Basic Data Types and Functions

Boolean Type

BOOLEAN: three-valued logic; possible values are true, false, and null, which means “unknown” (1 byte)

Logical Operations

and, **or**: logical “AND” (conjunction) and “OR” (disjunction)

```
                and or

f   f       f   f
f   t       f   t
t   f       f   t
t   t       t   t
N   f       f   N
f   N       f   N
N   t       N   t
t   N       N   t
N   N       N   N
```

not: logical “NOT” (negation)

```
      not

f     t
t     f

N     N
```

bool_and, **bool_or**: aggregate versions of logical “AND” and “OR”

```
select bool_and(b) from (values (true), (false)) t(b) → f
select bool_and(b) from (values (true), (null))  t(b) → t  — null is not taken into account
select bool_or(b)  from (values (true), (false)) t(b) → t
select bool_or(b)  from (values (false), (null)) t(b) → f  — null is not taken into account
```

Comparison Operators

Return a boolean type; these operators are defined for all types, for which it makes sense.

<, >, <=, >=, =, <> (**!=**): less than, greater than, less than or equal to, greater than or equal to, not equal

```
'Hello' < 'world' → f  — string comparison depends on collation
1 = 1     → t
1 = null → null  — most operators return null for null arguments
```

is [not] distinct from: (not) equal; null values are considered to be the same

```
1    is distinct from null → t
1    is distinct from 1    → f
null is distinct from null → f
```

is [not] null: check whether an expression is null

```
1    is null → f
null is null → t
```

is [not] true, is [not] false: check whether an expression is true or false

	is true	= true	is not false	is false	= false	is not true
f	f	f	f	**t**	**t**	**t**
t	**t**	**t**	**t**	f	f	f
N	f	N	**t**	f	N	**t**

[not] between: is located between two endpoints

```
X     between A and B  =  A <= X and X <= B
X not between A and B  =  X < A  or  X > B
```

```
2 between 1 and 3 → t
2 between 3 and 1 → f
1 between 2 and 3 → f
```

[not] between symmetric: is located between two endpoints (after sorting)

```
2 between symmetric 3 and 1 → t
```

Handling Null Values

coalesce: replace null with a value

```
coalesce(null, 'no')  → no
coalesce('yes', 'no') → yes
coalesce(null, null, 'no', f()) → no  — takes any number of arguments; f() will not be computed
```

nullif: substitute null for a value

```
nullif('no', 'no')  → N
nullif('yes', 'no') → yes
```

Numbers

Integer Types

SMALLINT: from −32768 to +32767 (2 bytes)

INTEGER, INT: from −2147483648 to +2147483647 (4 bytes)

BIGINT: from -10^{19} to $+10^{19}$, approximately (8 bytes)

SMALLSERIAL
SERIAL
BIGSERIAL
} autoincrementing, but better use **generated always as**

Floating-Point Types

REAL: about 6 decimal digits (4 bytes)

FLOAT, DOUBLE PRECISION: about 15 decimal digits (8 bytes)

```
-Infinity   negative infinity
+Infinity   positive infinity
NaN         not a number
```

```
0.1::real * 10 = 1.0::real → f — values are stored imprecisely, be careful with comparison operations!
```

Arbitrary Precision Numbers

NUMERIC, **DECIMAL**: up to 131072 digits before and 16383 digits after the decimal point (variable length)

NUMERIC(N): integers of up to N digits (N ≤ 1000)

NUMERIC(N, M): real numbers of up to N digits, M of which appear after the decimal point

```
NaN         not a number
```

Arithmetic Operators

+, −, *, /: addition, subtraction, multiplication, division

div: integer division

```
div(7.0,2.0) → 3 → trunc(7.0/2.0)
```

mod, **%**: modulo (remainder of division)

```
mod(7,2) → 7%2 → 1
```

power, ^: exponentiation

```
power(2,3) → 2^3 → 8
```

abs: absolute value

```
abs(-2.7) → 2.7
```

sign: sign of the argument

```
        -2.7  0   2.7
sign    -1    0   1
```

trunc, **ceil (ceiling)**, **round**, **floor**: rounding off

```
        -2.7  2.7
trunc   -2    2
ceil    -2    3
round   -3    3
floor   -3    2
```

Type Casting

to_number: convert a string to a number (see Template Patterns for Numeric Formatting)

```
9     digit                 0     digit with leading zeros
.     decimal point         D     dot or comma (depends on the locale)
,     group separator       G     group separator (depends on the locale)
MI    minus sign (<0)       PL    plus sign (>0)              SG    plus or minus sign

to_number('3,1416',    '99D00') → 3.14
to_number('3,1416','99D000000') → 3.1416
to_number('123,45',    '99D00') → numeric field overflow
```

Character Types

CHAR(N): fixed-length blank-padded string

VARCHAR(N): a variable-length string with limit

TEXT, VARCHAR: a string of unlimited length

```
'What''s up?'
$$What's up?$$
$function$BEGIN; RETURN $$What's up?$$; END;$function$
E'col1\tcol2\nval1\tval2'
```

Extracting a Substring

length, **char_length**: return the number of characters in a string

```
length('Hello, world!') → char_length('Hello, world!') → 13
```

octet_length: return the number of bytes in a string

```
octet_length('Hello, world!') → 13  — depends on the encoding; this value can
                                      differ from char_length for national alphabet characters
```

position, **strpos**: return the position of a substring

```
position('world' in 'Hello, world!') → strpos('Hello, world!', 'world') → 8
```

substring: extract a substring

```
substring('Hello, world!' from 8 for 5) → substr('Hello, world!', 8, 5) → world
substring('Hello, world!' from 8)       → substr('Hello, world!', 8)    → world!
```

left, **right**: extract a substring from the left or right

```
left( 'Hello, world!', 5) → Hello
right('Hello, world!', 6) → world!
```

Modification

overlay: replace a substring

```
overlay('Hello, world!' placing 'PostgreSQL' from 8 for 5) → Hello, PostgreSQL!
```

replace: replace all occurrences of a substring

```
replace('Hello, world!', 'o', 'ooo') → Hellooo, wooorld!
```

translate: replace characters in a string with the provided substitute

```
translate('Hello, world!', 'Hlwrdeo', 'hlwrd') → hll, wrld!
```

lower, **upper**, **initcap**: perform case conversion (uses CTYPE)

```
lower('Hello, world!')   → hello, world!
upper('Hello, world!')   → HELLO, WORLD!
initcap('Hello, world!') → Hello, World!
```

trim, **ltrim**, **rtrim**, **btrim**: remove characters from the ends of a string (a space by default)

```
trim( leading 'He!' from 'Hello, world!') → ltrim('Hello, world!', 'He!') → llo, world!
trim(trailing 'He!' from 'Hello, world!') → rtrim('Hello, world!', 'He!') → Hello, world
trim(    both 'He!' from 'Hello, world!') → btrim('Hello, world!', 'He!') → llo, world
```

lpad, **rpad**: add characters to a string from left or right (a space by default)

```
lpad('Hello, world!', 18, '. ') → . . .Hello, world!
rpad('Hello, world!', 18, '. ') → Hello, world!. . .
```

reverse: reverse the order of characters

```
reverse('Hello, world!') → !dlrow ,olleH
```

Construction

concat, **concat_ws**: concatenate strings (can take an arbitrary number of arguments)

```
concat('Hello,', ' ', 'world!') → 'Hello,' || ' ' || 'world!' → Hello, world!
concat_ws(', ', 'Hello', 'oh', 'world!') → Hello, oh, world!
```

string_agg: aggregate strings (see Aggregate Functions)

```
string_agg(s, ', ' order by id) from (values (2,'world!'), (1,'Hello')) v(id,s)
→ Hello, world!
```

repeat: repeat a string

```
repeat('Hello', 2) → HelloHello
```

chr: return the character with the given code (depends on the encoding)

```
chr(34) → "
```

Quoting and Escaping in Dynamic SQL

quote_ident: enclose a string into quotes to use it as an identifier

```
quote_ident('id') → id
quote_ident('foo bar') → "foo bar"
```

quote_literal, **quote_nullable**: enclose a string into quotes to use it as a string literal

```
quote_literal('id')           → quote_nullable('id')           → 'id'
quote_literal($$What's up?$$) → quote_nullable($$What's up?$$) → 'What''s up?'
quote_literal(null)  → N
quote_nullable(null) → NULL
```

format: return sprintf-like formatted text

```
format('Hello, %s!', 'world!') → Hello, world!
format('UPDATE %I SET s = %L', 'tbl', $$What's up?$$)
→ 'UPDATE '||quote_ident('tbl')||' SET s = '||quote_nullable($$What's up?$$)
→ UPDATE tbl SET s = 'What''s up?'
```

Type Casting

to_char: convert a number to a string (see Template Patterns for Numeric Formatting)

```
9     digit                 0     digit with leading zeros
.     decimal point         D     dot or comma (depends on the locale)
,     group separator       G     group separator (depends on the locale)
RN    Roman numerals
EEEE  exponent notation
MI    minus sign (<0)       PL    plus sign (>0)              SG    plus or minus sign
FM    without leading zeros and spaces

to_char(3.1416, 'FM99D00')        → 3,14
to_char(3.1416, 'FM99D000000')    → 3,141600
to_char(56789,     '999G999G999') →       56 789
to_char(123456789, '999G999G999') →  123 456 789
to_char(123456789, '999G999G999') → -123 456 789
```

to_char: convert a date to a string (see Template Patterns for Date/Time Formatting)

```
YYYY  year
MM    month (01-12)             MON   month (abbrv.)            MONTH month (a full name)
DD    day (01-31)
D     day of the week (1-7)     DY    day of the week (abbrv.)  DAY   day of the week (a full name)
HH    hour (01-12)              HH24  hour (00-23)
MI    minutes
SS    seconds
TZ    time zone                 OF    time-zone offset
FM    suppress padding blanks   TM    use localized day and month names

to_char(now(), 'DD.MM.YYYY HH24:MI:SSOF') → 05.10.2016 10:51:08+03
to_char(now(), 'FMDD TMMonth YYYY, Day')  → 5 October 2016, Wednesday
```

Pattern Matching

like (~~), **not like** (!~~): pattern-matching operators

```
_     any single symbol
%     ≥0 symbols

'Hello, world!' like 'Hello_ %'            → t
'Hello_world!'  like 'Hello\_%' escape '\' → t
```

ilike (~~*), **not ilike** (!~~*): case-insensitive pattern matching

```
'Hello, world!' ilike 'hello, world!'      → t
```

SQL Regular Expressions

similar to: a pattern-matching operator

```
_     any single character
%     ≥0 characters
*     repeat the prev.item ≥0 times  {m}   repeat m times             |     alternation
+     repeat ≥1 times                {m,}  repeat ≥m times            (…)   grouping
?     repeat 0 or 1 times            {m,n} repeat from m to n times   […]   a character class

'Hello, world!' similar to 'Hello_ %'                  → t
'-3.14' similar to '(#+|-)?[0-9]+(.[0-9]*)?' escape '#' → t
'24.sep.2016' similar to '_{1,2}._{3}._{4}'            → t
```

substring: extract a substring

```
substring('Hello, world!' from 'Hello, \"%\"!' for '\') → world
```

split_part: extract a substring on the specified delimiter

```
split_part('Hello, world!', ', ', 1) → Hello
split_part('Hello, world!', ', ', 2) → world!
```

POSIX Regular Expressions

~, **!~**: match operators

```
.     any single character
*     repeat the prev.item ≥0 times  {m}   repeat m times             |     alternation
+     repeat ≥1 times                {m,}  repeat ≥m times            (…)   grouping
?     repeat 0 or 1 times            {m,n} repeat from m to n times   […]   a character class
^     start of line                  $     end of line

'Hello, world!' ~ 'Hello'       → t
'Hello, world!' ~ '^Hello$'     → f
'Hello, world!' ~ '^Hello. .*$' → t
'-3.14' ~ '(\+|-)?[0-9]+(\.[0-9]*)?' → t
'24.sep.2016' ~ '.{1,2}\..{3}\..{4}' → t

\m    start of the word          \M    end of the word
\d    digit                      \D    not a digit
\s    space                      \S    not a space
\w    letter or digit            \W    neither a letter nor a digit
\n    end of line                \t    tabulation

'Hello, world!' ~ '\mworld\M' → t
'Helloworld!'   ~ '\mworld\M' → f
'-3.14' ~ '[+-]?\d+(\.\d*)?' → t
'24.sep.2016' ~ '\d{1,2}\.\w{3}\.\d{4}' → t
```

~*, **!~***: case-insensitive matching operators

```
'Hello, world!' ~* 'hello'     → t
'Hello, world!' ~  '(?i)hello' → t  — an alternative way is using (?i) before the pattern
```

substring: extract a substring

```
?     a “non-greedy” quantifier (for *, +, ?, {})

substring('Hello, world!' from '.*e')  → Hello, world
substring('Hello, world!' from '.*?e') → He

(?=…) positive lookahead
(?!…) negative lookahead

substring('Hello, world!' from '\w+,')     → Hello,
substring('Hello, world!' from '\w+(?=,)') → Hello
substring('Hello, world!' from '\w+(?!,)') → world
```

regexp_matches: extract substrings

```
regexp_matches('Hello, world!', '\w+')           → {Hello}
regexp_matches('Hello, world!', '(\w+).*?(\w+)') → {Hello,world}
regexp_matches('Hello, world!', '\w+', 'g')      → {Hello}
                                                   {world}     — 2 rows
regexp_matches('Hello, world!', '\d')            — 0 rows
```

regexp_replace: replace a substring

```
regexp_replace('Hello, world!', '\w+', 'Hi')               → Hi, world!
regexp_replace('Hello, world!', '(\w+).*?(\w+)', '\2, \1') → world, Hello!
regexp_replace('Hello, world!', '\w+', '_', 'g')           → _, _!
```

regexp_split_to_array: convert a string into an array

```
regexp_split_to_array('Hello,  world!', ',\s+') → {Hello,world!}
```

regexp_split_to_table: convert a string into a table

```
regexp_split_to_table('Hello,  world!', ',\s+') → Hello
                                                  world!   — 2 rows
```

Date/Time Types

DATE: date, without time of day (4 bytes)

```
date '2016-12-31'

make_date(2016,12,31)
```

TIME: time of day, without date (8 bytes)

```
time '23:59:59.999'

make_time(23,59,59.999)
```

TIME WITH TIME ZONE: time of day, with time zone (12 bytes)

TIMESTAMP: date and time (8 bytes)

```
timestamp '2016-12-31 23:59:59.999'

make_timestamp(2016,12,31,23,59,59.999)
```

TIMESTAMP WITH TIME ZONE, TIMESTAMPTZ: date and time, with time zone (8 bytes)

```
timestamptz '2016-12-31 23:59:59.999 MSK'

make_timestamptz(2016,12,31,23,59,59.999,'MSK')
```

INTERVAL: time interval (16 bytes)

```
interval '1 year 4 months 12 days 03:17:23 ago'

make_interval(-1,-4,0/*weeks*/,-12,-3,-17,-23)
```

Arithmetic Operators

+, −: modify date/time by adding (subtracting) an interval

```
date '2016-12-31' + 1 → 2017-01-01  — an interval for DATE is an integer number of days
timestamp '2016-12-31 23:50:01' + interval '9 minutes 59 seconds' → 2017-01-01 00:00:00
```

−: an interval between two dates/times

```
date '2016-12-31' - date '2016-12-01' → 30  — an interval for DATE is an integer number of days
timestamp '2016-12-31 23:59:59' - timestamp '2016-12-01 23:50:00' → 30 days 00:09:59
```

*, /: multiply (divide) an interval by the specified number

```
5 * interval '1 day' → 5 days
interval '5 day' / 2 → 2 days 12:00:00
```

Functions

overlaps: check whether intervals overlap

```
(time '12:00', time '14:00')       overlaps (time '13:00', time '15:00')       → t
(time '12:00', interval '2 hours') overlaps (time '13:00', interval '2 hours') → t
```

date_trunc: truncate date/time or interval

```
year         hour
month        minute
day          second

date_trunc('month', timestamp '2016-12-31 23:59:59')   → 2016-12-01 00:00:00
date_trunc('minutes', interval '9 minutes 59 seconds') → 00:09:00
```

Current Time

current_date, **localtime**, **localtimestamp**: transaction start time, without time zone

current_time, **current_timestamp = transaction_timestamp() = now()**: transaction start time, with time zone

statement_timestamp(): start time of the current statement, with time zone

clock_timestamp(): actual current time, with time zone

Extracting Date/Time Fields

extract, **date_part**: extract particular fields of date/time (returns a double-precision value)

```
year         hour         isodow
month        minute       timezone
day          second       timezone_hour

extract(year from   timestamp '2016-12-31 23:59:59')
→ date_part('year', timestamp '2016-12-31 23:59:59') → 2016
extract(isodow from timestamp '2016-12-31 23:59:59') → 6    — 1..7 (mon..sun)
```

Type Casting

to_date: convert a string to a date (see Template Patterns for Date/Time Formatting)

```
YYYY  year
MM    month (01-12)     MON   month (abbrv.)           MONTH month (a full name)
DD    day (01-31)       DY    day of week (abbrv.)     DAY   day of week

to_date('31.12.2016', 'DD.MM.YYYY') → 2016-12-31
```

to_timestamp: convert a string to a time stamp with time zone (see Template Patterns for Date/Time Formatting)

```
YYYY  year
MM    month (01-12)     MON   month (abbrv.)            MONTH month (a full name)
DD    day (01-31)       DY    day of week (abbrv.)      DAY   day of week
HH    hour (01-12)      HH24  hour (00-23)
MI    minutes
SS    seconds
```

```
to_timestamp('31.12.2016 23:59:59', 'DD.MM.YYYY HH24:MI:SS') → 2016-12-31 23:59:59+03
```

Composite Types

CREATE TYPE AS ( ): create a composite time as a database object

```
CREATE TYPE monetary AS (amount numeric, currency text);
CREATE TABLE uom(units text, value numeric); — implicitly define the corresponding composite type

monetary '(25.10,"USD")'
ROW(25.10,'USD')::monetary      (25.10,'USD')::monetary

((25.10,'USD')::monetary).amount   → 25.10
((25.10,'USD')::monetary).currency → USD
```

%ROWTYPE: a composite type that corresponds to a table row (PL/pgSQL)

```
DECLARE
  m monetary%rowtype;
BEGIN
  m := (25.10,'USD')::monetary;
END;
```

RECORD: an anonymous composite type without any predefined structure (PL/pgSQL)

```
DECLARE
  r record;
BEGIN
  r := (25.10,'USD')::monetary;
  SELECT * INTO r FROM uom LIMIT 1;
END;
```

Arrays

TYPE[]: an array of elements of the specified type

```
text[]
integer[][] — a two-dimensional array

'{"Hello","world!"}'            ARRAY['Hello','world!']
'{{1,2,3},{10,20,30}}'          ARRAY[[1,2,3], [10,20,30]]

(ARRAY['Hello','world!'])[1] → Hello   — array index starts with 1 by default
(ARRAY['Hello','world!'])[2] → world!
(ARRAY['Hello','world!'])[3] → N       — not an error

(ARRAY[[ 1, 2, 3],
       [10,20,30]])[2][1] → 10

'[-1:1]={10,20,30}'

('[-1:1]={5,6,7}'::int[])[-1]   → 5      — index can be arbitrary
('[-1:1]={5,6,7}'::int[])[-1:0] → {5,6}  — an array slice

(ARRAY[[ 1, 2, 3],
       [10,20,30]])[1:2][2:3] → {{2,3},{20,30}}
```

Dimensions

array_ndims: number of dimensions

```
\set A '[1:2][-1:1]={{10,20,30},{40,50,60}}'
```

```
    -1   0   1
1   10  20  30

2   40  50  60
```

```
array_ndims(:'A'::int[][]) → 2
```

array_length, **cardinality**: number of elements

```
array_length(:'A'::int[][], 1) → 2
array_length(:'A'::int[][], 2) → 3
array_length(:'A'::int[][])    → 6 = 2*3
```

array_lower, **array_upper**: index of the first (last) element

```
array_lower(:'A'::int[][], 2) → -1
array_upper(:'A'::int[][], 2) → 1
```

Search and Containment

@>, **<@**: check whether one of the arrays contains the other (supported by GIN index)

```
ARRAY[1,2,3,4] @> ARRAY[2,3] → t
ARRAY[2,3] <@ ARRAY[1,2,3,4] → t
```

&&: check whether arrays have at least one element in common (supported by GIN index)

```
ARRAY[1,3,5] && ARRAY[3,6,9] → t
```

array_position, **array_positions**: get element position (positions)

```
array_position (ARRAY[7,8,9,8,7], 8)    → 2
array_position (ARRAY[7,8,9,8,7], 8, 3) → 4
array_positions(ARRAY[7,8,9,8,7], 8)    → {2,4}
array_positions(ARRAY[null,null], null) → {1,2} — provides is not distinct from semantics
```

Modification

array_remove: remove an element

```
array_remove(ARRAY[7,8,9,8,7], 8)       → {7,9,7}
```

array_replace: replace an element

```
array_replace(ARRAY[7,8,9,8,7], 8, 0)   → {7,0,9,0,7}
```

Construction

array_fill: initialize an array with the specified value

```
array_fill(0, ARRAY[2,3])              → {{0,0,0},{0,0,0}}
array_fill(0, ARRAY[2,3], ARRAY[1,-1]) → [1:2][-1:1]={{0,0,0},{0,0,0}}
```

||, **array_append**, **array_prepend**, **array_cat**: perform concatenation

```
ARRAY[1,2,3] || 4        → array_append(ARRAY[1,2,3], 4)     → {1,2,3,4}
1 || ARRAY[2,3,4]        → array_prepend(1, ARRAY[2,3,4])    → {1,2,3,4}
ARRAY[1,2] || ARRAY[3,4] → array_cat(ARRAY[1,2], ARRAY[3,4]) → {1,2,3,4}
```

Type Conversion

array_to_string: concatenate array elements

```
array_to_string(ARRAY[1,2,3], ', ')            → 1, 2, 3
array_to_string(ARRAY[1,2,null,4], ', ', 'N/A) → 1, 2, N/A, 4
```

string_to_array: split a string into an array on delimiter

```
string_to_array('one two three', ' ')      → {one,two,three}
string_to_array('Hello', null)             → {H,e,l,l,}
string_to_array('1;2;N/A;4', ';', 'N/A')   → {1,2,NULL,4}
```

array_agg: convert a table into an array

```
select array_agg(a) from (values (1),(2)) t(a)                   → {1,2}
select array_agg(a) from (values (ARRAY[1,2]),(ARRAY[3,4])) t(a) → {{1,2},{3,4}}
```

unnest: expand an array into a table

```
unnest(ARRAY[1,2]) → 1
                     2 — 2 rows
```