`* `name`, *`lsn`* `pg_lsn` )<br>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 the current session. Temporary slots are also released upon any error. The optional fourth parameter, *`twophase`*, when set to true, specifies that the decoding of prepared transactions is enabled for this slot. The optional fifth parameter, *`failover`*, when set to true, specifies that this slot is enabled to be synced to the standbys so that logical replication can be resumed after failover. A call to this function has the same effect as the replication protocol command `CREATE_REPLICATION_SLOT ... LOGICAL`. |
| `pg_copy_physical_replication_slot` ( *`src_slot_name`* `name`, *`dst_slot_name`* `name` [, *`temporary`* `boolean` ] ) → `record` ( *`slot_name`* `name`, *`lsn`* `pg_lsn` )<br>Copies an existing physical replication slot named *`src_slot_name`* to a physical replication slot named *`dst_slot_name`*. The copied physical slot starts to reserve WAL from the same LSN as the source slot. *`temporary`* is optional. If *`temporary`* is omitted, the same value as the source slot is used. Copy of an invalidated slot is not allowed. |

| Function<br>Description |
|---|
| `pg_copy_logical_replication_slot` ( *`src_slot_name`* `name`, *`dst_slot_name`* `name` [, *`temporary`* `boolean` [, *`plugin`* `name` ]] ) → `record` ( *`slot_name`* `name`, *`lsn`* `pg_lsn` )<br>Copies an existing logical replication slot named *`src_slot_name`* to a logical replication slot named *`dst_slot_name`*, optionally changing the output plugin and persistence. The copied logical slot starts from the same LSN as the source logical slot. Both *`temporary`* and *`plugin`* are optional; if they are omitted, the values of the source slot are used. The `failover` option of the source logical slot is not copied and is set to `false` by default. This is to avoid the risk of being unable to continue logical replication after failover to standby where the slot is being synchronized. Copy of an invalidated slot is not allowed. |
| `pg_logical_slot_get_changes` ( *`slot_name`* `name`, *`upto_lsn`* `pg_lsn`, *`upto_nchanges`* `integer`, `VARIADIC` *`options`* `text[]` ) → `setof record` ( *`lsn`* `pg_lsn`, *`xid`* `xid`, *`data`* `text` )<br>Returns changes in the slot *`slot_name`*, starting from the point from which 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. If the specified slot is a logical failover slot then the function will not return until all physical slots specified in `synchronized_standby_slots` have confirmed WAL receipt. |
| `pg_logical_slot_peek_changes` ( *`slot_name`* `name`, *`upto_lsn`* `pg_lsn`, *`upto_nchanges`* `integer`, `VARIADIC` *`options`* `text[]` ) → `setof record` ( *`lsn`* `pg_lsn`, *`xid`* `xid`, *`data`* `text` )<br>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`* `integer`, `VARIADIC` *`options`* `text[]` ) → `setof record` ( *`lsn`* `pg_lsn`, *`xid`* `xid`, *`data`* `bytea` )<br>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`* `integer`, `VARIADIC` *`options`* `text[]` ) → `setof record` ( *`lsn`* `pg_lsn`, *`xid`* `xid`, *`data`* `bytea` )<br>Behaves just like the `pg_logical_slot_peek_changes()` function, except that changes are returned as `bytea`. |
| `pg_replication_slot_advance` ( *`slot_name`* `name`, *`upto_lsn`* `pg_lsn` ) → `record` ( *`slot_name`* `name`, *`end_lsn`* `pg_lsn` )<br>Advances the current confirmed position of a replication slot named *`slot_name`*. The slot will not be moved backwards, and it will not be moved beyond the current insert location. Returns the name of the slot and the actual position that it was advanced to. The updated slot position information is written out at the next checkpoint if any advancing is done. So in the event of a crash, the slot may return to an earlier position. If the specified slot is a logical failover slot then the function will not return until all physical slots specified in `synchronized_standby_slots` have confirmed WAL receipt. |
| `pg_replication_origin_create` ( *`node_name`* `text` ) → `oid`<br>Creates a replication origin with the given external name, and returns the internal ID assigned to it. The name must be no longer than 512 bytes. |
| `pg_replication_origin_drop` ( *`node_name`* `text` ) → `void`<br>Deletes a previously-created replication origin, including any associated replay progress. |

| Function<br>Description |
|---|
| `pg_replication_origin_oid` ( *node_name* `text` ) → `oid`<br>Looks up a replication origin by name and returns the internal ID. If no such replication origin is found, `NULL` is returned. |
| `pg_replication_origin_session_setup` ( *node_name* `text` ) → `void`<br>Marks the current session as replaying from the given origin, allowing replay progress to be tracked. Can only be used if no origin is currently selected. Use `pg_replication_origin_session_reset` to undo. |
| `pg_replication_origin_session_reset` () → `void`<br>Cancels the effects of `pg_replication_origin_session_setup()`. |
| `pg_replication_origin_session_is_setup` () → `boolean`<br>Returns true if a replication origin has been selected in the current session. |
| `pg_replication_origin_session_progress` ( *flush* `boolean` ) → `pg_lsn`<br>Returns the replay location for the replication origin selected 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* `timestamp with time zone` ) → `void`<br>Marks 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 been selected using `pg_replication_origin_session_setup`. |
| `pg_replication_origin_xact_reset` () → `void`<br>Cancels the effects of `pg_replication_origin_xact_setup()`. |
| `pg_replication_origin_advance` ( *node_name* `text`, *lsn* `pg_lsn` ) → `void`<br>Sets replication progress for the given node to the given location. This is primarily useful for setting up the initial location, or setting 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* `boolean` ) → `pg_lsn`<br>Returns 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* `boolean`, *prefix* `text`, *content* `text` [, *flush* `boolean` `DEFAULT` `false`] ) → `pg_lsn`<br>`pg_logical_emit_message` ( *transactional* `boolean`, *prefix* `text`, *content* `bytea` [, *flush* `boolean` `DEFAULT` `false`] ) → `pg_lsn`<br>Emits a logical decoding message. This can be used to pass generic messages to logical decoding plugins through WAL. The *transactional* parameter specifies if the message should be part of the current transaction, or if it should be written immediately and decoded as soon as the logical decoder reads the record. The *prefix* parameter is a textual prefix that can be used by logical decoding plugins to easily recognize messages that are interesting for them. The *content* parameter is the content of the message, given either in text or binary form. The *flush* parameter (default set to `false`) controls if the message is immediately flushed to WAL or not. *flush* has no effect with *transactional*, as the message's WAL record is flushed along with its transaction. |
| `pg_sync_replication_slots` () → `void`<br>Synchronize the logical failover replication slots from the primary server to the standby server. This function can only be executed on the standby server. Temporary synced slots, if any, cannot be used for logical decoding and must be dropped after promotion. See Section 47.2.3 for details. Note that this function cannot be executed if `sync_replication_slots` is enabled and the slotsync worker is already running to perform the synchronization of slots. |

| Function<br>Description |
|---|
| **Caution**<br>If, after executing the function, `hot_standby_feedback` is disabled on the standby or the physical slot configured in `primary_slot_name` is removed, then it is possible that the necessary rows of the synchronized slot will be removed by the VACUUM process on the primary server, resulting in the synchronized slot becoming invalidated. |

## 9.28.7. Database Object Management Functions

The functions shown in Table 9.102 calculate the disk space usage of database objects, or assist in presentation or understanding of usage results. `bigint` results are measured in bytes. If an OID that does not represent an existing object is passed to one of these functions, `NULL` is returned.

### Table 9.102. Database Object Size Functions

| Function<br>Description |
|---|
| `pg_column_size` ( `"any"` ) → `integer`<br>Shows the number of bytes used to store any individual data value. If applied directly to a table column value, this reflects any compression that was done. |
| `pg_column_compression` ( `"any"` ) → `text`<br>Shows the compression algorithm that was used to compress an individual variable-length value. Returns `NULL` if the value is not compressed. |
| `pg_column_toast_chunk_id` ( `"any"` ) → `oid`<br>Shows the `chunk_id` of an on-disk TOASTed value. Returns `NULL` if the value is un-TOASTed or not on-disk. See Section 66.2 for more information about TOAST. |
| `pg_database_size` ( `name` ) → `bigint`<br>`pg_database_size` ( `oid` ) → `bigint`<br>Computes the total disk space used by the database with the specified name or OID. To use this function, you must have `CONNECT` privilege on the specified database (which is granted by default) or have privileges of the `pg_read_all_stats` role. |
| `pg_indexes_size` ( `regclass` ) → `bigint`<br>Computes the total disk space used by indexes attached to the specified table. |
| `pg_relation_size` ( *`relation`* `regclass` [, *`fork`* `text` ] ) → `bigint`<br>Computes the disk space used by one "fork" of the specified 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, this returns the size of the main data fork of the relation. The second argument can be provided to specify which fork to examine:<br>• `main` returns the size of the main data fork of the relation.<br>• `fsm` returns the size of the Free Space Map (see Section 66.3) associated with the relation.<br>• `vm` returns the size of the Visibility Map (see Section 66.4) associated with the relation.<br>• `init` returns the size of the initialization fork, if any, associated with the relation. |
| `pg_size_bytes` ( `text` ) → `bigint`<br>Converts a size in human-readable format (as returned by `pg_size_pretty`) into bytes. Valid units are `bytes`, `B`, `kB`, `MB`, `GB`, `TB`, and `PB`. |
| `pg_size_pretty` ( `bigint` ) → `text` |

| Function<br>Description |
|---|
| `pg_size_pretty` ( `numeric` ) → `text`<br>Converts a size in bytes into a more easily human-readable format with size units (bytes, kB, MB, GB, TB, or PB as appropriate). Note that the units are powers of 2 rather than powers of 10, so 1kB is 1024 bytes, 1MB is $1024^2$ = 1048576 bytes, and so on. |
| `pg_table_size` ( `regclass` ) → `bigint`<br>Computes the disk space used by the specified table, excluding indexes (but including its TOAST table if any, free space map, and visibility map). |
| `pg_tablespace_size` ( `name` ) → `bigint`<br>`pg_tablespace_size` ( `oid` ) → `bigint`<br>Computes the total disk space used in the tablespace with the specified name or OID. To use this function, you must have `CREATE` privilege on the specified tablespace or have privileges of the `pg_read_all_stats` role, unless it is the default tablespace for the current database. |
| `pg_total_relation_size` ( `regclass` ) → `bigint`<br>Computes the total disk space used by the specified table, including all indexes and TOAST data. The result is equivalent to `pg_table_size` + `pg_indexes_size`. |

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. See Section 8.19 for details.

The functions shown in Table 9.103 assist in identifying the specific disk files associated with database objects.

### Table 9.103. Database Object Location Functions

| Function<br>Description |
|---|
| `pg_relation_filenode` ( *`relation`* `regclass` ) → `oid`<br>Returns the “filenode” number currently assigned to the specified relation. The filenode is the base component of the file name(s) used for the relation (see Section 66.1 for more information). For most relations 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` ( *`relation`* `regclass` ) → `text`<br>Returns the entire file path name (relative to the database cluster's data directory, `PGDATA`) of the relation. |
| `pg_filenode_relation` ( *`tablespace`* `oid`, *`filenode`* `oid` ) → `regclass`<br>Returns a relation's OID given the tablespace OID and filenode it is stored under. This is essentially the inverse mapping of `pg_relation_filepath`. For a relation in the database's default tablespace, the tablespace can be specified as zero. Returns `NULL` if no relation in the current database is associated with the given values. |

Table 9.104 lists functions used to manage collations.

### Table 9.104. Collation Management Functions

| Function<br>Description |
|---|
| `pg_collation_actual_version` ( `oid` ) → `text`<br>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. |

| Function<br>Description |
|---|
| `pg_database_collation_actual_version` ( `oid` ) → `text`<br>Returns the actual version of the database's collation as it is currently installed in the operating system. If this is different from the value in `pg_database.datcollversion`, then objects depending on the collation might need to be rebuilt. See also ALTER DATABASE. |
| `pg_import_system_collations` ( *`schema`* `regnamespace` ) → `integer`<br>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 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. Use of this function is restricted to superusers. |

Table 9.105 lists functions used to manipulate statistics. These functions cannot be executed during recovery.

## Warning

Changes made by these statistics manipulation functions are likely to be overwritten by autovacuum (or manual `VACUUM` or `ANALYZE`) and should be considered temporary.

### Table 9.105. Database Object Statistics Manipulation Functions

**Function**
**Description**

`pg_restore_relation_stats` ( `VARIADIC` *`kwargs`* `"any"` ) → `boolean`

Updates table-level statistics. Ordinarily, these statistics are collected automatically or updated as a part of VACUUM or ANALYZE, so it's not necessary to call this function. However, it is useful after a restore to enable the optimizer to choose better plans if `ANALYZE` has not been run yet.
The tracked statistics may change from version to version, so arguments are passed as pairs of *`argname`* and *`argvalue`* in the form:

```
SELECT pg_restore_relation_stats(
    'arg1name', 'arg1value'::arg1type,
    'arg2name', 'arg2value'::arg2type,
    'arg3name', 'arg3value'::arg3type);
```

For example, to set the `relpages` and `reltuples` values for the table `mytable`:

```
SELECT pg_restore_relation_stats(
    'schemaname', 'myschema',
    'relname',    'mytable',
    'relpages',   173::integer,
    'reltuples',  10000::real);
```

The arguments `schemaname` and `relname` are required, and specify the table. Other arguments are the names and values of statistics corresponding to certain columns in `pg_class`. The currently-supported relation statistics are `relpages` with a value of type `integer`, `reltuples` with a value of type `real`, `relallvisible` with a value of type `integer`, and `relallfrozen` with a value of type `integer`.

| Function<br>Description |
|---|
| Additionally, this function accepts argument name `version` of type `integer`, which specifies the server version from which the statistics originated. This is anticipated to be helpful in porting statistics from older versions of PostgreSQL.<br>Minor errors are reported as a `WARNING` and ignored, and remaining statistics will still be restored. If all specified statistics are successfully restored, returns `true`, otherwise `false`.<br>The caller must have the `MAINTAIN` privilege on the table or be the owner of the database. |
| `pg_clear_relation_stats` ( *schemaname* `text`, *relname* `text` ) → `void`<br>Clears table-level statistics for the given relation, as though the table was newly created.<br>The caller must have the `MAINTAIN` privilege on the table or be the owner of the database. |
| `pg_restore_attribute_stats` ( `VARIADIC` *kwargs* `"any"` ) → `boolean`<br>Creates or updates column-level statistics. Ordinarily, these statistics are collected automatically or updated as a part of VACUUM or ANALYZE, so it's not necessary to call this function. However, it is useful after a restore to enable the optimizer to choose better plans if `ANALYZE` has not been run yet.<br>The tracked statistics may change from version to version, so arguments are passed as pairs of *argname* and *argvalue* in the form:<br><pre>SELECT pg_restore_attribute_stats(<br>    'arg1name', 'arg1value'::arg1type,<br>    'arg2name', 'arg2value'::arg2type,<br>    'arg3name', 'arg3value'::arg3type);</pre>For example, to set the `avg_width` and `null_frac` values for the attribute `col1` of the table `mytable`:<br><pre>SELECT pg_restore_attribute_stats(<br>    'schemaname', 'myschema',<br>    'relname',    'mytable',<br>    'attname',    'col1',<br>    'inherited',  false,<br>    'avg_width',  125::integer,<br>    'null_frac',  0.5::real);</pre>The required arguments are `schemaname` and `relname` with a value of type `text` which specify the table; either `attname` with a value of type `text` or `attnum` with a value of type `smallint`, which specifies the column; and `inherited`, which specifies whether the statistics include values from child tables. Other arguments are the names and values of statistics corresponding to columns in `pg_stats`.<br>Additionally, this function accepts argument name `version` of type `integer`, which specifies the server version from which the statistics originated. This is anticipated to be helpful in porting statistics from older versions of PostgreSQL.<br>Minor errors are reported as a `WARNING` and ignored, and remaining statistics will still be restored. If all specified statistics are successfully restored, returns `true`, otherwise `false`.<br>The caller must have the `MAINTAIN` privilege on the table or be the owner of the database. |
| `pg_clear_attribute_stats` ( *schemaname* `text`, *relname* `text`, *attname* `text`, *inherited* `boolean` ) → `void`<br>Clears column-level statistics for the given relation and attribute, as though the table was newly created.<br>The caller must have the `MAINTAIN` privilege on the table or be the owner of the database. |

Table 9.106 lists functions that provide information about the structure of partitioned tables.

**Table 9.106. Partitioning Information Functions**

| Function<br>Description |
|---|
| `pg_partition_tree` ( `regclass` ) → `setof record` ( *`relid`* `regclass`, *`parentrelid`* `regclass`, *`isleaf`* `boolean`, *`level`* `integer` )<br>Lists the tables or indexes in the partition tree of the given partitioned table or partitioned index, with one row for each partition. Information provided includes the OID of the partition, the OID of its immediate parent, a boolean value telling if the partition is a leaf, and an integer telling its level in the hierarchy. The level value is 0 for the input table or index, 1 for its immediate child partitions, 2 for their partitions, and so on. Returns no rows if the relation does not exist or is not a partition or partitioned table. |
| `pg_partition_ancestors` ( `regclass` ) → `setof regclass`<br>Lists the ancestor relations of the given partition, including the relation itself. Returns no rows if the relation does not exist or is not a partition or partitioned table. |
| `pg_partition_root` ( `regclass` ) → `regclass`<br>Returns the top-most parent of the partition tree to which the given relation belongs. Returns `NULL` if the relation does not exist or is not a partition or partitioned table. |

For example, to check the total size of the data contained in a partitioned table `measurement`, one could use the following query:

```
SELECT pg_size_pretty(sum(pg_relation_size(relid))) AS total_size
  FROM pg_partition_tree('measurement');
```

# 9.28.8. Index Maintenance Functions

Table 9.107 shows the functions available for index maintenance tasks. (Note that these maintenance tasks are normally done automatically by autovacuum; use of these functions is only required in special cases.) These functions cannot be executed during recovery. Use of these functions is restricted to superusers and the owner of the given index.

**Table 9.107. Index Maintenance Functions**

| Function<br>Description |
|---|
| `brin_summarize_new_values` ( *`index`* `regclass` ) → `integer`<br>Scans the specified BRIN 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 those table pages. Returns the number of new page range summaries that were inserted into the index. |
| `brin_summarize_range` ( *`index`* `regclass`, *`blockNumber`* `bigint` ) → `integer`<br>Summarizes the page range covering the given block, if not already summarized. This is like `brin_summarize_new_values` except that it only processes the page range that covers the given table block number. |
| `brin_desummarize_range` ( *`index`* `regclass`, *`blockNumber`* `bigint` ) → `void`<br>Removes the BRIN index tuple that summarizes the page range covering the given table block, if there is one. |
| `gin_clean_pending_list` ( *`index`* `regclass` ) → `bigint`<br>Cleans up the “pending” list of the specified GIN index by moving entries in it, in bulk, to the main GIN data structure. Returns the number of pages removed from the pending list. If the argument is a GIN index built with the `fastupdate` option disabled, no cleanup happens and the result is zero, because the index doesn't have a pending list. See Section 65.4.4.1 and Section 65.4.5 for details about the pending list and `fastupdate` option. |

# 9.28.9. Generic File Access Functions

The functions shown in Table 9.108 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, unless the user is a superuser or is granted the role `pg_read_server_files`. Use a relative path for files in the cluster directory, and a path matching the `log_directory` configuration setting for log files.

Note that granting users the EXECUTE privilege on `pg_read_file()`, or related functions, allows them the ability to read any file on the server that the database server process can read; these functions bypass all in-database privilege checks. This means that, for example, a user with such access is able to read the contents of the `pg_authid` table where authentication information is stored, as well as read any table data in the database. Therefore, granting access to these functions should be carefully considered.

When granting privilege on these functions, note that the table entries showing optional parameters are mostly implemented as several physical functions with different parameter lists. Privilege must be granted separately on each such function, if it is to be used. psql's `\df` command can be useful to check what the actual function signatures are.

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` or an empty result set, as appropriate. If `false`, an error is raised. (Failure conditions other than “file not found” are reported as errors in any case.) The default is `false`.

### Table 9.108. Generic File Access Functions

| Function<br>Description |
|---|
| `pg_ls_dir` ( *`dirname`* `text` [, *`missing_ok`* `boolean`, *`include_dot_dirs`* `boolean` ] ) → `setof text`<br>Returns the names of all files (and directories and other special files) in the specified directory. The *`include_dot_dirs`* parameter indicates whether “.” and “..” are to be included in the result set; the default is to exclude them. Including them can be useful when *`missing_ok`* is `true`, to distinguish an empty directory from a non-existent directory.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_logdir` () → `setof record` ( *`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's log directory. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and roles with privileges of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_waldir` () → `setof record` ( *`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's write-ahead log (WAL) directory. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and roles with privileges of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_logicalmapdir` () → `setof record` ( *`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's `pg_logical/mappings` directory. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |

| Function<br>Description |
|---|
| `pg_ls_logicalsnapdir` () → `setof record` ( *name* `text`, *size* `bigint`, *modification* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's `pg_logical/snapshots` directory. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_replslotdir` ( *slot_name* `text` ) → `setof record` ( *name* `text`, *size* `bigint`, *modification* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's `pg_replslot/slot_name` directory, where *slot_name* is the name of the replication slot provided as input of the function. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_summariesdir` () → `setof record` ( *name* `text`, *size* `bigint`, *modification* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's WAL summaries directory (`pg_wal/summaries`). Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_archive_statusdir` () → `setof record` ( *name* `text`, *size* `bigint`, *modification* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the server's WAL archive status directory (`pg_wal/archive_status`). Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_ls_tmpdir` ( [ *tablespace* `oid` ] ) → `setof record` ( *name* `text`, *size* `bigint`, *modification* `timestamp with time zone` )<br>Returns the name, size, and last modification time (mtime) of each ordinary file in the temporary file directory for the specified *tablespace*. If *tablespace* is not provided, the `pg_default` tablespace is examined. Filenames beginning with a dot, directories, and other special files are excluded.<br>This function is restricted to superusers and members of the `pg_monitor` role by default, but other users can be granted EXECUTE to run the function. |
| `pg_read_file` ( *filename* `text` [, *offset* `bigint`, *length* `bigint` ] [, *missing_ok* `boolean` ] ) → `text`<br>Returns all or part of a text file, starting at the given byte *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 database's encoding; an error is thrown if they are not valid in that encoding.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |
| `pg_read_binary_file` ( *filename* `text` [, *offset* `bigint`, *length* `bigint` ] [, *missing_ok* `boolean` ] ) → `bytea`<br>Returns all or part of a file. This function is identical to `pg_read_file` except that it can read arbitrary binary data, returning the result as `bytea` not `text`; accordingly, no encoding checks are performed. |

| Function<br>Description |
|---|
| This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.<br>In combination with the `convert_from` function, this function can be used to read a text file in a specified encoding and convert to the database's encoding:<br><br>`SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');` |
| `pg_stat_file` ( *`filename`* `text` [, *`missing_ok`* `boolean` ] ) → `record` ( *`size`* `bigint`, *`access`* `timestamp with time zone`, *`modification`* `timestamp with time zone`, *`change`* `timestamp with time zone`, *`creation`* `timestamp with time zone`, *`isdir`* `boolean` )<br>Returns a record containing the file's size, last access time stamp, last modification time stamp, last file status change time stamp (Unix platforms only), file creation time stamp (Windows only), and a flag indicating if it is a directory.<br>This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function. |

## 9.28.10. Advisory Lock Functions

The functions shown in Table 9.109 manage advisory locks. For details about proper use of these functions, see Section 13.3.5.

All these functions are intended to be used to lock application-defined resources, 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 conflicting lock on the same resource identifier, the functions will either wait until the resource becomes available, or return a `false` result, as appropriate for the function. Locks can be either shared or exclusive: a shared lock does not conflict with other shared locks on the same resource, only with exclusive locks. Locks can be taken at session level (so that they are held until released or the session ends) or at transaction level (so that they are held until the current transaction ends; there is no provision for manual release). Multiple session-level lock requests stack, so that if the same resource identifier is locked three times there must then be three unlock requests to release the resource in advance of session end.

### Table 9.109. Advisory Lock Functions

| Function<br>Description |
|---|
| `pg_advisory_lock` ( *`key`* `bigint` ) → `void`<br>`pg_advisory_lock` ( *`key1`* `integer`, *`key2`* `integer` ) → `void`<br>Obtains an exclusive session-level advisory lock, waiting if necessary. |
| `pg_advisory_lock_shared` ( *`key`* `bigint` ) → `void`<br>`pg_advisory_lock_shared` ( *`key1`* `integer`, *`key2`* `integer` ) → `void`<br>Obtains a shared session-level advisory lock, waiting if necessary. |
| `pg_advisory_unlock` ( *`key`* `bigint` ) → `boolean`<br>`pg_advisory_unlock` ( *`key1`* `integer`, *`key2`* `integer` ) → `boolean`<br>Releases a previously-acquired exclusive session-level advisory lock. Returns `true` if the lock is successfully released. If the lock was not held, `false` is returned, and in addition, an SQL warning will be reported by the server. |
| `pg_advisory_unlock_all` () → `void`<br>Releases all session-level advisory locks held by the current session. (This function is implicitly invoked at session end, even if the client disconnects ungracefully.) |

| Function<br>Description |
|---|
| `pg_advisory_unlock_shared` ( *key* `bigint` ) → `boolean`<br>`pg_advisory_unlock_shared` ( *key1* `integer`, *key2* `integer` ) → `boolean`<br>Releases a previously-acquired shared session-level advisory lock. Returns `true` if the lock is successfully released. If the lock was not held, `false` is returned, and in addition, an SQL warning will be reported by the server. |
| `pg_advisory_xact_lock` ( *key* `bigint` ) → `void`<br>`pg_advisory_xact_lock` ( *key1* `integer`, *key2* `integer` ) → `void`<br>Obtains an exclusive transaction-level advisory lock, waiting if necessary. |
| `pg_advisory_xact_lock_shared` ( *key* `bigint` ) → `void`<br>`pg_advisory_xact_lock_shared` ( *key1* `integer`, *key2* `integer` ) → `void`<br>Obtains a shared transaction-level advisory lock, waiting if necessary. |
| `pg_try_advisory_lock` ( *key* `bigint` ) → `boolean`<br>`pg_try_advisory_lock` ( *key1* `integer`, *key2* `integer` ) → `boolean`<br>Obtains an exclusive session-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately. |
| `pg_try_advisory_lock_shared` ( *key* `bigint` ) → `boolean`<br>`pg_try_advisory_lock_shared` ( *key1* `integer`, *key2* `integer` ) → `boolean`<br>Obtains a shared session-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately. |
| `pg_try_advisory_xact_lock` ( *key* `bigint` ) → `boolean`<br>`pg_try_advisory_xact_lock` ( *key1* `integer`, *key2* `integer` ) → `boolean`<br>Obtains an exclusive transaction-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately. |
| `pg_try_advisory_xact_lock_shared` ( *key* `bigint` ) → `boolean`<br>`pg_try_advisory_xact_lock_shared` ( *key1* `integer`, *key2* `integer` ) → `boolean`<br>Obtains a shared transaction-level advisory lock if available. This will either obtain the lock immediately and return `true`, or return `false` without waiting if the lock cannot be acquired immediately. |

# 9.29. Trigger Functions

While many uses of triggers involve user-written trigger functions, PostgreSQL provides a few built-in trigger functions that can be used directly in user-defined triggers. These are summarized in Table 9.110. (Additional built-in trigger functions exist, which implement foreign key constraints and deferred index constraints. Those are not documented here since users need not use them directly.)

For more information about creating triggers, see CREATE TRIGGER.

### Table 9.110. Built-In Trigger Functions

| Function<br>Description<br>Example Usage |
|---|
| `suppress_redundant_updates_trigger` ( ) → `trigger`<br>Suppresses do-nothing update operations. See below for details.<br>`CREATE TRIGGER ... suppress_redundant_updates_trigger()` |
| `tsvector_update_trigger` ( ) → `trigger` |

| Function<br>Description<br>Example Usage |
|---|
| Automatically updates a `tsvector` column from associated plain-text document column(s). The text search configuration to use is specified by name as a trigger argument. See Section 12.4.3 for details.<br>`CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)` |
| `tsvector_update_trigger_column` ( ) → `trigger`<br>Automatically updates a `tsvector` column from associated plain-text document column(s). The text search configuration to use is taken from a `regconfig` column of the table. See Section 12.4.3 for details.<br>`CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)` |

The `suppress_redundant_updates_trigger` function, when applied as a row-level `BEFORE UPDATE` trigger, will prevent any update that does not actually change the data in the row from taking place. This overrides the normal behavior which always performs a physical row 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 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 updates do actually change, use of this trigger will make updates run slower on average.

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 FUNCTION suppress_redundant_updates_trigger();
```

In most cases, you need to fire this trigger last for each row, so that it does not override other triggers that might wish to alter the row. Bearing in mind that triggers fire in name order, you would therefore choose a trigger name that comes after the name of any other trigger you might have on the table. (Hence the “z” prefix in the example.)

# 9.30. Event Trigger Functions

PostgreSQL provides these helper functions to retrieve information from event triggers.

For more information about event triggers, see Chapter 38.

## 9.30.1. Capturing Changes at Command End

`pg_event_trigger_ddl_commands () → setof record`

`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 | boolean | 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. |

## 9.30.2. Processing Objects Dropped by a DDL Command

`pg_event_trigger_dropped_objects () → setof record`

`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, an error is raised. This function 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 | boolean | True if this was one of the root object(s) of the deletion |
| normal | boolean | True if there was a normal dependency relationship in the dependency graph leading to this object |
| is_temporary | boolean | 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. |

| Name | Type | Description |
|---|---|---|
| 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 FUNCTION test_event_trigger_for_drops();
```

## 9.30.3. Handling a Table Rewrite Event

The functions shown in Table 9.111 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 9.111. Table Rewrite Information Functions

| Function<br>Description |
|---|
| pg_event_trigger_table_rewrite_oid () → oid |

| Function<br>Description |
|---|
| Returns the OID of the table about to be rewritten. |
| `pg_event_trigger_table_rewrite_reason` () → `integer`<br>Returns a code explaining the reason(s) for rewriting. The value is a bitmap built from the following values: `1` (the table has changed its persistence), `2` (default value of a column has changed), `4` (a column has a new data type) and `8` (the table access method has changed). |

These functions 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 FUNCTION test_event_trigger_table_rewrite_oid();
```

# 9.31. Statistics Information Functions

PostgreSQL provides a function to inspect complex statistics defined using the `CREATE STATISTICS` command.

## 9.31.1. Inspecting MCV Lists

`pg_mcv_list_items ( pg_mcv_list )` → `setof record`

`pg_mcv_list_items` returns a set of records describing all items stored in a multi-column MCV list. It returns the following columns:

| Name | Type | Description |
|---|---|---|
| `index` | `integer` | index of the item in the MCV list |
| `values` | `text[]` | values stored in the MCV item |
| `nulls` | `boolean[]` | flags identifying `NULL` values |
| `frequency` | `double precision` | frequency of this MCV item |
| `base_frequency` | `double precision` | base frequency of this MCV item |

The `pg_mcv_list_items` function can be used like this:

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid =
 stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts';
```

Values of the `pg_mcv_list` type can be obtained only from the `pg_statistic_ext_data.stxdmcv` column.