Built-in Functions

!

! expr - Logical not.

Examples:

```
> SELECT ! true;
 false
> SELECT ! false;
 true
> SELECT ! NULL;
 NULL
```

Since: 1.0.0

!=

expr1 != expr2 - Returns true if `expr1` is not equal to `expr2`, or false otherwise.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be used in equality comparison. Map type is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 1 != 2;
 true
> SELECT 1 != '2';
 true
> SELECT true != NULL;
 NULL
> SELECT NULL != NULL;
 NULL
```

Since: 1.0.0

%

expr1 % expr2 - Returns the remainder after `expr1` / `expr2`.

Examples:

```
> SELECT 2 % 1.8;
 0.2
> SELECT MOD(2, 1.8);
 0.2
```

Since: 1.0.0

&

expr1 & expr2 - Returns the result of bitwise AND of `expr1` and `expr2`.

Examples:

```
> SELECT 3 & 5;
 1
```

Since: 1.4.0

*

expr1 * expr2 - Returns `expr1` * `expr2`.

Examples:

```
> SELECT 2 * 3;
 6
```

Since: 1.0.0

+

expr1 + expr2 - Returns `expr1` + `expr2`.

Examples:

```
> SELECT 1 + 2;
 3
```

Since: 1.0.0

-

expr1 - expr2 - Returns `expr1` - `expr2`.

Examples:

```
> SELECT 2 - 1;
 1
```

Since: 1.0.0

/

expr1 / expr2 - Returns `expr1` / `expr2`. It always performs floating point division.

Examples:

```
> SELECT 3 / 2;
 1.5
> SELECT 2L / 2L;
 1.0
```

Since: 1.0.0

<

expr1 < expr2 - Returns true if `expr1` is less than `expr2`.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be ordered. For example, map type is not orderable, so it is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 1 < 2;
 true
> SELECT 1.1 < '1';
 false
> SELECT to_date('2009-07-30 04:17:52') < to_date('2009-07-30 04:17:52');
 false
> SELECT to_date('2009-07-30 04:17:52') < to_date('2009-08-01 04:17:52');
 true
> SELECT 1 < NULL;
 NULL
```

Since: 1.0.0

<=

expr1 <= expr2 - Returns true if `expr1` is less than or equal to `expr2`.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be ordered. For example, map type is not orderable, so it is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 <= 2;
 true
> SELECT 1.0 <= '1';
 true
> SELECT to_date('2009-07-30 04:17:52') <= to_date('2009-07-30 04:17:52');
 true
> SELECT to_date('2009-07-30 04:17:52') <= to_date('2009-08-01 04:17:52');
 true
> SELECT 1 <= NULL;
 NULL
```

Since: 1.0.0

<=>

expr1 <=> expr2 - Returns same result as the EQUAL(=) operator for non-null operands, but returns true if both are null, false if one of the them is null.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be used in equality comparison. Map type is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 <=> 2;
 true
> SELECT 1 <=> '1';
 true
> SELECT true <=> NULL;
 false
> SELECT NULL <=> NULL;
 true
```

Since: 1.1.0

<>

expr1 != expr2 - Returns true if `expr1` is not equal to `expr2`, or false otherwise.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be used in equality comparison. Map type is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 1 != 2;
 true
> SELECT 1 != '2';
 true
> SELECT true != NULL;
 NULL
> SELECT NULL != NULL;
 NULL
```

Since: 1.0.0

=

expr1 = expr2 - Returns true if `expr1` equals `expr2`, or false otherwise.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be used in equality comparison. Map type is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 = 2;
 true
> SELECT 1 = '1';
 true
> SELECT true = NULL;
 NULL
> SELECT NULL = NULL;
 NULL
```

Since: 1.0.0

==

expr1 == expr2 - Returns true if `expr1` equals `expr2`, or false otherwise.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be used in equality comparison. Map type is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 == 2;
 true
> SELECT 1 == '1';
 true
> SELECT true == NULL;
 NULL
> SELECT NULL == NULL;
 NULL
```

Since: 1.0.0

>

expr1 > expr2 - Returns true if `expr1` is greater than `expr2`.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be ordered. For example, map type is not orderable, so it is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 > 1;
 true
> SELECT 2 > 1.1;
 true
> SELECT to_date('2009-07-30 04:17:52') > to_date('2009-07-30 04:17:52');
 false
> SELECT to_date('2009-07-30 04:17:52') > to_date('2009-08-01 04:17:52');
 false
> SELECT 1 > NULL;
 NULL
```

Since: 1.0.0

>=

expr1 >= expr2 - Returns true if `expr1` is greater than or equal to `expr2`.

Arguments:

- expr1, expr2 - the two expressions must be same type or can be casted to a common type, and must be a type that can be ordered. For example, map type is not orderable, so it is not supported. For complex types such array/struct, the data types of fields must be orderable.

Examples:

```
> SELECT 2 >= 1;
 true
> SELECT 2.0 >= '2.1';
 false
> SELECT to_date('2009-07-30 04:17:52') >= to_date('2009-07-30 04:17:52');
 true
> SELECT to_date('2009-07-30 04:17:52') >= to_date('2009-08-01 04:17:52');
 false
> SELECT 1 >= NULL;
 NULL
```

Since: 1.0.0

^

expr1 ^ expr2 - Returns the result of bitwise exclusive OR of `expr1` and `expr2`.

Examples:

```
> SELECT 3 ^ 5;
 6
```

Since: 1.4.0

abs

abs(expr) - Returns the absolute value of the numeric or interval value.

Examples:

```
> SELECT abs(-1);
 1
> SELECT abs(INTERVAL -'1-1' YEAR TO MONTH);
 1-1
```

Since: 1.2.0

acos

acos(expr) - Returns the inverse cosine (a.k.a. arc cosine) of `expr`, as if computed by `java.lang.Math.acos`.

Examples:

```
> SELECT acos(1);
 0.0
> SELECT acos(2);
 NaN
```

Since: 1.4.0

acosh

acosh(expr) - Returns inverse hyperbolic cosine of `expr`.

Examples:

```
> SELECT acosh(1);
 0.0
> SELECT acosh(0);
 NaN
```

Since: 3.0.0

add_months

add_months(start_date, num_months) - Returns the date that is `num_months` after `start_date`.

Examples:

```
> SELECT add_months('2016-08-31', 1);
 2016-09-30
```

Since: 1.5.0

aes_decrypt

aes_decrypt(expr, key[, mode[, padding]]) - Returns a decrypted value of `expr` using AES in `mode` with `padding`. Key lengths of 16, 24 and 32 bits are supported. Supported combinations of (`mode`, `padding`) are ('ECB', 'PKCS') and ('GCM', 'NONE'). The default mode is GCM.

Arguments:

- expr - The binary value to decrypt.
- key - The passphrase to use to decrypt the data.
- mode - Specifies which block cipher mode should be used to decrypt messages. Valid modes: ECB, GCM.
- padding - Specifies how to pad messages whose length is not a multiple of the block size. Valid values: PKCS, NONE, DEFAULT. The DEFAULT padding means PKCS for ECB and NONE for GCM.

Examples:

```
> SELECT aes_decrypt(unhex('83F16B2AA704794132802D248E6BFD4E380078182D1544813898AC97E709B28A
 Spark
> SELECT aes_decrypt(unhex('6E7CA17BBB468D3084B5744BCA729FB7B2B7BCB8E4472847D02670489D95FA97
 Spark SQL
> SELECT aes_decrypt(unbase64('3lmwu+Mw0H3fi5NDvcu9lg=='), '1234567890abcdef', 'ECB', 'PKCS'
 Spark SQL
```

Since: 3.3.0

aes_encrypt

aes_encrypt(expr, key[, mode[, padding]]) - Returns an encrypted value of `expr` using AES in given `mode` with the specified `padding`. Key lengths of 16, 24 and 32 bits are supported. Supported combinations of (`mode`, `padding`) are ('ECB', 'PKCS') and ('GCM', 'NONE'). The default mode is GCM.

Arguments:

- expr - The binary value to encrypt.
- key - The passphrase to use to encrypt the data.

- mode - Specifies which block cipher mode should be used to encrypt messages. Valid modes: ECB, GCM.
- padding - Specifies how to pad messages whose length is not a multiple of the block size. Valid values: PKCS, NONE, DEFAULT. The DEFAULT padding means PKCS for ECB and NONE for GCM.

Examples:

```
> SELECT hex(aes_encrypt('Spark', '0000111122223333'));
 83F16B2AA704794132802D248E6BFD4E380078182D1544813898AC97E709B28A94
> SELECT hex(aes_encrypt('Spark SQL', '0000111122223333', 'GCM'));
 6E7CA17BBB468D3084B5744BCA729FB7B2B7BCB8E4472847D02670489D95FA97DBBA7D3210
> SELECT base64(aes_encrypt('Spark SQL', '1234567890abcdef', 'ECB', 'PKCS'));
 3lmwu+Mw0H3fi5NDvcu9lg==
```

Since: 3.3.0

aggregate

aggregate(expr, start, merge, finish) - Applies a binary operator to an initial state and all elements in the array, and reduces this to a single state. The final state is converted into the final result by applying a finish function.

Examples:

```
> SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x);
 6
> SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x, acc -> acc * 10);
 60
```

Since: 2.4.0

and

expr1 and expr2 - Logical AND.

Examples:

```
> SELECT true and true;
 true
> SELECT true and false;
 false
> SELECT true and NULL;
 NULL
> SELECT false and NULL;
 false
```

Since: 1.0.0

any

any(expr) - Returns true if at least one value of `expr` is true.

Examples:

```
> SELECT any(col) FROM VALUES (true), (false), (false) AS tab(col);
 true
> SELECT any(col) FROM VALUES (NULL), (true), (false) AS tab(col);
 true
> SELECT any(col) FROM VALUES (false), (false), (NULL) AS tab(col);
 false
```

Since: 3.0.0

approx_count_distinct

approx_count_distinct(expr[, relativeSD]) - Returns the estimated cardinality by HyperLogLog++. `relativeSD` defines the maximum relative standard deviation allowed.

Examples:

```
> SELECT approx_count_distinct(col1) FROM VALUES (1), (1), (2), (2), (3) tab(col1);
 3
```

Since: 1.6.0

approx_percentile

approx_percentile(col, percentage [, accuracy]) - Returns the approximate `percentile` of the numeric or ansi interval column `col` which is the smallest value in the ordered `col` values (sorted from least to greatest) such that no more than `percentage` of `col` values is less than the value or equal to that value. The value of percentage must be between 0.0 and 1.0. The `accuracy` parameter (default: 10000) is a positive numeric literal which controls approximation accuracy at the cost of memory. Higher value of `accuracy` yields better accuracy, `1.0/accuracy` is the relative error of the approximation. When `percentage` is an array, each value of the percentage array must be between 0.0 and 1.0. In this case, returns the approximate percentile array of column `col` at the given percentage array.

Examples:

```
> SELECT approx_percentile(col, array(0.5, 0.4, 0.1), 100) FROM VALUES (0), (1), (2), (10) A
 [1,1,0]
> SELECT approx_percentile(col, 0.5, 100) FROM VALUES (0), (6), (7), (9), (10) AS tab(col);
 7
> SELECT approx_percentile(col, 0.5, 100) FROM VALUES (INTERVAL '0' MONTH), (INTERVAL '1' MO
 0-1
> SELECT approx_percentile(col, array(0.5, 0.7), 100) FROM VALUES (INTERVAL '0' SECOND), (IN
 [0 00:00:01.000000000,0 00:00:02.000000000]
```

Since: 2.1.0

array

array(expr, ...) - Returns an array with the given elements.

Examples:

```
> SELECT array(1, 2, 3);
 [1,2,3]
```

Since: 1.1.0

array_agg

array_agg(expr) - Collects and returns a list of non-unique elements.

Examples:

```
> SELECT array_agg(col) FROM VALUES (1), (2), (1) AS tab(col);
 [1,2,1]
```

Note:

The function is non-deterministic because the order of collected results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

array_contains

array_contains(array, value) - Returns true if the array contains the value.

Examples:

```
> SELECT array_contains(array(1, 2, 3), 2);
 true
```

Since: 1.5.0

array_distinct

array_distinct(array) - Removes duplicate values from the array.

Examples:

```
> SELECT array_distinct(array(1, 2, 3, null, 3));
 [1,2,3,null]
```

Since: 2.4.0

array_except

array_except(array1, array2) - Returns an array of the elements in array1 but not in array2, without duplicates.

Examples:

```
> SELECT array_except(array(1, 2, 3), array(1, 3, 5));
 [2]
```

Since: 2.4.0

array_intersect

array_intersect(array1, array2) - Returns an array of the elements in the intersection of array1 and array2, without duplicates.

Examples:

```
> SELECT array_intersect(array(1, 2, 3), array(1, 3, 5));
 [1,3]
```

Since: 2.4.0

array_join

array_join(array, delimiter[, nullReplacement]) - Concatenates the elements of the given array using the delimiter and an optional string to replace nulls. If no value is set for nullReplacement, any null value is filtered.

Examples:

```
> SELECT array_join(array('hello', 'world'), ' ');
 hello world
> SELECT array_join(array('hello', null ,'world'), ' ');
 hello world
> SELECT array_join(array('hello', null ,'world'), ' ', ',');
 hello , world
```

Since: 2.4.0

array_max

array_max(array) - Returns the maximum value in the array. NaN is greater than any non-NaN elements for double/float type. NULL elements are skipped.

Examples:

```
> SELECT array_max(array(1, 20, null, 3));
 20
```

Since: 2.4.0

array_min

array_min(array) - Returns the minimum value in the array. NaN is greater than any non-NaN elements for double/float type. NULL elements are skipped.

Examples:

```
> SELECT array_min(array(1, 20, null, 3));
 1
```

Since: 2.4.0

array_position

array_position(array, element) - Returns the (1-based) index of the first element of the array as long.

Examples:

```
> SELECT array_position(array(3, 2, 1), 1);
 3
```

Since: 2.4.0

array_remove

array_remove(array, element) - Remove all elements that equal to element from array.

Examples:

```
> SELECT array_remove(array(1, 2, 3, null, 3), 3);
 [1,2,null]
```

Since: 2.4.0

array_repeat

array_repeat(element, count) - Returns the array containing element count times.

Examples:

```
> SELECT array_repeat('123', 2);
 ["123","123"]
```

Since: 2.4.0

array_size

array_size(expr) - Returns the size of an array. The function returns null for null input.

Examples:

```
> SELECT array_size(array('b', 'd', 'c', 'a'));
 4
```

Since: 3.3.0

array_sort

array_sort(expr, func) - Sorts the input array. If func is omitted, sort in ascending order. The elements of the input array must be orderable. NaN is greater than any non-NaN elements for double/float type. Null elements will be placed at the end of the returned array. Since 3.0.0 this function also sorts and returns the array based on the given comparator function. The

comparator will take two arguments representing two elements of the array. It returns -1, 0, or 1 as the first element is less than, equal to, or greater than the second element. If the comparator function returns other values (including null), the function will fail and raise an error.

Examples:

```
> SELECT array_sort(array(5, 6, 1), (left, right) -> case when left < right then -1 when lef
 [1,5,6]
> SELECT array_sort(array('bc', 'ab', 'dc'), (left, right) -> case when left is null and rig
 ["dc","bc","ab"]
> SELECT array_sort(array('b', 'd', null, 'c', 'a'));
 ["a","b","c","d",null]
```

Since: 2.4.0

array_union

array_union(array1, array2) - Returns an array of the elements in the union of array1 and array2, without duplicates.

Examples:

```
> SELECT array_union(array(1, 2, 3), array(1, 3, 5));
 [1,2,3,5]
```

Since: 2.4.0

arrays_overlap

arrays_overlap(a1, a2) - Returns true if a1 contains at least a non-null element present also in a2. If the arrays have no common element and they are both non-empty and either of them contains a null element null is returned, false otherwise.

Examples:

```
> SELECT arrays_overlap(array(1, 2, 3), array(3, 4, 5));
 true
```

Since: 2.4.0

arrays_zip

arrays_zip(a1, a2, ...) - Returns a merged array of structs in which the N-th struct contains all N-th values of input arrays.

Examples:

```
> SELECT arrays_zip(array(1, 2, 3), array(2, 3, 4));
 [{"0":1,"1":2},{"0":2,"1":3},{"0":3,"1":4}]
> SELECT arrays_zip(array(1, 2), array(2, 3), array(3, 4));
 [{"0":1,"1":2,"2":3},{"0":2,"1":3,"2":4}]
```

Since: 2.4.0

ascii

ascii(str) - Returns the numeric value of the first character of `str`.

Examples:

```
> SELECT ascii('222');
 50
> SELECT ascii(2);
 50
```

Since: 1.5.0

asin

asin(expr) - Returns the inverse sine (a.k.a. arc sine) the arc sin of `expr`, as if computed by `java.lang.Math.asin`.

Examples:

```
> SELECT asin(0);
 0.0
> SELECT asin(2);
 NaN
```

Since: 1.4.0

asinh

asinh(expr) - Returns inverse hyperbolic sine of `expr`.

Examples:

```
> SELECT asinh(0);
 0.0
```

Since: 3.0.0

assert_true

assert_true(expr) - Throws an exception if `expr` is not true.

Examples:

```
> SELECT assert_true(0 < 1);
 NULL
```

Since: 2.0.0

atan

atan(expr) - Returns the inverse tangent (a.k.a. arc tangent) of `expr`, as if computed by `java.lang.Math.atan`

Examples:

```
> SELECT atan(0);
 0.0
```

Since: 1.4.0

atan2

atan2(exprY, exprX) - Returns the angle in radians between the positive x-axis of a plane and the point given by the coordinates (`exprX` , `exprY`), as if computed by `java.lang.Math.atan2` .

Arguments:

- exprY - coordinate on y-axis
- exprX - coordinate on x-axis

Examples:

```
> SELECT atan2(0, 0);
 0.0
```

Since: 1.4.0

atanh

atanh(expr) - Returns inverse hyperbolic tangent of `expr` .

Examples:

```
> SELECT atanh(0);
 0.0
> SELECT atanh(2);
 NaN
```

Since: 3.0.0

avg

avg(expr) - Returns the mean calculated from values of a group.

Examples:

```
> SELECT avg(col) FROM VALUES (1), (2), (3) AS tab(col);
 2.0
> SELECT avg(col) FROM VALUES (1), (2), (NULL) AS tab(col);
 1.5
```

Since: 1.0.0

base64

base64(bin) - Converts the argument from a binary `bin` to a base 64 string.

Examples:

```
> SELECT base64('Spark SQL');
 U3BhcmsgU1FM
```

Since: 1.5.0

between

expr1 [NOT] BETWEEN expr2 AND expr3 - evaluate if `expr1` is [not] in between `expr2` and `expr3`.

Examples:

```
> SELECT col1 FROM VALUES 1, 3, 5, 7 WHERE col1 BETWEEN 2 AND 5;
 3
 5
```

Since: 1.0.0

bigint

bigint(expr) - Casts the value `expr` to the target data type `bigint`.

Since: 2.0.1

bin

bin(expr) - Returns the string representation of the long value `expr` represented in binary.

Examples:

```
> SELECT bin(13);
 1101
> SELECT bin(-13);
 1111111111111111111111111111111111111111111111111111111111110011
> SELECT bin(13.3);
 1101
```

Since: 1.5.0

binary

binary(expr) - Casts the value `expr` to the target data type `binary`.

Since: 2.0.1

bit_and

bit_and(expr) - Returns the bitwise AND of all non-null input values, or null if none.

Examples:

```
> SELECT bit_and(col) FROM VALUES (3), (5) AS tab(col);
 1
```

Since: 3.0.0

bit_count

bit_count(expr) - Returns the number of bits that are set in the argument expr as an unsigned 64-bit integer, or NULL if the argument is NULL.

Examples:

```
> SELECT bit_count(0);
 0
```

Since: 3.0.0

bit_get

bit_get(expr, pos) - Returns the value of the bit (0 or 1) at the specified position. The positions are numbered from right to left, starting at zero. The position argument cannot be negative.

Examples:

```
> SELECT bit_get(11, 0);
 1
> SELECT bit_get(11, 2);
 0
```

Since: 3.2.0

bit_length

bit_length(expr) - Returns the bit length of string data or number of bits of binary data.

Examples:

```
> SELECT bit_length('Spark SQL');
 72
```

Since: 2.3.0

bit_or

bit_or(expr) - Returns the bitwise OR of all non-null input values, or null if none.

Examples:

```
> SELECT bit_or(col) FROM VALUES (3), (5) AS tab(col);
 7
```

Since: 3.0.0

bit_xor

bit_xor(expr) - Returns the bitwise XOR of all non-null input values, or null if none.

Examples:

```
> SELECT bit_xor(col) FROM VALUES (3), (5) AS tab(col);
 6
```

Since: 3.0.0

bool_and

bool_and(expr) - Returns true if all values of `expr` are true.

Examples:

```
> SELECT bool_and(col) FROM VALUES (true), (true), (true) AS tab(col);
 true
> SELECT bool_and(col) FROM VALUES (NULL), (true), (true) AS tab(col);
 true
> SELECT bool_and(col) FROM VALUES (true), (false), (true) AS tab(col);
 false
```

Since: 3.0.0

bool_or

bool_or(expr) - Returns true if at least one value of `expr` is true.

Examples:

```
> SELECT bool_or(col) FROM VALUES (true), (false), (false) AS tab(col);
 true
> SELECT bool_or(col) FROM VALUES (NULL), (true), (false) AS tab(col);
 true
> SELECT bool_or(col) FROM VALUES (false), (false), (NULL) AS tab(col);
 false
```

Since: 3.0.0

boolean

boolean(expr) - Casts the value `expr` to the target data type `boolean`.

Since: 2.0.1

bround

bround(expr, d) - Returns `expr` rounded to `d` decimal places using HALF_EVEN rounding mode.

Examples:

```
> SELECT bround(2.5, 0);
 2
> SELECT bround(25, -1);
 20
```

Since: 2.0.0

btrim

btrim(str) - Removes the leading and trailing space characters from `str`.

btrim(str, trimStr) - Remove the leading and trailing `trimStr` characters from `str`.

Arguments:

- str - a string expression
- trimStr - the trim string characters to trim, the default value is a single space

Examples:

```
> SELECT btrim('    SparkSQL   ');
 SparkSQL
> SELECT btrim(encode('    SparkSQL   ', 'utf-8'));
 SparkSQL
> SELECT btrim('SSparkSQLS', 'SL');
 parkSQ
> SELECT btrim(encode('SSparkSQLS', 'utf-8'), encode('SL', 'utf-8'));
 parkSQ
```

Since: 3.2.0

cardinality

cardinality(expr) - Returns the size of an array or a map. The function returns null for null input if spark.sql.legacy.sizeOfNull is set to false or spark.sql.ansi.enabled is set to true. Otherwise, the function returns -1 for null input. With the default settings, the function returns -1 for null input.

Examples:

```
> SELECT cardinality(array('b', 'd', 'c', 'a'));
 4
> SELECT cardinality(map('a', 1, 'b', 2));
 2
```

Since: 1.5.0

case

CASE expr1 WHEN expr2 THEN expr3 [WHEN expr4 THEN expr5]* [ELSE expr6] END - When `expr1` = `expr2`, returns `expr3`; when `expr1` = `expr4`, return `expr5`; else return `expr6`.

Arguments:

- expr1 - the expression which is one operand of comparison.
- expr2, expr4 - the expressions each of which is the other operand of comparison.
- expr3, expr5, expr6 - the branch value expressions and else value expression should all be same type or coercible to a common type.

Examples:

```
> SELECT CASE col1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE '?' END FROM VALUES 1, 2, 3;
 one
 two
 ?
> SELECT CASE col1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' END FROM VALUES 1, 2, 3;
 one
 two
 NULL
```

Since: 1.0.1

cast

cast(expr AS type) - Casts the value `expr` to the target data type `type`.

Examples:

```
> SELECT cast('10' as int);
 10
```

Since: 1.0.0

cbrt

cbrt(expr) - Returns the cube root of `expr`.

Examples:

```
> SELECT cbrt(27.0);
 3.0
```

Since: 1.4.0

ceil

ceil(expr[, scale]) - Returns the smallest number after rounding up that is not smaller than `expr`. An optional `scale` parameter can be specified to control the rounding behavior.

Examples:

```
> SELECT ceil(-0.1);
 0
> SELECT ceil(5);
 5
> SELECT ceil(3.1411, 3);
 3.142
> SELECT ceil(3.1411, -3);
 1000
```

Since: 3.3.0

ceiling

ceiling(expr[, scale]) - Returns the smallest number after rounding up that is not smaller than `expr`. An optional `scale` parameter can be specified to control the rounding behavior.

Examples:

```
> SELECT ceiling(-0.1);
 0
> SELECT ceiling(5);
 5
> SELECT ceiling(3.1411, 3);
 3.142
> SELECT ceiling(3.1411, -3);
 1000
```

Since: 3.3.0

char

char(expr) - Returns the ASCII character having the binary equivalent to `expr`. If n is larger than 256 the result is equivalent to chr(n % 256)

Examples:

```
> SELECT char(65);
 A
```

Since: 2.3.0

char_length

char_length(expr) - Returns the character length of string data or number of bytes of binary data. The length of string data includes the trailing spaces. The length of binary data includes binary zeros.

Examples:

```
> SELECT char_length('Spark SQL ');
 10
> SELECT CHAR_LENGTH('Spark SQL ');
 10
> SELECT CHARACTER_LENGTH('Spark SQL ');
 10
```

Since: 1.5.0

character_length

character_length(expr) - Returns the character length of string data or number of bytes of binary data. The length of string data includes the trailing spaces. The length of binary data includes binary zeros.

Examples:

```
> SELECT character_length('Spark SQL ');
 10
> SELECT CHAR_LENGTH('Spark SQL ');
 10
> SELECT CHARACTER_LENGTH('Spark SQL ');
 10
```

Since: 1.5.0

chr

chr(expr) - Returns the ASCII character having the binary equivalent to `expr`. If n is larger than 256 the result is equivalent to chr(n % 256)

Examples:

```
> SELECT chr(65);
 A
```

Since: 2.3.0

coalesce

coalesce(expr1, expr2, ...) - Returns the first non-null argument if exists. Otherwise, null.

Examples:

```
> SELECT coalesce(NULL, 1, NULL);
 1
```

Since: 1.0.0

collect_list

collect_list(expr) - Collects and returns a list of non-unique elements.

Examples:

```
> SELECT collect_list(col) FROM VALUES (1), (2), (1) AS tab(col);
 [1,2,1]
```

Note:

The function is non-deterministic because the order of collected results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

collect_set

collect_set(expr) - Collects and returns a set of unique elements.

Examples:

```
> SELECT collect_set(col) FROM VALUES (1), (2), (1) AS tab(col);
 [1,2]
```

Note:

The function is non-deterministic because the order of collected results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

concat

concat(col1, col2, ..., colN) - Returns the concatenation of col1, col2, ..., colN.

Examples:

```
> SELECT concat('Spark', 'SQL');
 SparkSQL
> SELECT concat(array(1, 2, 3), array(4, 5), array(6));
 [1,2,3,4,5,6]
```

Note:

Concat logic for arrays is available since 2.4.0.

Since: 1.5.0

concat_ws

concat_ws(sep[, str | array(str)]+) - Returns the concatenation of the strings separated by `sep`.

Examples:

```
> SELECT concat_ws(' ', 'Spark', 'SQL');
  Spark SQL
> SELECT concat_ws('s');
```

Since: 1.5.0

contains

contains(left, right) - Returns a boolean. The value is True if right is found inside left. Returns NULL if either input expression is NULL. Otherwise, returns False. Both left or right must be of STRING or BINARY type.

Examples:

```
> SELECT contains('Spark SQL', 'Spark');
 true
> SELECT contains('Spark SQL', 'SPARK');
 false
> SELECT contains('Spark SQL', null);
 NULL
> SELECT contains(x'537061726b2053514c', x'537061726b');
 true
```

Since: 3.3.0

conv

conv(num, from_base, to_base) - Convert `num` from `from_base` to `to_base`.

Examples:

```
> SELECT conv('100', 2, 10);
 4
> SELECT conv(-10, 16, -10);
 -16
```

Since: 1.5.0

corr

corr(expr1, expr2) - Returns Pearson coefficient of correlation between a set of number pairs.

Examples:

```
> SELECT corr(c1, c2) FROM VALUES (3, 2), (3, 3), (6, 4) as tab(c1, c2);
 0.8660254037844387
```

Since: 1.6.0

cos

cos(expr) - Returns the cosine of `expr`, as if computed by `java.lang.Math.cos`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT cos(0);
 1.0
```

Since: 1.4.0

cosh

cosh(expr) - Returns the hyperbolic cosine of `expr`, as if computed by `java.lang.Math.cosh`.

Arguments:

- expr - hyperbolic angle

Examples:

```
> SELECT cosh(0);
 1.0
```

Since: 1.4.0

cot

cot(expr) - Returns the cotangent of `expr`, as if computed by `1/java.lang.Math.tan`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT cot(1);
 0.6420926159343306
```

Since: 2.3.0

count

count(*) - Returns the total number of retrieved rows, including rows containing null.

count(expr[, expr...]) - Returns the number of rows for which the supplied expression(s) are all non-null.

count(DISTINCT expr[, expr...]) - Returns the number of rows for which the supplied expression(s) are unique and non-null.

Examples:

```
> SELECT count(*) FROM VALUES (NULL), (5), (5), (20) AS tab(col);
 4
> SELECT count(col) FROM VALUES (NULL), (5), (5), (20) AS tab(col);
 3
> SELECT count(DISTINCT col) FROM VALUES (NULL), (5), (5), (10) AS tab(col);
 2
```

Since: 1.0.0

count_if

count_if(expr) - Returns the number of `TRUE` values for the expression.

Examples:

```
> SELECT count_if(col % 2 = 0) FROM VALUES (NULL), (0), (1), (2), (3) AS tab(col);
 2
> SELECT count_if(col IS NULL) FROM VALUES (NULL), (0), (1), (2), (3) AS tab(col);
 1
```

Since: 3.0.0

count_min_sketch

count_min_sketch(col, eps, confidence, seed) - Returns a count-min sketch of a column with the given esp, confidence and seed. The result is an array of bytes, which can be deserialized to a `CountMinSketch` before usage. Count-min sketch is a probabilistic data structure used for cardinality estimation using sub-linear space.

Examples:

```
> SELECT hex(count_min_sketch(col, 0.5d, 0.5d, 1)) FROM VALUES (1), (2), (1) AS tab(col);
 0000000100000000000000030000000100000004000000005D8D6AB900000000000000000000000000000002000
```

Since: 2.2.0

covar_pop

covar_pop(expr1, expr2) - Returns the population covariance of a set of number pairs.

Examples:

```
> SELECT covar_pop(c1, c2) FROM VALUES (1,1), (2,2), (3,3) AS tab(c1, c2);
 0.6666666666666666
```

Since: 2.0.0

covar_samp

covar_samp(expr1, expr2) - Returns the sample covariance of a set of number pairs.

Examples:

```
> SELECT covar_samp(c1, c2) FROM VALUES (1,1), (2,2), (3,3) AS tab(c1, c2);
 1.0
```

Since: 2.0.0

crc32

crc32(expr) - Returns a cyclic redundancy check value of the `expr` as a bigint.

Examples:

```
> SELECT crc32('Spark');
 1557323817
```

Since: 1.5.0

csc

csc(expr) - Returns the cosecant of `expr`, as if computed by `1/java.lang.Math.sin`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT csc(1);
 1.1883951057781212
```

Since: 3.3.0

cume_dist

cume_dist() - Computes the position of a value relative to all values in the partition.

Examples:

```
> SELECT a, b, cume_dist() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1)
 A1 1   0.6666666666666666
 A1 1   0.6666666666666666
 A1 2   1.0
 A2 3   1.0
```

Since: 2.0.0

current_catalog

current_catalog() - Returns the current catalog.

Examples:

```
> SELECT current_catalog();
 spark_catalog
```

Since: 3.1.0

current_database

current_database() - Returns the current database.

Examples:

```
> SELECT current_database();
 default
```

Since: 1.6.0

current_date

current_date() - Returns the current date at the start of query evaluation. All calls of current_date within the same query return the same value.

current_date - Returns the current date at the start of query evaluation.

Examples:

```
> SELECT current_date();
 2020-04-25
> SELECT current_date;
 2020-04-25
```

Note:

The syntax without braces has been supported since 2.0.1.

Since: 1.5.0

current_timestamp

current_timestamp() - Returns the current timestamp at the start of query evaluation. All calls of current_timestamp within the same query return the same value.

current_timestamp - Returns the current timestamp at the start of query evaluation.

Examples:

```
> SELECT current_timestamp();
 2020-04-25 15:49:11.914
> SELECT current_timestamp;
 2020-04-25 15:49:11.914
```

Note:

The syntax without braces has been supported since 2.0.1.

Since: 1.5.0

current_timezone

current_timezone() - Returns the current session local timezone.

Examples:

```
> SELECT current_timezone();
 Asia/Shanghai
```

Since: 3.1.0

current_user

current_user() - user name of current execution context.

Examples:

```
> SELECT current_user();
 mockingjay
```

Since: 3.2.0

date

date(expr) - Casts the value `expr` to the target data type `date`.

Since: 2.0.1

date_add

date_add(start_date, num_days) - Returns the date that is `num_days` after `start_date`.

Examples:

```
> SELECT date_add('2016-07-30', 1);
 2016-07-31
```

Since: 1.5.0

date_format

date_format(timestamp, fmt) - Converts `timestamp` to a value of string in the format specified by the date format `fmt`.

Arguments:

- timestamp - A date/timestamp or string to be converted to the given format.
- fmt - Date/time format pattern to follow. See Datetime Patterns for valid date and time format patterns.

Examples:

```
> SELECT date_format('2016-04-08', 'y');
 2016
```

Since: 1.5.0

date_from_unix_date

date_from_unix_date(days) - Create date from the number of days since 1970-01-01.

Examples:

```
> SELECT date_from_unix_date(1);
 1970-01-02
```

Since: 3.1.0

date_part

date_part(field, source) - Extracts a part of the date/timestamp or interval source.

Arguments:

- field - selects which part of the source should be extracted, and supported string values are as same as the fields of the equivalent function `EXTRACT`.
- source - a date/timestamp or interval column from where `field` should be extracted

Examples:

```
> SELECT date_part('YEAR', TIMESTAMP '2019-08-12 01:00:00.123456');
 2019
> SELECT date_part('week', timestamp'2019-08-12 01:00:00.123456');
 33
> SELECT date_part('doy', DATE'2019-08-12');
 224
> SELECT date_part('SECONDS', timestamp'2019-10-01 00:00:01.000001');
 1.000001
> SELECT date_part('days', interval 5 days 3 hours 7 minutes);
 5
> SELECT date_part('seconds', interval 5 hours 30 seconds 1 milliseconds 1 microseconds);
 30.001001
> SELECT date_part('MONTH', INTERVAL '2021-11' YEAR TO MONTH);
 11
> SELECT date_part('MINUTE', INTERVAL '123 23:55:59.002001' DAY TO SECOND);
 55
```

Note:

The date_part function is equivalent to the SQL-standard function `EXTRACT(field FROM source)`

Since: 3.0.0

date_sub

date_sub(start_date, num_days) - Returns the date that is `num_days` before `start_date`.

Examples:

```
> SELECT date_sub('2016-07-30', 1);
 2016-07-29
```

Since: 1.5.0

date_trunc

date_trunc(fmt, ts) - Returns timestamp `ts` truncated to the unit specified by the format model `fmt`.

Arguments:

- fmt - the format representing the unit to be truncated to
  - "YEAR", "YYYY", "YY" - truncate to the first date of the year that the `ts` falls in, the time part will be zero out

- "QUARTER" - truncate to the first date of the quarter that the `ts` falls in, the time part will be zero out
- "MONTH", "MM", "MON" - truncate to the first date of the month that the `ts` falls in, the time part will be zero out
- "WEEK" - truncate to the Monday of the week that the `ts` falls in, the time part will be zero out
- "DAY", "DD" - zero out the time part
- "HOUR" - zero out the minute and second with fraction part
- "MINUTE"- zero out the second with fraction part
- "SECOND" - zero out the second fraction part
- "MILLISECOND" - zero out the microseconds
- "MICROSECOND" - everything remains

- ts - datetime value or valid timestamp string

Examples:

```
> SELECT date_trunc('YEAR', '2015-03-05T09:32:05.359');
 2015-01-01 00:00:00
> SELECT date_trunc('MM', '2015-03-05T09:32:05.359');
 2015-03-01 00:00:00
> SELECT date_trunc('DD', '2015-03-05T09:32:05.359');
 2015-03-05 00:00:00
> SELECT date_trunc('HOUR', '2015-03-05T09:32:05.359');
 2015-03-05 09:00:00
> SELECT date_trunc('MILLISECOND', '2015-03-05T09:32:05.123456');
 2015-03-05 09:32:05.123
```

Since: 2.3.0

datediff

datediff(endDate, startDate) - Returns the number of days from `startDate` to `endDate`.

Examples:

```
> SELECT datediff('2009-07-31', '2009-07-30');
 1

> SELECT datediff('2009-07-30', '2009-07-31');
 -1
```

Since: 1.5.0

day

day(date) - Returns the day of month of the date/timestamp.

Examples:

```
> SELECT day('2009-07-30');
 30
```

Since: 1.5.0

dayofmonth

dayofmonth(date) - Returns the day of month of the date/timestamp.

Examples:

```
> SELECT dayofmonth('2009-07-30');
 30
```

Since: 1.5.0

dayofweek

dayofweek(date) - Returns the day of the week for date/timestamp (1 = Sunday, 2 = Monday, ..., 7 = Saturday).

Examples:

```
> SELECT dayofweek('2009-07-30');
 5
```

Since: 2.3.0

dayofyear

dayofyear(date) - Returns the day of year of the date/timestamp.

Examples:

```
> SELECT dayofyear('2016-04-09');
 100
```

Since: 1.5.0

decimal

decimal(expr) - Casts the value `expr` to the target data type `decimal`.

Since: 2.0.1

decode

decode(bin, charset) - Decodes the first argument using the second argument character set.

decode(expr, search, result [, search, result] ... [, default]) - Compares expr to each search value in order. If expr is equal to a search value, decode returns the corresponding result. If no match is found, then it returns default. If default is omitted, it returns null.

Examples:

```
> SELECT decode(encode('abc', 'utf-8'), 'utf-8');
 abc
> SELECT decode(2, 1, 'Southlake', 2, 'San Francisco', 3, 'New Jersey', 4, 'Seattle', 'Non d
 San Francisco
> SELECT decode(6, 1, 'Southlake', 2, 'San Francisco', 3, 'New Jersey', 4, 'Seattle', 'Non d
 Non domestic
> SELECT decode(6, 1, 'Southlake', 2, 'San Francisco', 3, 'New Jersey', 4, 'Seattle');
 NULL
```

Since: 3.2.0

degrees

degrees(expr) - Converts radians to degrees.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT degrees(3.141592653589793);
 180.0
```

Since: 1.4.0

dense_rank

dense_rank() - Computes the rank of a value in a group of values. The result is one plus the previously assigned rank value. Unlike the function rank, dense_rank will not produce gaps in the ranking sequence.

Arguments:

- children - this is to base the rank on; a change in the value of one the children will trigger a change in rank. This is an internal parameter and will be assigned by the Analyser.

Examples:

```
> SELECT a, b, dense_rank(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1',
 A1 1   1
 A1 1   1
 A1 2   2
 A2 3   1
```

Since: 2.0.0

div

expr1 div expr2 - Divide `expr1` by `expr2`. It returns NULL if an operand is NULL or `expr2` is 0. The result is casted to long.

Examples:

```
> SELECT 3 div 2;
 1
> SELECT INTERVAL '1-1' YEAR TO MONTH div INTERVAL '-1' MONTH;
 -13
```

Since: 3.0.0

double

double(expr) - Casts the value `expr` to the target data type `double`.

Since: 2.0.1

e

e() - Returns Euler's number, e.

Examples:

```
> SELECT e();
 2.718281828459045
```

Since: 1.5.0

element_at

element_at(array, index) - Returns element of array at given (1-based) index. If Index is 0, Spark will throw an error. If index < 0, accesses elements from the last to the first. The function returns NULL if the index exceeds the length of the array and `spark.sql.ansi.enabled` is set to false. If `spark.sql.ansi.enabled` is set to true, it throws ArrayIndexOutOfBoundsException for invalid indices.

element_at(map, key) - Returns value for given key. The function returns NULL if the key is not contained in the map and `spark.sql.ansi.enabled` is set to false. If `spark.sql.ansi.enabled` is set to true, it throws NoSuchElementException instead.

Examples:

```
> SELECT element_at(array(1, 2, 3), 2);
 2
> SELECT element_at(map(1, 'a', 2, 'b'), 2);
 b
```

Since: 2.4.0

elt

elt(n, input1, input2, ...) - Returns the `n`-th input, e.g., returns `input2` when `n` is 2. The function returns NULL if the index exceeds the length of the array and `spark.sql.ansi.enabled` is set to false. If `spark.sql.ansi.enabled` is set to true, it throws ArrayIndexOutOfBoundsException for invalid indices.

Examples:

```
> SELECT elt(1, 'scala', 'java');
 scala
```

Since: 2.0.0

encode

encode(str, charset) - Encodes the first argument using the second argument character set.

Examples:

```
> SELECT encode('abc', 'utf-8');
 abc
```

Since: 1.5.0

endswith

endswith(left, right) - Returns a boolean. The value is True if left ends with right. Returns NULL if either input expression is NULL. Otherwise, returns False. Both left or right must be of STRING or BINARY type.

Examples:

```
> SELECT endswith('Spark SQL', 'SQL');
 true
> SELECT endswith('Spark SQL', 'Spark');
 false
> SELECT endswith('Spark SQL', null);
 NULL
> SELECT endswith(x'537061726b2053514c', x'537061726b');
 false
> SELECT endswith(x'537061726b2053514c', x'53514c');
 true
```

Since: 3.3.0

every

every(expr) - Returns true if all values of `expr` are true.

Examples:

```
> SELECT every(col) FROM VALUES (true), (true), (true) AS tab(col);
 true
> SELECT every(col) FROM VALUES (NULL), (true), (true) AS tab(col);
 true
> SELECT every(col) FROM VALUES (true), (false), (true) AS tab(col);
 false
```

Since: 3.0.0

exists

exists(expr, pred) - Tests whether a predicate holds for one or more elements in the array.

Examples:

```
> SELECT exists(array(1, 2, 3), x -> x % 2 == 0);
 true
> SELECT exists(array(1, 2, 3), x -> x % 2 == 10);
 false
> SELECT exists(array(1, null, 3), x -> x % 2 == 0);
 NULL
> SELECT exists(array(0, null, 2, 3, null), x -> x IS NULL);
 true
> SELECT exists(array(1, 2, 3), x -> x IS NULL);
 false
```

Since: 2.4.0

exp

exp(expr) - Returns e to the power of `expr`.

Examples:

```
> SELECT exp(0);
 1.0
```

Since: 1.4.0

explode

explode(expr) - Separates the elements of array `expr` into multiple rows, or the elements of map `expr` into multiple rows and columns. Unless specified otherwise, uses the default column name `col` for elements of the array or `key` and `value` for the elements of the map.

Examples:

```
> SELECT explode(array(10, 20));
 10
 20
```

Since: 1.0.0

explode_outer

explode_outer(expr) - Separates the elements of array `expr` into multiple rows, or the elements of map `expr` into multiple rows and columns. Unless specified otherwise, uses the default column name `col` for elements of the array or `key` and `value` for the elements of the map.

Examples:

```
> SELECT explode_outer(array(10, 20));
 10
 20
```

Since: 1.0.0

expm1

expm1(expr) - Returns exp(`expr`) - 1.

Examples:

```
> SELECT expm1(0);
 0.0
```

Since: 1.4.0

extract

extract(field FROM source) - Extracts a part of the date/timestamp or interval source.

Arguments:

- field - selects which part of the source should be extracted
  - Supported string values of `field` for dates and timestamps are(case insensitive):
    - "YEAR", ("Y", "YEARS", "YR", "YRS") - the year field
    - "YEAROFWEEK" - the ISO 8601 week-numbering year that the datetime falls in. For example, 2005-01-02 is part of the 53rd week of year 2004, so the result is 2004
    - "QUARTER", ("QTR") - the quarter (1 - 4) of the year that the datetime falls in
    - "MONTH", ("MON", "MONS", "MONTHS") - the month field (1 - 12)
    - "WEEK", ("W", "WEEKS") - the number of the ISO 8601 week-of-week-based-year. A week is considered to start on a Monday and week 1 is the first week with >3 days. 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, 2005-01-02 is part of the 53rd week of year 2004, while 2012-12-31 is part of the first week of 2013
    - "DAY", ("D", "DAYS") - the day of the month field (1 - 31)
    - "DAYOFWEEK",("DOW") - the day of the week for datetime as Sunday(1) to Saturday(7)

        - "DAYOFWEEK_ISO",("DOW_ISO") - ISO 8601 based day of the week for datetime as Monday(1) to Sunday(7)
        - "DOY" - the day of the year (1 - 365/366)
        - "HOUR", ("H", "HOURS", "HR", "HRS") - The hour field (0 - 23)
        - "MINUTE", ("M", "MIN", "MINS", "MINUTES") - the minutes field (0 - 59)
        - "SECOND", ("S", "SEC", "SECONDS", "SECS") - the seconds field, including fractional parts
    - Supported string values of `field` for interval(which consists of `months`, `days`, `microseconds`) are(case insensitive):
        - "YEAR", ("Y", "YEARS", "YR", "YRS") - the total `months` / 12
        - "MONTH", ("MON", "MONS", "MONTHS") - the total `months` % 12
        - "DAY", ("D", "DAYS") - the `days` part of interval
        - "HOUR", ("H", "HOURS", "HR", "HRS") - how many hours the `microseconds` contains
        - "MINUTE", ("M", "MIN", "MINS", "MINUTES") - how many minutes left after taking hours from `microseconds`
        - "SECOND", ("S", "SEC", "SECONDS", "SECS") - how many second with fractions left after taking hours and minutes from `microseconds`
- source - a date/timestamp or interval column from where `field` should be extracted

Examples:

```
> SELECT extract(YEAR FROM TIMESTAMP '2019-08-12 01:00:00.123456');
 2019
> SELECT extract(week FROM timestamp'2019-08-12 01:00:00.123456');
 33
> SELECT extract(doy FROM DATE'2019-08-12');
 224
> SELECT extract(SECONDS FROM timestamp'2019-10-01 00:00:01.000001');
 1.000001
> SELECT extract(days FROM interval 5 days 3 hours 7 minutes);
 5
> SELECT extract(seconds FROM interval 5 hours 30 seconds 1 milliseconds 1 microseconds);
 30.001001
> SELECT extract(MONTH FROM INTERVAL '2021-11' YEAR TO MONTH);
 11
> SELECT extract(MINUTE FROM INTERVAL '123 23:55:59.002001' DAY TO SECOND);
 55
```

Note:

The extract function is equivalent to `date_part(field, source)`.

Since: 3.0.0

factorial

factorial(expr) - Returns the factorial of `expr`. `expr` is [0..20]. Otherwise, null.

Examples:

```
> SELECT factorial(5);
 120
```

Since: 1.5.0

filter

filter(expr, func) - Filters the input array using the given predicate.

Examples:

```
> SELECT filter(array(1, 2, 3), x -> x % 2 == 1);
 [1,3]
> SELECT filter(array(0, 2, 3), (x, i) -> x > i);
 [2,3]
> SELECT filter(array(0, null, 2, 3, null), x -> x IS NOT NULL);
 [0,2,3]
```

Note:

The inner function may use the index argument since 3.0.0.

Since: 2.4.0

find_in_set

find_in_set(str, str_array) - Returns the index (1-based) of the given string (`str`) in the comma-delimited list (`str_array`). Returns 0, if the string was not found or if the given string (`str`) contains a comma.

Examples:

```
> SELECT find_in_set('ab','abc,b,ab,c,def');
 3
```

Since: 1.5.0

first

first(expr[, isIgnoreNull]) - Returns the first value of `expr` for a group of rows. If `isIgnoreNull` is true, returns only non-null values.

Examples:

```
> SELECT first(col) FROM VALUES (10), (5), (20) AS tab(col);
 10
> SELECT first(col) FROM VALUES (NULL), (5), (20) AS tab(col);
 NULL
> SELECT first(col, true) FROM VALUES (NULL), (5), (20) AS tab(col);
 5
```

Note:

The function is non-deterministic because its results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

first_value

first_value(expr[, isIgnoreNull]) - Returns the first value of `expr` for a group of rows. If `isIgnoreNull` is true, returns only non-null values.

Examples:

```
> SELECT first_value(col) FROM VALUES (10), (5), (20) AS tab(col);
 10
> SELECT first_value(col) FROM VALUES (NULL), (5), (20) AS tab(col);
 NULL
> SELECT first_value(col, true) FROM VALUES (NULL), (5), (20) AS tab(col);
 5
```

Note:

The function is non-deterministic because its results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

flatten

flatten(arrayOfArrays) - Transforms an array of arrays into a single array.

Examples:

```
> SELECT flatten(array(array(1, 2), array(3, 4)));
 [1,2,3,4]
```

Since: 2.4.0

float

float(expr) - Casts the value `expr` to the target data type `float`.

Since: 2.0.1

floor

floor(expr[, scale]) - Returns the largest number after rounding down that is not greater than `expr`. An optional `scale` parameter can be specified to control the rounding behavior.

Examples:

```
> SELECT floor(-0.1);
 -1
> SELECT floor(5);
 5
> SELECT floor(3.1411, 3);
 3.141
> SELECT floor(3.1411, -3);
 0
```

Since: 3.3.0

forall

forall(expr, pred) - Tests whether a predicate holds for all elements in the array.

Examples:

```
> SELECT forall(array(1, 2, 3), x -> x % 2 == 0);
 false
> SELECT forall(array(2, 4, 8), x -> x % 2 == 0);
 true
> SELECT forall(array(1, null, 3), x -> x % 2 == 0);
 false
> SELECT forall(array(2, null, 8), x -> x % 2 == 0);
 NULL
```

Since: 3.0.0

format_number

format_number(expr1, expr2) - Formats the number `expr1` like '#,###,###.##', rounded to `expr2` decimal places. If `expr2` is 0, the result has no decimal point or fractional part. `expr2` also accept a user specified format. This is supposed to function like MySQL's FORMAT.

Examples:

```
> SELECT format_number(12332.123456, 4);
 12,332.1235
> SELECT format_number(12332.123456, '##################.###');
 12332.123
```

Since: 1.5.0

format_string

format_string(strfmt, obj, ...) - Returns a formatted string from printf-style format strings.

Examples:

```
> SELECT format_string("Hello World %d %s", 100, "days");
 Hello World 100 days
```

Since: 1.5.0

from_csv

from_csv(csvStr, schema[, options]) - Returns a struct value with the given `csvStr` and `schema`.

Examples:

```
> SELECT from_csv('1, 0.8', 'a INT, b DOUBLE');
 {"a":1,"b":0.8}
> SELECT from_csv('26/08/2015', 'time Timestamp', map('timestampFormat', 'dd/MM/yyyy'));
 {"time":2015-08-26 00:00:00}
```

Since: 3.0.0

from_json

from_json(jsonStr, schema[, options]) - Returns a struct value with the given `jsonStr` and `schema`.

Examples:

```
> SELECT from_json('{"a":1, "b":0.8}', 'a INT, b DOUBLE');
 {"a":1,"b":0.8}
> SELECT from_json('{"time":"26/08/2015"}', 'time Timestamp', map('timestampFormat', 'dd/MM/
 {"time":2015-08-26 00:00:00}
> SELECT from_json('{"teacher": "Alice", "student": [{"name": "Bob", "rank": 1}, {"name": "C
 {"teacher":"Alice","student":[{"name":"Bob","rank":1},{"name":"Charlie","rank":2}]}
```

Since: 2.2.0

from_unixtime

from_unixtime(unix_time[, fmt]) - Returns `unix_time` in the specified `fmt`.

Arguments:

- unix_time - UNIX Timestamp to be converted to the provided format.
- fmt - Date/time format pattern to follow. See Datetime Patterns for valid date and time format patterns. The 'yyyy-MM-dd HH:mm:ss' pattern is used if omitted.

Examples:

```
> SELECT from_unixtime(0, 'yyyy-MM-dd HH:mm:ss');
 1969-12-31 16:00:00

> SELECT from_unixtime(0);
 1969-12-31 16:00:00
```

Since: 1.5.0

from_utc_timestamp

from_utc_timestamp(timestamp, timezone) - Given a timestamp like '2017-07-14 02:40:00.0', interprets it as a time in UTC, and renders that time as a timestamp in the given time zone. For example, 'GMT+1' would yield '2017-07-14 03:40:00.0'.

Examples:

```
> SELECT from_utc_timestamp('2016-08-31', 'Asia/Seoul');
 2016-08-31 09:00:00
```

Since: 1.5.0

get_json_object

get_json_object(json_txt, path) - Extracts a json object from `path`.

Examples:

```
> SELECT get_json_object('{"a":"b"}', '$.a');
 b
```

Since: 1.5.0

getbit

getbit(expr, pos) - Returns the value of the bit (0 or 1) at the specified position. The positions are numbered from right to left, starting at zero. The position argument cannot be negative.

Examples:

```
> SELECT getbit(11, 0);
 1
> SELECT getbit(11, 2);
 0
```

Since: 3.2.0

greatest

greatest(expr, ...) - Returns the greatest value of all parameters, skipping null values.

Examples:

```
> SELECT greatest(10, 9, 2, 4, 3);
 10
```

Since: 1.5.0

grouping

grouping(col) - indicates whether a specified column in a GROUP BY is aggregated or not, returns 1 for aggregated or 0 for not aggregated in the result set.",

Examples:

```
> SELECT name, grouping(name), sum(age) FROM VALUES (2, 'Alice'), (5, 'Bob') people(age, nam
  Alice 0   2
  Bob   0   5
  NULL  1   7
```

Since: 2.0.0

grouping_id

grouping_id([col1[, col2 ..]]) - returns the level of grouping, equals to `(grouping(c1) << (n-1)) + (grouping(c2) << (n-2)) + ... + grouping(cn)`

Examples:

```
> SELECT name, grouping_id(), sum(age), avg(height) FROM VALUES (2, 'Alice', 165), (5, 'Bob'
  Alice 0   2   165.0
  Alice 1   2   165.0
  NULL  3   7   172.5
  Bob   0   5   180.0
  Bob   1   5   180.0
  NULL  2   2   165.0
  NULL  2   5   180.0
```

Note:

Input columns should match with grouping columns exactly, or empty (means all the grouping columns).

Since: 2.0.0

hash

hash(expr1, expr2, ...) - Returns a hash value of the arguments.

Examples:

```
> SELECT hash('Spark', array(123), 2);
 -1321691492
```

Since: 2.0.0

hex

hex(expr) - Converts `expr` to hexadecimal.

Examples:

```
> SELECT hex(17);
 11
> SELECT hex('Spark SQL');
 537061726B2053514C
```

Since: 1.5.0

histogram_numeric

histogram_numeric(expr, nb) - Computes a histogram on numeric 'expr' using nb bins. The return value is an array of (x,y) pairs representing the centers of the histogram's bins. As the value of 'nb' is increased, the histogram approximation gets finer-grained, but may yield artifacts around outliers. In practice, 20-40 histogram bins appear to work well, with more bins being required for skewed or smaller datasets. Note that this function creates a histogram with non-uniform bin widths. It offers no guarantees in terms of the mean-squared-error of the histogram, but in practice is comparable to the histograms produced by the R/S-Plus statistical computing packages. Note: the output type of the 'x' field in the return value is propagated from the input value consumed in the aggregate function.

Examples:

```
> SELECT histogram_numeric(col, 5) FROM VALUES (0), (1), (2), (10) AS tab(col);
 [{"x":0,"y":1.0},{"x":1,"y":1.0},{"x":2,"y":1.0},{"x":10,"y":1.0}]
```

Since: 3.3.0

hour

hour(timestamp) - Returns the hour component of the string/timestamp.

Examples:

```
> SELECT hour('2009-07-30 12:58:59');
 12
```

Since: 1.5.0

hypot

hypot(expr1, expr2) - Returns sqrt(`expr1`**2 +** `expr2`2).

Examples:

```
> SELECT hypot(3, 4);
 5.0
```

Since: 1.4.0

if

if(expr1, expr2, expr3) - If `expr1` evaluates to true, then returns `expr2`; otherwise returns `expr3`.

Examples:

```
> SELECT if(1 < 2, 'a', 'b');
 a
```

Since: 1.0.0

ifnull

ifnull(expr1, expr2) - Returns `expr2` if `expr1` is null, or `expr1` otherwise.

Examples:

```
> SELECT ifnull(NULL, array('2'));
 ["2"]
```

Since: 2.0.0

ilike

str ilike pattern[ESCAPE escape] - Returns true if str matches `pattern` with `escape` case-insensitively, null if any arguments are null, false otherwise.

Arguments:

- str - a string expression
- pattern - a string expression. The pattern is a string which is matched literally and case-insensitively, with exception to the following special symbols:

  _ matches any one character in the input (similar to . in posix regular expressions)

% matches zero or more characters in the input (similar to .* in posix regular expressions)

Since Spark 2.0, string literals are unescaped in our SQL parser. For example, in order to match "\abc", the pattern should be "\abc".

When SQL config 'spark.sql.parser.escapedStringLiterals' is enabled, it falls back to Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the pattern to match "\abc" should be "\abc". * escape - an character added since Spark 3.0. The default escape character is the '\'. If an escape character precedes a special symbol or another escape character, the following character is matched literally. It is invalid to escape any other character.

Examples:

```
> SELECT ilike('Spark', '_Park');
true
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT '%SystemDrive%\Users\John' ilike '\%SystemDrive\%\\users%';
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT '%SystemDrive%\\USERS\\John' ilike '\%SystemDrive\%\\\\Users%';
true
> SELECT '%SystemDrive%/Users/John' ilike '/%SYSTEMDrive/%//Users%' ESCAPE '/';
true
```

Note:

Use RLIKE to match with standard regular expressions.

Since: 3.3.0

in

expr1 in(expr2, expr3, ...) - Returns true if `expr` equals to any valN.

Arguments:

- expr1, expr2, expr3, ... - the arguments must be same type.

Examples:

```
> SELECT 1 in(1, 2, 3);
 true
> SELECT 1 in(2, 3, 4);
 false
> SELECT named_struct('a', 1, 'b', 2) in(named_struct('a', 1, 'b', 1), named_struct('a', 1,
 false
> SELECT named_struct('a', 1, 'b', 2) in(named_struct('a', 1, 'b', 2), named_struct('a', 1,
 true
```

Since: 1.0.0

initcap

initcap(str) - Returns `str` with the first letter of each word in uppercase. All other letters are in lowercase. Words are delimited by white space.

Examples:

```
> SELECT initcap('sPark sql');
 Spark Sql
```

Since: 1.5.0

inline

inline(expr) - Explodes an array of structs into a table. Uses column names col1, col2, etc. by default unless specified otherwise.

Examples:

```
> SELECT inline(array(struct(1, 'a'), struct(2, 'b')));
 1  a
 2  b
```

Since: 2.0.0

inline_outer

inline_outer(expr) - Explodes an array of structs into a table. Uses column names col1, col2, etc. by default unless specified otherwise.

Examples:

```
> SELECT inline_outer(array(struct(1, 'a'), struct(2, 'b')));
 1  a
 2  b
```

Since: 2.0.0

input_file_block_length

input_file_block_length() - Returns the length of the block being read, or -1 if not available.

Examples:

```
> SELECT input_file_block_length();
 -1
```

Since: 2.2.0

input_file_block_start

input_file_block_start() - Returns the start offset of the block being read, or -1 if not available.

Examples:

```
> SELECT input_file_block_start();
 -1
```

Since: 2.2.0

input_file_name

input_file_name() - Returns the name of the file being read, or empty string if not available.

Examples:

```
> SELECT input_file_name();
```

Since: 1.5.0

instr

instr(str, substr) - Returns the (1-based) index of the first occurrence of `substr` in `str`.

Examples:

```
> SELECT instr('SparkSQL', 'SQL');
 6
```

Since: 1.5.0

int

int(expr) - Casts the value `expr` to the target data type `int`.

Since: 2.0.1

isnan

isnan(expr) - Returns true if `expr` is NaN, or false otherwise.

Examples:

```
> SELECT isnan(cast('NaN' as double));
 true
```

Since: 1.5.0

isnotnull

isnotnull(expr) - Returns true if `expr` is not null, or false otherwise.

Examples:

```
> SELECT isnotnull(1);
 true
```

Since: 1.0.0

isnull

isnull(expr) - Returns true if `expr` is null, or false otherwise.

Examples:

```
> SELECT isnull(1);
 false
```

Since: 1.0.0

java_method

java_method(class, method[, arg1[, arg2 ..]]) - Calls a method with reflection.

Examples:

```
> SELECT java_method('java.util.UUID', 'randomUUID');
 c33fb387-8500-4bfa-81d2-6e0e3e930df2
> SELECT java_method('java.util.UUID', 'fromString', 'a5cf6c42-0c85-418f-af6c-3e4e5b1328f2')
 a5cf6c42-0c85-418f-af6c-3e4e5b1328f2
```

Since: 2.0.0

json_array_length

json_array_length(jsonArray) - Returns the number of elements in the outermost JSON array.

Arguments:

- jsonArray - A JSON array. `NULL` is returned in case of any other valid JSON string, `NULL` or an invalid JSON.

Examples:

```
> SELECT json_array_length('[1,2,3,4]');
  4
> SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');
  5
> SELECT json_array_length('[1,2');
  NULL
```

Since: 3.1.0

json_object_keys

json_object_keys(json_object) - Returns all the keys of the outermost JSON object as an array.

Arguments:

- json_object - A JSON object. If a valid JSON object is given, all the keys of the outermost object will be returned as an array. If it is any other valid JSON string, an invalid JSON string or an empty string, the function returns null.

Examples:

```
> SELECT json_object_keys('{}');
  []
> SELECT json_object_keys('{"key": "value"}');
  ["key"]
> SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}');
  ["f1","f2"]
```

Since: 3.1.0

json_tuple

json_tuple(jsonStr, p1, p2, ..., pn) - Returns a tuple like the function get_json_object, but it takes multiple names. All the input parameters and output column types are string.

Examples:

```
> SELECT json_tuple('{"a":1, "b":2}', 'a', 'b');
 1  2
```

Since: 1.6.0

kurtosis

kurtosis(expr) - Returns the kurtosis value calculated from values of a group.

Examples:

```
> SELECT kurtosis(col) FROM VALUES (-10), (-20), (100), (1000) AS tab(col);
 -0.7014368047529627
> SELECT kurtosis(col) FROM VALUES (1), (10), (100), (10), (1) as tab(col);
 0.19432323191699075
```

Since: 1.6.0

lag

lag(input[, offset[, default]]) - Returns the value of `input` at the `offset`th row before the current row in the window. The default value of `offset` is 1 and the default value of `default` is null. If the value of `input` at the `offset`th row is null, null is returned. If there is no such offset row (e.g., when the offset is 1, the first row of the window does not have any previous row), `default` is returned.

Arguments:

- input - a string expression to evaluate `offset` rows before the current row.
- offset - an int expression which is rows to jump back in the partition.
- default - a string expression which is to use when the offset row does not exist.

Examples:

```
> SELECT a, b, lag(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1), ('A
 A1 1   NULL
 A1 1   1
 A1 2   1
 A2 3   NULL
```

Since: 2.0.0

last

last(expr[, isIgnoreNull]) - Returns the last value of `expr` for a group of rows. If `isIgnoreNull` is true, returns only non-null values

Examples:

```
> SELECT last(col) FROM VALUES (10), (5), (20) AS tab(col);
 20
> SELECT last(col) FROM VALUES (10), (5), (NULL) AS tab(col);
 NULL
> SELECT last(col, true) FROM VALUES (10), (5), (NULL) AS tab(col);
 5
```

Note:

The function is non-deterministic because its results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

last_day

last_day(date) - Returns the last day of the month which the date belongs to.

Examples:

```
> SELECT last_day('2009-01-12');
 2009-01-31
```

Since: 1.5.0

last_value

last_value(expr[, isIgnoreNull]) - Returns the last value of `expr` for a group of rows. If `isIgnoreNull` is true, returns only non-null values

Examples:

```
> SELECT last_value(col) FROM VALUES (10), (5), (20) AS tab(col);
 20
> SELECT last_value(col) FROM VALUES (10), (5), (NULL) AS tab(col);
 NULL
> SELECT last_value(col, true) FROM VALUES (10), (5), (NULL) AS tab(col);
 5
```

Note:

The function is non-deterministic because its results depends on the order of the rows which may be non-deterministic after a shuffle.

Since: 2.0.0

lcase

lcase(str) - Returns `str` with all characters changed to lowercase.

Examples:

```
> SELECT lcase('SparkSql');
 sparksql
```

Since: 1.0.1

lead

lead(input[, offset[, default]]) - Returns the value of `input` at the `offset`th row after the current row in the window. The default value of `offset` is 1 and the default value of `default` is null. If the value of `input` at the `offset`th row is null, null is returned. If there is no such an offset row (e.g., when the offset is 1, the last row of the window does not have any subsequent row), `default` is returned.

Arguments:

- input - a string expression to evaluate `offset` rows after the current row.
- offset - an int expression which is rows to jump ahead in the partition.
- default - a string expression which is to use when the offset is larger than the window. The default value is null.

Examples:

```
> SELECT a, b, lead(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1), ('
 A1 1   1
 A1 1   2
 A1 2   NULL
 A2 3   NULL
```

Since: 2.0.0

least

least(expr, ...) - Returns the least value of all parameters, skipping null values.

Examples:

```
> SELECT least(10, 9, 2, 4, 3);
 2
```

Since: 1.5.0

left

left(str, len) - Returns the leftmost `len`(`len` can be string type) characters from the string `str`,if `len` is less or equal than 0 the result is an empty string.

Examples:

```
> SELECT left('Spark SQL', 3);
 Spa
```

Since: 2.3.0

length

length(expr) - Returns the character length of string data or number of bytes of binary data. The length of string data includes the trailing spaces. The length of binary data includes binary zeros.

Examples:

```
> SELECT length('Spark SQL ');
 10
> SELECT CHAR_LENGTH('Spark SQL ');
 10
> SELECT CHARACTER_LENGTH('Spark SQL ');
 10
```

Since: 1.5.0

levenshtein

levenshtein(str1, str2) - Returns the Levenshtein distance between the two given strings.

Examples:

```
> SELECT levenshtein('kitten', 'sitting');
 3
```

Since: 1.5.0

like

str like pattern[ESCAPE escape] - Returns true if str matches `pattern` with `escape`, null if any arguments are null, false otherwise.

Arguments:

- str - a string expression
- pattern - a string expression. The pattern is a string which is matched literally, with exception to the following special symbols:

  _ matches any one character in the input (similar to . in posix regular expressions)

% matches zero or more characters in the input (similar to .* in posix regular expressions)

Since Spark 2.0, string literals are unescaped in our SQL parser. For example, in order to match "\abc", the pattern should be "\abc".

When SQL config 'spark.sql.parser.escapedStringLiterals' is enabled, it falls back to Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the pattern to match "\abc" should be "\abc". * escape - an character added since Spark 3.0. The default escape character is the '\'. If an escape character precedes a special symbol or another escape character, the following character is matched literally. It is invalid to escape any other character.

Examples:

```
> SELECT like('Spark', '_park');
true
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT '%SystemDrive%\Users\John' like '\%SystemDrive\%\\Users%';
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT '%SystemDrive%\\Users\\John' like '\%SystemDrive\%\\\\Users%';
true
> SELECT '%SystemDrive%/Users/John' like '/%SystemDrive/%//Users%' ESCAPE '/';
true
```

Note:

Use RLIKE to match with standard regular expressions.

Since: 1.0.0

ln

ln(expr) - Returns the natural logarithm (base e) of `expr`.

Examples:

```
> SELECT ln(1);
 0.0
```

Since: 1.4.0

locate

locate(substr, str[, pos]) - Returns the position of the first occurrence of `substr` in `str` after position `pos`. The given `pos` and return value are 1-based.

Examples:

```
> SELECT locate('bar', 'foobarbar');
 4
> SELECT locate('bar', 'foobarbar', 5);
 7
> SELECT POSITION('bar' IN 'foobarbar');
 4
```

Since: 1.5.0

log

log(base, expr) - Returns the logarithm of `expr` with `base`.

Examples:

```
> SELECT log(10, 100);
 2.0
```

Since: 1.5.0

log10

log10(expr) - Returns the logarithm of `expr` with base 10.

Examples:

```
> SELECT log10(10);
 1.0
```

Since: 1.4.0

log1p

log1p(expr) - Returns log(1 + `expr`).

Examples:

```
> SELECT log1p(0);
 0.0
```

Since: 1.4.0

log2

log2(expr) - Returns the logarithm of `expr` with base 2.

Examples:

```
> SELECT log2(2);
 1.0
```

Since: 1.4.0

lower

lower(str) - Returns `str` with all characters changed to lowercase.

Examples:

```
> SELECT lower('SparkSql');
 sparksql
```

Since: 1.0.1

lpad

lpad(str, len[, pad]) - Returns `str`, left-padded with `pad` to a length of `len`. If `str` is longer than `len`, the return value is shortened to `len` characters or bytes. If `pad` is not specified, `str` will be padded to the left with space characters if it is a character string, and with zeros if it is a byte sequence.

Examples:

```
> SELECT lpad('hi', 5, '??');
 ???hi
> SELECT lpad('hi', 1, '??');
 h
> SELECT lpad('hi', 5);
    hi
> SELECT hex(lpad(unhex('aabb'), 5));
 000000AABB
> SELECT hex(lpad(unhex('aabb'), 5, unhex('1122')));
 112211AABB
```

Since: 1.5.0

ltrim

ltrim(str) - Removes the leading space characters from `str`.

Arguments:

- str - a string expression
- trimStr - the trim string characters to trim, the default value is a single space

Examples:

```
> SELECT ltrim('    SparkSQL   ');
 SparkSQL
```

Since: 1.5.0

make_date

make_date(year, month, day) - Create date from year, month and day fields. If the configuration `spark.sql.ansi.enabled` is false, the function returns NULL on invalid inputs. Otherwise, it will throw an error instead.

Arguments:

- year - the year to represent, from 1 to 9999
- month - the month-of-year to represent, from 1 (January) to 12 (December)
- day - the day-of-month to represent, from 1 to 31

Examples:

```
> SELECT make_date(2013, 7, 15);
 2013-07-15
> SELECT make_date(2019, 7, NULL);
 NULL
```

Since: 3.0.0

make_dt_interval

make_dt_interval([days[, hours[, mins[, secs]]]]) - Make DayTimeIntervalType duration from days, hours, mins and secs.

Arguments:

- days - the number of days, positive or negative
- hours - the number of hours, positive or negative
- mins - the number of minutes, positive or negative
- secs - the number of seconds with the fractional part in microsecond precision.

Examples:

```
> SELECT make_dt_interval(1, 12, 30, 01.001001);
 1 12:30:01.001001000
> SELECT make_dt_interval(2);
 2 00:00:00.000000000
> SELECT make_dt_interval(100, null, 3);
 NULL
```

Since: 3.2.0

make_interval

make_interval([years[, months[, weeks[, days[, hours[, mins[, secs]]]]]]]) - Make interval from years, months, weeks, days, hours, mins and secs.

Arguments:

- years - the number of years, positive or negative
- months - the number of months, positive or negative
- weeks - the number of weeks, positive or negative
- days - the number of days, positive or negative
- hours - the number of hours, positive or negative
- mins - the number of minutes, positive or negative
- secs - the number of seconds with the fractional part in microsecond precision.

Examples:

```
> SELECT make_interval(100, 11, 1, 1, 12, 30, 01.001001);
 100 years 11 months 8 days 12 hours 30 minutes 1.001001 seconds
> SELECT make_interval(100, null, 3);
 NULL
> SELECT make_interval(0, 1, 0, 1, 0, 0, 100.000001);
 1 months 1 days 1 minutes 40.000001 seconds
```

Since: 3.0.0

make_timestamp

make_timestamp(year, month, day, hour, min, sec[, timezone]) - Create timestamp from year, month, day, hour, min, sec and timezone fields. The result data type is consistent with the value of configuration `spark.sql.timestampType`. If the configuration `spark.sql.ansi.enabled` is false, the function returns NULL on invalid inputs. Otherwise, it will throw an error instead.

Arguments:

- year - the year to represent, from 1 to 9999
- month - the month-of-year to represent, from 1 (January) to 12 (December)
- day - the day-of-month to represent, from 1 to 31
- hour - the hour-of-day to represent, from 0 to 23
- min - the minute-of-hour to represent, from 0 to 59
- sec - the second-of-minute and its micro-fraction to represent, from 0 to 60. The value can be either an integer like 13 , or a fraction like 13.123. If the sec argument equals to 60, the seconds field is set to 0 and 1 minute is added to the final timestamp.
- timezone - the time zone identifier. For example, CET, UTC and etc.

Examples:

```
> SELECT make_timestamp(2014, 12, 28, 6, 30, 45.887);
 2014-12-28 06:30:45.887
> SELECT make_timestamp(2014, 12, 28, 6, 30, 45.887, 'CET');
 2014-12-27 21:30:45.887
> SELECT make_timestamp(2019, 6, 30, 23, 59, 60);
 2019-07-01 00:00:00
> SELECT make_timestamp(2019, 6, 30, 23, 59, 1);
 2019-06-30 23:59:01
> SELECT make_timestamp(null, 7, 22, 15, 30, 0);
 NULL
```

Since: 3.0.0

make_ym_interval

make_ym_interval([years[, months]]) - Make year-month interval from years, months.

Arguments:

- years - the number of years, positive or negative
- months - the number of months, positive or negative

Examples:

```
> SELECT make_ym_interval(1, 2);
 1-2
> SELECT make_ym_interval(1, 0);
 1-0
> SELECT make_ym_interval(-1, 1);
 -0-11
> SELECT make_ym_interval(2);
 2-0
```

Since: 3.2.0

map

map(key0, value0, key1, value1, ...) - Creates a map with the given key/value pairs.

Examples:

```
> SELECT map(1.0, '2', 3.0, '4');
 {1.0:"2",3.0:"4"}
```

Since: 2.0.0

map_concat

map_concat(map, ...) - Returns the union of all the given maps

Examples:

```
> SELECT map_concat(map(1, 'a', 2, 'b'), map(3, 'c'));
 {1:"a",2:"b",3:"c"}
```

Since: 2.4.0

map_contains_key

map_contains_key(map, key) - Returns true if the map contains the key.

Examples:

```
> SELECT map_contains_key(map(1, 'a', 2, 'b'), 1);
 true
> SELECT map_contains_key(map(1, 'a', 2, 'b'), 3);
 false
```

Since: 3.3.0

map_entries

map_entries(map) - Returns an unordered array of all entries in the given map.

Examples:

```
> SELECT map_entries(map(1, 'a', 2, 'b'));
 [{"key":1,"value":"a"},{"key":2,"value":"b"}]
```

Since: 3.0.0

map_filter

map_filter(expr, func) - Filters entries in a map using the function.

Examples:

```
> SELECT map_filter(map(1, 0, 2, 2, 3, -1), (k, v) -> k > v);
 {1:0,3:-1}
```

Since: 3.0.0

map_from_arrays

map_from_arrays(keys, values) - Creates a map with a pair of the given key/value arrays. All elements in keys should not be null

Examples:

```
> SELECT map_from_arrays(array(1.0, 3.0), array('2', '4'));
 {1.0:"2",3.0:"4"}
```

Since: 2.4.0

map_from_entries

map_from_entries(arrayOfEntries) - Returns a map created from the given array of entries.

Examples:

```
> SELECT map_from_entries(array(struct(1, 'a'), struct(2, 'b')));
 {1:"a",2:"b"}
```

Since: 2.4.0

map_keys

map_keys(map) - Returns an unordered array containing the keys of the map.

Examples:

```
> SELECT map_keys(map(1, 'a', 2, 'b'));
 [1,2]
```

Since: 2.0.0

map_values

map_values(map) - Returns an unordered array containing the values of the map.

Examples:

```
> SELECT map_values(map(1, 'a', 2, 'b'));
 ["a","b"]
```

Since: 2.0.0

map_zip_with

map_zip_with(map1, map2, function) - Merges two given maps into a single map by applying function to the pair of values with the same key. For keys only presented in one map, NULL will be passed as the value for the missing key. If an input map contains duplicated keys, only the first entry of the duplicated key is passed into the lambda function.

Examples:

```
> SELECT map_zip_with(map(1, 'a', 2, 'b'), map(1, 'x', 2, 'y'), (k, v1, v2) -> concat(v1, v2
 {1:"ax",2:"by"}
```

Since: 3.0.0

max

max(expr) - Returns the maximum value of `expr`.

Examples:

```
> SELECT max(col) FROM VALUES (10), (50), (20) AS tab(col);
 50
```

Since: 1.0.0

max_by

max_by(x, y) - Returns the value of `x` associated with the maximum value of `y`.

Examples:

```
> SELECT max_by(x, y) FROM VALUES (('a', 10)), (('b', 50)), (('c', 20)) AS tab(x, y);
 b
```

Since: 3.0.0

md5

md5(expr) - Returns an MD5 128-bit checksum as a hex string of `expr`.

Examples:

```
> SELECT md5('Spark');
 8cde774d6f7333752ed72cacddb05126
```

Since: 1.5.0

mean

mean(expr) - Returns the mean calculated from values of a group.

Examples:

```
> SELECT mean(col) FROM VALUES (1), (2), (3) AS tab(col);
 2.0
> SELECT mean(col) FROM VALUES (1), (2), (NULL) AS tab(col);
 1.5
```

Since: 1.0.0

min

min(expr) - Returns the minimum value of `expr`.

Examples:

```
> SELECT min(col) FROM VALUES (10), (-1), (20) AS tab(col);
 -1
```

Since: 1.0.0

min_by

min_by(x, y) - Returns the value of `x` associated with the minimum value of `y`.

Examples:

```
> SELECT min_by(x, y) FROM VALUES (('a', 10)), (('b', 50)), (('c', 20)) AS tab(x, y);
 a
```

Since: 3.0.0

minute

minute(timestamp) - Returns the minute component of the string/timestamp.

Examples:

```
> SELECT minute('2009-07-30 12:58:59');
 58
```

Since: 1.5.0

mod

expr1 mod expr2 - Returns the remainder after `expr1` / `expr2`.

Examples:

```
> SELECT 2 % 1.8;
 0.2
> SELECT MOD(2, 1.8);
 0.2
```

Since: 1.0.0

monotonically_increasing_id

monotonically_increasing_id() - Returns monotonically increasing 64-bit integers. The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive. The current implementation puts the partition ID in the upper 31 bits, and the lower 33 bits represent the record number within each partition. The assumption is that the data frame has less than 1 billion partitions, and each partition has less than 8 billion records. The function is non-deterministic because its result depends on partition IDs.

Examples:

```
> SELECT monotonically_increasing_id();
 0
```

Since: 1.4.0

month

month(date) - Returns the month component of the date/timestamp.

Examples:

```
> SELECT month('2016-07-30');
 7
```

Since: 1.5.0

months_between

months_between(timestamp1, timestamp2[, roundOff]) - If `timestamp1` is later than `timestamp2`, then the result is positive. If `timestamp1` and `timestamp2` are on the same day of month, or both are the last day of month, time of day will be ignored. Otherwise, the difference is calculated based on 31 days per month, and rounded to 8 digits unless roundOff=false.

Examples:

```
> SELECT months_between('1997-02-28 10:30:00', '1996-10-30');
 3.94959677
> SELECT months_between('1997-02-28 10:30:00', '1996-10-30', false);
 3.9495967741935485
```

Since: 1.5.0

named_struct

named_struct(name1, val1, name2, val2, ...) - Creates a struct with the given field names and values.

Examples:

```
> SELECT named_struct("a", 1, "b", 2, "c", 3);
 {"a":1,"b":2,"c":3}
```

Since: 1.5.0

nanvl

nanvl(expr1, expr2) - Returns `expr1` if it's not NaN, or `expr2` otherwise.

Examples:

```
> SELECT nanvl(cast('NaN' as double), 123);
 123.0
```

Since: 1.5.0

negative

negative(expr) - Returns the negated value of `expr`.

Examples:

```
> SELECT negative(1);
 -1
```

Since: 1.0.0

next_day

next_day(start_date, day_of_week) - Returns the first date which is later than `start_date` and named as indicated. The function returns NULL if at least one of the input parameters is NULL. When both of the input parameters are not NULL and day_of_week is an invalid input, the function throws IllegalArgumentException if `spark.sql.ansi.enabled` is set to true, otherwise NULL.

Examples:

```
> SELECT next_day('2015-01-14', 'TU');
 2015-01-20
```

Since: 1.5.0

not

not expr - Logical not.

Examples:

```
> SELECT not true;
 false
> SELECT not false;
 true
> SELECT not NULL;
 NULL
```

Since: 1.0.0

now

now() - Returns the current timestamp at the start of query evaluation.

Examples:

```
> SELECT now();
 2020-04-25 15:49:11.914
```

Since: 1.6.0

nth_value

nth_value(input[, offset]) - Returns the value of `input` at the row that is the `offset`th row from beginning of the window frame. Offset starts at 1. If ignoreNulls=true, we will skip nulls when finding the `offset`th row. Otherwise, every row counts for the `offset`. If there is no such an `offset`th row (e.g., when the offset is 10, size of the window frame is less than 10), null is returned.

Arguments:

- input - the target column or expression that the function operates on.
- offset - a positive int literal to indicate the offset in the window frame. It starts with 1.
- ignoreNulls - an optional specification that indicates the NthValue should skip null values in the determination of which row to use.

Examples:

```
> SELECT a, b, nth_value(b, 2) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1'
 A1 1   1
 A1 1   1
 A1 2   1
 A2 3   NULL
```

Since: 3.1.0

ntile

ntile(n) - Divides the rows for each window partition into `n` buckets ranging from 1 to at most `n`.

Arguments:

- buckets - an int expression which is number of buckets to divide the rows in. Default value is 1.

Examples:

```
> SELECT a, b, ntile(2) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1), (
 A1 1   1
 A1 1   1
 A1 2   2
 A2 3   1
```

Since: 2.0.0

nullif

nullif(expr1, expr2) - Returns null if `expr1` equals to `expr2`, or `expr1` otherwise.

Examples:

```
> SELECT nullif(2, 2);
 NULL
```

Since: 2.0.0

nvl

nvl(expr1, expr2) - Returns `expr2` if `expr1` is null, or `expr1` otherwise.

Examples:

```
> SELECT nvl(NULL, array('2'));
 ["2"]
```

Since: 2.0.0

nvl2

nvl2(expr1, expr2, expr3) - Returns `expr2` if `expr1` is not null, or `expr3` otherwise.

Examples:

```
> SELECT nvl2(NULL, 2, 1);
 1
```

Since: 2.0.0

octet_length

octet_length(expr) - Returns the byte length of string data or number of bytes of binary data.

Examples:

```
> SELECT octet_length('Spark SQL');
 9
```

Since: 2.3.0

or

expr1 or expr2 - Logical OR.

Examples:

```
> SELECT true or false;
 true
> SELECT false or false;
 false
> SELECT true or NULL;
 true
> SELECT false or NULL;
 NULL
```

Since: 1.0.0

overlay

overlay(input, replace, pos[, len]) - Replace `input` with `replace` that starts at `pos` and is of length `len`.

Examples:

```
> SELECT overlay('Spark SQL' PLACING '_' FROM 6);
 Spark_SQL
> SELECT overlay('Spark SQL' PLACING 'CORE' FROM 7);
 Spark CORE
> SELECT overlay('Spark SQL' PLACING 'ANSI ' FROM 7 FOR 0);
 Spark ANSI SQL
> SELECT overlay('Spark SQL' PLACING 'tructured' FROM 2 FOR 4);
 Structured SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('_', 'utf-8') FROM 6);
 Spark_SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('CORE', 'utf-8') FROM 7);
 Spark CORE
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('ANSI ', 'utf-8') FROM 7 FOR 0)
 Spark ANSI SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('tructured', 'utf-8') FROM 2 FO
 Structured SQL
```

Since: 3.0.0

parse_url

parse_url(url, partToExtract[, key]) - Extracts a part from a URL.

Examples:

```
> SELECT parse_url('http://spark.apache.org/path?query=1', 'HOST');
 spark.apache.org
> SELECT parse_url('http://spark.apache.org/path?query=1', 'QUERY');
 query=1
> SELECT parse_url('http://spark.apache.org/path?query=1', 'QUERY', 'query');
 1
```

Since: 2.0.0

percent_rank

percent_rank() - Computes the percentage ranking of a value in a group of values.

Arguments:

- children - this is to base the rank on; a change in the value of one the children will trigger a change in rank. This is an internal parameter and will be assigned by the Analyser.

Examples:

```
> SELECT a, b, percent_rank(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1'
 A1 1   0.0
 A1 1   0.0
 A1 2   1.0
 A2 3   0.0
```

Since: 2.0.0

percentile

percentile(col, percentage [, frequency]) - Returns the exact percentile value of numeric column `col` at the given percentage. The value of percentage must be between 0.0 and 1.0. The value of frequency should be positive integral percentile(col, array(percentage1 [, percentage2]...) [, frequency]) - Returns the exact percentile value array of numeric column `col` at the given percentage(s). Each value of the percentage array must be between 0.0 and 1.0. The value of frequency should be positive integral

Examples:

```
> SELECT percentile(col, 0.3) FROM VALUES (0), (10) AS tab(col);
 3.0
> SELECT percentile(col, array(0.25, 0.75)) FROM VALUES (0), (10) AS tab(col);
 [2.5,7.5]
```

Since: 2.1.0

percentile_approx

percentile_approx(col, percentage [, accuracy]) - Returns the approximate `percentile` of the numeric or ansi interval column `col` which is the smallest value in the ordered `col` values (sorted from least to greatest) such that no more than `percentage` of `col` values is less than the value or equal to that value. The value of percentage must be between 0.0 and 1.0. The `accuracy` parameter (default: 10000) is a positive numeric literal which controls approximation accuracy at the cost of memory. Higher value of `accuracy` yields better accuracy, `1.0/accuracy` is the relative error of the approximation. When `percentage` is an array, each value of the percentage array must be between 0.0 and 1.0. In this case, returns the approximate percentile array of column `col` at the given percentage array.

Examples:

```
> SELECT percentile_approx(col, array(0.5, 0.4, 0.1), 100) FROM VALUES (0), (1), (2), (10) A
 [1,1,0]
> SELECT percentile_approx(col, 0.5, 100) FROM VALUES (0), (6), (7), (9), (10) AS tab(col);
 7
> SELECT percentile_approx(col, 0.5, 100) FROM VALUES (INTERVAL '0' MONTH), (INTERVAL '1' MO
 0-1
> SELECT percentile_approx(col, array(0.5, 0.7), 100) FROM VALUES (INTERVAL '0' SECOND), (IN
 [0 00:00:01.000000000,0 00:00:02.000000000]
```

Since: 2.1.0

pi

pi() - Returns pi.

Examples:

```
> SELECT pi();
 3.141592653589793
```

Since: 1.5.0

pmod

pmod(expr1, expr2) - Returns the positive value of `expr1` mod `expr2`.

Examples:

```
> SELECT pmod(10, 3);
 1
> SELECT pmod(-10, 3);
 2
```

Since: 1.5.0

posexplode

posexplode(expr) - Separates the elements of array `expr` into multiple rows with positions, or the elements of map `expr` into multiple rows and columns with positions. Unless specified otherwise, uses the column name `pos` for position, `col` for elements of the array or `key` and `value` for elements of the map.

Examples:

```
> SELECT posexplode(array(10,20));
 0  10
 1  20
```

Since: 2.0.0

posexplode_outer

posexplode_outer(expr) - Separates the elements of array `expr` into multiple rows with positions, or the elements of map `expr` into multiple rows and columns with positions. Unless specified otherwise, uses the column name `pos` for position, `col` for elements of the array or `key` and `value` for elements of the map.

Examples:

```
> SELECT posexplode_outer(array(10,20));
 0  10
 1  20
```

Since: 2.0.0

position

position(substr, str[, pos]) - Returns the position of the first occurrence of `substr` in `str` after position `pos`. The given `pos` and return value are 1-based.

Examples:

```
> SELECT position('bar', 'foobarbar');
 4
> SELECT position('bar', 'foobarbar', 5);
 7
> SELECT POSITION('bar' IN 'foobarbar');
 4
```

Since: 1.5.0

positive

positive(expr) - Returns the value of `expr`.

Examples:

```
> SELECT positive(1);
 1
```

Since: 1.5.0

pow

pow(expr1, expr2) - Raises `expr1` to the power of `expr2`.

Examples:

```
> SELECT pow(2, 3);
 8.0
```

Since: 1.4.0

power

power(expr1, expr2) - Raises `expr1` to the power of `expr2`.

Examples:

```
> SELECT power(2, 3);
 8.0
```

Since: 1.4.0

printf

printf(strfmt, obj, ...) - Returns a formatted string from printf-style format strings.

Examples:

```
> SELECT printf("Hello World %d %s", 100, "days");
 Hello World 100 days
```

Since: 1.5.0

quarter

quarter(date) - Returns the quarter of the year for date, in the range 1 to 4.

Examples:

```
> SELECT quarter('2016-08-31');
 3
```

Since: 1.5.0

radians

radians(expr) - Converts degrees to radians.

Arguments:

- expr - angle in degrees

Examples:

```
> SELECT radians(180);
 3.141592653589793
```

Since: 1.4.0

raise_error

raise_error(expr) - Throws an exception with `expr`.

Examples:

```
> SELECT raise_error('custom error message');
 java.lang.RuntimeException
 custom error message
```

Since: 3.1.0

rand

rand([seed]) - Returns a random value with independent and identically distributed (i.i.d.) uniformly distributed values in [0, 1).

Examples:

```
> SELECT rand();
 0.9629742951434543
> SELECT rand(0);
 0.7604953758285915
> SELECT rand(null);
 0.7604953758285915
```

Note:

The function is non-deterministic in general case.

Since: 1.5.0

randn

randn([seed]) - Returns a random value with independent and identically distributed (i.i.d.) values drawn from the standard normal distribution.

Examples:

```
> SELECT randn();
 -0.3254147983080288
> SELECT randn(0);
 1.6034991609278433
> SELECT randn(null);
 1.6034991609278433
```

Note:

The function is non-deterministic in general case.

Since: 1.5.0

random

random([seed]) - Returns a random value with independent and identically distributed (i.i.d.) uniformly distributed values in [0, 1).

Examples:

```
> SELECT random();
 0.9629742951434543
> SELECT random(0);
 0.7604953758285915
> SELECT random(null);
 0.7604953758285915
```

Note:

The function is non-deterministic in general case.

Since: 1.5.0

rank

rank() - Computes the rank of a value in a group of values. The result is one plus the number of rows preceding or equal to the current row in the ordering of the partition. The values will produce gaps in the sequence.

Arguments:

- children - this is to base the rank on; a change in the value of one the children will trigger a change in rank. This is an internal parameter and will be assigned by the Analyser.

Examples:

```
> SELECT a, b, rank(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1), ('
 A1 1   1
 A1 1   1
 A1 2   3
 A2 3   1
```

Since: 2.0.0

reflect

reflect(class, method[, arg1[, arg2 ..]]) - Calls a method with reflection.

Examples:

```
> SELECT reflect('java.util.UUID', 'randomUUID');
 c33fb387-8500-4bfa-81d2-6e0e3e930df2
> SELECT reflect('java.util.UUID', 'fromString', 'a5cf6c42-0c85-418f-af6c-3e4e5b1328f2');
 a5cf6c42-0c85-418f-af6c-3e4e5b1328f2
```

Since: 2.0.0

regexp

regexp(str, regexp) - Returns true if `str` matches `regexp`, or false otherwise.

Arguments:

- str - a string expression
- regexp - a string expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

  There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$".

Examples:

```
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT regexp('%SystemDrive%\Users\John', '%SystemDrive%\\Users.*');
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT regexp('%SystemDrive%\\Users\\John', '%SystemDrive%\\\\Users.*');
true
```

Note:

Use LIKE to match with simple string pattern.

Since: 3.2.0

regexp_extract

regexp_extract(str, regexp[, idx]) - Extract the first string in the `str` that match the `regexp` expression and corresponding to the regex group index.

Arguments:

- str - a string expression.
- regexp - a string representing a regular expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

  There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$". * idx - an integer expression that representing the group index. The regex maybe contains multiple groups. `idx` indicates which regex group to extract. The group index should be non-negative. The minimum value of `idx` is 0, which means matching the entire regular expression. If `idx` is not specified, the default group index value is 1. The `idx` parameter is the Java regex Matcher group() method index.

Examples:

```
> SELECT regexp_extract('100-200', '(\\d+)-(\\d+)', 1);
 100
```

Since: 1.5.0

regexp_extract_all

regexp_extract_all(str, regexp[, idx]) - Extract all strings in the `str` that match the `regexp` expression and corresponding to the regex group index.

Arguments:

- str - a string expression.
- regexp - a string representing a regular expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$". * idx - an integer expression that representing the group index. The regex may contains multiple groups. `idx` indicates which regex group to extract. The group index should be non-negative. The minimum value of `idx` is 0, which means matching the entire regular expression. If `idx` is not specified, the default group index value is 1. The `idx` parameter is the Java regex Matcher group() method index.

Examples:

```
> SELECT regexp_extract_all('100-200, 300-400', '(\\d+)-(\\d+)', 1);
 ["100","300"]
```

Since: 3.1.0

regexp_like

regexp_like(str, regexp) - Returns true if `str` matches `regexp`, or false otherwise.

Arguments:

- str - a string expression
- regexp - a string expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

  There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$".

Examples:

```
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT regexp_like('%SystemDrive%\Users\John', '%SystemDrive%\\Users.*');
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT regexp_like('%SystemDrive%\\Users\\John', '%SystemDrive%\\\\Users.*');
true
```

Note:

Use LIKE to match with simple string pattern.

Since: 3.2.0

regexp_replace

regexp_replace(str, regexp, rep[, position]) - Replaces all substrings of `str` that match `regexp` with `rep`.

Arguments:

- str - a string expression to search for a regular expression pattern match.
- regexp - a string representing a regular expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

  There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$". * rep - a string expression to replace matched substrings. * position - a positive integer literal that indicates the position within `str` to begin searching. The default is 1. If position is greater than the number of characters in `str`, the result is `str`.

Examples:

```
> SELECT regexp_replace('100-200', '(\\d+)', 'num');
 num-num
```

Since: 1.5.0

regr_avgx

regr_avgx(y, x) - Returns the average of the independent variable for non-null pairs in a group, where `y` is the dependent variable and `x` is the independent variable.

Examples:

```
> SELECT regr_avgx(y, x) FROM VALUES (1, 2), (2, 2), (2, 3), (2, 4) AS tab(y, x);
 2.75
> SELECT regr_avgx(y, x) FROM VALUES (1, null) AS tab(y, x);
 NULL
> SELECT regr_avgx(y, x) FROM VALUES (null, 1) AS tab(y, x);
 NULL
> SELECT regr_avgx(y, x) FROM VALUES (1, 2), (2, null), (2, 3), (2, 4) AS tab(y, x);
 3.0
> SELECT regr_avgx(y, x) FROM VALUES (1, 2), (2, null), (null, 3), (2, 4) AS tab(y, x);
 3.0
```

Since: 3.3.0

regr_avgy

regr_avgy(y, x) - Returns the average of the dependent variable for non-null pairs in a group, where `y` is the dependent variable and `x` is the independent variable.

Examples:

```
> SELECT regr_avgy(y, x) FROM VALUES (1, 2), (2, 2), (2, 3), (2, 4) AS tab(y, x);
 1.75
> SELECT regr_avgy(y, x) FROM VALUES (1, null) AS tab(y, x);
 NULL
> SELECT regr_avgy(y, x) FROM VALUES (null, 1) AS tab(y, x);
 NULL
> SELECT regr_avgy(y, x) FROM VALUES (1, 2), (2, null), (2, 3), (2, 4) AS tab(y, x);
 1.6666666666666667
> SELECT regr_avgy(y, x) FROM VALUES (1, 2), (2, null), (null, 3), (2, 4) AS tab(y, x);
 1.5
```

Since: 3.3.0

regr_count

regr_count(y, x) - Returns the number of non-null number pairs in a group, where `y` is the dependent variable and `x` is the independent variable.

Examples:

```
> SELECT regr_count(y, x) FROM VALUES (1, 2), (2, 2), (2, 3), (2, 4) AS tab(y, x);
 4
> SELECT regr_count(y, x) FROM VALUES (1, 2), (2, null), (2, 3), (2, 4) AS tab(y, x);
 3
> SELECT regr_count(y, x) FROM VALUES (1, 2), (2, null), (null, 3), (2, 4) AS tab(y, x);
 2
```

Since: 3.3.0

regr_r2

regr_r2(y, x) - Returns the coefficient of determination for non-null pairs in a group, where `y` is the dependent variable and `x` is the independent variable.

Examples:

```
> SELECT regr_r2(y, x) FROM VALUES (1, 2), (2, 2), (2, 3), (2, 4) AS tab(y, x);
 0.2727272727272727
> SELECT regr_r2(y, x) FROM VALUES (1, null) AS tab(y, x);
 NULL
> SELECT regr_r2(y, x) FROM VALUES (null, 1) AS tab(y, x);
 NULL
> SELECT regr_r2(y, x) FROM VALUES (1, 2), (2, null), (2, 3), (2, 4) AS tab(y, x);
 0.7500000000000001
> SELECT regr_r2(y, x) FROM VALUES (1, 2), (2, null), (null, 3), (2, 4) AS tab(y, x);
 1.0
```

Since: 3.3.0

repeat

repeat(str, n) - Returns the string which repeats the given string value n times.

Examples:

```
> SELECT repeat('123', 2);
 123123
```

Since: 1.5.0

replace

replace(str, search[, replace]) - Replaces all occurrences of `search` with `replace`.

Arguments:

- str - a string expression

- search - a string expression. If `search` is not found in `str`, `str` is returned unchanged.
- replace - a string expression. If `replace` is not specified or is an empty string, nothing replaces the string that is removed from `str`.

Examples:

```
> SELECT replace('ABCabc', 'abc', 'DEF');
 ABCDEF
```

Since: 2.3.0

reverse

reverse(array) - Returns a reversed string or an array with reverse order of elements.

Examples:

```
> SELECT reverse('Spark SQL');
 LQS krapS
> SELECT reverse(array(2, 1, 4, 3));
 [3,4,1,2]
```

Note:

Reverse logic for arrays is available since 2.4.0.

Since: 1.5.0

right

right(str, len) - Returns the rightmost `len` (`len` can be string type) characters from the string `str`,if `len` is less or equal than 0 the result is an empty string.

Examples:

```
> SELECT right('Spark SQL', 3);
 SQL
```

Since: 2.3.0

rint

rint(expr) - Returns the double value that is closest in value to the argument and is equal to a mathematical integer.

Examples:

```
> SELECT rint(12.3456);
 12.0
```

Since: 1.4.0

rlike

rlike(str, regexp) - Returns true if `str` matches `regexp`, or false otherwise.

Arguments:

- str - a string expression
- regexp - a string expression. The regex string should be a Java regular expression.

  Since Spark 2.0, string literals (including regex patterns) are unescaped in our SQL parser. For example, to match "\abc", a regular expression for `regexp` can be "^\abc$".

  There is a SQL config 'spark.sql.parser.escapedStringLiterals' that can be used to fallback to the Spark 1.6 behavior regarding string literal parsing. For example, if the config is enabled, the `regexp` that can match "\abc" is "^\abc$".

Examples:

```
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT rlike('%SystemDrive%\Users\John', '%SystemDrive%\\Users.*');
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT rlike('%SystemDrive%\\Users\\John', '%SystemDrive%\\\\Users.*');
true
```

Note:

Use LIKE to match with simple string pattern.

Since: 1.0.0

round

round(expr, d) - Returns `expr` rounded to `d` decimal places using HALF_UP rounding mode.

Examples:

```
> SELECT round(2.5, 0);
 3
> SELECT round(25, -1);
 30
```

Since: 1.5.0

row_number

row_number() - Assigns a unique, sequential number to each row, starting with one, according to the ordering of rows within the window partition.

Examples:

```
> SELECT a, b, row_number() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1', 1
 A1 1   1
 A1 1   2
 A1 2   3
 A2 3   1
```

Since: 2.0.0

rpad

rpad(str, len[, pad]) - Returns `str`, right-padded with `pad` to a length of `len`. If `str` is longer than `len`, the return value is shortened to `len` characters. If `pad` is not specified, `str` will be padded to the right with space characters if it is a character string, and with zeros if it is a binary string.

Examples:

```
> SELECT rpad('hi', 5, '??');
 hi???
> SELECT rpad('hi', 1, '??');
 h
> SELECT rpad('hi', 5);
 hi
> SELECT hex(rpad(unhex('aabb'), 5));
 AABB000000
> SELECT hex(rpad(unhex('aabb'), 5, unhex('1122')));
 AABB112211
```

Since: 1.5.0

rtrim

rtrim(str) - Removes the trailing space characters from `str`.

Arguments:

- str - a string expression
- trimStr - the trim string characters to trim, the default value is a single space

Examples:

```
> SELECT rtrim('    SparkSQL   ');
 SparkSQL
```

Since: 1.5.0

schema_of_csv

schema_of_csv(csv[, options]) - Returns schema in the DDL format of CSV string.

Examples:

```
> SELECT schema_of_csv('1,abc');
 STRUCT<_c0: INT, _c1: STRING>
```

Since: 3.0.0

schema_of_json

schema_of_json(json[, options]) - Returns schema in the DDL format of JSON string.

Examples:

```
> SELECT schema_of_json('[{"col":0}]');
 ARRAY<STRUCT<col: BIGINT>>
> SELECT schema_of_json('[{"col":01}]', map('allowNumericLeadingZeros', 'true'));
 ARRAY<STRUCT<col: BIGINT>>
```

Since: 2.4.0

sec

sec(expr) - Returns the secant of `expr`, as if computed by `1/java.lang.Math.cos`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT sec(0);
 1.0
```

Since: 3.3.0

second

second(timestamp) - Returns the second component of the string/timestamp.

Examples:

```
> SELECT second('2009-07-30 12:58:59');
 59
```

Since: 1.5.0

sentences

sentences(str[, lang, country]) - Splits `str` into an array of array of words.

Examples:

```
> SELECT sentences('Hi there! Good morning.');
 [["Hi","there"],["Good","morning"]]
```

Since: 2.0.0

sequence

sequence(start, stop, step) - Generates an array of elements from start to stop (inclusive), incrementing by step. The type of the returned elements is the same as the type of argument expressions.

Supported types are: byte, short, integer, long, date, timestamp.

The start and stop expressions must resolve to the same type. If start and stop expressions resolve to the 'date' or 'timestamp' type then the step expression must resolve to the 'interval' or 'year-month interval' or 'day-time interval' type, otherwise to the same type as the start and stop expressions.

Arguments:

- start - an expression. The start of the range.
- stop - an expression. The end the range (inclusive).
- step - an optional expression. The step of the range. By default step is 1 if start is less than or equal to stop, otherwise -1. For the temporal sequences it's 1 day and -1 day respectively. If start is greater than stop then the step must be negative, and vice versa.

Examples:

```
> SELECT sequence(1, 5);
 [1,2,3,4,5]
> SELECT sequence(5, 1);
 [5,4,3,2,1]
> SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval 1 month);
 [2018-01-01,2018-02-01,2018-03-01]
> SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval '0-1' year to month
 [2018-01-01,2018-02-01,2018-03-01]
```

Since: 2.4.0

session_window

session_window(time_column, gap_duration) - Generates session window given a timestamp specifying column and gap duration. See 'Types of time windows' in Structured Streaming guide doc for detailed explanation and examples.

Arguments:

- time_column - The column or the expression to use as the timestamp for windowing by time. The time column must be of TimestampType.
- gap_duration - A string specifying the timeout of the session represented as "interval value" (See Interval Literal for more details.) for the fixed gap duration, or an expression which is applied for each input and evaluated to the "interval value" for the dynamic gap duration.

Examples:

```
> SELECT a, session_window.start, session_window.end, count(*) as cnt FROM VALUES ('A1', '20
  A1    2021-01-01 00:00:00 2021-01-01 00:09:30 2
  A1    2021-01-01 00:10:00 2021-01-01 00:15:00 1
  A2    2021-01-01 00:01:00 2021-01-01 00:06:00 1
> SELECT a, session_window.start, session_window.end, count(*) as cnt FROM VALUES ('A1', '20
  A1    2021-01-01 00:00:00 2021-01-01 00:09:30 2
  A1    2021-01-01 00:10:00 2021-01-01 00:15:00 1
  A2    2021-01-01 00:01:00 2021-01-01 00:02:00 1
  A2    2021-01-01 00:04:30 2021-01-01 00:05:30 1
```

Since: 3.2.0

sha

sha(expr) - Returns a sha1 hash value as a hex string of the `expr`.

Examples:

```
> SELECT sha('Spark');
 85f5955f4b27a9a4c2aab6ffe5d7189fc298b92c
```

Since: 1.5.0

sha1

sha1(expr) - Returns a sha1 hash value as a hex string of the `expr`.

Examples:

```
> SELECT sha1('Spark');
 85f5955f4b27a9a4c2aab6ffe5d7189fc298b92c
```

Since: 1.5.0

sha2

sha2(expr, bitLength) - Returns a checksum of SHA-2 family as a hex string of `expr`. SHA-224, SHA-256, SHA-384, and SHA-512 are supported. Bit length of 0 is equivalent to 256.

Examples:

```
> SELECT sha2('Spark', 256);
 529bc3b07127ecb7e53a4dcf1991d9152c24537d919178022b2c42657f79a26b
```

Since: 1.5.0

shiftleft

shiftleft(base, expr) - Bitwise left shift.

Examples:

```
> SELECT shiftleft(2, 1);
 4
```

Since: 1.5.0

shiftright

shiftright(base, expr) - Bitwise (signed) right shift.

Examples:

```
> SELECT shiftright(4, 1);
 2
```

Since: 1.5.0

shiftrightunsigned

shiftrightunsigned(base, expr) - Bitwise unsigned right shift.

Examples:

```
> SELECT shiftrightunsigned(4, 1);
 2
```

Since: 1.5.0

shuffle

shuffle(array) - Returns a random permutation of the given array.

Examples:

```
> SELECT shuffle(array(1, 20, 3, 5));
 [3,1,5,20]
> SELECT shuffle(array(1, 20, null, 3));
 [20,null,3,1]
```

Note:

The function is non-deterministic.

Since: 2.4.0

sign

sign(expr) - Returns -1.0, 0.0 or 1.0 as `expr` is negative, 0 or positive.

Examples:

```
> SELECT sign(40);
 1.0
> SELECT sign(INTERVAL -'100' YEAR);
 -1.0
```

Since: 1.4.0

signum

signum(expr) - Returns -1.0, 0.0 or 1.0 as `expr` is negative, 0 or positive.

Examples:

```
> SELECT signum(40);
 1.0
> SELECT signum(INTERVAL -'100' YEAR);
 -1.0
```

Since: 1.4.0

sin

sin(expr) - Returns the sine of `expr`, as if computed by `java.lang.Math.sin`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT sin(0);
 0.0
```

Since: 1.4.0

sinh

sinh(expr) - Returns hyperbolic sine of `expr`, as if computed by `java.lang.Math.sinh`.

Arguments:

- expr - hyperbolic angle

Examples:

```
> SELECT sinh(0);
 0.0
```

Since: 1.4.0

size

size(expr) - Returns the size of an array or a map. The function returns null for null input if spark.sql.legacy.sizeOfNull is set to false or spark.sql.ansi.enabled is set to true. Otherwise, the function returns -1 for null input. With the default settings, the function returns -1 for null input.

Examples:

```
> SELECT size(array('b', 'd', 'c', 'a'));
 4
> SELECT size(map('a', 1, 'b', 2));
 2
```

Since: 1.5.0

skewness

skewness(expr) - Returns the skewness value calculated from values of a group.

Examples:

```
> SELECT skewness(col) FROM VALUES (-10), (-20), (100), (1000) AS tab(col);
 1.1135657469022011
> SELECT skewness(col) FROM VALUES (-1000), (-100), (10), (20) AS tab(col);
 -1.1135657469022011
```

Since: 1.6.0

slice

slice(x, start, length) - Subsets array x starting from index start (array indices start at 1, or starting from the end if start is negative) with the specified length.

Examples:

```
> SELECT slice(array(1, 2, 3, 4), 2, 2);
 [2,3]
> SELECT slice(array(1, 2, 3, 4), -2, 2);
 [3,4]
```

Since: 2.4.0

smallint

smallint(expr) - Casts the value `expr` to the target data type `smallint`.

Since: 2.0.1

some

some(expr) - Returns true if at least one value of `expr` is true.

Examples:

```
> SELECT some(col) FROM VALUES (true), (false), (false) AS tab(col);
 true
> SELECT some(col) FROM VALUES (NULL), (true), (false) AS tab(col);
 true
> SELECT some(col) FROM VALUES (false), (false), (NULL) AS tab(col);
 false
```

Since: 3.0.0

sort_array

sort_array(array[, ascendingOrder]) - Sorts the input array in ascending or descending order according to the natural ordering of the array elements. NaN is greater than any non-NaN elements for double/float type. Null elements will be placed at the beginning of the returned array in ascending order or at the end of the returned array in descending order.

Examples:

```
> SELECT sort_array(array('b', 'd', null, 'c', 'a'), true);
 [null,"a","b","c","d"]
```

Since: 1.5.0

soundex

soundex(str) - Returns Soundex code of the string.

Examples:

```
> SELECT soundex('Miller');
 M460
```

Since: 1.5.0

space

space(n) - Returns a string consisting of `n` spaces.

Examples:

```
> SELECT concat(space(2), '1');
   1
```

Since: 1.5.0

spark_partition_id

spark_partition_id() - Returns the current partition id.

Examples:

```
> SELECT spark_partition_id();
 0
```

Since: 1.4.0

split

split(str, regex, limit) - Splits `str` around occurrences that match `regex` and returns an array with a length of at most `limit`

Arguments:

- str - a string expression to split.
- regex - a string representing a regular expression. The regex string should be a Java regular expression.
- limit - an integer expression which controls the number of times the regex is applied.
  - limit > 0: The resulting array's length will not be more than `limit`, and the resulting array's last entry will contain all input beyond the last matched regex.
  - limit <= 0: `regex` will be applied as many times as possible, and the resulting array can be of any size.

Examples:

```
> SELECT split('oneAtwoBthreeC', '[ABC]');
 ["one","two","three",""]
> SELECT split('oneAtwoBthreeC', '[ABC]', -1);
 ["one","two","three",""]
> SELECT split('oneAtwoBthreeC', '[ABC]', 2);
 ["one","twoBthreeC"]
```

Since: 1.5.0

split_part

split_part(str, delimiter, partNum) - Splits `str` by delimiter and return requested part of the split (1-based). If any input is null, returns null. if `partNum` is out of range of split parts, returns empty string. If `partNum` is 0, throws an error. If `partNum` is negative, the parts are counted backward from the end of the string. If the `delimiter` is an empty string, the `str` is not split.

Examples:

```
> SELECT split_part('11.12.13', '.', 3);
 13
```

Since: 3.3.0

sqrt

sqrt(expr) - Returns the square root of `expr`.

Examples:

```
> SELECT sqrt(4);
 2.0
```

Since: 1.1.1

stack

stack(n, expr1, ..., exprk) - Separates `expr1`, ..., `exprk` into `n` rows. Uses column names col0, col1, etc. by default unless specified otherwise.

Examples:

```
> SELECT stack(2, 1, 2, 3);
 1  2
 3  NULL
```

Since: 2.0.0

startswith

startswith(left, right) - Returns a boolean. The value is True if left starts with right. Returns NULL if either input expression is NULL. Otherwise, returns False. Both left or right must be of STRING or BINARY type.

Examples:

```
> SELECT startswith('Spark SQL', 'Spark');
 true
> SELECT startswith('Spark SQL', 'SQL');
 false
> SELECT startswith('Spark SQL', null);
 NULL
> SELECT startswith(x'537061726b2053514c', x'537061726b');
 true
> SELECT startswith(x'537061726b2053514c', x'53514c');
 false
```

Since: 3.3.0

std

std(expr) - Returns the sample standard deviation calculated from values of a group.

Examples:

```
> SELECT std(col) FROM VALUES (1), (2), (3) AS tab(col);
 1.0
```

Since: 1.6.0

stddev

stddev(expr) - Returns the sample standard deviation calculated from values of a group.

Examples:

```
> SELECT stddev(col) FROM VALUES (1), (2), (3) AS tab(col);
 1.0
```

Since: 1.6.0

stddev_pop

stddev_pop(expr) - Returns the population standard deviation calculated from values of a group.

Examples:

```
> SELECT stddev_pop(col) FROM VALUES (1), (2), (3) AS tab(col);
 0.816496580927726
```

Since: 1.6.0

stddev_samp

stddev_samp(expr) - Returns the sample standard deviation calculated from values of a group.

Examples:

```
> SELECT stddev_samp(col) FROM VALUES (1), (2), (3) AS tab(col);
 1.0
```

Since: 1.6.0

str_to_map

str_to_map(text[, pairDelim[, keyValueDelim]]) - Creates a map after splitting the text into key/value pairs using delimiters. Default delimiters are ',' for `pairDelim` and ':' for `keyValueDelim`. Both `pairDelim` and `keyValueDelim` are treated as regular expressions.

Examples:

```
> SELECT str_to_map('a:1,b:2,c:3', ',', ':');
 {"a":"1","b":"2","c":"3"}
> SELECT str_to_map('a');
 {"a":null}
```

Since: 2.0.1

string

string(expr) - Casts the value `expr` to the target data type `string`.

Since: 2.0.1

struct

struct(col1, col2, col3, ...) - Creates a struct with the given field values.

Examples:

```
> SELECT struct(1, 2, 3);
 {"col1":1,"col2":2,"col3":3}
```

Since: 1.4.0

substr

substr(str, pos[, len]) - Returns the substring of `str` that starts at `pos` and is of length `len`, or the slice of byte array that starts at `pos` and is of length `len`.

substr(str FROM pos[FOR len]]) - Returns the substring of `str` that starts at `pos` and is of length `len`, or the slice of byte array that starts at `pos` and is of length `len`.

Examples:

```
> SELECT substr('Spark SQL', 5);
 k SQL
> SELECT substr('Spark SQL', -3);
 SQL
> SELECT substr('Spark SQL', 5, 1);
 k
> SELECT substr('Spark SQL' FROM 5);
 k SQL
> SELECT substr('Spark SQL' FROM -3);
 SQL
> SELECT substr('Spark SQL' FROM 5 FOR 1);
 k
```

Since: 1.5.0

substring

substring(str, pos[, len]) - Returns the substring of `str` that starts at `pos` and is of length `len`, or the slice of byte array that starts at `pos` and is of length `len`.

substring(str FROM pos[FOR len]]) - Returns the substring of `str` that starts at `pos` and is of length `len`, or the slice of byte array that starts at `pos` and is of length `len`.

Examples:

```
> SELECT substring('Spark SQL', 5);
 k SQL
> SELECT substring('Spark SQL', -3);
 SQL
> SELECT substring('Spark SQL', 5, 1);
 k
> SELECT substring('Spark SQL' FROM 5);
 k SQL
> SELECT substring('Spark SQL' FROM -3);
 SQL
> SELECT substring('Spark SQL' FROM 5 FOR 1);
 k
```

Since: 1.5.0

substring_index

substring_index(str, delim, count) - Returns the substring from `str` before `count` occurrences of the delimiter `delim`. If `count` is positive, everything to the left of the final delimiter (counting from the left) is returned. If `count` is negative, everything to the right of the final delimiter (counting from the right) is returned. The function substring_index performs a case-sensitive match when searching for `delim`.

Examples:

```
> SELECT substring_index('www.apache.org', '.', 2);
 www.apache
```

Since: 1.5.0

sum

sum(expr) - Returns the sum calculated from values of a group.

Examples:

```
> SELECT sum(col) FROM VALUES (5), (10), (15) AS tab(col);
 30
> SELECT sum(col) FROM VALUES (NULL), (10), (15) AS tab(col);
 25
> SELECT sum(col) FROM VALUES (NULL), (NULL) AS tab(col);
 NULL
```

Since: 1.0.0

tan

tan(expr) - Returns the tangent of `expr`, as if computed by `java.lang.Math.tan`.

Arguments:

- expr - angle in radians

Examples:

```
> SELECT tan(0);
 0.0
```

Since: 1.4.0

tanh

tanh(expr) - Returns the hyperbolic tangent of `expr`, as if computed by `java.lang.Math.tanh`.

Arguments:

- expr - hyperbolic angle

Examples:

```
> SELECT tanh(0);
 0.0
```

Since: 1.4.0

timestamp

timestamp(expr) - Casts the value `expr` to the target data type `timestamp`.

Since: 2.0.1

timestamp_micros

timestamp_micros(microseconds) - Creates timestamp from the number of microseconds since UTC epoch.

Examples:

```
> SELECT timestamp_micros(1230219000123123);
 2008-12-25 07:30:00.123123
```

Since: 3.1.0

timestamp_millis

timestamp_millis(milliseconds) - Creates timestamp from the number of milliseconds since UTC epoch.

Examples:

```
> SELECT timestamp_millis(1230219000123);
 2008-12-25 07:30:00.123
```

Since: 3.1.0

timestamp_seconds

timestamp_seconds(seconds) - Creates timestamp from the number of seconds (can be fractional) since UTC epoch.

Examples:

```
> SELECT timestamp_seconds(1230219000);
 2008-12-25 07:30:00
> SELECT timestamp_seconds(1230219000.123);
 2008-12-25 07:30:00.123
```

Since: 3.1.0

tinyint

tinyint(expr) - Casts the value `expr` to the target data type `tinyint`.

Since: 2.0.1

to_binary

to_binary(str[, fmt]) - Converts the input `str` to a binary value based on the supplied `fmt`. `fmt` can be a case-insensitive string literal of "hex", "utf-8", or "base64". By default, the binary format for conversion is "hex" if `fmt` is omitted. The function returns NULL if at least one of the input parameters is NULL.

Examples:

```
> SELECT to_binary('abc', 'utf-8');
 abc
```

Since: 3.3.0

to_csv

to_csv(expr[, options]) - Returns a CSV string with a given struct value

Examples:

```
> SELECT to_csv(named_struct('a', 1, 'b', 2));
 1,2
> SELECT to_csv(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')), map('timesta
 26/08/2015
```

Since: 3.0.0

to_date

to_date(date_str[, fmt]) - Parses the `date_str` expression with the `fmt` expression to a date. Returns null with invalid input. By default, it follows casting rules to a date if the `fmt` is omitted.

Arguments:

- date_str - A string to be parsed to date.
- fmt - Date format pattern to follow. See Datetime Patterns for valid date and time format patterns.

Examples:

```
> SELECT to_date('2009-07-30 04:17:52');
 2009-07-30
> SELECT to_date('2016-12-31', 'yyyy-MM-dd');
 2016-12-31
```

Since: 1.5.0

to_json

to_json(expr[, options]) - Returns a JSON string with a given struct value

Examples:

```
> SELECT to_json(named_struct('a', 1, 'b', 2));
 {"a":1,"b":2}
> SELECT to_json(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')), map('timest
 {"time":"26/08/2015"}
> SELECT to_json(array(named_struct('a', 1, 'b', 2)));
 [{"a":1,"b":2}]
> SELECT to_json(map('a', named_struct('b', 1)));
 {"a":{"b":1}}
> SELECT to_json(map(named_struct('a', 1),named_struct('b', 2)));
 {"[1]":{"b":2}}
> SELECT to_json(map('a', 1));
 {"a":1}
> SELECT to_json(array((map('a', 1))));
 [{"a":1}]
```

Since: 2.2.0

to_number

to_number(expr, fmt) - Convert string 'expr' to a number based on the string format 'fmt'. Throws an exception if the conversion fails. The format can consist of the following characters, case insensitive: '0' or '9': Specifies an expected digit between 0 and 9. A sequence of 0 or 9 in the format string matches a sequence of digits in the input string. If the 0/9 sequence starts with 0 and is before the decimal point, it can only match a digit sequence of the same size. Otherwise, if the sequence starts with 9 or is after the decimal poin, it can match a digit sequence that has the same or smaller size. '.' or 'D': Specifies the position of the decimal point (optional, only allowed once). ',' or 'G': Specifies the position of the grouping (thousands) separator (,). There must be one or more 0 or 9 to the left of the rightmost grouping separator. 'expr' must match the grouping separator relevant for the size of the number. '$': Specifies the location of the $ currency sign. This character may only be specified once. 'S' or 'MI': Specifies the position of a '-' or '+' sign

(optional, only allowed once at the beginning or end of the format string). Note that 'S' allows '-' but 'MI' does not. 'PR': Only allowed at the end of the format string; specifies that 'expr' indicates a negative number with wrapping angled brackets. ('<1>').

Examples:

```
> SELECT to_number('454', '999');
 454
> SELECT to_number('454.00', '000.00');
 454.00
> SELECT to_number('12,454', '99,999');
 12454
> SELECT to_number('$78.12', '$99.99');
 78.12
> SELECT to_number('12,454.8-', '99,999.9S');
 -12454.8
```

Since: 3.3.0

to_timestamp

to_timestamp(timestamp_str[, fmt]) - Parses the `timestamp_str` expression with the `fmt` expression to a timestamp. Returns null with invalid input. By default, it follows casting rules to a timestamp if the `fmt` is omitted. The result data type is consistent with the value of configuration `spark.sql.timestampType`.

Arguments:

- timestamp_str - A string to be parsed to timestamp.
- fmt - Timestamp format pattern to follow. See Datetime Patterns for valid date and time format patterns.

Examples:

```
> SELECT to_timestamp('2016-12-31 00:12:00');
 2016-12-31 00:12:00
> SELECT to_timestamp('2016-12-31', 'yyyy-MM-dd');
 2016-12-31 00:00:00
```

Since: 2.2.0

to_unix_timestamp

to_unix_timestamp(timeExp[, fmt]) - Returns the UNIX timestamp of the given time.

Arguments:

- timeExp - A date/timestamp or string which is returned as a UNIX timestamp.
- fmt - Date/time format pattern to follow. Ignored if `timeExp` is not a string. Default value is "yyyy-MM-dd HH:mm:ss". See Datetime Patterns for valid date and time format patterns.

Examples:

```
> SELECT to_unix_timestamp('2016-04-08', 'yyyy-MM-dd');
 1460098800
```

Since: 1.6.0

to_utc_timestamp

to_utc_timestamp(timestamp, timezone) - Given a timestamp like '2017-07-14 02:40:00.0', interprets it as a time in the given time zone, and renders that time as a timestamp in UTC. For example, 'GMT+1' would yield '2017-07-14 01:40:00.0'.

Examples:

```
> SELECT to_utc_timestamp('2016-08-31', 'Asia/Seoul');
 2016-08-30 15:00:00
```

Since: 1.5.0

transform

transform(expr, func) - Transforms elements in an array using the function.

Examples:

```
> SELECT transform(array(1, 2, 3), x -> x + 1);
 [2,3,4]
> SELECT transform(array(1, 2, 3), (x, i) -> x + i);
 [1,3,5]
```

Since: 2.4.0

transform_keys

transform_keys(expr, func) - Transforms elements in a map using the function.

Examples:

```
> SELECT transform_keys(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -> k + 1);
 {2:1,3:2,4:3}
> SELECT transform_keys(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -> k + v);
 {2:1,4:2,6:3}
```

Since: 3.0.0

transform_values

transform_values(expr, func) - Transforms values in the map using the function.

Examples:

```
> SELECT transform_values(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -> v + 1);
 {1:2,2:3,3:4}
> SELECT transform_values(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -> k + v);
 {1:2,2:4,3:6}
```

Since: 3.0.0

translate

translate(input, from, to) - Translates the `input` string by replacing the characters present in the `from` string with the corresponding characters in the `to` string.

Examples:

```
> SELECT translate('AaBbCc', 'abc', '123');
 A1B2C3
```

Since: 1.5.0

trim

trim(str) - Removes the leading and trailing space characters from `str`.

trim(BOTH FROM str) - Removes the leading and trailing space characters from `str`.

trim(LEADING FROM str) - Removes the leading space characters from `str`.

trim(TRAILING FROM str) - Removes the trailing space characters from `str`.

trim(trimStr FROM str) - Remove the leading and trailing `trimStr` characters from `str`.

trim(BOTH trimStr FROM str) - Remove the leading and trailing `trimStr` characters from `str`.

trim(LEADING trimStr FROM str) - Remove the leading `trimStr` characters from `str`.

trim(TRAILING trimStr FROM str) - Remove the trailing `trimStr` characters from `str`.

Arguments:

- str - a string expression
- trimStr - the trim string characters to trim, the default value is a single space
- BOTH, FROM - these are keywords to specify trimming string characters from both ends of the string
- LEADING, FROM - these are keywords to specify trimming string characters from the left end of the string
- TRAILING, FROM - these are keywords to specify trimming string characters from the right end of the string

Examples:

```
> SELECT trim('    SparkSQL   ');
 SparkSQL
> SELECT trim(BOTH FROM '    SparkSQL   ');
 SparkSQL
> SELECT trim(LEADING FROM '    SparkSQL   ');
 SparkSQL
> SELECT trim(TRAILING FROM '    SparkSQL   ');
     SparkSQL
> SELECT trim('SL' FROM 'SSparkSQLS');
 parkSQ
> SELECT trim(BOTH 'SL' FROM 'SSparkSQLS');
 parkSQ
> SELECT trim(LEADING 'SL' FROM 'SSparkSQLS');
 parkSQLS
> SELECT trim(TRAILING 'SL' FROM 'SSparkSQLS');
 SSparkSQ
```

Since: 1.5.0

trunc

trunc(date, fmt) - Returns `date` with the time portion of the day truncated to the unit specified by the format model `fmt`.

Arguments:

- date - date value or valid date string
- fmt - the format representing the unit to be truncated to
  - "YEAR", "YYYY", "YY" - truncate to the first date of the year that the `date` falls in
  - "QUARTER" - truncate to the first date of the quarter that the `date` falls in
  - "MONTH", "MM", "MON" - truncate to the first date of the month that the `date` falls in
  - "WEEK" - truncate to the Monday of the week that the `date` falls in

Examples:

```
> SELECT trunc('2019-08-04', 'week');
 2019-07-29
> SELECT trunc('2019-08-04', 'quarter');
 2019-07-01
> SELECT trunc('2009-02-12', 'MM');
 2009-02-01
> SELECT trunc('2015-10-27', 'YEAR');
 2015-01-01
```

Since: 1.5.0

try_add

try_add(expr1, expr2) - Returns the sum of `expr1` and `expr2` and the result is null on overflow. The acceptable input types are the same with the `+` operator.

Examples:

```
> SELECT try_add(1, 2);
 3
> SELECT try_add(2147483647, 1);
 NULL
> SELECT try_add(date'2021-01-01', 1);
 2021-01-02
> SELECT try_add(date'2021-01-01', interval 1 year);
 2022-01-01
> SELECT try_add(timestamp'2021-01-01 00:00:00', interval 1 day);
 2021-01-02 00:00:00
> SELECT try_add(interval 1 year, interval 2 year);
 3-0
```

Since: 3.2.0

try_avg

try_avg(expr) - Returns the mean calculated from values of a group and the result is null on overflow.

Examples:

```
> SELECT try_avg(col) FROM VALUES (1), (2), (3) AS tab(col);
 2.0
> SELECT try_avg(col) FROM VALUES (1), (2), (NULL) AS tab(col);
 1.5
> SELECT try_avg(col) FROM VALUES (interval '2147483647 months'), (interval '1 months') AS t
 NULL
```

Since: 3.3.0

try_divide

try_divide(dividend, divisor) - Returns `dividend`/`divisor`. It always performs floating point division. Its result is always null if `expr2` is 0. `dividend` must be a numeric or an interval. `divisor` must be a numeric.

Examples:

```
> SELECT try_divide(3, 2);
 1.5
> SELECT try_divide(2L, 2L);
 1.0
> SELECT try_divide(1, 0);
 NULL
> SELECT try_divide(interval 2 month, 2);
 0-1
> SELECT try_divide(interval 2 month, 0);
 NULL
```

Since: 3.2.0

try_element_at

try_element_at(array, index) - Returns element of array at given (1-based) index. If Index is 0, Spark will throw an error. If index < 0, accesses elements from the last to the first. The function always returns NULL if the index exceeds the length of the array.

try_element_at(map, key) - Returns value for given key. The function always returns NULL if the key is not contained in the map.

Examples:

```
> SELECT try_element_at(array(1, 2, 3), 2);
 2
> SELECT try_element_at(map(1, 'a', 2, 'b'), 2);
 b
```

Since: 3.3.0

try_multiply

try_multiply(expr1, expr2) - Returns `expr1`*`expr2` and the result is null on overflow. The acceptable input types are the same with the `*` operator.

Examples:

```
> SELECT try_multiply(2, 3);
 6
> SELECT try_multiply(-2147483648, 10);
 NULL
> SELECT try_multiply(interval 2 year, 3);
 6-0
```

Since: 3.3.0

try_subtract

try_subtract(expr1, expr2) - Returns `expr1` - `expr2` and the result is null on overflow. The acceptable input types are the same with the `-` operator.

Examples:

```
> SELECT try_subtract(2, 1);
 1
> SELECT try_subtract(-2147483648, 1);
 NULL
> SELECT try_subtract(date'2021-01-02', 1);
 2021-01-01
> SELECT try_subtract(date'2021-01-01', interval 1 year);
 2020-01-01
> SELECT try_subtract(timestamp'2021-01-02 00:00:00', interval 1 day);
 2021-01-01 00:00:00
> SELECT try_subtract(interval 2 year, interval 1 year);
 1-0
```

Since: 3.3.0

try_sum

try_sum(expr) - Returns the sum calculated from values of a group and the result is null on overflow.

Examples:

```
> SELECT try_sum(col) FROM VALUES (5), (10), (15) AS tab(col);
 30
> SELECT try_sum(col) FROM VALUES (NULL), (10), (15) AS tab(col);
 25
> SELECT try_sum(col) FROM VALUES (NULL), (NULL) AS tab(col);
 NULL
> SELECT try_sum(col) FROM VALUES (9223372036854775807L), (1L) AS tab(col);
 NULL
```

Since: 3.3.0

try_to_binary

try_to_binary(str[, fmt]) - This is a special version of `to_binary` that performs the same operation, but returns a NULL value instead of raising an error if the conversion cannot be performed.

Examples:

```
> SELECT try_to_binary('abc', 'utf-8');
 abc
> select try_to_binary('a!', 'base64');
 NULL
> select try_to_binary('abc', 'invalidFormat');
 NULL
```

Since: 3.3.0

try_to_number

try_to_number(expr, fmt) - Convert string 'expr' to a number based on the string format `fmt`. Returns NULL if the string 'expr' does not match the expected format. The format follows the same semantics as the to_number function.

Examples:

```
> SELECT try_to_number('454', '999');
 454
> SELECT try_to_number('454.00', '000.00');
 454.00
> SELECT try_to_number('12,454', '99,999');
 12454
> SELECT try_to_number('$78.12', '$99.99');
 78.12
> SELECT try_to_number('12,454.8-', '99,999.9S');
 -12454.8
```

Since: 3.3.0

typeof

typeof(expr) - Return DDL-formatted type string for the data type of the input.

Examples:

```
> SELECT typeof(1);
 int
> SELECT typeof(array(1));
 array<int>
```

Since: 3.0.0

ucase

ucase(str) - Returns `str` with all characters changed to uppercase.

Examples:

```
> SELECT ucase('SparkSql');
 SPARKSQL
```

Since: 1.0.1

unbase64

unbase64(str) - Converts the argument from a base 64 string `str` to a binary.

Examples:

```
> SELECT unbase64('U3BhcmsgU1FM');
 Spark SQL
```

Since: 1.5.0

unhex

unhex(expr) - Converts hexadecimal `expr` to binary.

Examples:

```
> SELECT decode(unhex('537061726B2053514C'), 'UTF-8');
 Spark SQL
```

Since: 1.5.0

unix_date

unix_date(date) - Returns the number of days since 1970-01-01.

Examples:

```
> SELECT unix_date(DATE("1970-01-02"));
 1
```

Since: 3.1.0

unix_micros

unix_micros(timestamp) - Returns the number of microseconds since 1970-01-01 00:00:00 UTC.

Examples:

```
> SELECT unix_micros(TIMESTAMP('1970-01-01 00:00:01Z'));
 1000000
```

Since: 3.1.0

unix_millis

unix_millis(timestamp) - Returns the number of milliseconds since 1970-01-01 00:00:00 UTC. Truncates higher levels of precision.

Examples:

```
> SELECT unix_millis(TIMESTAMP('1970-01-01 00:00:01Z'));
 1000
```

Since: 3.1.0

unix_seconds

unix_seconds(timestamp) - Returns the number of seconds since 1970-01-01 00:00:00 UTC. Truncates higher levels of precision.

Examples:

```
> SELECT unix_seconds(TIMESTAMP('1970-01-01 00:00:01Z'));
 1
```

Since: 3.1.0

unix_timestamp

unix_timestamp([timeExp[, fmt]]) - Returns the UNIX timestamp of current or specified time.

Arguments:

- timeExp - A date/timestamp or string. If not provided, this defaults to current time.
- fmt - Date/time format pattern to follow. Ignored if `timeExp` is not a string. Default value is "yyyy-MM-dd HH:mm:ss". See Datetime Patterns for valid date and time format patterns.

Examples:

```
> SELECT unix_timestamp();
 1476884637
> SELECT unix_timestamp('2016-04-08', 'yyyy-MM-dd');
 1460041200
```

Since: 1.5.0

upper

upper(str) - Returns `str` with all characters changed to uppercase.

Examples:

```
> SELECT upper('SparkSql');
 SPARKSQL
```

Since: 1.0.1

uuid

uuid() - Returns an universally unique identifier (UUID) string. The value is returned as a canonical UUID 36-character string.

Examples:

```
> SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

Note:

The function is non-deterministic.

Since: 2.3.0

var_pop

var_pop(expr) - Returns the population variance calculated from values of a group.

Examples:

```
> SELECT var_pop(col) FROM VALUES (1), (2), (3) AS tab(col);
 0.6666666666666666
```

Since: 1.6.0

var_samp

var_samp(expr) - Returns the sample variance calculated from values of a group.

Examples:

```
> SELECT var_samp(col) FROM VALUES (1), (2), (3) AS tab(col);
 1.0
```

Since: 1.6.0

variance

variance(expr) - Returns the sample variance calculated from values of a group.

Examples:

```
> SELECT variance(col) FROM VALUES (1), (2), (3) AS tab(col);
 1.0
```

Since: 1.6.0

version

version() - Returns the Spark version. The string contains 2 fields, the first being a release version and the second being a git revision.

Examples:

```
> SELECT version();
 3.1.0 a6d6ea3efedbad14d99c24143834cd4e2e52fb40
```

Since: 3.0.0

weekday

weekday(date) - Returns the day of the week for date/timestamp (0 = Monday, 1 = Tuesday, ..., 6 = Sunday).

Examples:

```
> SELECT weekday('2009-07-30');
 3
```

Since: 2.4.0

weekofyear

weekofyear(date) - Returns the week of the year of the given date. A week is considered to start on a Monday and week 1 is the first week with >3 days.

Examples:

```
> SELECT weekofyear('2008-02-20');
 8
```

Since: 1.5.0

when

CASE WHEN expr1 THEN expr2 [WHEN expr3 THEN expr4]* [ELSE expr5] END - When `expr1` = true, returns `expr2`; else when `expr3` = true, returns `expr4`; else returns `expr5`.

Arguments:

- expr1, expr3 - the branch condition expressions should all be boolean type.
- expr2, expr4, expr5 - the branch value expressions and else value expression should all be same type or coercible to a common type.

Examples:

```
> SELECT CASE WHEN 1 > 0 THEN 1 WHEN 2 > 0 THEN 2.0 ELSE 1.2 END;
 1.0
> SELECT CASE WHEN 1 < 0 THEN 1 WHEN 2 > 0 THEN 2.0 ELSE 1.2 END;
 2.0
> SELECT CASE WHEN 1 < 0 THEN 1 WHEN 2 < 0 THEN 2.0 END;
 NULL
```

Since: 1.0.1

width_bucket

width_bucket(value, min_value, max_value, num_bucket) - Returns the bucket number to which `value` would be assigned in an equiwidth histogram with `num_bucket` buckets, in the range `min_value` to `max_value`."

Examples:

```
> SELECT width_bucket(5.3, 0.2, 10.6, 5);
 3
> SELECT width_bucket(-2.1, 1.3, 3.4, 3);
 0
> SELECT width_bucket(8.1, 0.0, 5.7, 4);
 5
> SELECT width_bucket(-0.9, 5.2, 0.5, 2);
 3
> SELECT width_bucket(INTERVAL '0' YEAR, INTERVAL '0' YEAR, INTERVAL '10' YEAR, 10);
 1
> SELECT width_bucket(INTERVAL '1' YEAR, INTERVAL '0' YEAR, INTERVAL '10' YEAR, 10);
 2
> SELECT width_bucket(INTERVAL '0' DAY, INTERVAL '0' DAY, INTERVAL '10' DAY, 10);
 1
> SELECT width_bucket(INTERVAL '1' DAY, INTERVAL '0' DAY, INTERVAL '10' DAY, 10);
 2
```

Since: 3.1.0

window

window(time_column, window_duration[, slide_duration[, start_time]]) - Bucketize rows into one or more time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported. See 'Window Operations on Event Time' in Structured Streaming guide doc for detailed explanation and examples.

Arguments:

- time_column - The column or the expression to use as the timestamp for windowing by time. The time column must be of TimestampType.
- window_duration - A string specifying the width of the window represented as "interval value". (See Interval Literal for more details.) Note that the duration is a fixed length of time, and does not vary over time according to a calendar.
- slide_duration - A string specifying the sliding interval of the window represented as "interval value". A new window will be generated every `slide_duration`. Must be less than or equal to the `window_duration`. This duration is likewise absolute, and does not vary according to a calendar.
- start_time - The offset with respect to 1970-01-01 00:00:00 UTC with which to start window intervals. For example, in order to have hourly tumbling windows that start 15 minutes past the hour, e.g. 12:15-13:15, 13:15-14:15... provide `start_time` as `15 minutes`.

Examples:

```
> SELECT a, window.start, window.end, count(*) as cnt FROM VALUES ('A1', '2021-01-01 00:00:0
  A1    2021-01-01 00:00:00 2021-01-01 00:05:00 2
  A1    2021-01-01 00:05:00 2021-01-01 00:10:00 1
  A2    2021-01-01 00:00:00 2021-01-01 00:05:00 1
> SELECT a, window.start, window.end, count(*) as cnt FROM VALUES ('A1', '2021-01-01 00:00:0
  A1    2020-12-31 23:55:00 2021-01-01 00:05:00 2
  A1    2021-01-01 00:00:00 2021-01-01 00:10:00 3
  A1    2021-01-01 00:05:00 2021-01-01 00:15:00 1
  A2    2020-12-31 23:55:00 2021-01-01 00:05:00 1
  A2    2021-01-01 00:00:00 2021-01-01 00:10:00 1
```

Since: 2.0.0

xpath

xpath(xml, xpath) - Returns a string array of values within the nodes of xml that match the XPath expression.

Examples:

```
> SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>','a/b/text()');
 ["b1","b2","b3"]
```

Since: 2.0.0

xpath_boolean

xpath_boolean(xml, xpath) - Returns true if the XPath expression evaluates to true, or if a matching node is found.

Examples:

```
> SELECT xpath_boolean('<a><b>1</b></a>','a/b');
 true
```

Since: 2.0.0

xpath_double

xpath_double(xml, xpath) - Returns a double value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_double('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

Since: 2.0.0

xpath_float

xpath_float(xml, xpath) - Returns a float value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_float('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

Since: 2.0.0

xpath_int

xpath_int(xml, xpath) - Returns an integer value, or the value zero if no match is found, or a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_int('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

Since: 2.0.0

xpath_long

xpath_long(xml, xpath) - Returns a long integer value, or the value zero if no match is found, or a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_long('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

Since: 2.0.0

xpath_number

xpath_number(xml, xpath) - Returns a double value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_number('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

Since: 2.0.0

xpath_short

xpath_short(xml, xpath) - Returns a short integer value, or the value zero if no match is found, or a match is found but the value is non-numeric.

Examples:

```
> SELECT xpath_short('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

Since: 2.0.0

xpath_string

xpath_string(xml, xpath) - Returns the text contents of the first xml node that matches the XPath expression.

Examples:

```
> SELECT xpath_string('<a><b>b</b><c>cc</c></a>','a/c');
 cc
```

Since: 2.0.0

xxhash64

xxhash64(expr1, expr2, ...) - Returns a 64-bit hash value of the arguments.

Examples:

```
> SELECT xxhash64('Spark', array(123), 2);
 5602566077635097486
```

Since: 3.0.0

year

year(date) - Returns the year component of the date/timestamp.

Examples:

```
> SELECT year('2016-07-30');
 2016
```

Since: 1.5.0

zip_with

zip_with(left, right, func) - Merges the two given arrays, element-wise, into a single array using function. If one array is shorter, nulls are appended at the end to match the length of the longer array, before applying function.

Examples:

```
> SELECT zip_with(array(1, 2, 3), array('a', 'b', 'c'), (x, y) -> (y, x));
 [{"y":"a","x":1},{"y":"b","x":2},{"y":"c","x":3}]
> SELECT zip_with(array(1, 2), array(3, 4), (x, y) -> x + y);
 [4,6]
> SELECT zip_with(array('a', 'b', 'c'), array('d', 'e', 'f'), (x, y) -> concat(x, y));
 ["ad","be","cf"]
```

Since: 2.4.0

|

expr1 | expr2 - Returns the result of bitwise OR of `expr1` and `expr2`.

Examples:

```
> SELECT 3 | 5;
 7
```

Since: 1.4.0

||

expr1 || expr2 - Returns the concatenation of `expr1` and `expr2`.

Examples:

```
> SELECT 'Spark' || 'SQL';
 SparkSQL
> SELECT array(1, 2, 3) || array(4, 5) || array(6);
 [1,2,3,4,5,6]
```

Note:

|| for arrays is available since 2.4.0.

Since: 2.3.0

~

~ expr - Returns the result of bitwise NOT of `expr`.

Examples:

```
> SELECT ~ 0;
 -1
```

Since: 1.4.0