Functions and Operators

Overview

- Logical operators
- Comparison functions and operators
- Mathematical functions and operators
- String functions and operators

Logical Operators

- AND
- OR
- NOT

Logical Operators

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

Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Comparison Predicates

Predicate	Description
a `BETWEEN` *x* `AND` *y*	between
a `NOT BETWEEN` *x* `AND` *y*	not between
a `BETWEEN SYMMETRIC` *x* `AND` *y*	between, after sorting the comparison values
a `NOT BETWEEN SYMMETRIC` *x* `AND` *y*	not between, after sorting the comparison values
a `IS DISTINCT FROM` *b*	not equal, treating null like an ordinary value
a `IS NOT DISTINCT FROM` *b*	equal, treating null like an ordinary value
expression `IS NULL`	is null
expression `IS NOT NULL`	is not null
expression `ISNULL`	is null (nonstandard syntax)
expression `NOTNULL`	is not null (nonstandard syntax)
boolean_expression `IS TRUE`	is true
boolean_expression `IS NOT TRUE`	is false or unknown
boolean_expression `IS FALSE`	is false
boolean_expression `IS NOT FALSE`	is true or unknown
boolean_expression `IS UNKNOWN`	is unknown
boolean_expression `IS NOT UNKNOWN`	is true or false

Comparison Predicates

```
a BETWEEN x AND y
```

```
a >= x AND a <= y
```

Comparison Predicates

```
a NOT BETWEEN x AND y
```

```
a < x OR a > y
```

Comparison Predicates

Function	Description	Example	Example Result
`num_nonnulls(VARIADIC "any")`	returns the number of non-null arguments	`num_nonnulls(1, NULL, 2)`	2
`num_nulls(VARIADIC "any")`	returns the number of null arguments	`num_nulls(1, NULL, 2)`	1

Mathematical Operators

Operator	Description	Example	Result
+	addition	`2 + 3`	5
–	subtraction	`2 - 3`	-1
*	multiplication	`2 * 3`	6
/	division (integer division truncates the result)	`4 / 2`	2
%	modulo (remainder)	`5 % 4`	1
^	exponentiation (associates left to right)	`2.0 ^ 3.0`	8
|/	square root	`|/ 25.0`	5
||/	cube root	`||/ 27.0`	3
!	factorial	`5 !`	120
!!	factorial (prefix operator)	`!! 5`	120
@	absolute value	`@ -5.0`	5
&	bitwise AND	`91 & 15`	11
|	bitwise OR	`32 | 3`	35
#	bitwise XOR	`17 # 5`	20
~	bitwise NOT	`~1`	-2
<<	bitwise shift left	`1 << 4`	16
>>	bitwise shift right	`8 >> 2`	2

Mathematical Functions

Function	Return Type	Description	Example	Result
`abs(x)`	(same as input)	absolute value	`abs(-17.4)`	`17.4`
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	`3`
`ceil(dp or numeric)`	(same as input)	nearest integer greater than or equal to argument	`ceil(-42.8)`	`-42`
`ceiling(dp or numeric)`	(same as input)	nearest integer greater than or equal to argument (same as `ceil`)	`ceiling(-95.3)`	`-95`
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	`28.6478897565412`
`div(y numeric, x numeric)`	`numeric`	integer quotient of *y*/*x*	`div(9,4)`	`2`
`exp(dp or numeric)`	(same as input)	exponential	`exp(1.0)`	`2.71828182845905`
`floor(dp or numeric)`	(same as input)	nearest integer less than or equal to argument	`floor(-42.8)`	`-43`
`ln(dp or numeric)`	(same as input)	natural logarithm	`ln(2.0)`	`0.693147180559945`
`log(dp or numeric)`	(same as input)	base 10 logarithm	`log(100.0)`	`2`
`log(b numeric, x numeric)`	`numeric`	logarithm to base *b*	`log(2.0, 64.0)`	`6.0000000000`
`mod(y, x)`	(same as argument types)	remainder of *y*/*x*	`mod(9,4)`	`1`

Mathematical Functions

`pi()`	`dp`	"π" constant	`pi()`	`3.14159265358979`
`power(`*a* `dp, `*b* `dp)`	`dp`	*a* raised to the power of *b*	`power(9.0, 3.0)`	`729`
`power(`*a* `numeric, `*b* `numeric)`	`numeric`	*a* raised to the power of *b*	`power(9.0, 3.0)`	`729`
`radians(dp)`	`dp`	degrees to radians	`radians(45.0)`	`0.785398163397448`
`round(dp or numeric)`	(same as input)	round to nearest integer	`round(42.4)`	`42`
`round(`*v* `numeric, `*s* `int)`	`numeric`	round to *s* decimal places	`round(42.4382, 2)`	`42.44`
`scale(numeric)`	`integer`	scale of the argument (the number of decimal digits in the fractional part)	`scale(8.41)`	`2`
`sign(dp or numeric)`	(same as input)	sign of the argument (-1, 0, +1)	`sign(-8.4)`	`-1`
`sqrt(dp or numeric)`	(same as input)	square root	`sqrt(2.0)`	`1.4142135623731`
`trunc(dp or numeric)`	(same as input)	truncate toward zero	`trunc(42.8)`	`42`
`trunc(`*v* `numeric, `*s* `int)`	`numeric`	truncate to *s* decimal places	`trunc(42.4382, 2)`	`42.43`

Mathematical Functions

`width_bucket(`*`operand`* `dp, `*`b1`* `dp, `*`b2`* `dp, `*`count`* `int)`	`int`	return the bucket number to which *`operand`* would be assigned in a histogram having *`count`* equal-width buckets spanning the range *`b1`* to *`b2`*; returns `0` or *`count`*`+1` for an input outside the range	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`
`width_bucket(`*`operand`* `numeric, `*`b1`* `numeric, `*`b2`* `numeric, `*`count`* `int)`	`int`	return the bucket number to which *`operand`* would be assigned in a histogram having *`count`* equal-width buckets spanning the range *`b1`* to *`b2`*; returns `0` or *`count`*`+1` for an input outside the range	`width_bucket(5.35, 0.024, 10.06, 5)`	`3`
`width_bucket(`*`operand`* `anyelement, `*`thresholds`* `anyarray)`	`int`	return the bucket number to which *`operand`* would be assigned given an array listing the lower bounds of the buckets; returns `0` for an input less than the first lower bound; the *`thresholds`* array ***must be sorted***, smallest first, or unexpected results will be obtained	`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])`	`2`

Random Functions

Function	Return Type	Description
`random()`	`dp`	random value in the range 0.0 <= x < 1.0
`setseed(dp)`	`void`	set seed for subsequent `random()` calls (value between -1.0 and 1.0, inclusive)

Trigonometric Functions

Function (radians)	Function (degrees)	Description
`acos(`*x*`)`	`acosd(`*x*`)`	inverse cosine
`asin(`*x*`)`	`asind(`*x*`)`	inverse sine
`atan(`*x*`)`	`atand(`*x*`)`	inverse tangent
`atan2(`*y*`, `*x*`)`	`atan2d(`*y*`, `*x*`)`	inverse tangent of *y*/*x*
`cos(`*x*`)`	`cosd(`*x*`)`	cosine
`cot(`*x*`)`	`cotd(`*x*`)`	cotangent
`sin(`*x*`)`	`sind(`*x*`)`	sine
`tan(`*x*`)`	`tand(`*x*`)`	tangent

String Functions and Operators

Function	Return Type	Description	Example	Result
string `||` *string*	`text`	String concatenation	`'Post' || 'greSQL'`	`PostgreSQL`
string `||` *non-string* or *non-string* `||` *string*	`text`	String concatenation with one non-string input	`'Value: ' || 42`	`Value: 42`
`bit_length(`*string*`)`	`int`	Number of bits in string	`bit_length('jose')`	`32`
`char_length(`*string*`)` or `character_length(`*string*`)`	`int`	Number of characters in string	`char_length('jose')`	`4`
`lower(`*string*`)`	`text`	Convert string to lower case	`lower('TOM')`	`tom`
`octet_length(`*string*`)`	`int`	Number of bytes in string	`octet_length('jose')`	`4`
`overlay(`*string* `placing` *string* `from int [for int])`	`text`	Replace substring	`overlay('Txxxxas' placing 'hom' from 2 for 4)`	`Thomas`
`position(`*substring* `in` *string*`)`	`int`	Location of specified substring	`position('om' in 'Thomas')`	`3`
`substring(`*string* `[from int] [for int])`	`text`	Extract substring	`substring('Thomas' from 2 for 3)`	`hom`
`substring(`*string* `from` *pattern*`)`	`text`	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '...$')`	`mas`
`substring(`*string* `from` *pattern* `for` *escape*`)`	`text`	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '%#"o_a#"_' for '#')`	`oma`
`trim([leading | trailing | both] [`*characters*`] from` *string*`)`	`text`	Remove the longest string containing only characters from *characters* (a space by default) from the start, end, or both ends (`both` is the default) of *string*	`trim(both 'xyz' from 'yxTomxx')`	`Tom`
`trim([leading | trailing | both] [from]` *string* `[,` *characters*`] )`	`text`	Non-standard syntax for `trim()`	`trim(both from 'yxTomxx', 'xyz')`	`Tom`
`upper(`*string*`)`	`text`	Convert string to upper case	`upper('tom')`	`TOM`

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.

format()

```
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', E'C:\\Program Files');
Result: INSERT INTO locations VALUES(E'C:\\Program Files')
```

Bit String Operators

Operator	Description	Example	Result
||	concatenation	`B'10001' || B'011'`	10001011
&	bitwise AND	`B'10001' & B'01101'`	00001
|	bitwise OR	`B'10001' | B'01101'`	11101
#	bitwise XOR	`B'10001' # B'01101'`	11100
~	bitwise NOT	`~ B'10001'`	01110
<<	bitwise shift left	`B'10001' << 3`	01000
>>	bitwise shift right	`B'10001' >> 2`	00100

LIKE

- 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*).)

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

LIKE

- 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.

```
'abc' LIKE 'abc'     true
'abc' LIKE 'a%'      true
'abc' LIKE '_b_'     true
'abc' LIKE 'c'       false
```

Data Type Formatting Functions

Function	Return Type	Description	Example
to_char(timestamp, text)	text	convert time stamp to string	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	convert interval to string	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	convert integer to string	to_char(125, '999')
to_char(double precision, text)	text	convert real/double precision to string	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	convert numeric to string	to_char(-125.8, '999D99S')
to_date(text, text)	date	convert string to date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convert string to numeric	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	convert string to time stamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

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

```
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
```

CASE

- 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
```

CASE

```
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
```

COALESCE

- The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

```
COALESCE(value [, ...])
```

```
SELECT COALESCE(description, short_description, '(none)') ...
```

NULLIF

- 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:

```
NULLIF(value1, value2)
```

```
SELECT NULLIF(value, '(none)') ...
```

GREATEST & LEAST

- The `GREATEST` and `LEAST` functions select the largest or smallest value from a list of any number of expressions.

```
GREATEST(value [, ...])

LEAST(value [, ...])
```

Array Operators

Operator	Description	Example	Result
=	equal	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`	t
<>	not equal	`ARRAY[1,2,3] <> ARRAY[1,2,4]`	t
<	less than	`ARRAY[1,2,3] < ARRAY[1,2,4]`	t
>	greater than	`ARRAY[1,4,3] > ARRAY[1,2,4]`	t
<=	less than or equal	`ARRAY[1,2,3] <= ARRAY[1,2,3]`	t
>=	greater than or equal	`ARRAY[1,4,3] >= ARRAY[1,4,3]`	t
@>	contains	`ARRAY[1,4,3] @> ARRAY[3,1]`	t
<@	is contained by	`ARRAY[2,7] <@ ARRAY[1,7,4,2,6]`	t
&&	overlap (have elements in common)	`ARRAY[1,4,3] && ARRAY[2,1]`	t
||	array-to-array concatenation	`ARRAY[1,2,3] || ARRAY[4,5,6]`	{1,2,3,4,5,6}
||	array-to-array concatenation	`ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]`	{{1,2,3},{4,5,6},{7,8,9}}
||	element-to-array concatenation	`3 || ARRAY[4,5,6]`	{3,4,5,6}
||	array-to-element concatenation	`ARRAY[4,5,6] || 7`	{4,5,6,7}

Array Functions

Function	Return Type	Description	Example	Result
`array_append(anyarray, anyelement)`	`anyarray`	append an element to the end of an array	`array_append(ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat(anyarray, anyarray)`	`anyarray`	concatenate two arrays	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_ndims(anyarray)`	`int`	returns the number of dimensions of the array	`array_ndims(ARRAY[[1,2,3], [4,5,6]])`	`2`
`array_dims(anyarray)`	`text`	returns a text representation of array's dimensions	`array_dims(ARRAY[[1,2,3], [4,5,6]])`	`[1:2][1:3]`
`array_fill(anyelement, int[], [, int[]])`	`anyarray`	returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1	`array_fill(7, ARRAY[3], ARRAY[2])`	`[2:4]= {7,7,7}`
`array_length(anyarray, int)`	`int`	returns the length of the requested array dimension	`array_length(array[1,2,3], 1)`	`3`
`array_lower(anyarray, int)`	`int`	returns lower bound of the requested array dimension	`array_lower('[0:2]={1,2,3}'::int[], 1)`	`0`
`array_position(anyarray, anyelement [, int])`	`int`	returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional)	`array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon')`	`2`
`array_positions(anyarray, anyelement)`	`int[]`	returns an array of subscripts of all occurrences of the second argument in the array given as first argument (array must be one-dimensional)	`array_positions(ARRAY['A','A','B','A'], 'A')`	`{1,2,4}`
`array_prepend(anyelement, anyarray)`	`anyarray`	append an element to the beginning of an array	`array_prepend(1, ARRAY[2,3])`	`{1,2,3}`
`array_remove(anyarray, anyelement)`	`anyarray`	remove all elements equal to the given value from the array (array must be one-dimensional)	`array_remove(ARRAY[1,2,3,2], 2)`	`{1,3}`
`array_replace(anyarray, anyelement, anyelement)`	`anyarray`	replace each array element equal to the given value with a new value	`array_replace(ARRAY[1,2,5,4], 5, 3)`	`{1,2,3,4}`

Array Functions

`array_to_string(anyarray, text [, text])`	`text`	concatenates array elements using supplied delimiter and optional null string	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')`	`1,2,3,*,5`
`array_upper(anyarray, int)`	`int`	returns upper bound of the requested array dimension	`array_upper(ARRAY[1,8,3,7], 1)`	`4`
`cardinality(anyarray)`	`int`	returns the total number of elements in the array, or 0 if the array is empty	`cardinality(ARRAY[[1,2],[3,4]])`	`4`
`string_to_array(text, text [, text])`	`text[]`	splits string into array elements using supplied delimiter and optional null string	`string_to_array('xx~^~yy~^~zz', '~^~', 'yy')`	`{xx,NULL,zz}`

`unnest(anyarray)`	`setof anyelement`	expand an array to a set of rows	`unnest(ARRAY[1,2])`	`1`
				`2`
				(2 rows)

`unnest(anyarray, anyarray [, ...])`	`setof anyelement, anyelement [, ...]`	expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see Section 7.2.1.4	`unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])`	`1  foo`
				`2  bar`
				`NULL  baz`
				(3 rows)

Range Operators

Operator	Description	Example	Result
=	equal	`int4range(1,5) = '[1,4]'::int4range`	t
<>	not equal	`numrange(1.1,2.2) <> numrange(1.1,2.3)`	t
<	less than	`int4range(1,10) < int4range(2,3)`	t
>	greater than	`int4range(1,10) > int4range(1,5)`	t
<=	less than or equal	`numrange(1.1,2.2) <= numrange(1.1,2.2)`	t
>=	greater than or equal	`numrange(1.1,2.2) >= numrange(1.1,2.0)`	t
@>	contains range	`int4range(2,4) @> int4range(2,3)`	t
@>	contains element	`'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp`	t
<@	range is contained by	`int4range(2,4) <@ int4range(1,7)`	t
<@	element is contained by	`42 <@ int4range(1,7)`	f
&&	overlap (have points in common)	`int8range(3,7) && int8range(4,12)`	t
<<	strictly left of	`int8range(1,10) << int8range(100,110)`	t
>>	strictly right of	`int8range(50,60) >> int8range(20,30)`	t
&<	does not extend to the right of	`int8range(1,20) &< int8range(18,20)`	t
&>	does not extend to the left of	`int8range(7,20) &> int8range(5,10)`	t
-|-	is adjacent to	`numrange(1.1,2.2) -|- numrange(2.2,3.3)`	t
+	union	`numrange(5,15) + numrange(10,20)`	[5,20)
*	intersection	`int8range(5,15) * int8range(10,20)`	[10,15)
-	difference	`int8range(5,15) - int8range(10,20)`	[5,10)

Range Functions

Function	Return Type	Description	Example	Result
`lower(anyrange)`	range's element type	lower bound of range	`lower(numrange(1.1,2.2))`	`1.1`
`upper(anyrange)`	range's element type	upper bound of range	`upper(numrange(1.1,2.2))`	`2.2`
`isempty(anyrange)`	`boolean`	is the range empty?	`isempty(numrange(1.1,2.2))`	`false`
`lower_inc(anyrange)`	`boolean`	is the lower bound inclusive?	`lower_inc(numrange(1.1,2.2))`	`true`
`upper_inc(anyrange)`	`boolean`	is the upper bound inclusive?	`upper_inc(numrange(1.1,2.2))`	`false`
`lower_inf(anyrange)`	`boolean`	is the lower bound infinite?	`lower_inf('(,)'::daterange)`	`true`
`upper_inf(anyrange)`	`boolean`	is the upper bound infinite?	`upper_inf('(,)'::daterange)`	`true`
`range_merge(anyrange, anyrange)`	`anyrange`	the smallest range which includes both of the given ranges	`range_merge('[1,2)'::int4range, '[3,4)'::int4range)`	`[1,4)`

Aggregate Functions

Function	Argument Type(s)	Return Type	Partial Mode	Description
`array_agg(`*`expression`*`)`	any non-array type	array of the argument type	No	input values, including nulls, concatenated into an array
`array_agg(`*`expression`*`)`	any array type	same as argument data type	No	input arrays concatenated into array of one higher dimension (inputs must all have same dimensionality, and cannot be empty or NULL)
`avg(`*`expression`*`)`	`smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, or `interval`	`numeric` for any integer-type argument, `double precision` for a floating-point argument, otherwise the same as the argument data type	Yes	the average (arithmetic mean) of all input values
`bit_and(`*`expression`*`)`	`smallint`, `int`, `bigint`, or `bit`	same as argument data type	Yes	the bitwise AND of all non-null input values, or null if none
`bit_or(`*`expression`*`)`	`smallint`, `int`, `bigint`, or `bit`	same as argument data type	Yes	the bitwise OR of all non-null input values, or null if none
`bool_and(`*`expression`*`)`	`bool`	`bool`	Yes	true if all input values are true, otherwise false
`bool_or(`*`expression`*`)`	`bool`	`bool`	Yes	true if at least one input value is true, otherwise false
`count(*)`		`bigint`	Yes	number of input rows
`count(`*`expression`*`)`	any	`bigint`	Yes	number of input rows for which the value of *`expression`* is not null
`every(`*`expression`*`)`	`bool`	`bool`	Yes	equivalent to `bool_and`
`json_agg(`*`expression`*`)`	any	`json`	No	aggregates values as a JSON array
`jsonb_agg(`*`expression`*`)`	any	`jsonb`	No	aggregates values as a JSON array
`json_object_agg(`*`name`*`, `*`value`*`)`	`(any, any)`	`json`	No	aggregates name/value pairs as a JSON object
`jsonb_object_agg(`*`name`*`, `*`value`*`)`	`(any, any)`	`jsonb`	No	aggregates name/value pairs as a JSON object

Aggregate Functions

`max(`*`expression`*`)`	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	Yes	maximum value of *`expression`* across all input values
`min(`*`expression`*`)`	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	Yes	minimum value of *`expression`* across all input values
`string_agg(`*`expression`*`, `*`delimiter`*`)`	(`text`, `text`) or (`bytea`, `bytea`)	same as argument types	No	input values concatenated into a string, separated by delimiter
`sum(`*`expression`*`)`	`smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, `interval`, or `money`	`bigint` for `smallint` or `int` arguments, `numeric` for `bigint` arguments, otherwise the same as the argument data type	Yes	sum of *`expression`* across all input values
`xmlagg(`*`expression`*`)`	`xml`	`xml`	No	concatenation of XML values (see also Section 9.14.1.7)

EXISTS

- 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”

```
EXISTS (subquery)
```

EXISTS

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN

- 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

```
expression IN (subquery)
```

IN

```
SELECT col1
FROM tab1
WHERE col1 IN (SELECT col2 FROM tab2);
```

NOT IN

- 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

- The result is “false” if any equal row is found

```
expression NOT IN (subquery)
```

NOT IN

```
SELECT col1
FROM tab1
WHERE col1 NOT IN (SELECT col2 FROM tab2);
```

ANY/SOME

- 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/SOME is “true” if any true result is obtained.

- The result is “false” if no true result is found

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

ANY/SOME

```sql
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

ALL

- 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

- The result is “false” if any false result is found

```
expression operator ALL (subquery)
```

ALL

```sql
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

Questions?