Package: dd (via r-universe)

May 23, 2026

Title Functions Provided by DuckDB

Version 0.0.0.9000

Date 2025-04-10

Description Lists DuckDB functions for integration in R's help system.

License MIT + file LICENSE

URL https://github.com/krlmlr/dd, https://cynkra.github.io/dd

BugReports https://github.com/cynkra/dd/issues

Depends R (>= 4.4)

Config/Needs/website cynkra/cynkratemplate

Encoding UTF-8

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.0.0.9000

Repository https://cynkra.r-universe.dev

Date/Publication 2026-05-23 18:04:23 UTC

RemoteUrl https://github.com/cynkra/dd

RemoteRef HEAD

RemoteSha 4452d176f59cf222e2b7331c07f62141536c4e91

Contents

`*` *DuckDB function **

Description

DuckDB function `*()`.

Arguments

`col0`	`TINYINT` | `SMALLINT` | `INTEGER` | `BIGINT` | `HUGEINT` | `FLOAT` | `DOUBLE` | `DECIMAL` | `UTINYINT` | `USMALLINT` | `UINTEGER` | `UBIGINT` | `UHUGEINT` | `INTERVAL`
`col1`	`TINYINT` | `SMALLINT` | `INTEGER` | `BIGINT` | `HUGEINT` | `FLOAT` | `DOUBLE` | `DECIMAL` | `UTINYINT` | `USMALLINT` | `UINTEGER` | `UBIGINT` | `UHUGEINT` | `INTERVAL`

Value

`TINYINT` | `SMALLINT` | `INTEGER` | `BIGINT` | `HUGEINT` | `FLOAT` | `DOUBLE` | `DECIMAL` | `UTINYINT` | `USMALLINT` | `UINTEGER` | `UBIGINT` | `UHUGEINT` | `INTERVAL`

Overloads

- `` `*`(col0 = TINYINT, col1 = TINYINT) ``
- `` `*`(col0 = SMALLINT, col1 = SMALLINT) ``
- `` `*`(col0 = INTEGER, col1 = INTEGER) ``
- `` `*`(col0 = BIGINT, col1 = BIGINT) ``
- `` `*`(col0 = HUGEINT, col1 = HUGEINT) ``
- `` `*`(col0 = FLOAT, col1 = FLOAT) ``
- `` `*`(col0 = DOUBLE, col1 = DOUBLE) ``
- `` `*`(col0 = DECIMAL, col1 = DECIMAL) ``
- `` `*`(col0 = UTINYINT, col1 = UTINYINT) ``
- `` `*`(col0 = USMALLINT, col1 = USMALLINT) ``
- `` `*`(col0 = UINTEGER, col1 = UINTEGER) ``
- `` `*`(col0 = UBIGINT, col1 = UBIGINT) ``
- `` `*`(col0 = UHUGEINT, col1 = UHUGEINT) ``

- `*`(col0 = INTERVAL, col1 = DOUBLE)
- `*`(col0 = DOUBLE, col1 = INTERVAL)
- `*`(col0 = BIGINT, col1 = INTERVAL)
- `*`(col0 = INTERVAL, col1 = BIGINT)

** *DuckDB function ***

Description

Computes x to the power of y.

Usage

```
`**`(x = DOUBLE, y = DOUBLE)
```

Arguments

x	DOUBLE
y	DOUBLE

Value

DOUBLE

SQL examples

```
2 ** 3
```

/ *DuckDB function /*

Description

DuckDB function /().

Arguments

col0	FLOAT | DOUBLE | INTERVAL
col1	FLOAT | DOUBLE

Value

FLOAT | DOUBLE | INTERVAL

Overloads

- `/`(col0 = FLOAT, col1 = FLOAT)
- `/`(col0 = DOUBLE, col1 = DOUBLE)
- `/`(col0 = INTERVAL, col1 = DOUBLE)

// *DuckDB function //*

Description

DuckDB function //().

Arguments

col0	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Overloads

- `//`(col0 = TINYINT, col1 = TINYINT)
- `//`(col0 = SMALLINT, col1 = SMALLINT)
- `//`(col0 = INTEGER, col1 = INTEGER)
- `//`(col0 = BIGINT, col1 = BIGINT)
- `//`(col0 = HUGEINT, col1 = HUGEINT)
- `//`(col0 = FLOAT, col1 = FLOAT)
- `//`(col0 = DOUBLE, col1 = DOUBLE)
- `//`(col0 = UTINYINT, col1 = UTINYINT)
- `//`(col0 = USMALLINT, col1 = USMALLINT)
- `//`(col0 = UINTEGER, col1 = UINTEGER)
- `//`(col0 = UBIGINT, col1 = UBIGINT)
- `//`(col0 = UHUGEINT, col1 = UHUGEINT)

`&` *DuckDB function &*

Description

Bitwise AND.

Arguments

`left`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`
`right`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Overloads

- `` `&`(left = TINYINT, right = TINYINT) ``
- `` `&`(left = SMALLINT, right = SMALLINT) ``
- `` `&`(left = INTEGER, right = INTEGER) ``
- `` `&`(left = BIGINT, right = BIGINT) ``
- `` `&`(left = HUGEINT, right = HUGEINT) ``
- `` `&`(left = UTINYINT, right = UTINYINT) ``
- `` `&`(left = USMALLINT, right = USMALLINT) ``
- `` `&`(left = UINTEGER, right = UINTEGER) ``
- `` `&`(left = UBIGINT, right = UBIGINT) ``
- `` `&`(left = UHUGEINT, right = UHUGEINT) ``
- `` `&`(left = BIT, right = BIT) ``

SQL examples

```
91 & 15
```

`&&` *DuckDB function &&*

Description

Returns true if the lists have any element in common. NULLs are ignored.

Usage

```
`&&`(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
list_has_any([1, 2, 3], [2, 3, 4])
```

`%` *DuckDB function %*

Description

DuckDB function `%()`.

Arguments

col0	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

^-at *DuckDB function ^@*

Description

Returns `true` if `string` begins with `search_string`.

Usage

```
`^@`(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

`BOOLEAN`

SQL examples

```
starts_with('abc', 'a')
```

^ *DuckDB function ^*

Description

Computes x to the power of y.

Computes x to the power of y.

Usage

```
`^`(x = DOUBLE, y = DOUBLE)

pow(x = DOUBLE, y = DOUBLE)
```

Arguments

`x`	`DOUBLE`
`y`	`DOUBLE`

Value

`DOUBLE`

`DOUBLE`

SQL examples

```
2 ^ 3

pow(2, 3)
power(2, 3)
```

`~~*` *DuckDB function ~~**

Description

DuckDB function `~~*()`.

Usage

```
`~~*`(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

col0	VARCHAR
col1	VARCHAR

Value

BOOLEAN

`~~~` *DuckDB function ~~~*

Description

DuckDB function `~~~()`.

Usage

```
`~~~`(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

col0	VARCHAR
col1	VARCHAR

Value

BOOLEAN

~~ *DuckDB function ~~*

Description

DuckDB function `~~()`.

Usage

```
`~~`(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

col0	VARCHAR
col1	VARCHAR

Value

BOOLEAN

~ *DuckDB function ~*

Description

Bitwise NOT.

Arguments

input	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT

Overloads

- `~`(input = TINYINT)
- `~`(input = SMALLINT)
- `~`(input = INTEGER)
- `~`(input = BIGINT)
- `~`(input = HUGEINT)
- `~`(input = UTINYINT)

- `~`(input = USMALLINT)
- `~`(input = UINTEGER)
- `~`(input = UBIGINT)
- `~`(input = UHUGEINT)
- `~`(input = BIT)

SQL examples

```
~15
```

<-at *DuckDB function <@*

Description

Returns true if all elements of list2 are in list1. NULLs are ignored.

Usage

```
`<@`(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
list_has_all([1, 2, 3], [2, 3])
```

<< *DuckDB function «*

Description

Bitwise shift left.

Arguments

input	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT

Overloads

- `<<`(input = TINYINT, col1 = TINYINT)
- `<<`(input = SMALLINT, col1 = SMALLINT)
- `<<`(input = INTEGER, col1 = INTEGER)
- `<<`(input = BIGINT, col1 = BIGINT)
- `<<`(input = HUGEINT, col1 = HUGEINT)
- `<<`(input = UTINYINT, col1 = UTINYINT)
- `<<`(input = USMALLINT, col1 = USMALLINT)
- `<<`(input = UINTEGER, col1 = UINTEGER)
- `<<`(input = UBIGINT, col1 = UBIGINT)
- `<<`(input = UHUGEINT, col1 = UHUGEINT)
- `<<`(input = BIT, col1 = INTEGER)

SQL examples

```
1 << 4
```

`<=>` *DuckDB function <=>*

Description

Computes the cosine distance between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `` `<=>`(list1 = `FLOAT[]`, list2 = `FLOAT[]`) ``
- `` `<=>`(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`) ``

SQL examples

```
list_cosine_distance([1, 2, 3], [1, 2, 3])
```

`>>` *DuckDB function »*

Description

Bitwise shift right.

Arguments

`input`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`
`col1`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Overloads

- `>>`(input = TINYINT, col1 = TINYINT)
- `>>`(input = SMALLINT, col1 = SMALLINT)
- `>>`(input = INTEGER, col1 = INTEGER)
- `>>`(input = BIGINT, col1 = BIGINT)
- `>>`(input = HUGEINT, col1 = HUGEINT)
- `>>`(input = UTINYINT, col1 = UTINYINT)
- `>>`(input = USMALLINT, col1 = USMALLINT)
- `>>`(input = UINTEGER, col1 = UINTEGER)
- `>>`(input = UBIGINT, col1 = UBIGINT)
- `>>`(input = UHUGEINT, col1 = UHUGEINT)
- `>>`(input = BIT, col1 = INTEGER)

SQL examples

```
8 >> 2
```

---

abs *DuckDB function abs*

Description

Absolute value.

Arguments

x	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Overloads

- abs(x = TINYINT)
- abs(x = SMALLINT)
- abs(x = INTEGER)
- abs(x = BIGINT)
- abs(x = HUGEINT)
- abs(x = FLOAT)

- abs(x = DOUBLE)
- abs(x = DECIMAL)
- abs(x = UTINYINT)
- abs(x = USMALLINT)
- abs(x = UINTEGER)
- abs(x = UBIGINT)
- abs(x = UHUGEINT)

SQL examples

```
abs(-17.4)
```

---

acos *DuckDB function acos*

Description

Computes the arccosine of x.

Usage

```
acos(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
acos(0.5)
```

`acosh` *DuckDB function acosh*

Description

Computes the inverse hyperbolic cos of x.

Usage

```
acosh(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
acosh(2.3)
```

`add` *DuckDB function add*

Description

DuckDB function `add()`.

Arguments

col0	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTI
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTI

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLI

Overloads

- add(col0 = TINYINT)
- add(col0 = TINYINT, col1 = TINYINT)
- add(col0 = SMALLINT)
- add(col0 = SMALLINT, col1 = SMALLINT)
- add(col0 = INTEGER)
- add(col0 = INTEGER, col1 = INTEGER)
- add(col0 = BIGINT)
- add(col0 = BIGINT, col1 = BIGINT)
- add(col0 = HUGEINT)
- add(col0 = HUGEINT, col1 = HUGEINT)
- add(col0 = FLOAT)
- add(col0 = FLOAT, col1 = FLOAT)
- add(col0 = DOUBLE)
- add(col0 = DOUBLE, col1 = DOUBLE)
- add(col0 = DECIMAL)
- add(col0 = DECIMAL, col1 = DECIMAL)
- add(col0 = UTINYINT)
- add(col0 = UTINYINT, col1 = UTINYINT)
- add(col0 = USMALLINT)
- add(col0 = USMALLINT, col1 = USMALLINT)
- add(col0 = UINTEGER)
- add(col0 = UINTEGER, col1 = UINTEGER)
- add(col0 = UBIGINT)
- add(col0 = UBIGINT, col1 = UBIGINT)
- add(col0 = UHUGEINT)
- add(col0 = UHUGEINT, col1 = UHUGEINT)
- add(col0 = DATE, col1 = INTEGER)
- add(col0 = INTEGER, col1 = DATE)
- add(col0 = INTERVAL, col1 = INTERVAL)
- add(col0 = DATE, col1 = INTERVAL)
- add(col0 = INTERVAL, col1 = DATE)
- add(col0 = TIME, col1 = INTERVAL)
- add(col0 = INTERVAL, col1 = TIME)
- add(col0 = TIMESTAMP, col1 = INTERVAL)
- add(col0 = INTERVAL, col1 = TIMESTAMP)
- add(col0 = `TIME WITH TIME ZONE`, col1 = INTERVAL)

- `add(col0 = INTERVAL, col1 = `TIME WITH TIME ZONE`)`
- `add(col0 = TIME, col1 = DATE)`
- `add(col0 = DATE, col1 = TIME)`
- `add(col0 = `TIME WITH TIME ZONE`, col1 = DATE)`
- `add(col0 = DATE, col1 = `TIME WITH TIME ZONE`)`
- `add()`
- `add(col0 = BIGNUM, col1 = BIGNUM)`

`add_parquet_key` *DuckDB function add_parquet_key*

Description

DuckDB function `add_parquet_key()`.

Usage

```
add_parquet_key(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`
`col1`	`VARCHAR`

Value

Unspecified.

`age` *DuckDB function age*

Description

Subtract arguments, resulting in the time difference between the two timestamps.

Arguments

`timestamp`	`TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`INTERVAL`

Overloads

- age(timestamp = TIMESTAMP)
- age(timestamp = TIMESTAMP, timestamp = TIMESTAMP)
- age(timestamp = `TIMESTAMP WITH TIME ZONE`)
- age(timestamp = `TIMESTAMP WITH TIME ZONE`, timestamp = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')
```

---

aggregate *DuckDB function aggregate*

Description

Executes the aggregate function `function_name` on the elements of `list`.

Usage

```
aggregate(list = `ANY[]`, function_name = VARCHAR)
```

Arguments

list	ANY[]
function_name	VARCHAR

Value

ANY

SQL examples

```
aggregate([1, 2, NULL], 'min')
```

alias *DuckDB function alias*

Description

Returns the name of a given expression.

Usage

```
alias(expr = ANY)
```

Arguments

expr	ANY

Value

VARCHAR

SQL examples

```
alias(42 + 1)
```

all_profiling_output *DuckDB function all_profiling_output*

Description

DuckDB function `all_profiling_output()`.

Usage

```
all_profiling_output()
```

Value

Unspecified.

`any_value` *DuckDB function any_value*

Description

Returns the first non-NULL value from arg. This function is affected by ordering.

Arguments

`arg`	`DECIMAL | ANY`

Value

`DECIMAL | ANY`

Overloads

- `any_value(arg = DECIMAL)`
- `any_value(arg = ANY)`

`apply` *DuckDB function apply*

Description

Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function.

Usage

```
apply(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
apply([1, 2, 3], lambda x : x + 1)
```

`approx_count_distinct` *DuckDB function approx_count_distinct*

Description

Computes the approximate count of distinct elements using HyperLogLog.

Usage

```
approx_count_distinct(any = ANY)
```

Arguments

any	ANY

Value

BIGINT

SQL examples

```
approx_count_distinct(A)
```

`approx_quantile` *DuckDB function approx_quantile*

Description

Computes the approximate quantile using T-Digest.

Arguments

x	DECIMAL | SMALLINT | INTEGER | BIGINT | HUGEINT | DOUBLE | DATE | TIME | TIME WI
pos	FLOAT | FLOAT[]

Value

DECIMAL | SMALLINT | INTEGER | BIGINT | HUGEINT | DOUBLE | DATE | TIME | TIME WITH TIME ZONE |

Overloads

- approx_quantile(x = DECIMAL, pos = FLOAT)
- approx_quantile(x = SMALLINT, pos = FLOAT)
- approx_quantile(x = INTEGER, pos = FLOAT)
- approx_quantile(x = BIGINT, pos = FLOAT)
- approx_quantile(x = HUGEINT, pos = FLOAT)
- approx_quantile(x = DOUBLE, pos = FLOAT)
- approx_quantile(x = DATE, pos = FLOAT)
- approx_quantile(x = TIME, pos = FLOAT)
- approx_quantile(x = `TIME WITH TIME ZONE`, pos = FLOAT)
- approx_quantile(x = TIMESTAMP, pos = FLOAT)
- approx_quantile(x = `TIMESTAMP WITH TIME ZONE`, pos = FLOAT)
- approx_quantile(x = DECIMAL, pos = `FLOAT[]`)
- approx_quantile(x = TINYINT, pos = `FLOAT[]`)
- approx_quantile(x = SMALLINT, pos = `FLOAT[]`)
- approx_quantile(x = INTEGER, pos = `FLOAT[]`)
- approx_quantile(x = BIGINT, pos = `FLOAT[]`)
- approx_quantile(x = HUGEINT, pos = `FLOAT[]`)
- approx_quantile(x = FLOAT, pos = `FLOAT[]`)
- approx_quantile(x = DOUBLE, pos = `FLOAT[]`)
- approx_quantile(x = DATE, pos = `FLOAT[]`)
- approx_quantile(x = TIME, pos = `FLOAT[]`)
- approx_quantile(x = `TIME WITH TIME ZONE`, pos = `FLOAT[]`)
- approx_quantile(x = TIMESTAMP, pos = `FLOAT[]`)
- approx_quantile(x = `TIMESTAMP WITH TIME ZONE`, pos = `FLOAT[]`)

SQL examples

```
approx_quantile(x, 0.5)
```

`approx_top_k` *DuckDB function approx_top_k*

Description

Finds the k approximately most occurring values in the data set.

Usage

```
approx_top_k(val = ANY, k = BIGINT)
```

Arguments

val	ANY
k	BIGINT

Value

```
ANY[]
```

SQL examples

```
approx_top_k(x, 5)
```

`arbitrary` *DuckDB function arbitrary*

Description

Returns the first value (NULL or non-NULL) from arg. This function is affected by ordering.

Arguments

arg	DECIMAL | ANY

Value

```
DECIMAL | ANY
```

Overloads

- `arbitrary(arg = DECIMAL)`
- `arbitrary(arg = ANY)`

SQL examples

```
arbitrary(A)
```

arg_max *DuckDB function arg_max*

Description

Finds the row with the maximum val. Calculates the non-NULL arg expression at that row.

Arguments

arg	INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON
val	INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT
col2	BIGINT

Value

INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC

Overloads

- arg_max(arg = INTEGER, val = INTEGER)
- arg_max(arg = INTEGER, val = BIGINT)
- arg_max(arg = INTEGER, val = HUGEINT)
- arg_max(arg = INTEGER, val = DOUBLE)
- arg_max(arg = INTEGER, val = VARCHAR)
- arg_max(arg = INTEGER, val = DATE)
- arg_max(arg = INTEGER, val = TIMESTAMP)
- arg_max(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = INTEGER, val = BLOB)
- arg_max(arg = BIGINT, val = INTEGER)
- arg_max(arg = BIGINT, val = BIGINT)
- arg_max(arg = BIGINT, val = HUGEINT)
- arg_max(arg = BIGINT, val = DOUBLE)
- arg_max(arg = BIGINT, val = VARCHAR)
- arg_max(arg = BIGINT, val = DATE)
- arg_max(arg = BIGINT, val = TIMESTAMP)
- arg_max(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = BIGINT, val = BLOB)
- arg_max(arg = DOUBLE, val = INTEGER)
- arg_max(arg = DOUBLE, val = BIGINT)
- arg_max(arg = DOUBLE, val = HUGEINT)

- arg_max(arg = DOUBLE, val = DOUBLE)
- arg_max(arg = DOUBLE, val = VARCHAR)
- arg_max(arg = DOUBLE, val = DATE)
- arg_max(arg = DOUBLE, val = TIMESTAMP)
- arg_max(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = DOUBLE, val = BLOB)
- arg_max(arg = VARCHAR, val = INTEGER)
- arg_max(arg = VARCHAR, val = BIGINT)
- arg_max(arg = VARCHAR, val = HUGEINT)
- arg_max(arg = VARCHAR, val = DOUBLE)
- arg_max(arg = VARCHAR, val = VARCHAR)
- arg_max(arg = VARCHAR, val = DATE)
- arg_max(arg = VARCHAR, val = TIMESTAMP)
- arg_max(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = VARCHAR, val = BLOB)
- arg_max(arg = DATE, val = INTEGER)
- arg_max(arg = DATE, val = BIGINT)
- arg_max(arg = DATE, val = HUGEINT)
- arg_max(arg = DATE, val = DOUBLE)
- arg_max(arg = DATE, val = VARCHAR)
- arg_max(arg = DATE, val = DATE)
- arg_max(arg = DATE, val = TIMESTAMP)
- arg_max(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = DATE, val = BLOB)
- arg_max(arg = TIMESTAMP, val = INTEGER)
- arg_max(arg = TIMESTAMP, val = BIGINT)
- arg_max(arg = TIMESTAMP, val = HUGEINT)
- arg_max(arg = TIMESTAMP, val = DOUBLE)
- arg_max(arg = TIMESTAMP, val = VARCHAR)
- arg_max(arg = TIMESTAMP, val = DATE)
- arg_max(arg = TIMESTAMP, val = TIMESTAMP)
- arg_max(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = TIMESTAMP, val = BLOB)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)

- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- arg_max(arg = BLOB, val = INTEGER)
- arg_max(arg = BLOB, val = BIGINT)
- arg_max(arg = BLOB, val = HUGEINT)
- arg_max(arg = BLOB, val = DOUBLE)
- arg_max(arg = BLOB, val = VARCHAR)
- arg_max(arg = BLOB, val = DATE)
- arg_max(arg = BLOB, val = TIMESTAMP)
- arg_max(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = BLOB, val = BLOB)
- arg_max(arg = DECIMAL, val = INTEGER)
- arg_max(arg = DECIMAL, val = BIGINT)
- arg_max(arg = DECIMAL, val = HUGEINT)
- arg_max(arg = DECIMAL, val = DOUBLE)
- arg_max(arg = DECIMAL, val = VARCHAR)
- arg_max(arg = DECIMAL, val = DATE)
- arg_max(arg = DECIMAL, val = TIMESTAMP)
- arg_max(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = DECIMAL, val = BLOB)
- arg_max(arg = ANY, val = INTEGER)
- arg_max(arg = ANY, val = BIGINT)
- arg_max(arg = ANY, val = HUGEINT)
- arg_max(arg = ANY, val = DOUBLE)
- arg_max(arg = ANY, val = VARCHAR)
- arg_max(arg = ANY, val = DATE)
- arg_max(arg = ANY, val = TIMESTAMP)
- arg_max(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max(arg = ANY, val = BLOB)
- arg_max(arg = ANY, val = ANY)
- arg_max(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
arg_max(A, B)
```

`arg_max_null` *DuckDB function arg_max_null*

Description

Finds the row with the maximum val. Calculates the arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `arg_max_null(arg = INTEGER, val = INTEGER)`
- `arg_max_null(arg = INTEGER, val = BIGINT)`
- `arg_max_null(arg = INTEGER, val = HUGEINT)`
- `arg_max_null(arg = INTEGER, val = DOUBLE)`
- `arg_max_null(arg = INTEGER, val = VARCHAR)`
- `arg_max_null(arg = INTEGER, val = DATE)`
- `arg_max_null(arg = INTEGER, val = TIMESTAMP)`
- `arg_max_null(arg = INTEGER, val = ` `TIMESTAMP WITH TIME ZONE` `)`
- `arg_max_null(arg = INTEGER, val = BLOB)`
- `arg_max_null(arg = BIGINT, val = INTEGER)`
- `arg_max_null(arg = BIGINT, val = BIGINT)`
- `arg_max_null(arg = BIGINT, val = HUGEINT)`
- `arg_max_null(arg = BIGINT, val = DOUBLE)`
- `arg_max_null(arg = BIGINT, val = VARCHAR)`
- `arg_max_null(arg = BIGINT, val = DATE)`
- `arg_max_null(arg = BIGINT, val = TIMESTAMP)`
- `arg_max_null(arg = BIGINT, val = ` `TIMESTAMP WITH TIME ZONE` `)`
- `arg_max_null(arg = BIGINT, val = BLOB)`
- `arg_max_null(arg = DOUBLE, val = INTEGER)`
- `arg_max_null(arg = DOUBLE, val = BIGINT)`
- `arg_max_null(arg = DOUBLE, val = HUGEINT)`
- `arg_max_null(arg = DOUBLE, val = DOUBLE)`
- `arg_max_null(arg = DOUBLE, val = VARCHAR)`

- arg_max_null(arg = DOUBLE, val = DATE)
- arg_max_null(arg = DOUBLE, val = TIMESTAMP)
- arg_max_null(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = DOUBLE, val = BLOB)
- arg_max_null(arg = VARCHAR, val = INTEGER)
- arg_max_null(arg = VARCHAR, val = BIGINT)
- arg_max_null(arg = VARCHAR, val = HUGEINT)
- arg_max_null(arg = VARCHAR, val = DOUBLE)
- arg_max_null(arg = VARCHAR, val = VARCHAR)
- arg_max_null(arg = VARCHAR, val = DATE)
- arg_max_null(arg = VARCHAR, val = TIMESTAMP)
- arg_max_null(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = VARCHAR, val = BLOB)
- arg_max_null(arg = DATE, val = INTEGER)
- arg_max_null(arg = DATE, val = BIGINT)
- arg_max_null(arg = DATE, val = HUGEINT)
- arg_max_null(arg = DATE, val = DOUBLE)
- arg_max_null(arg = DATE, val = VARCHAR)
- arg_max_null(arg = DATE, val = DATE)
- arg_max_null(arg = DATE, val = TIMESTAMP)
- arg_max_null(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = DATE, val = BLOB)
- arg_max_null(arg = TIMESTAMP, val = INTEGER)
- arg_max_null(arg = TIMESTAMP, val = BIGINT)
- arg_max_null(arg = TIMESTAMP, val = HUGEINT)
- arg_max_null(arg = TIMESTAMP, val = DOUBLE)
- arg_max_null(arg = TIMESTAMP, val = VARCHAR)
- arg_max_null(arg = TIMESTAMP, val = DATE)
- arg_max_null(arg = TIMESTAMP, val = TIMESTAMP)
- arg_max_null(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = TIMESTAMP, val = BLOB)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)

- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- arg_max_null(arg = BLOB, val = INTEGER)
- arg_max_null(arg = BLOB, val = BIGINT)
- arg_max_null(arg = BLOB, val = HUGEINT)
- arg_max_null(arg = BLOB, val = DOUBLE)
- arg_max_null(arg = BLOB, val = VARCHAR)
- arg_max_null(arg = BLOB, val = DATE)
- arg_max_null(arg = BLOB, val = TIMESTAMP)
- arg_max_null(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = BLOB, val = BLOB)
- arg_max_null(arg = DECIMAL, val = INTEGER)
- arg_max_null(arg = DECIMAL, val = BIGINT)
- arg_max_null(arg = DECIMAL, val = HUGEINT)
- arg_max_null(arg = DECIMAL, val = DOUBLE)
- arg_max_null(arg = DECIMAL, val = VARCHAR)
- arg_max_null(arg = DECIMAL, val = DATE)
- arg_max_null(arg = DECIMAL, val = TIMESTAMP)
- arg_max_null(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = DECIMAL, val = BLOB)
- arg_max_null(arg = ANY, val = INTEGER)
- arg_max_null(arg = ANY, val = BIGINT)
- arg_max_null(arg = ANY, val = HUGEINT)
- arg_max_null(arg = ANY, val = DOUBLE)
- arg_max_null(arg = ANY, val = VARCHAR)
- arg_max_null(arg = ANY, val = DATE)
- arg_max_null(arg = ANY, val = TIMESTAMP)
- arg_max_null(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- arg_max_null(arg = ANY, val = BLOB)
- arg_max_null(arg = ANY, val = ANY)

SQL examples

```
arg_max_null(A, B)
```

`arg_min` *DuckDB function arg_min*

Description

Finds the row with the minimum val. Calculates the non-NULL arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT`
`col2`	`BIGINT`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `arg_min(arg = INTEGER, val = INTEGER)`
- `arg_min(arg = INTEGER, val = BIGINT)`
- `arg_min(arg = INTEGER, val = HUGEINT)`
- `arg_min(arg = INTEGER, val = DOUBLE)`
- `arg_min(arg = INTEGER, val = VARCHAR)`
- `arg_min(arg = INTEGER, val = DATE)`
- `arg_min(arg = INTEGER, val = TIMESTAMP)`
- `arg_min(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min(arg = INTEGER, val = BLOB)`
- `arg_min(arg = BIGINT, val = INTEGER)`
- `arg_min(arg = BIGINT, val = BIGINT)`
- `arg_min(arg = BIGINT, val = HUGEINT)`
- `arg_min(arg = BIGINT, val = DOUBLE)`
- `arg_min(arg = BIGINT, val = VARCHAR)`
- `arg_min(arg = BIGINT, val = DATE)`
- `arg_min(arg = BIGINT, val = TIMESTAMP)`
- `arg_min(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min(arg = BIGINT, val = BLOB)`
- `arg_min(arg = DOUBLE, val = INTEGER)`
- `arg_min(arg = DOUBLE, val = BIGINT)`
- `arg_min(arg = DOUBLE, val = HUGEINT)`
- `arg_min(arg = DOUBLE, val = DOUBLE)`

- arg_min(arg = DOUBLE, val = VARCHAR)
- arg_min(arg = DOUBLE, val = DATE)
- arg_min(arg = DOUBLE, val = TIMESTAMP)
- arg_min(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = DOUBLE, val = BLOB)
- arg_min(arg = VARCHAR, val = INTEGER)
- arg_min(arg = VARCHAR, val = BIGINT)
- arg_min(arg = VARCHAR, val = HUGEINT)
- arg_min(arg = VARCHAR, val = DOUBLE)
- arg_min(arg = VARCHAR, val = VARCHAR)
- arg_min(arg = VARCHAR, val = DATE)
- arg_min(arg = VARCHAR, val = TIMESTAMP)
- arg_min(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = VARCHAR, val = BLOB)
- arg_min(arg = DATE, val = INTEGER)
- arg_min(arg = DATE, val = BIGINT)
- arg_min(arg = DATE, val = HUGEINT)
- arg_min(arg = DATE, val = DOUBLE)
- arg_min(arg = DATE, val = VARCHAR)
- arg_min(arg = DATE, val = DATE)
- arg_min(arg = DATE, val = TIMESTAMP)
- arg_min(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = DATE, val = BLOB)
- arg_min(arg = TIMESTAMP, val = INTEGER)
- arg_min(arg = TIMESTAMP, val = BIGINT)
- arg_min(arg = TIMESTAMP, val = HUGEINT)
- arg_min(arg = TIMESTAMP, val = DOUBLE)
- arg_min(arg = TIMESTAMP, val = VARCHAR)
- arg_min(arg = TIMESTAMP, val = DATE)
- arg_min(arg = TIMESTAMP, val = TIMESTAMP)
- arg_min(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = TIMESTAMP, val = BLOB)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)

- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- arg_min(arg = BLOB, val = INTEGER)
- arg_min(arg = BLOB, val = BIGINT)
- arg_min(arg = BLOB, val = HUGEINT)
- arg_min(arg = BLOB, val = DOUBLE)
- arg_min(arg = BLOB, val = VARCHAR)
- arg_min(arg = BLOB, val = DATE)
- arg_min(arg = BLOB, val = TIMESTAMP)
- arg_min(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = BLOB, val = BLOB)
- arg_min(arg = DECIMAL, val = INTEGER)
- arg_min(arg = DECIMAL, val = BIGINT)
- arg_min(arg = DECIMAL, val = HUGEINT)
- arg_min(arg = DECIMAL, val = DOUBLE)
- arg_min(arg = DECIMAL, val = VARCHAR)
- arg_min(arg = DECIMAL, val = DATE)
- arg_min(arg = DECIMAL, val = TIMESTAMP)
- arg_min(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = DECIMAL, val = BLOB)
- arg_min(arg = ANY, val = INTEGER)
- arg_min(arg = ANY, val = BIGINT)
- arg_min(arg = ANY, val = HUGEINT)
- arg_min(arg = ANY, val = DOUBLE)
- arg_min(arg = ANY, val = VARCHAR)
- arg_min(arg = ANY, val = DATE)
- arg_min(arg = ANY, val = TIMESTAMP)
- arg_min(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min(arg = ANY, val = BLOB)
- arg_min(arg = ANY, val = ANY)
- arg_min(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
arg_min(A, B)
```

`arg_min_null` *DuckDB function arg_min_null*

Description

Finds the row with the minimum val. Calculates the arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WI`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `arg_min_null(arg = INTEGER, val = INTEGER)`
- `arg_min_null(arg = INTEGER, val = BIGINT)`
- `arg_min_null(arg = INTEGER, val = HUGEINT)`
- `arg_min_null(arg = INTEGER, val = DOUBLE)`
- `arg_min_null(arg = INTEGER, val = VARCHAR)`
- `arg_min_null(arg = INTEGER, val = DATE)`
- `arg_min_null(arg = INTEGER, val = TIMESTAMP)`
- `arg_min_null(arg = INTEGER, val = ` `TIMESTAMP WITH TIME ZONE` `)`
- `arg_min_null(arg = INTEGER, val = BLOB)`
- `arg_min_null(arg = BIGINT, val = INTEGER)`
- `arg_min_null(arg = BIGINT, val = BIGINT)`
- `arg_min_null(arg = BIGINT, val = HUGEINT)`
- `arg_min_null(arg = BIGINT, val = DOUBLE)`
- `arg_min_null(arg = BIGINT, val = VARCHAR)`
- `arg_min_null(arg = BIGINT, val = DATE)`
- `arg_min_null(arg = BIGINT, val = TIMESTAMP)`
- `arg_min_null(arg = BIGINT, val = ` `TIMESTAMP WITH TIME ZONE` `)`
- `arg_min_null(arg = BIGINT, val = BLOB)`
- `arg_min_null(arg = DOUBLE, val = INTEGER)`
- `arg_min_null(arg = DOUBLE, val = BIGINT)`
- `arg_min_null(arg = DOUBLE, val = HUGEINT)`
- `arg_min_null(arg = DOUBLE, val = DOUBLE)`
- `arg_min_null(arg = DOUBLE, val = VARCHAR)`

- arg_min_null(arg = DOUBLE, val = DATE)
- arg_min_null(arg = DOUBLE, val = TIMESTAMP)
- arg_min_null(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min_null(arg = DOUBLE, val = BLOB)
- arg_min_null(arg = VARCHAR, val = INTEGER)
- arg_min_null(arg = VARCHAR, val = BIGINT)
- arg_min_null(arg = VARCHAR, val = HUGEINT)
- arg_min_null(arg = VARCHAR, val = DOUBLE)
- arg_min_null(arg = VARCHAR, val = VARCHAR)
- arg_min_null(arg = VARCHAR, val = DATE)
- arg_min_null(arg = VARCHAR, val = TIMESTAMP)
- arg_min_null(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min_null(arg = VARCHAR, val = BLOB)
- arg_min_null(arg = DATE, val = INTEGER)
- arg_min_null(arg = DATE, val = BIGINT)
- arg_min_null(arg = DATE, val = HUGEINT)
- arg_min_null(arg = DATE, val = DOUBLE)
- arg_min_null(arg = DATE, val = VARCHAR)
- arg_min_null(arg = DATE, val = DATE)
- arg_min_null(arg = DATE, val = TIMESTAMP)
- arg_min_null(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min_null(arg = DATE, val = BLOB)
- arg_min_null(arg = TIMESTAMP, val = INTEGER)
- arg_min_null(arg = TIMESTAMP, val = BIGINT)
- arg_min_null(arg = TIMESTAMP, val = HUGEINT)
- arg_min_null(arg = TIMESTAMP, val = DOUBLE)
- arg_min_null(arg = TIMESTAMP, val = VARCHAR)
- arg_min_null(arg = TIMESTAMP, val = DATE)
- arg_min_null(arg = TIMESTAMP, val = TIMESTAMP)
- arg_min_null(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- arg_min_null(arg = TIMESTAMP, val = BLOB)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)

- `arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)`
- `arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min_null(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)`
- `arg_min_null(arg = BLOB, val = INTEGER)`
- `arg_min_null(arg = BLOB, val = BIGINT)`
- `arg_min_null(arg = BLOB, val = HUGEINT)`
- `arg_min_null(arg = BLOB, val = DOUBLE)`
- `arg_min_null(arg = BLOB, val = VARCHAR)`
- `arg_min_null(arg = BLOB, val = DATE)`
- `arg_min_null(arg = BLOB, val = TIMESTAMP)`
- `arg_min_null(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min_null(arg = BLOB, val = BLOB)`
- `arg_min_null(arg = DECIMAL, val = INTEGER)`
- `arg_min_null(arg = DECIMAL, val = BIGINT)`
- `arg_min_null(arg = DECIMAL, val = HUGEINT)`
- `arg_min_null(arg = DECIMAL, val = DOUBLE)`
- `arg_min_null(arg = DECIMAL, val = VARCHAR)`
- `arg_min_null(arg = DECIMAL, val = DATE)`
- `arg_min_null(arg = DECIMAL, val = TIMESTAMP)`
- `arg_min_null(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min_null(arg = DECIMAL, val = BLOB)`
- `arg_min_null(arg = ANY, val = INTEGER)`
- `arg_min_null(arg = ANY, val = BIGINT)`
- `arg_min_null(arg = ANY, val = HUGEINT)`
- `arg_min_null(arg = ANY, val = DOUBLE)`
- `arg_min_null(arg = ANY, val = VARCHAR)`
- `arg_min_null(arg = ANY, val = DATE)`
- `arg_min_null(arg = ANY, val = TIMESTAMP)`
- `arg_min_null(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)`
- `arg_min_null(arg = ANY, val = BLOB)`
- `arg_min_null(arg = ANY, val = ANY)`

SQL examples

```
arg_min_null(A, B)
```

`argmax` *DuckDB function argmax*

Description

Finds the row with the maximum val. Calculates the non-NULL arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT`
`col2`	`BIGINT`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `argmax(arg = INTEGER, val = INTEGER)`
- `argmax(arg = INTEGER, val = BIGINT)`
- `argmax(arg = INTEGER, val = HUGEINT)`
- `argmax(arg = INTEGER, val = DOUBLE)`
- `argmax(arg = INTEGER, val = VARCHAR)`
- `argmax(arg = INTEGER, val = DATE)`
- `argmax(arg = INTEGER, val = TIMESTAMP)`
- `argmax(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)`
- `argmax(arg = INTEGER, val = BLOB)`
- `argmax(arg = BIGINT, val = INTEGER)`
- `argmax(arg = BIGINT, val = BIGINT)`
- `argmax(arg = BIGINT, val = HUGEINT)`
- `argmax(arg = BIGINT, val = DOUBLE)`
- `argmax(arg = BIGINT, val = VARCHAR)`
- `argmax(arg = BIGINT, val = DATE)`
- `argmax(arg = BIGINT, val = TIMESTAMP)`
- `argmax(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)`
- `argmax(arg = BIGINT, val = BLOB)`
- `argmax(arg = DOUBLE, val = INTEGER)`
- `argmax(arg = DOUBLE, val = BIGINT)`
- `argmax(arg = DOUBLE, val = HUGEINT)`

- argmax(arg = DOUBLE, val = DOUBLE)
- argmax(arg = DOUBLE, val = VARCHAR)
- argmax(arg = DOUBLE, val = DATE)
- argmax(arg = DOUBLE, val = TIMESTAMP)
- argmax(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = DOUBLE, val = BLOB)
- argmax(arg = VARCHAR, val = INTEGER)
- argmax(arg = VARCHAR, val = BIGINT)
- argmax(arg = VARCHAR, val = HUGEINT)
- argmax(arg = VARCHAR, val = DOUBLE)
- argmax(arg = VARCHAR, val = VARCHAR)
- argmax(arg = VARCHAR, val = DATE)
- argmax(arg = VARCHAR, val = TIMESTAMP)
- argmax(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = VARCHAR, val = BLOB)
- argmax(arg = DATE, val = INTEGER)
- argmax(arg = DATE, val = BIGINT)
- argmax(arg = DATE, val = HUGEINT)
- argmax(arg = DATE, val = DOUBLE)
- argmax(arg = DATE, val = VARCHAR)
- argmax(arg = DATE, val = DATE)
- argmax(arg = DATE, val = TIMESTAMP)
- argmax(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = DATE, val = BLOB)
- argmax(arg = TIMESTAMP, val = INTEGER)
- argmax(arg = TIMESTAMP, val = BIGINT)
- argmax(arg = TIMESTAMP, val = HUGEINT)
- argmax(arg = TIMESTAMP, val = DOUBLE)
- argmax(arg = TIMESTAMP, val = VARCHAR)
- argmax(arg = TIMESTAMP, val = DATE)
- argmax(arg = TIMESTAMP, val = TIMESTAMP)
- argmax(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = TIMESTAMP, val = BLOB)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)

- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- argmax(arg = BLOB, val = INTEGER)
- argmax(arg = BLOB, val = BIGINT)
- argmax(arg = BLOB, val = HUGEINT)
- argmax(arg = BLOB, val = DOUBLE)
- argmax(arg = BLOB, val = VARCHAR)
- argmax(arg = BLOB, val = DATE)
- argmax(arg = BLOB, val = TIMESTAMP)
- argmax(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = BLOB, val = BLOB)
- argmax(arg = DECIMAL, val = INTEGER)
- argmax(arg = DECIMAL, val = BIGINT)
- argmax(arg = DECIMAL, val = HUGEINT)
- argmax(arg = DECIMAL, val = DOUBLE)
- argmax(arg = DECIMAL, val = VARCHAR)
- argmax(arg = DECIMAL, val = DATE)
- argmax(arg = DECIMAL, val = TIMESTAMP)
- argmax(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = DECIMAL, val = BLOB)
- argmax(arg = ANY, val = INTEGER)
- argmax(arg = ANY, val = BIGINT)
- argmax(arg = ANY, val = HUGEINT)
- argmax(arg = ANY, val = DOUBLE)
- argmax(arg = ANY, val = VARCHAR)
- argmax(arg = ANY, val = DATE)
- argmax(arg = ANY, val = TIMESTAMP)
- argmax(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- argmax(arg = ANY, val = BLOB)
- argmax(arg = ANY, val = ANY)
- argmax(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
argmax(A, B)
```

`argmin` *DuckDB function argmin*

Description

Finds the row with the minimum val. Calculates the non-NULL arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT`
`col2`	`BIGINT`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `argmin(arg = INTEGER, val = INTEGER)`
- `argmin(arg = INTEGER, val = BIGINT)`
- `argmin(arg = INTEGER, val = HUGEINT)`
- `argmin(arg = INTEGER, val = DOUBLE)`
- `argmin(arg = INTEGER, val = VARCHAR)`
- `argmin(arg = INTEGER, val = DATE)`
- `argmin(arg = INTEGER, val = TIMESTAMP)`
- `argmin(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)`
- `argmin(arg = INTEGER, val = BLOB)`
- `argmin(arg = BIGINT, val = INTEGER)`
- `argmin(arg = BIGINT, val = BIGINT)`
- `argmin(arg = BIGINT, val = HUGEINT)`
- `argmin(arg = BIGINT, val = DOUBLE)`
- `argmin(arg = BIGINT, val = VARCHAR)`
- `argmin(arg = BIGINT, val = DATE)`
- `argmin(arg = BIGINT, val = TIMESTAMP)`
- `argmin(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)`
- `argmin(arg = BIGINT, val = BLOB)`
- `argmin(arg = DOUBLE, val = INTEGER)`
- `argmin(arg = DOUBLE, val = BIGINT)`
- `argmin(arg = DOUBLE, val = HUGEINT)`
- `argmin(arg = DOUBLE, val = DOUBLE)`

- argmin(arg = DOUBLE, val = VARCHAR)
- argmin(arg = DOUBLE, val = DATE)
- argmin(arg = DOUBLE, val = TIMESTAMP)
- argmin(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = DOUBLE, val = BLOB)
- argmin(arg = VARCHAR, val = INTEGER)
- argmin(arg = VARCHAR, val = BIGINT)
- argmin(arg = VARCHAR, val = HUGEINT)
- argmin(arg = VARCHAR, val = DOUBLE)
- argmin(arg = VARCHAR, val = VARCHAR)
- argmin(arg = VARCHAR, val = DATE)
- argmin(arg = VARCHAR, val = TIMESTAMP)
- argmin(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = VARCHAR, val = BLOB)
- argmin(arg = DATE, val = INTEGER)
- argmin(arg = DATE, val = BIGINT)
- argmin(arg = DATE, val = HUGEINT)
- argmin(arg = DATE, val = DOUBLE)
- argmin(arg = DATE, val = VARCHAR)
- argmin(arg = DATE, val = DATE)
- argmin(arg = DATE, val = TIMESTAMP)
- argmin(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = DATE, val = BLOB)
- argmin(arg = TIMESTAMP, val = INTEGER)
- argmin(arg = TIMESTAMP, val = BIGINT)
- argmin(arg = TIMESTAMP, val = HUGEINT)
- argmin(arg = TIMESTAMP, val = DOUBLE)
- argmin(arg = TIMESTAMP, val = VARCHAR)
- argmin(arg = TIMESTAMP, val = DATE)
- argmin(arg = TIMESTAMP, val = TIMESTAMP)
- argmin(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = TIMESTAMP, val = BLOB)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)

- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- argmin(arg = BLOB, val = INTEGER)
- argmin(arg = BLOB, val = BIGINT)
- argmin(arg = BLOB, val = HUGEINT)
- argmin(arg = BLOB, val = DOUBLE)
- argmin(arg = BLOB, val = VARCHAR)
- argmin(arg = BLOB, val = DATE)
- argmin(arg = BLOB, val = TIMESTAMP)
- argmin(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = BLOB, val = BLOB)
- argmin(arg = DECIMAL, val = INTEGER)
- argmin(arg = DECIMAL, val = BIGINT)
- argmin(arg = DECIMAL, val = HUGEINT)
- argmin(arg = DECIMAL, val = DOUBLE)
- argmin(arg = DECIMAL, val = VARCHAR)
- argmin(arg = DECIMAL, val = DATE)
- argmin(arg = DECIMAL, val = TIMESTAMP)
- argmin(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = DECIMAL, val = BLOB)
- argmin(arg = ANY, val = INTEGER)
- argmin(arg = ANY, val = BIGINT)
- argmin(arg = ANY, val = HUGEINT)
- argmin(arg = ANY, val = DOUBLE)
- argmin(arg = ANY, val = VARCHAR)
- argmin(arg = ANY, val = DATE)
- argmin(arg = ANY, val = TIMESTAMP)
- argmin(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- argmin(arg = ANY, val = BLOB)
- argmin(arg = ANY, val = ANY)
- argmin(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
argmin(A, B)
```

`array_agg` *DuckDB function array_agg*

Description

Returns a LIST containing all the values of a column.

Usage

```
array_agg(arg = T)
```

Arguments

arg	T

Value

`T[]`

SQL examples

```
array_agg(A)
```

`array_aggr` *DuckDB function array_aggr*

Description

Executes the aggregate function `function_name` on the elements of `list`.

Usage

```
array_aggr(list = `ANY[]`, function_name = VARCHAR)
```

Arguments

list	ANY[]
function_name	VARCHAR

Value

`ANY`

SQL examples

```
array_aggr([1, 2, NULL], 'min')
```

`array_aggregate` *DuckDB function array_aggregate*

Description

Executes the aggregate function `function_name` on the elements of `list`.

Usage

```
array_aggregate(list = `ANY[]`, function_name = VARCHAR)
```

Arguments

`list`	`ANY[]`
`function_name`	`VARCHAR`

Value

`ANY`

SQL examples

```
array_aggregate([1, 2, NULL], 'min')
```

`array_append` *DuckDB function array_append*

Description

DuckDB macro `array_append()`.

Usage

```
array_append(arr, el)
```

Arguments

`arr`	Unspecified.
`el`	Unspecified.

Value

Unspecified.

`array_apply` *DuckDB function array_apply*

Description

Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function.

Usage

```
array_apply(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
array_apply([1, 2, 3], lambda x : x + 1)
```

`array_cat` *DuckDB function array_cat*

Description

Concatenates lists. `NULL` inputs are skipped. See also operator `||`.

Usage

```
array_cat()
```

Value

`ANY[]`

SQL examples

```
array_cat([2, 3], [4, 5, 6], [7])
```

`array_concat` *DuckDB function array_concat*

Description

Concatenates lists. `NULL` inputs are skipped. See also operator `||`.

Usage

```
array_concat()
```

Value

`ANY[]`

SQL examples

```
array_concat([2, 3], [4, 5, 6], [7])
```

`array_contains` *DuckDB function array_contains*

Description

Returns true if the list contains the element.

Usage

```
array_contains(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

`BOOLEAN`

SQL examples

```
array_contains([1, 2, NULL], 1)
```

`array_cosine_distance`

DuckDB function array_cosine_distance

Description

Computes the cosine distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.

Arguments

`array1`	`FLOAT[ANY] | DOUBLE[ANY]`
`array2`	`FLOAT[ANY] | DOUBLE[ANY]`

Value

`FLOAT | DOUBLE`

Overloads

- `array_cosine_distance(array1 = \`FLOAT[ANY]\`, array2 = \`FLOAT[ANY]\`)`
- `array_cosine_distance(array1 = \`DOUBLE[ANY]\`, array2 = \`DOUBLE[ANY]\`)`

SQL examples

```
array_cosine_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT,
```

`array_cosine_similarity`

DuckDB function array_cosine_similarity

Description

Computes the cosine similarity between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.

Arguments

`array1`	`FLOAT[ANY] | DOUBLE[ANY]`
`array2`	`FLOAT[ANY] | DOUBLE[ANY]`

Value

`FLOAT | DOUBLE`

Overloads

- `array_cosine_similarity(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)`
- `array_cosine_similarity(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)`

SQL examples

```
array_cosine_similarity(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOA
```

---

array_cross_product *DuckDB function array_cross_product*

Description

Computes the cross product of two arrays of size 3. The array elements can not be NULL.

Arguments

array	FLOAT[3] | DOUBLE[3]

Value

FLOAT[3] | DOUBLE[3]

Overloads

- `array_cross_product(array = `FLOAT[3]`, array = `FLOAT[3]`)`
- `array_cross_product(array = `DOUBLE[3]`, array = `DOUBLE[3]`)`

SQL examples

```
array_cross_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3
```

---

array_distance *DuckDB function array_distance*

Description

Computes the distance between two arrays of the same size. The array elements can not be NULL. The arrays can have any size as long as the size is the same for both arguments.

Arguments

array1	FLOAT[ANY] | DOUBLE[ANY]
array2	FLOAT[ANY] | DOUBLE[ANY]

Value

FLOAT | DOUBLE

Overloads

- array_distance(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)
- array_distance(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)

SQL examples

```
array_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FL
```

---

array_distinct *DuckDB function array_distinct*

Description

Removes all duplicates and NULL values from a list. Does not preserve the original order.

Usage

```
array_distinct(list = `T[]`)
```

Arguments

list	T[]

Value

T[]

SQL examples

```
array_distinct([1, 1, NULL, -3, 1, 5])
```

`array_dot_product` *DuckDB function array_dot_product*

Description

Computes the inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.

Arguments

`array1`	`FLOAT[ANY] | DOUBLE[ANY]`
`array2`	`FLOAT[ANY] | DOUBLE[ANY]`

Value

`FLOAT | DOUBLE`

Overloads

- `array_dot_product(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)`
- `array_dot_product(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)`

SQL examples

```
array_dot_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0
```

`array_extract` *DuckDB function array_extract*

Description

Extracts a single character from a `string` using a (1-based) `index`.

Extracts the named `entry` from the `STRUCT`.

Extracts the entry from an unnamed `STRUCT` (tuple) using an index (1-based).

Arguments

`col0`	`T[]`
`col1`	`BIGINT`
`string`	`VARCHAR`
`index`	`BIGINT`
`struct`	`STRUCT`
`entry`	`VARCHAR`

Value

`T | VARCHAR | ANY`

Overloads

- `array_extract(col0 = \`T[]\`, col1 = BIGINT)`
- `array_extract(string = VARCHAR, index = BIGINT)`
- `array_extract(struct = STRUCT, entry = VARCHAR)`
- `array_extract(struct = STRUCT, index = BIGINT)`

SQL examples

```
array_extract('DuckDB', 2)
array_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')
array_extract(row(42, 84), 1)
```

---

array_filter *DuckDB function array_filter*

Description

Constructs a list from those elements of the input `list` for which the `lambda` function returns `true`. DuckDB must be able to cast the `lambda` function's return type to `BOOL`. The return type of `list_filter` is the same as the input list's.

Usage

```
array_filter(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
array_filter([3, 4, 5], lambda x : x > 4)
```

`array_grade_up` *DuckDB function array_grade_up*

Description

Works like list_sort, but the results are the indexes that correspond to the position in the original list instead of the actual values.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`
`col2`	`VARCHAR`

Value

`ANY[]`

Overloads

- `array_grade_up(list = \`ANY[]\`)`
- `array_grade_up(list = \`ANY[]\`, col1 = VARCHAR)`
- `array_grade_up(list = \`ANY[]\`, col1 = VARCHAR, col2 = VARCHAR)`

SQL examples

```
array_grade_up([3, 6, 1, 2])
```

`array_has` *DuckDB function array_has*

Description

Returns true if the list contains the element.

Usage

```
array_has(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

BOOLEAN

SQL examples

```
array_has([1, 2, NULL], 1)
```

---

array_has_all *DuckDB function array_has_all*

Description

Returns true if all elements of list2 are in list1. NULLs are ignored.

Usage

```
array_has_all(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
array_has_all([1, 2, 3], [2, 3])
```

---

array_has_any *DuckDB function array_has_any*

Description

Returns true if the lists have any element in common. NULLs are ignored.

Usage

```
array_has_any(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
array_has_any([1, 2, 3], [2, 3, 4])
```

`array_indexof` *DuckDB function array_indexof*

Description

Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.

Usage

```
array_indexof(list = `T[]`, element = T)
```

Arguments

list	T[]
element	T

Value

INTEGER

SQL examples

```
array_indexof([1, 2, NULL], 2)
```

`array_inner_product` *DuckDB function array_inner_product*

Description

Computes the inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.

Arguments

array1	FLOAT[ANY] | DOUBLE[ANY]
array2	FLOAT[ANY] | DOUBLE[ANY]

Value

FLOAT | DOUBLE

Overloads

- array_inner_product(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)
- array_inner_product(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)

SQL examples

```
array_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3
```

---

array_intersect *DuckDB function array_intersect*

Description

DuckDB macro array_intersect().

Usage

```
array_intersect(l1, l2)
```

Arguments

l1	Unspecified.
l2	Unspecified.

Value

Unspecified.

---

array_length *DuckDB function array_length*

Description

Returns the length of the list.

array_length for lists with dimensions other than 1 not implemented.

Arguments

list	ANY[]
dimension	BIGINT

Value

BIGINT

Overloads

- array_length(list = `ANY[]`)
- array_length(list = `ANY[]`, dimension = BIGINT)

SQL examples

```
array_length([1, 2, 3])
```

---

array_negative_dot_product

DuckDB function array_negative_dot_product

Description

Computes the negative inner product between two arrays of the same size. The array elements can not be NULL. The arrays can have any size as long as the size is the same for both arguments.

Arguments

array1	FLOAT[ANY] | DOUBLE[ANY]
array2	FLOAT[ANY] | DOUBLE[ANY]

Value

FLOAT | DOUBLE

Overloads

- array_negative_dot_product(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)
- array_negative_dot_product(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)

SQL examples

```
array_negative_dot_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FL
```

`array_negative_inner_product`
DuckDB function array_negative_inner_product

Description

Computes the negative inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.

Arguments

`array1`	`FLOAT[ANY] | DOUBLE[ANY]`
`array2`	`FLOAT[ANY] | DOUBLE[ANY]`

Value

`FLOAT | DOUBLE`

Overloads

- `array_negative_inner_product(array1 = `FLOAT[ANY]`, array2 = `FLOAT[ANY]`)`
- `array_negative_inner_product(array1 = `DOUBLE[ANY]`, array2 = `DOUBLE[ANY]`)`

SQL examples

```
array_negative_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0:
```

`array_pop_back` *DuckDB function array_pop_back*

Description

DuckDB macro `array_pop_back()`.

Usage

```
array_pop_back(arr)
```

Arguments

`arr`	Unspecified.

Value

Unspecified.

`array_pop_front` *DuckDB function array_pop_front*

Description

DuckDB macro `array_pop_front()`.

Usage

```
array_pop_front(arr)
```

Arguments

`arr`	Unspecified.

Value

Unspecified.

`array_position` *DuckDB function array_position*

Description

Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.

Usage

```
array_position(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

`INTEGER`

SQL examples

```
array_position([1, 2, NULL], 2)
```

array_prepend *DuckDB function array_prepend*

Description

DuckDB macro `array_prepend()`.

Usage

```
array_prepend(el, arr)
```

Arguments

`el`	Unspecified.
`arr`	Unspecified.

Value

Unspecified.

array_push_back *DuckDB function array_push_back*

Description

DuckDB macro `array_push_back()`.

Usage

```
array_push_back(arr, e)
```

Arguments

`arr`	Unspecified.
`e`	Unspecified.

Value

Unspecified.

`array_push_front` *DuckDB function array_push_front*

Description

DuckDB macro `array_push_front()`.

Usage

```
array_push_front(arr, e)
```

Arguments

`arr`	Unspecified.
`e`	Unspecified.

Value

Unspecified.

`array_reduce` *DuckDB function array_reduce*

Description

Reduces all elements of the input `list` into a single scalar value by executing the `lambda` function on a running result and the next list element. The `lambda` function has an optional `initial_value` argument.

Arguments

`list`	`ANY[]`
`initial_value`	`ANY`
`lambda(x, y)`	`LAMBDA`

Value

`ANY`

Overloads

- `array_reduce(list = `ANY[]`, `lambda(x,y)` = LAMBDA)`
- `array_reduce(list = `ANY[]`, `lambda(x,y)` = LAMBDA, initial_value = ANY)`

SQL examples

```
array_reduce([1, 2, 3], lambda x, y : x + y)
```

`array_resize` *DuckDB function array_resize*

Description

Resizes the `list` to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.

Arguments

`list`	`ANY[]`
`size[`	`ANY`
`value]`	`ANY`

Value

`ANY[]`

Overloads

- `array_resize(list = `ANY[]`, `size[` = ANY)`
- `array_resize(list = `ANY[]`, `size[` = ANY, `value]` = ANY)`

SQL examples

```
array_resize([1, 2, 3], 5, 0)
```

`array_reverse` *DuckDB function array_reverse*

Description

DuckDB macro `array_reverse()`.

Usage

```
array_reverse(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`array_reverse_sort` *DuckDB function array_reverse_sort*

Description

Sorts the elements of the list in reverse order.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`

Value

`ANY[]`

Overloads

- `array_reverse_sort(list = `ANY[]`)`
- `array_reverse_sort(list = `ANY[]`, col1 = VARCHAR)`

SQL examples

```
array_reverse_sort([3, 6, 1, 2])
```

`array_select` *DuckDB function array_select*

Description

Returns a list based on the elements selected by the `index_list`.

Usage

```
array_select(value_list = `T[]`, index_list = `BIGINT[]`)
```

Arguments

`value_list`	`T[]`
`index_list`	`BIGINT[]`

Value

`T[]`

SQL examples

```
array_select([10, 20, 30, 40], [1, 4])
```

`array_slice` *DuckDB function array_slice*

Description

Extracts a sublist or substring using slice conventions. Negative values are accepted. list_slice with added step feature.

Arguments

`list`	`ANY`
`begin`	`ANY`
`end`	`ANY`
`step`	`BIGINT`

Value

`ANY`

Overloads

- `array_slice(list = ANY, begin = ANY, end = ANY)`
- `array_slice(list = ANY, begin = ANY, end = ANY, step = BIGINT)`

SQL examples

```
array_slice('DuckDB', 3, 4)
array_slice('DuckDB', 3, NULL)
array_slice('DuckDB', 0, -3)
array_slice([4, 5, 6], 1, 3, 2)
```

`array_sort` *DuckDB function array_sort*

Description

Sorts the elements of the list.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`
`col2`	`VARCHAR`

Value

`ANY[]`

Overloads

- `array_sort(list = `ANY[]`)`
- `array_sort(list = `ANY[]`, col1 = VARCHAR)`
- `array_sort(list = `ANY[]`, col1 = VARCHAR, col2 = VARCHAR)`

SQL examples

```
array_sort([3, 6, 1, 2])
```

`array_to_string` *DuckDB function array_to_string*

Description

DuckDB macro `array_to_string()`.

Usage

```
array_to_string(arr, sep)
```

Arguments

`arr`	Unspecified.
`sep`	Unspecified.

Value

Unspecified.

`array_to_string_comma_default` *DuckDB function array_to_string_comma_default*

Description

DuckDB macro `array_to_string_comma_default()`.

Usage

```
array_to_string_comma_default(arr, sep)
```

Arguments

arr	Unspecified.
sep	Unspecified.

Value

Unspecified.

---

array_transform *DuckDB function array_transform*

Description

Returns a list that is the result of applying the lambda function to each element of the input list. The return type is defined by the return type of the lambda function.

Usage

```
array_transform(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

list	ANY[]
lambda(x)	LAMBDA

Value

ANY[]

SQL examples

```
array_transform([1, 2, 3], lambda x : x + 1)
```

---

array_unique *DuckDB function array_unique*

Description

Counts the unique elements of a list.

Usage

```
array_unique(list = `ANY[]`)
```

Arguments

`list`	`ANY[]`

Value

`UBIGINT`

SQL examples

```
array_unique([1, 1, NULL, -3, 1, 5])
```

`array_value` *DuckDB function array_value*

Description

Creates an `ARRAY` containing the argument values.

Usage

```
array_value()
```

Value

`ARRAY`

SQL examples

```
array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)
```

`array_where` *DuckDB function array_where*

Description

Returns a list with the `BOOLEAN`s in `mask_list` applied as a mask to the `value_list`.

Usage

```
array_where(value_list = `T[]`, mask_list = `BOOLEAN[]`)
```

Arguments

`value_list`	`T[]`
`mask_list`	`BOOLEAN[]`

Value

T[]

SQL examples

```
array_where([10, 20, 30, 40], [true, false, false, true])
```

---

array_zip *DuckDB function array_zip*

Description

Zips n LISTs to a new LIST whose length will be that of the longest list. Its elements are structs of n elements from each list list_1, …, list_n, missing elements are replaced with NULL. If truncate is set, all lists are truncated to the smallest list length.

Usage

```
array_zip()
```

Value

STRUCT[]

SQL examples

```
array_zip([1, 2], [3, 4], [5, 6])
array_zip([1, 2], [3, 4], [5, 6, 7])
array_zip([1, 2], [3, 4], [5, 6, 7], true)
```

---

arrow_scan *DuckDB function arrow_scan*

Description

DuckDB function arrow_scan().

Usage

```
arrow_scan(col0 = POINTER, col1 = POINTER, col2 = POINTER)
```

Arguments

col0	POINTER
col1	POINTER
col2	POINTER

Value

Unspecified.

`arrow_scan_dumb` *DuckDB function arrow_scan_dumb*

Description

DuckDB function `arrow_scan_dumb()`.

Usage

```
arrow_scan_dumb(col0 = POINTER, col1 = POINTER, col2 = POINTER)
```

Arguments

`col0`	`POINTER`
`col1`	`POINTER`
`col2`	`POINTER`

Value

Unspecified.

`ascii` *DuckDB function ascii*

Description

Returns an integer that represents the Unicode code point of the first character of the `string`.

Usage

```
ascii(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`INTEGER`

SQL examples

```
ascii('Ω')
```

`asin` *DuckDB function asin*

Description

Computes the arcsine of x.

Usage

```
asin(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
asin(0.5)
```

`asinh` *DuckDB function asinh*

Description

Computes the inverse hyperbolic sin of x.

Usage

```
asinh(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
asinh(0.5)
```

`at-` *DuckDB function @*

Description

Absolute value.

Arguments

x	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Overloads

- `` `@`(x = TINYINT) ``
- `` `@`(x = SMALLINT) ``
- `` `@`(x = INTEGER) ``
- `` `@`(x = BIGINT) ``
- `` `@`(x = HUGEINT) ``
- `` `@`(x = FLOAT) ``
- `` `@`(x = DOUBLE) ``
- `` `@`(x = DECIMAL) ``
- `` `@`(x = UTINYINT) ``
- `` `@`(x = USMALLINT) ``
- `` `@`(x = UINTEGER) ``
- `` `@`(x = UBIGINT) ``
- `` `@`(x = UHUGEINT) ``

SQL examples

```
abs(-17.4)
```

`at->` *DuckDB function @>*

Description

Returns true if all elements of list2 are in list1. NULLs are ignored.

Usage

```
`@>`(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
list_has_all([1, 2, 3], [2, 3])
```

`atan` *DuckDB function atan*

Description

Computes the arctangent of x.

Usage

```
atan(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
atan(0.5)
```

`atan2` *DuckDB function atan2*

Description

Computes the arctangent (y, x).

Usage

```
atan2(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

`DOUBLE`

SQL examples

```
atan2(1.0, 0.0)
```

`atanh` *DuckDB function atanh*

Description

Computes the inverse hyperbolic tan of x.

Usage

```
atanh(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

`DOUBLE`

SQL examples

```
atanh(0.5)
```

`avg` *DuckDB function avg*

Description

Calculates the average value for all tuples in x.

Arguments

x	DECIMAL | SMALLINT | INTEGER | BIGINT | HUGEINT | INTERVAL | DOUBLE | TIMESTAMP

Value

DECIMAL | DOUBLE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE | TIME | TIME WITH TIME ZONE

Overloads

- avg(x = DECIMAL)
- avg(x = SMALLINT)
- avg(x = INTEGER)
- avg(x = BIGINT)
- avg(x = HUGEINT)
- avg(x = INTERVAL)
- avg(x = DOUBLE)
- avg(x = TIMESTAMP)
- avg(x = `TIMESTAMP WITH TIME ZONE`)
- avg(x = TIME)
- avg(x = `TIME WITH TIME ZONE`)

SQL examples

```
SUM(x) / COUNT(*)
```

`bar` *DuckDB function bar*

Description

Draws a band whose width is proportional to (`x - min`) and equal to `width` characters when `x = max`. `width` defaults to 80.

Arguments

x	DOUBLE
min	DOUBLE
max	DOUBLE
width	DOUBLE

Value

VARCHAR

Overloads

- bar(x = DOUBLE, min = DOUBLE, max = DOUBLE, width = DOUBLE)
- bar(x = DOUBLE, min = DOUBLE, max = DOUBLE)

SQL examples

```
bar(5, 0, 20, 10)
```

---

base64 *DuckDB function base64*

Description

Converts a blob to a base64 encoded string.

Usage

```
base64(blob = BLOB)
```

Arguments

blob	BLOB

Value

VARCHAR

SQL examples

```
base64('A'::BLOB)
```

`bin` *DuckDB function bin*

Description

Converts the `string` to binary representation.

Converts the `value` to binary representation.

Arguments

`string`	`VARCHAR`
`value`	`BIGNUM | UBIGINT | BIGINT | HUGEINT | UHUGEINT`

Value

`VARCHAR`

Overloads

- `bin(string = VARCHAR)`
- `bin(value = BIGNUM)`
- `bin(value = UBIGINT)`
- `bin(value = BIGINT)`
- `bin(value = HUGEINT)`
- `bin(value = UHUGEINT)`

SQL examples

```
bin('Aa')
bin(42)
```

`bit_and` *DuckDB function bit_and*

Description

Returns the bitwise AND of all bits in a given expression.

Arguments

`arg`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT

Overloads

- bit_and(arg = TINYINT)
- bit_and(arg = SMALLINT)
- bit_and(arg = INTEGER)
- bit_and(arg = BIGINT)
- bit_and(arg = HUGEINT)
- bit_and(arg = UTINYINT)
- bit_and(arg = USMALLINT)
- bit_and(arg = UINTEGER)
- bit_and(arg = UBIGINT)
- bit_and(arg = UHUGEINT)
- bit_and(arg = BIT)

SQL examples

```
bit_and(A)
```

---

bit_count *DuckDB function bit_count*

Description

Returns the number of bits that are set.

Arguments

x	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | BIT

Value

TINYINT | BIGINT

Overloads

- bit_count(x = TINYINT)
- bit_count(x = SMALLINT)
- bit_count(x = INTEGER)
- bit_count(x = BIGINT)
- bit_count(x = HUGEINT)
- bit_count(x = BIT)

SQL examples

```
bit_count(31)
```

`bit_length` *DuckDB function bit_length*

Description

Number of bits in a `string`.

Returns the bit-length of the `bit` argument.

Arguments

`string`	`VARCHAR`
`bit`	`BIT`

Value

`BIGINT`

Overloads

- `bit_length(string = VARCHAR)`
- `bit_length(bit = BIT)`

SQL examples

```
bit_length('abc')
bit_length(42::TINYINT::BIT)
```

`bit_or` *DuckDB function bit_or*

Description

Returns the bitwise OR of all bits in a given expression.

Arguments

`arg`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Overloads

- `bit_or(arg = TINYINT)`
- `bit_or(arg = SMALLINT)`
- `bit_or(arg = INTEGER)`
- `bit_or(arg = BIGINT)`
- `bit_or(arg = HUGEINT)`
- `bit_or(arg = UTINYINT)`
- `bit_or(arg = USMALLINT)`
- `bit_or(arg = UINTEGER)`
- `bit_or(arg = UBIGINT)`
- `bit_or(arg = UHUGEINT)`
- `bit_or(arg = BIT)`

SQL examples

```
bit_or(A)
```

`bit_position` *DuckDB function bit_position*

Description

Returns first starting index of the specified substring within bits, or zero if it is not present. The first (leftmost) bit is indexed 1.

Usage

```
bit_position(substring = BIT, bitstring = BIT)
```

Arguments

`substring`	`BIT`
`bitstring`	`BIT`

Value

`INTEGER`

SQL examples

```
bit_position('010'::BIT, '1110101'::BIT)
```

`bit_xor` *DuckDB function bit_xor*

Description

Returns the bitwise XOR of all bits in a given expression.

Arguments

`arg`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Overloads

- `bit_xor(arg = TINYINT)`
- `bit_xor(arg = SMALLINT)`
- `bit_xor(arg = INTEGER)`
- `bit_xor(arg = BIGINT)`
- `bit_xor(arg = HUGEINT)`
- `bit_xor(arg = UTINYINT)`
- `bit_xor(arg = USMALLINT)`
- `bit_xor(arg = UINTEGER)`
- `bit_xor(arg = UBIGINT)`
- `bit_xor(arg = UHUGEINT)`
- `bit_xor(arg = BIT)`

SQL examples

```
bit_xor(A)
```

`bitstring` *DuckDB function bitstring*

Description

Pads the bitstring until the specified length.

Arguments

`bitstring`	`VARCHAR | BIT`
`length`	`INTEGER`

Value

`BIT`

Overloads

- `bitstring(bitstring = VARCHAR, length = INTEGER)`
- `bitstring(bitstring = BIT, length = INTEGER)`

SQL examples

```
bitstring('1010'::BIT, 7)
```

`bitstring_agg` *DuckDB function bitstring_agg*

Description

Returns a bitstring with bits set for each distinct value.

Arguments

`arg`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`
`col1`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`
`col2`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`

Value

`BIT`

Overloads

- bitstring_agg(arg = TINYINT)
- bitstring_agg(arg = TINYINT, col1 = TINYINT, col2 = TINYINT)
- bitstring_agg(arg = SMALLINT)
- bitstring_agg(arg = SMALLINT, col1 = SMALLINT, col2 = SMALLINT)
- bitstring_agg(arg = INTEGER)
- bitstring_agg(arg = INTEGER, col1 = INTEGER, col2 = INTEGER)
- bitstring_agg(arg = BIGINT)
- bitstring_agg(arg = BIGINT, col1 = BIGINT, col2 = BIGINT)
- bitstring_agg(arg = HUGEINT)
- bitstring_agg(arg = HUGEINT, col1 = HUGEINT, col2 = HUGEINT)
- bitstring_agg(arg = UTINYINT)
- bitstring_agg(arg = UTINYINT, col1 = UTINYINT, col2 = UTINYINT)
- bitstring_agg(arg = USMALLINT)
- bitstring_agg(arg = USMALLINT, col1 = USMALLINT, col2 = USMALLINT)
- bitstring_agg(arg = UINTEGER)
- bitstring_agg(arg = UINTEGER, col1 = UINTEGER, col2 = UINTEGER)
- bitstring_agg(arg = UBIGINT)
- bitstring_agg(arg = UBIGINT, col1 = UBIGINT, col2 = UBIGINT)
- bitstring_agg(arg = UHUGEINT)
- bitstring_agg(arg = UHUGEINT, col1 = UHUGEINT, col2 = UHUGEINT)

SQL examples

```
bitstring_agg(A)
```

---

bool_and *DuckDB function bool_and*

Description

Returns TRUE if every input value is TRUE, otherwise FALSE.

Usage

```
bool_and(arg = BOOLEAN)
```

Arguments

arg	BOOLEAN

Value

BOOLEAN

SQL examples

```
bool_and(A)
```

`bool_or` *DuckDB function bool_or*

Description

Returns TRUE if any input value is TRUE, otherwise FALSE.

Usage

```
bool_or(arg = BOOLEAN)
```

Arguments

arg	BOOLEAN

Value

BOOLEAN

SQL examples

```
bool_or(A)
```

`can_cast_implicitly` *DuckDB function can_cast_implicitly*

Description

Whether or not we can implicitly cast from the source type to the other type.

Usage

```
can_cast_implicitly(source_type = ANY, target_type = ANY)
```

Arguments

source_type	ANY
target_type	ANY

Value

BOOLEAN

SQL examples

```
can_cast_implicitly(NULL::INTEGER, NULL::BIGINT)
```

---

cardinality *DuckDB function cardinality*

Description

Returns the size of the map (or the number of entries in the map).

Usage

```
cardinality(map = ANY)
```

Arguments

map	ANY

Value

UBIGINT

SQL examples

```
cardinality( map([4, 2], ['a', 'b']) );
```

---

cast_to_type *DuckDB function cast_to_type*

Description

Casts the first argument to the type of the second argument.

Usage

```
cast_to_type(param = ANY, type = ANY)
```

Arguments

param	ANY
type	ANY

Value

ANY

SQL examples

```
cast_to_type('42', NULL::INTEGER)
```

`cbrt` *DuckDB function cbrt*

Description

Returns the cube root of x.

Usage

```
cbrt(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
cbrt(8)
```

`ceil` *DuckDB function ceil*

Description

Rounds the number up.

Arguments

x	FLOAT | DOUBLE | DECIMAL

Value

FLOAT | DOUBLE | DECIMAL

Overloads

- `ceil(x = FLOAT)`
- `ceil(x = DOUBLE)`
- `ceil(x = DECIMAL)`

SQL examples

```
ceil(17.4)
```

`ceiling` *DuckDB function ceiling*

Description

Rounds the number up.

Arguments

`x`	`FLOAT | DOUBLE | DECIMAL`

Value

`FLOAT | DOUBLE | DECIMAL`

Overloads

- `ceiling(x = FLOAT)`
- `ceiling(x = DOUBLE)`
- `ceiling(x = DECIMAL)`

SQL examples

```
ceiling(17.4)
```

`century` *DuckDB function century*

Description

Extract the century component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `century(ts = DATE)`
- `century(ts = INTERVAL)`
- `century(ts = TIMESTAMP)`
- `century(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
century(timestamp '2021-08-03 11:59:44.123456')
```

`char_length` *DuckDB function char_length*

Description

Number of characters in `string`.

Returns the bit-length of the `bit` argument.

Returns the length of the `list`.

Arguments

`string`	`VARCHAR`
`bit`	`BIT`
`list`	`ANY[]`

Value

`BIGINT`

Overloads

- `char_length(string = VARCHAR)`
- `char_length(bit = BIT)`
- `char_length(list = `ANY[]`)`

SQL examples

```
char_length('Hello ')
char_length(42::TINYINT::BIT)
char_length([1,2,3])
```

`character_length` *DuckDB function character_length*

Description

Number of characters in `string`.

Returns the bit-length of the `bit` argument.

Returns the length of the `list`.

Arguments

`string`	`VARCHAR`
`bit`	`BIT`
`list`	`ANY[]`

Value

`BIGINT`

Overloads

- `character_length(string = VARCHAR)`
- `character_length(bit = BIT)`
- `character_length(list = `ANY[]`)`

SQL examples

```
character_length('Hello ')
character_length(42::TINYINT::BIT)
character_length([1,2,3])
```

`checkpoint` *DuckDB function checkpoint*

Description

DuckDB function `checkpoint()`.

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

Overloads

- `checkpoint()`
- `checkpoint(col0 = VARCHAR)`

`chr` *DuckDB function chr*

Description

Returns a character which is corresponding the ASCII code value or Unicode code point.

Usage

```
chr(code_point = INTEGER)
```

Arguments

`code_point`	`INTEGER`

Value

`VARCHAR`

SQL examples

```
chr(65)
```

col_description *DuckDB function col_description*

Description

DuckDB macro `col_description()`.

Usage

```
col_description(table_oid, column_number)
```

Arguments

`table_oid`	Unspecified.
`column_number`	Unspecified.

Value

Unspecified.

collations *DuckDB function collations*

Description

DuckDB function `collations()`.

Usage

```
collations()
```

Value

Unspecified.

`combine` *DuckDB function combine*

Description

DuckDB function `combine()`.

Usage

```
combine(col0 = `AGGREGATE_STATE<?>`, col1 = ANY)
```

Arguments

col0	AGGREGATE_STATE<?>
col1	ANY

Value

AGGREGATE_STATE<?>

`concat` *DuckDB function concat*

Description

Concatenates multiple strings or lists. `NULL` inputs are skipped. See also operator `||`.

Usage

```
concat(value = ANY)
```

Arguments

value	ANY

Value

ANY

SQL examples

```
concat('Hello', ' ', 'World')
concat([1, 2, 3], NULL, [4, 5, 6])
```

`concat_ws` *DuckDB function concat_ws*

Description

Concatenates many strings, separated by `separator`. `NULL` inputs are skipped.

Usage

```
concat_ws(separator = VARCHAR, string = ANY)
```

Arguments

`separator`	`VARCHAR`
`string`	`ANY`

Value

`VARCHAR`

SQL examples

```
concat_ws(', ', 'Banana', 'Apple', 'Melon')
```

`constant_or_null` *DuckDB function constant_or_null*

Description

If arg2 is NULL, return NULL. Otherwise, return arg1.

Usage

```
constant_or_null(arg1 = ANY, arg2 = ANY)
```

Arguments

`arg1`	`ANY`
`arg2`	`ANY`

Value

`ANY`

SQL examples

```
constant_or_null(42, NULL)
```

contains *DuckDB function contains*

Description

Returns `true` if `search_string` is found within `string`.

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`
`col0`	`T[] | MAP(K, V) | STRUCT`
`col1`	`T | K | ANY`

Value

`BOOLEAN`

Overloads

- `contains(string = VARCHAR, search_string = VARCHAR)`
- `contains(col0 = \`T[]\`, col1 = T)`
- `contains(col0 = \`MAP(K, V)\`, col1 = K)`
- `contains(col0 = STRUCT, col1 = ANY)`

SQL examples

```
contains('abc', 'a')
```

copy_database *DuckDB function copy_database*

Description

DuckDB function `copy_database()`.

Usage

```
copy_database(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`
`col1`	`VARCHAR`

Value

Unspecified.

`corr` *DuckDB function corr*

Description

Returns the correlation coefficient for non-NULL pairs in a group.

Usage

```
corr(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
COVAR_POP(y, x) / (STDDEV_POP(x) * STDDEV_POP(y))
```

`cos` *DuckDB function cos*

Description

Computes the cos of x.

Usage

```
cos(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
cos(90)
```

`cosh` *DuckDB function cosh*

Description

Computes the hyperbolic cos of x.

Usage

```
cosh(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
cosh(1)
```

`cot` *DuckDB function cot*

Description

Computes the cotangent of x.

Usage

```
cot(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
cot(0.5)
```

`count` *DuckDB function count*

Description

Returns the number of non-NULL values in arg.

Arguments

`arg`	`ANY`

Value

`BIGINT`

Overloads

- `count(arg = ANY)`
- `count()`

SQL examples

```
count(A)
```

`count_if` *DuckDB function count_if*

Description

Counts the total number of TRUE values for a boolean column.

Usage

```
count_if(arg = BOOLEAN)
```

Arguments

`arg`	`BOOLEAN`

Value

`HUGEINT`

SQL examples

```
count_if(A)
```

count_star *DuckDB function count_star*

Description

DuckDB function `count_star()`.

Usage

```
count_star()
```

Value

```
BIGINT
```

countif *DuckDB function countif*

Description

Counts the total number of TRUE values for a boolean column.

Usage

```
countif(arg = BOOLEAN)
```

Arguments

```
arg             BOOLEAN
```

Value

```
HUGEINT
```

SQL examples

```
countif(A)
```

`covar_pop` *DuckDB function covar_pop*

Description

Returns the population covariance of input values.

Usage

```
covar_pop(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / COUNT(*)
```

`covar_samp` *DuckDB function covar_samp*

Description

Returns the sample covariance for non-NULL pairs in a group.

Usage

```
covar_samp(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / (COUNT(*) - 1)
```

`create_sort_key` *DuckDB function create_sort_key*

Description

Constructs a binary-comparable sort key based on a set of input parameters and sort qualifiers.

Usage

```
create_sort_key(parameters... = ANY)
```

Arguments

```
parameters...  ANY
```

Value

```
BLOB
```

SQL examples

```
create_sort_key('A', 'DESC')
```

`current_catalog` *DuckDB function current_catalog*

Description

DuckDB macro `current_catalog()`.

Usage

```
current_catalog()
```

Value

Unspecified.

`current_connection_id`
DuckDB function current_connection_id

Description

Get the current connection_id.

Usage

```
current_connection_id()
```

Value

`UBIGINT`

SQL examples

```
current_connection_id()
```

`current_database` *DuckDB function current_database*

Description

Returns the name of the currently active database.

Value

`VARCHAR`

Overloads

- `current_database()`
- `current_database()`

SQL examples

```
current_database()
```

`current_query` *DuckDB function current_query*

Description

Returns the current query as a string.

Value

`VARCHAR`

Overloads

- `current_query()`
- `current_query()`

SQL examples

```
current_query()
```

`current_query_id` *DuckDB function current_query_id*

Description

Get the current query_id.

Usage

```
current_query_id()
```

Value

`UBIGINT`

SQL examples

```
current_query_id()
```

current_role *DuckDB function current_role*

Description

DuckDB macro `current_role()`.

Usage

```
current_role()
```

Value

Unspecified.

current_schema *DuckDB function current_schema*

Description

Returns the name of the currently active schema. Default is main.

Value

`VARCHAR`

Overloads

- `current_schema()`
- `current_schema()`

SQL examples

```
current_schema()
```

`current_schemas` *DuckDB function current_schemas*

Description

Returns list of schemas. Pass a parameter of True to include implicit schemas.

Arguments

```
include_implicit
                BOOLEAN
```

Value

```
VARCHAR[]
```

Overloads

- `current_schemas(include_implicit = BOOLEAN)`
- `current_schemas(include_implicit)`

SQL examples

```
current_schemas(true)
```

`current_setting` *DuckDB function current_setting*

Description

Returns the current value of the configuration setting.

Usage

```
current_setting(setting_name = VARCHAR)
```

Arguments

```
setting_name   VARCHAR
```

Value

```
ANY
```

SQL examples

```
current_setting('access_mode')
```

`current_transaction_id` *DuckDB function current_transaction_id*

Description

Get the current global transaction_id.

Usage

```
current_transaction_id()
```

Value

```
UBIGINT
```

SQL examples

```
current_transaction_id()
```

`current_user` *DuckDB function current_user*

Description

DuckDB macro `current_user()`.

Usage

```
current_user()
```

Value

Unspecified.

`currval` *DuckDB function currval*

Description

Return the current value of the sequence. Note that nextval must be called at least once prior to calling currval.

Usage

```
currval(`'sequence_name'` = VARCHAR)
```

Arguments

`'sequence_name'`	VARCHAR

Value

BIGINT

SQL examples

```
currval('my_sequence_name')
```

`damerau_levenshtein` *DuckDB function damerau_levenshtein*

Description

Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Characters of different cases (e.g., `a` and `A`) are considered different.

Usage

```
damerau_levenshtein(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

s1	VARCHAR
s2	VARCHAR

Value

BIGINT

SQL examples

```
damerau_levenshtein('duckdb', 'udckbd')
```

---

database_list *DuckDB function database_list*

Description

DuckDB function `database_list()`.

Usage

```
database_list()
```

Value

Unspecified.

---

database_size *DuckDB function database_size*

Description

DuckDB function `database_size()`.

Usage

```
database_size()
```

Value

Unspecified.

`date_add` *DuckDB function date_add*

Description

DuckDB macro `date_add()`.

Usage

```
date_add(date, interval)
```

Arguments

`date`	Unspecified.
`interval`	Unspecified.

Value

Unspecified.

`date_diff` *DuckDB function date_diff*

Description

The number of partition boundaries between the timestamps.

Arguments

`part`	`VARCHAR`
`startdate`	`DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE`
`enddate`	`DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `date_diff(part = VARCHAR, startdate = DATE, enddate = DATE)`
- `date_diff(part = VARCHAR, startdate = TIME, enddate = TIME)`
- `date_diff(part = VARCHAR, startdate = TIMESTAMP, enddate = TIMESTAMP)`
- `date_diff(part = VARCHAR, startdate = \`TIMESTAMP WITH TIME ZONE\`, enddate = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
date_diff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')
```

`date_part` *DuckDB function date_part*

Description

Get subfield (equivalent to extract).

Arguments

ts	VARCHAR[] | VARCHAR
col1	DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W

Value

STRUCT() | BIGINT

Overloads

- date_part(ts = `VARCHAR[]`, col1 = DATE)
- date_part(ts = `VARCHAR[]`, col1 = INTERVAL)
- date_part(ts = `VARCHAR[]`, col1 = TIME)
- date_part(ts = `VARCHAR[]`, col1 = TIMESTAMP)
- date_part(ts = `VARCHAR[]`, col1 = `TIME WITH TIME ZONE`)
- date_part(ts = `VARCHAR[]`, col1 = TIME_NS)
- date_part(ts = VARCHAR, col1 = DATE)
- date_part(ts = VARCHAR, col1 = INTERVAL)
- date_part(ts = VARCHAR, col1 = TIME)
- date_part(ts = VARCHAR, col1 = TIMESTAMP)
- date_part(ts = VARCHAR, col1 = `TIME WITH TIME ZONE`)
- date_part(ts = VARCHAR, col1 = TIME_NS)
- date_part(ts = `VARCHAR[]`, col1 = `TIMESTAMP WITH TIME ZONE`)
- date_part(ts = VARCHAR, col1 = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
date_part('minute', TIMESTAMP '1992-09-20 20:38:40')
```

`date_sub` *DuckDB function date_sub*

Description

The number of complete partitions between the timestamps.

Arguments

part	VARCHAR
startdate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE
enddate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- date_sub(part = VARCHAR, startdate = DATE, enddate = DATE)
- date_sub(part = VARCHAR, startdate = TIME, enddate = TIME)
- date_sub(part = VARCHAR, startdate = TIMESTAMP, enddate = TIMESTAMP)
- date_sub(part = VARCHAR, startdate = `TIMESTAMP WITH TIME ZONE`, enddate = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
date_sub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')
```

`date_trunc` *DuckDB function date_trunc*

Description

Truncate to specified precision.

Arguments

part	VARCHAR
timestamp	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

TIMESTAMP | INTERVAL | TIMESTAMP WITH TIME ZONE

Overloads

- date_trunc(part = VARCHAR, timestamp = DATE)
- date_trunc(part = VARCHAR, timestamp = INTERVAL)
- date_trunc(part = VARCHAR, timestamp = TIMESTAMP)
- date_trunc(part = VARCHAR, timestamp = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
date_trunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')
```

---

datediff *DuckDB function datediff*

Description

The number of partition boundaries between the timestamps.

Arguments

part	VARCHAR
startdate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE
enddate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- datediff(part = VARCHAR, startdate = DATE, enddate = DATE)
- datediff(part = VARCHAR, startdate = TIME, enddate = TIME)
- datediff(part = VARCHAR, startdate = TIMESTAMP, enddate = TIMESTAMP)
- datediff(part = VARCHAR, startdate = `TIMESTAMP WITH TIME ZONE`, enddate = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
datediff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')
```

`datepart` *DuckDB function datepart*

Description

Get subfield (equivalent to extract).

Arguments

`ts`	`VARCHAR[] | VARCHAR`
`col1`	`DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W`

Value

`STRUCT() | BIGINT`

Overloads

- `datepart(ts = VARCHAR[], col1 = DATE)`
- `datepart(ts = VARCHAR[], col1 = INTERVAL)`
- `datepart(ts = VARCHAR[], col1 = TIME)`
- `datepart(ts = VARCHAR[], col1 = TIMESTAMP)`
- `datepart(ts = VARCHAR[], col1 = TIME WITH TIME ZONE)`
- `datepart(ts = VARCHAR[], col1 = TIME_NS)`
- `datepart(ts = VARCHAR, col1 = DATE)`
- `datepart(ts = VARCHAR, col1 = INTERVAL)`
- `datepart(ts = VARCHAR, col1 = TIME)`
- `datepart(ts = VARCHAR, col1 = TIMESTAMP)`
- `datepart(ts = VARCHAR, col1 = TIME WITH TIME ZONE)`
- `datepart(ts = VARCHAR, col1 = TIME_NS)`
- `datepart(ts = VARCHAR[], col1 = TIMESTAMP WITH TIME ZONE)`
- `datepart(ts = VARCHAR, col1 = TIMESTAMP WITH TIME ZONE)`

SQL examples

```
datepart('minute', TIMESTAMP '1992-09-20 20:38:40')
```

`datesub` *DuckDB function datesub*

Description

The number of complete partitions between the timestamps.

Arguments

part	VARCHAR
startdate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE
enddate	DATE | TIME | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- datesub(part = VARCHAR, startdate = DATE, enddate = DATE)
- datesub(part = VARCHAR, startdate = TIME, enddate = TIME)
- datesub(part = VARCHAR, startdate = TIMESTAMP, enddate = TIMESTAMP)
- datesub(part = VARCHAR, startdate = `TIMESTAMP WITH TIME ZONE`, enddate = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
datesub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')
```

`datetrunc` *DuckDB function datetrunc*

Description

Truncate to specified precision.

Arguments

part	VARCHAR
timestamp	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

TIMESTAMP | INTERVAL | TIMESTAMP WITH TIME ZONE

Overloads

- `datetrunc(part = VARCHAR, timestamp = DATE)`
- `datetrunc(part = VARCHAR, timestamp = INTERVAL)`
- `datetrunc(part = VARCHAR, timestamp = TIMESTAMP)`
- `datetrunc(part = VARCHAR, timestamp = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
datetrunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')
```

`day` *DuckDB function day*

Description

Extract the day component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `day(ts = DATE)`
- `day(ts = INTERVAL)`
- `day(ts = TIMESTAMP)`
- `day(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
day(timestamp '2021-08-03 11:59:44.123456')
```

`dayname` *DuckDB function dayname*

Description

The (English) name of the weekday.

Arguments

`ts`	`DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`VARCHAR`

Overloads

- `dayname(ts = DATE)`
- `dayname(ts = TIMESTAMP)`
- `dayname(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
dayname(TIMESTAMP '1992-03-22')
```

`dayofmonth` *DuckDB function dayofmonth*

Description

Extract the dayofmonth component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `dayofmonth(ts = DATE)`
- `dayofmonth(ts = INTERVAL)`
- `dayofmonth(ts = TIMESTAMP)`
- `dayofmonth(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
dayofmonth(timestamp '2021-08-03 11:59:44.123456')
```

`dayofweek` *DuckDB function dayofweek*

Description

Extract the dayofweek component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `dayofweek(ts = DATE)`
- `dayofweek(ts = INTERVAL)`
- `dayofweek(ts = TIMESTAMP)`
- `dayofweek(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
dayofweek(timestamp '2021-08-03 11:59:44.123456')
```

`dayofyear` *DuckDB function dayofyear*

Description

Extract the dayofyear component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- dayofyear(ts = DATE)
- dayofyear(ts = INTERVAL)
- dayofyear(ts = TIMESTAMP)
- dayofyear(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
dayofyear(timestamp '2021-08-03 11:59:44.123456')
```

---

dd *DuckDB functions*

Description

A list of known DuckDB functions.

Usage

```
dd
```

Examples

```
dd[1:3]
```

---

decade *DuckDB function decade*

Description

Extract the decade component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- decade(ts = DATE)
- decade(ts = INTERVAL)
- decade(ts = TIMESTAMP)
- decade(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
decade(timestamp '2021-08-03 11:59:44.123456')
```

`decode` *DuckDB function decode*

Description

Converts `blob` to `VARCHAR`. Fails if `blob` is not valid UTF-8.

Usage

```
decode(blob = BLOB)
```

Arguments

`blob`	`BLOB`

Value

`VARCHAR`

SQL examples

```
decode('\xC3\xBC'::BLOB)
```

`degrees` *DuckDB function degrees*

Description

Converts radians to degrees.

Usage

```
degrees(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
degrees(pi())
```

`disable_checkpoint_on_shutdown` *DuckDB function disable_checkpoint_on_shutdown*

Description

DuckDB function `disable_checkpoint_on_shutdown()`.

Usage

```
disable_checkpoint_on_shutdown()
```

Value

Unspecified.

`disable_logging` *DuckDB function disable_logging*

Description

DuckDB function `disable_logging()`.

Usage

```
disable_logging()
```

Value

Unspecified.

`disable_object_cache` *DuckDB function disable_object_cache*

Description

DuckDB function `disable_object_cache()`.

Usage

```
disable_object_cache()
```

Value

Unspecified.

disable_optimizer *DuckDB function disable_optimizer*

Description

DuckDB function `disable_optimizer()`.

Usage

```
disable_optimizer()
```

Value

Unspecified.

disable_print_progress_bar *DuckDB function disable_print_progress_bar*

Description

DuckDB function `disable_print_progress_bar()`.

Usage

```
disable_print_progress_bar()
```

Value

Unspecified.

disable_profile *DuckDB function disable_profile*

Description

DuckDB function `disable_profile()`.

Usage

```
disable_profile()
```

Value

Unspecified.

disable_profiling *DuckDB function disable_profiling*

Description

DuckDB function `disable_profiling()`.

Usage

```
disable_profiling()
```

Value

Unspecified.

disable_progress_bar *DuckDB function disable_progress_bar*

Description

DuckDB function `disable_progress_bar()`.

Usage

```
disable_progress_bar()
```

Value

Unspecified.

disable_verification *DuckDB function disable_verification*

Description

DuckDB function `disable_verification()`.

Usage

```
disable_verification()
```

Value

Unspecified.

`disable_verify_external`
DuckDB function disable_verify_external

Description

DuckDB function `disable_verify_external()`.

Usage

```
disable_verify_external()
```

Value

Unspecified.

`disable_verify_fetch_row`
DuckDB function disable_verify_fetch_row

Description

DuckDB function `disable_verify_fetch_row()`.

Usage

```
disable_verify_fetch_row()
```

Value

Unspecified.

`disable_verify_parallelism`
DuckDB function disable_verify_parallelism

Description

DuckDB function `disable_verify_parallelism()`.

Usage

```
disable_verify_parallelism()
```

Value

Unspecified.

`disable_verify_serializer` *DuckDB function disable_verify_serializer*

Description

DuckDB function `disable_verify_serializer()`.

Usage

```
disable_verify_serializer()
```

Value

Unspecified.

`divide` *DuckDB function divide*

Description

DuckDB function `divide()`.

Arguments

`col0`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`
`col1`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`

Overloads

- `divide(col0 = TINYINT, col1 = TINYINT)`
- `divide(col0 = SMALLINT, col1 = SMALLINT)`
- `divide(col0 = INTEGER, col1 = INTEGER)`
- `divide(col0 = BIGINT, col1 = BIGINT)`
- `divide(col0 = HUGEINT, col1 = HUGEINT)`
- `divide(col0 = FLOAT, col1 = FLOAT)`
- `divide(col0 = DOUBLE, col1 = DOUBLE)`

- divide(col0 = UTINYINT, col1 = UTINYINT)
- divide(col0 = USMALLINT, col1 = USMALLINT)
- divide(col0 = UINTEGER, col1 = UINTEGER)
- divide(col0 = UBIGINT, col1 = UBIGINT)
- divide(col0 = UHUGEINT, col1 = UHUGEINT)

---

duckdb_approx_database_count

DuckDB function duckdb_approx_database_count

Description

DuckDB function `duckdb_approx_database_count()`.

Usage

```
duckdb_approx_database_count()
```

Value

Unspecified.

---

duckdb_columns

DuckDB function duckdb_columns

Description

DuckDB function `duckdb_columns()`.

Usage

```
duckdb_columns()
```

Value

Unspecified.

`duckdb_constraints` *DuckDB function duckdb_constraints*

Description

DuckDB function `duckdb_constraints()`.

Usage

```
duckdb_constraints()
```

Value

Unspecified.

`duckdb_databases` *DuckDB function duckdb_databases*

Description

DuckDB function `duckdb_databases()`.

Usage

```
duckdb_databases()
```

Value

Unspecified.

`duckdb_dependencies` *DuckDB function duckdb_dependencies*

Description

DuckDB function `duckdb_dependencies()`.

Usage

```
duckdb_dependencies()
```

Value

Unspecified.

duckdb_extensions *DuckDB function duckdb_extensions*

Description

DuckDB function `duckdb_extensions()`.

Usage

```
duckdb_extensions()
```

Value

Unspecified.

duckdb_external_file_cache *DuckDB function duckdb_external_file_cache*

Description

DuckDB function `duckdb_external_file_cache()`.

Usage

```
duckdb_external_file_cache()
```

Value

Unspecified.

duckdb_functions *DuckDB function duckdb_functions*

Description

DuckDB function `duckdb_functions()`.

Usage

```
duckdb_functions()
```

Value

Unspecified.

duckdb_indexes *DuckDB function duckdb_indexes*

Description

DuckDB function `duckdb_indexes()`.

Usage

```
duckdb_indexes()
```

Value

Unspecified.

duckdb_keywords *DuckDB function duckdb_keywords*

Description

DuckDB function `duckdb_keywords()`.

Usage

```
duckdb_keywords()
```

Value

Unspecified.

duckdb_log_contexts *DuckDB function duckdb_log_contexts*

Description

DuckDB function `duckdb_log_contexts()`.

Usage

```
duckdb_log_contexts()
```

Value

Unspecified.

duckdb_logs *DuckDB function duckdb_logs*

Description

DuckDB function `duckdb_logs()`.

Usage

```
duckdb_logs(denormalized_table = BOOLEAN)
```

Arguments

```
denormalized_table
               BOOLEAN
```

Value

Unspecified.

duckdb_logs_parsed *DuckDB function duckdb_logs_parsed*

Description

DuckDB macro `duckdb_logs_parsed()`.

Usage

```
duckdb_logs_parsed(log_type)
```

Arguments

`log_type`	Unspecified.

Value

Unspecified.

`duckdb_memory` *DuckDB function duckdb_memory*

Description

DuckDB function `duckdb_memory()`.

Usage

```
duckdb_memory()
```

Value

Unspecified.

`duckdb_optimizers` *DuckDB function duckdb_optimizers*

Description

DuckDB function `duckdb_optimizers()`.

Usage

```
duckdb_optimizers()
```

Value

Unspecified.

`duckdb_prepared_statements`
DuckDB function duckdb_prepared_statements

Description

DuckDB function `duckdb_prepared_statements()`.

Usage

```
duckdb_prepared_statements()
```

Value

Unspecified.

`duckdb_schemas` *DuckDB function duckdb_schemas*

Description

DuckDB function `duckdb_schemas()`.

Usage

```
duckdb_schemas()
```

Value

Unspecified.

`duckdb_secret_types` *DuckDB function duckdb_secret_types*

Description

DuckDB function `duckdb_secret_types()`.

Usage

```
duckdb_secret_types()
```

Value

Unspecified.

`duckdb_secrets` *DuckDB function duckdb_secrets*

Description

DuckDB function `duckdb_secrets()`.

Usage

```
duckdb_secrets(redact = BOOLEAN)
```

Arguments

`redact`	`BOOLEAN`

Value

Unspecified.

`duckdb_sequences` *DuckDB function duckdb_sequences*

Description

DuckDB function `duckdb_sequences()`.

Usage

```
duckdb_sequences()
```

Value

Unspecified.

`duckdb_settings` *DuckDB function duckdb_settings*

Description

DuckDB function `duckdb_settings()`.

Usage

```
duckdb_settings()
```

Value

Unspecified.

`duckdb_table_sample` *DuckDB function duckdb_table_sample*

Description

DuckDB function `duckdb_table_sample()`.

Usage

```
duckdb_table_sample(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

`duckdb_tables` *DuckDB function duckdb_tables*

Description

DuckDB function `duckdb_tables()`.

Usage

```
duckdb_tables()
```

Value

Unspecified.

`duckdb_temporary_files`
DuckDB function duckdb_temporary_files

Description

DuckDB function `duckdb_temporary_files()`.

Usage

```
duckdb_temporary_files()
```

Value

Unspecified.

`duckdb_types` *DuckDB function duckdb_types*

Description

DuckDB function `duckdb_types()`.

Usage

```
duckdb_types()
```

Value

Unspecified.

duckdb_variables *DuckDB function duckdb_variables*

Description

DuckDB function `duckdb_variables()`.

Usage

```
duckdb_variables()
```

Value

Unspecified.

duckdb_views *DuckDB function duckdb_views*

Description

DuckDB function `duckdb_views()`.

Usage

```
duckdb_views()
```

Value

Unspecified.

editdist3 *DuckDB function editdist3*

Description

The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., `a` and `A`) are considered different.

Usage

```
editdist3(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

s1	VARCHAR
s2	VARCHAR

Value

BIGINT

SQL examples

```
editdist3('duck', 'db')
```

---

element_at *DuckDB function element_at*

Description

Returns a list containing the value for a given key or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.

Usage

```
element_at(map = `MAP(K, V)`, key = K)
```

Arguments

map	MAP(K, V)
key	K

Value

V[]

SQL examples

```
element_at(map(['key'], ['val']), 'key')
```

enable_checkpoint_on_shutdown

DuckDB function enable_checkpoint_on_shutdown

Description

DuckDB function `enable_checkpoint_on_shutdown()`.

Usage

```
enable_checkpoint_on_shutdown()
```

Value

Unspecified.

enable_object_cache

DuckDB function enable_object_cache

Description

DuckDB function `enable_object_cache()`.

Usage

```
enable_object_cache()
```

Value

Unspecified.

enable_optimizer

DuckDB function enable_optimizer

Description

DuckDB function `enable_optimizer()`.

Usage

```
enable_optimizer()
```

Value

Unspecified.

enable_print_progress_bar

DuckDB function enable_print_progress_bar

Description

DuckDB function `enable_print_progress_bar()`.

Usage

```
enable_print_progress_bar()
```

Value

Unspecified.

enable_profile

DuckDB function enable_profile

Description

DuckDB function `enable_profile()`.

Usage

```
enable_profile()
```

Value

Unspecified.

enable_profiling

DuckDB function enable_profiling

Description

DuckDB function `enable_profiling()`.

Usage

```
enable_profiling()
```

Value

Unspecified.

`enable_progress_bar` *DuckDB function enable_progress_bar*

Description

DuckDB function `enable_progress_bar()`.

Usage

```
enable_progress_bar()
```

Value

Unspecified.

`enable_verification` *DuckDB function enable_verification*

Description

DuckDB function `enable_verification()`.

Usage

```
enable_verification()
```

Value

Unspecified.

`encode` *DuckDB function encode*

Description

Converts the `string` to `BLOB`. Converts UTF-8 characters into literal encoding.

Usage

```
encode(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

BLOB

SQL examples

```
encode('my_string_with_ü')
```

`ends_with` *DuckDB function ends_with*

Description

Returns `true` if `string` ends with `search_string`.

Usage

```
ends_with(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	VARCHAR
`search_string`	VARCHAR

Value

BOOLEAN

SQL examples

```
ends_with('abc', 'bc')
```

`entropy` *DuckDB function entropy*

Description

Returns the log-2 entropy of count input-values.

Usage

```
entropy(x = ANY)
```

Arguments

`x`	ANY

Value

DOUBLE

`enum_code` *DuckDB function enum_code*

Description

Returns the numeric value backing the given enum value.

Usage

```
enum_code(enum = ANY)
```

Arguments

enum	ANY

Value

ANY

SQL examples

```
enum_code('happy'::mood)
```

`enum_first` *DuckDB function enum_first*

Description

Returns the first value of the input enum type.

Usage

```
enum_first(enum = ANY)
```

Arguments

enum	ANY

Value

VARCHAR

SQL examples

```
enum_first(NULL::mood)
```

`enum_last` *DuckDB function enum_last*

Description

Returns the last value of the input enum type.

Usage

```
enum_last(enum = ANY)
```

Arguments

`enum`	`ANY`

Value

`VARCHAR`

SQL examples

```
enum_last(NULL::mood)
```

`enum_range` *DuckDB function enum_range*

Description

Returns all values of the input enum type as an array.

Usage

```
enum_range(enum = ANY)
```

Arguments

`enum`	`ANY`

Value

`VARCHAR[]`

SQL examples

```
enum_range(NULL::mood)
```

`enum_range_boundary` *DuckDB function enum_range_boundary*

Description

Returns the range between the two given enum values as an array. The values must be of the same enum type. When the first parameter is NULL, the result starts with the first value of the enum type. When the second parameter is NULL, the result ends with the last value of the enum type.

Usage

```
enum_range_boundary(start = ANY, end = ANY)
```

Arguments

`start`	`ANY`
`end`	`ANY`

Value

`VARCHAR[]`

SQL examples

```
enum_range_boundary(NULL, 'happy'::mood)
```

`epoch` *DuckDB function epoch*

Description

Extract the epoch component from a temporal type.

Arguments

`temporal`	`DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W`

Value

`DOUBLE`

Overloads

- `epoch(temporal = DATE)`
- `epoch(temporal = INTERVAL)`
- `epoch(temporal = TIME)`
- `epoch(temporal = TIMESTAMP)`
- `epoch(temporal = \`TIME WITH TIME ZONE\`)`
- `epoch(temporal = TIME_NS)`
- `epoch(temporal = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
epoch(timestamp '2021-08-03 11:59:44.123456')
```

`epoch_ms` *DuckDB function epoch_ms*

Description

Extract the epoch component in milliseconds from a temporal type.

Arguments

`temporal`	`DATE | TIMESTAMP | INTERVAL | TIME | TIME_NS | TIME WITH TIME ZONE | TIMESTAMP W`

Value

`BIGINT | TIMESTAMP`

Overloads

- `epoch_ms(temporal = DATE)`
- `epoch_ms(temporal = TIMESTAMP)`
- `epoch_ms(temporal = INTERVAL)`
- `epoch_ms(temporal = TIME)`
- `epoch_ms(temporal = TIME_NS)`
- `epoch_ms(temporal = \`TIME WITH TIME ZONE\`)`
- `epoch_ms(temporal = \`TIMESTAMP WITH TIME ZONE\`)`
- `epoch_ms(temporal = BIGINT)`

SQL examples

```
epoch_ms(timestamp '2021-08-03 11:59:44.123456')
```

`epoch_ns` *DuckDB function epoch_ns*

Description

Extract the epoch component in nanoseconds from a temporal type.

Arguments

`temporal`	`DATE | TIMESTAMP | INTERVAL | TIME | TIME_NS | TIME WITH TIME ZONE | TIMESTAMP W`

Value

`BIGINT`

Overloads

- `epoch_ns(temporal = DATE)`
- `epoch_ns(temporal = TIMESTAMP)`
- `epoch_ns(temporal = INTERVAL)`
- `epoch_ns(temporal = TIME)`
- `epoch_ns(temporal = TIME_NS)`
- `epoch_ns(temporal = `TIME WITH TIME ZONE`)`
- `epoch_ns(temporal = `TIMESTAMP WITH TIME ZONE`)`
- `epoch_ns(temporal = TIMESTAMP_NS)`

SQL examples

```
epoch_ns(timestamp '2021-08-03 11:59:44.123456')
```

`epoch_us` *DuckDB function epoch_us*

Description

Extract the epoch component in microseconds from a temporal type.

Arguments

`temporal`	`DATE | TIMESTAMP | INTERVAL | TIME | TIME_NS | TIME WITH TIME ZONE | TIMESTAMP W`

Value

`BIGINT`

Overloads

- `epoch_us(temporal = DATE)`
- `epoch_us(temporal = TIMESTAMP)`
- `epoch_us(temporal = INTERVAL)`
- `epoch_us(temporal = TIME)`
- `epoch_us(temporal = TIME_NS)`
- `epoch_us(temporal = `TIME WITH TIME ZONE`)`
- `epoch_us(temporal = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
epoch_us(timestamp '2021-08-03 11:59:44.123456')
```

`equi_width_bins` *DuckDB function equi_width_bins*

Description

Generates bin_count equi-width bins between the min and max. If enabled nice_rounding makes the numbers more readable/less jagged.

Arguments

`min`	`BIGINT | DOUBLE | TIMESTAMP | ANY`
`max`	`BIGINT | DOUBLE | TIMESTAMP | ANY`
`bin_count`	`BIGINT`
`nice_rounding`	`BOOLEAN`

Value

`ANY[]`

Overloads

- `equi_width_bins(min = BIGINT, max = BIGINT, bin_count = BIGINT, nice_rounding = BOOLEAN)`
- `equi_width_bins(min = DOUBLE, max = DOUBLE, bin_count = BIGINT, nice_rounding = BOOLEAN)`
- `equi_width_bins(min = TIMESTAMP, max = TIMESTAMP, bin_count = BIGINT, nice_rounding = BOOLEAN)`
- `equi_width_bins(min = ANY, max = ANY, bin_count = BIGINT, nice_rounding = BOOLEAN)`

SQL examples

```
equi_width_bins(0, 10, 2, true)
```

`era` *DuckDB function era*

Description

Extract the era component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `era(ts = DATE)`
- `era(ts = INTERVAL)`
- `era(ts = TIMESTAMP)`
- `era(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
era(timestamp '2021-08-03 11:59:44.123456')
```

`error` *DuckDB function error*

Description

Throws the given error message.

Usage

```
error(message = VARCHAR)
```

Arguments

`message`	`VARCHAR`

Value

`"NULL"`

SQL examples

```
error('access_mode')
```

`even` *DuckDB function even*

Description

Rounds x to next even number by rounding away from zero.

Usage

```
even(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
even(2.9)
```

`exp` *DuckDB function exp*

Description

Computes e to the power of x.

Usage

```
exp(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
exp(1)
```

extension_versions *DuckDB function extension_versions*

Description

DuckDB function `extension_versions()`.

Usage

```
extension_versions()
```

Value

Unspecified.

factorial *DuckDB function factorial*

Description

Factorial of x. Computes the product of the current integer and all integers below it.

Usage

```
factorial(x = INTEGER)
```

Arguments

x	INTEGER

Value

`HUGEINT`

SQL examples

```
4!
```

`favg` *DuckDB function favg*

Description

Calculates the average using a more accurate floating point summation (Kahan Sum).

Usage

```
favg(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
favg(A)
```

`fdiv` *DuckDB function fdiv*

Description

DuckDB macro `fdiv()`.

Usage

```
fdiv(x, y)
```

Arguments

`x`	Unspecified.
`y`	Unspecified.

Value

Unspecified.

filter — *DuckDB function filter*

Description

Constructs a list from those elements of the input `list` for which the `lambda` function returns `true`. DuckDB must be able to cast the `lambda` function's return type to `BOOL`. The return type of `list_filter` is the same as the input list's.

Usage

```
filter(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
filter([3, 4, 5], lambda x : x > 4)
```

finalize — *DuckDB function finalize*

Description

DuckDB function `finalize()`.

Usage

```
finalize(col0 = `AGGREGATE_STATE<?>`)
```

Arguments

`col0`	`AGGREGATE_STATE<?>`

Value

`INVALID`

`first` *DuckDB function first*

Description

Returns the first value (NULL or non-NULL) from arg. This function is affected by ordering.

Arguments

`arg`	`DECIMAL | ANY`

Value

`DECIMAL | ANY`

Overloads

- `first(arg = DECIMAL)`
- `first(arg = ANY)`

SQL examples

```
first(A)
```

`flatten` *DuckDB function flatten*

Description

Flattens a nested list by one level.

Usage

```
flatten(nested_list = `T[][]`)
```

Arguments

`nested_list`	`T[][]`

Value

`T[]`

SQL examples

```
flatten([[1, 2, 3], [4, 5]])
```

`floor` *DuckDB function floor*

Description

Rounds the number down.

Arguments

`x`	`FLOAT | DOUBLE | DECIMAL`

Value

`FLOAT | DOUBLE | DECIMAL`

Overloads

- `floor(x = FLOAT)`
- `floor(x = DOUBLE)`
- `floor(x = DECIMAL)`

SQL examples

```
floor(17.4)
```

`fmod` *DuckDB function fmod*

Description

DuckDB macro `fmod()`.

Usage

```
fmod(x, y)
```

Arguments

`x`	Unspecified.
`y`	Unspecified.

Value

Unspecified.

`force_checkpoint` *DuckDB function force_checkpoint*

Description

DuckDB function `force_checkpoint()`.

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

Overloads

- `force_checkpoint()`
- `force_checkpoint(col0 = VARCHAR)`
- `force_checkpoint()`

`format_bytes` *DuckDB function format_bytes*

Description

Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).

Usage

```
format_bytes(integer = BIGINT)
```

Arguments

`integer`	`BIGINT`

Value

`VARCHAR`

SQL examples

```
format_bytes(16_000)
```

format_pg_type *DuckDB function format_pg_type*

Description

DuckDB macro `format_pg_type()`.

Usage

```
format_pg_type(logical_type, type_name)
```

Arguments

`logical_type`	Unspecified.
`type_name`	Unspecified.

Value

Unspecified.

format_type *DuckDB function format_type*

Description

DuckDB macro `format_type()`.

Usage

```
format_type(type_oid, typemod)
```

Arguments

`type_oid`	Unspecified.
`typemod`	Unspecified.

Value

Unspecified.

`formatReadableDecimalSize`
DuckDB function formatReadableDecimalSize

Description

Converts `integer` to a human-readable representation using units based on powers of 10 (KB, MB, GB, etc.).

Usage

```
formatReadableDecimalSize(integer = BIGINT)
```

Arguments

```
integer        BIGINT
```

Value

```
VARCHAR
```

SQL examples

```
formatReadableDecimalSize(16_000)
```

`formatReadableSize` *DuckDB function formatReadableSize*

Description

Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).

Usage

```
formatReadableSize(integer = BIGINT)
```

Arguments

```
integer        BIGINT
```

Value

```
VARCHAR
```

SQL examples

```
formatReadableSize(16_000)
```

`from_base64` *DuckDB function from_base64*

Description

Converts a base64 encoded `string` to a character string (`BLOB`).

Usage

```
from_base64(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`BLOB`

SQL examples

```
from_base64('QQ==')
```

`from_binary` *DuckDB function from_binary*

Description

Converts a `value` from binary representation to a blob.

Usage

```
from_binary(value = VARCHAR)
```

Arguments

`value`	`VARCHAR`

Value

`BLOB`

SQL examples

```
from_binary('0110')
```

`from_hex` *DuckDB function from_hex*

Description

Converts a `value` from hexadecimal representation to a blob.

Usage

```
from_hex(value = VARCHAR)
```

Arguments

`value`	`VARCHAR`

Value

`BLOB`

SQL examples

```
from_hex('2A')
```

`fsum` *DuckDB function fsum*

Description

Calculates the sum using a more accurate floating point summation (Kahan Sum).

Usage

```
fsum(arg = DOUBLE)
```

Arguments

`arg`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
fsum(A)
```

functions *DuckDB function functions*

Description

DuckDB function `functions()`.

Usage

```
functions()
```

Value

Unspecified.

gamma *DuckDB function gamma*

Description

Interpolation of (x-1) factorial (so decimal inputs are allowed).

Usage

```
gamma(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
gamma(5.5)
```

`gcd` *DuckDB function gcd*

Description

Computes the greatest common divisor of x and y.

Arguments

x	BIGINT | HUGEINT
y	BIGINT | HUGEINT

Value

`BIGINT | HUGEINT`

Overloads

- `gcd(x = BIGINT, y = BIGINT)`
- `gcd(x = HUGEINT, y = HUGEINT)`

SQL examples

```
gcd(42, 57)
```

`gen_random_uuid` *DuckDB function gen_random_uuid*

Description

Returns a random UUID v4 similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.

Usage

```
gen_random_uuid()
```

Value

`UUID`

SQL examples

```
gen_random_uuid()
```

`generate_series` *DuckDB function generate_series*

Description

Creates a list of values between `start` and `stop` - the stop parameter is inclusive.

Arguments

col0	BIGINT | TIMESTAMP
col1	BIGINT | TIMESTAMP
col2	BIGINT | INTERVAL
start	BIGINT | TIMESTAMP | TIMESTAMP WITH TIME ZONE
stop	BIGINT | TIMESTAMP | TIMESTAMP WITH TIME ZONE
step	BIGINT | INTERVAL

Value

BIGINT[] | TIMESTAMP[] | TIMESTAMP WITH TIME ZONE[]

Overloads

- generate_series(col0 = BIGINT)
- generate_series(col0 = BIGINT, col1 = BIGINT)
- generate_series(col0 = BIGINT, col1 = BIGINT, col2 = BIGINT)
- generate_series(col0 = TIMESTAMP, col1 = TIMESTAMP, col2 = INTERVAL)
- generate_series(start = BIGINT)
- generate_series(start = BIGINT, stop = BIGINT)
- generate_series(start = BIGINT, stop = BIGINT, step = BIGINT)
- generate_series(start = TIMESTAMP, stop = TIMESTAMP, step = INTERVAL)
- generate_series(start = `TIMESTAMP WITH TIME ZONE`, stop = `TIMESTAMP WITH TIME ZONE`, step = INTERVAL)

SQL examples

```
generate_series(2, 5, 3)
```

generate_subscripts *DuckDB function generate_subscripts*

Description

DuckDB macro `generate_subscripts()`.

Usage

```
generate_subscripts(arr, dim)
```

Arguments

`arr`	Unspecified.
`dim`	Unspecified.

Value

Unspecified.

geomean *DuckDB function geomean*

Description

DuckDB macro `geomean()`.

Usage

```
geomean(x)
```

Arguments

`x`	Unspecified.

Value

Unspecified.

geometric_mean *DuckDB function geometric_mean*

Description

DuckDB macro `geometric_mean()`.

Usage

```
geometric_mean(x)
```

Arguments

`x`	Unspecified.

Value

Unspecified.

get_bit *DuckDB function get_bit*

Description

Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.

Usage

```
get_bit(bitstring = BIT, index = INTEGER)
```

Arguments

`bitstring`	`BIT`
`index`	`INTEGER`

Value

`INTEGER`

SQL examples

```
get_bit('0110010'::BIT, 2)
```

`get_block_size` *DuckDB function get_block_size*

Description

DuckDB macro `get_block_size()`.

Usage

```
get_block_size(db_name)
```

Arguments

`db_name`	Unspecified.

Value

Unspecified.

`get_current_timestamp`
DuckDB function get_current_timestamp

Description

Returns the current timestamp.

Usage

```
get_current_timestamp()
```

Value

`TIMESTAMP WITH TIME ZONE`

SQL examples

```
get_current_timestamp()
```

`getvariable` *DuckDB function getvariable*

Description

DuckDB function `getvariable()`.

Usage

```
getvariable(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

`ANY`

`glob` *DuckDB function glob*

Description

DuckDB function `glob()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`

Value

Unspecified.

Overloads

- `glob(col0 = VARCHAR)`
- `glob(col0 = \`VARCHAR[]\`)`

`grade_up` *DuckDB function grade_up*

Description

Works like list_sort, but the results are the indexes that correspond to the position in the original list instead of the actual values.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`
`col2`	`VARCHAR`

Value

`ANY[]`

Overloads

- `grade_up(list = \`ANY[]\`)`
- `grade_up(list = \`ANY[]\`, col1 = VARCHAR)`
- `grade_up(list = \`ANY[]\`, col1 = VARCHAR, col2 = VARCHAR)`

SQL examples

```
grade_up([3, 6, 1, 2])
```

`greatest` *DuckDB function greatest*

Description

Returns the largest value. For strings lexicographical ordering is used. Note that lowercase characters are considered "larger" than uppercase characters and collations are not supported.

Usage

```
greatest(arg1 = ANY)
```

Arguments

`arg1`	`ANY`

Value

ANY

SQL examples

```
greatest(42, 84)
greatest('abc', 'bcd', 'cde', 'EFG')
```

---

greatest_common_divisor

DuckDB function greatest_common_divisor

Description

Computes the greatest common divisor of x and y.

Arguments

x	BIGINT | HUGEINT
y	BIGINT | HUGEINT

Value

BIGINT | HUGEINT

Overloads

- greatest_common_divisor(x = BIGINT, y = BIGINT)
- greatest_common_divisor(x = HUGEINT, y = HUGEINT)

SQL examples

```
greatest_common_divisor(42, 57)
```

`group_concat` *DuckDB function group_concat*

Description

Concatenates the column string values with an optional separator.

Arguments

`str`	`ANY`
`arg`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `group_concat(str = ANY)`
- `group_concat(str = ANY, arg = VARCHAR)`

SQL examples

```
group_concat(A, '-')
```

`hamming` *DuckDB function hamming*

Description

The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., `a` and `A`) are considered different.

Usage

```
hamming(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

`s1`	`VARCHAR`
`s2`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
hamming('duck', 'luck')
```

---

has_any_column_privilege

DuckDB function has_any_column_privilege

Description

DuckDB function `has_any_column_privilege()`.

Arguments

`table`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_any_column_privilege(table, privilege)`
- `has_any_column_privilege(user, table, privilege)`

---

has_column_privilege *DuckDB function has_column_privilege*

Description

DuckDB function `has_column_privilege()`.

Arguments

`table`	Unspecified.
`column`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_column_privilege(table, column, privilege)`
- `has_column_privilege(user, table, column, privilege)`

`has_database_privilege`
DuckDB function has_database_privilege

Description

DuckDB function `has_database_privilege()`.

Arguments

`database`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_database_privilege(database, privilege)`
- `has_database_privilege(user, database, privilege)`

`has_foreign_data_wrapper_privilege`
DuckDB function has_foreign_data_wrapper_privilege

Description

DuckDB function `has_foreign_data_wrapper_privilege()`.

Arguments

`fdw`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_foreign_data_wrapper_privilege(fdw, privilege)`
- `has_foreign_data_wrapper_privilege(user, fdw, privilege)`

`has_function_privilege`

DuckDB function has_function_privilege

Description

DuckDB function `has_function_privilege()`.

Arguments

`function`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_function_privilege(` `function` `, privilege)`
- `has_function_privilege(user, ` `function` `, privilege)`

`has_language_privilege`

DuckDB function has_language_privilege

Description

DuckDB function `has_language_privilege()`.

Arguments

`language`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_language_privilege(language, privilege)`
- `has_language_privilege(user, language, privilege)`

`has_schema_privilege` *DuckDB function has_schema_privilege*

Description

DuckDB function `has_schema_privilege()`.

Arguments

`schema`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_schema_privilege(schema, privilege)`
- `has_schema_privilege(user, schema, privilege)`

`has_sequence_privilege`
DuckDB function has_sequence_privilege

Description

DuckDB function `has_sequence_privilege()`.

Arguments

`sequence`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_sequence_privilege(sequence, privilege)`
- `has_sequence_privilege(user, sequence, privilege)`

`has_server_privilege` *DuckDB function has_server_privilege*

Description

DuckDB function `has_server_privilege()`.

Arguments

`server`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_server_privilege(server, privilege)`
- `has_server_privilege(user, server, privilege)`

`has_table_privilege` *DuckDB function has_table_privilege*

Description

DuckDB function `has_table_privilege()`.

Arguments

`table`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_table_privilege(table, privilege)`
- `has_table_privilege(user, table, privilege)`

`has_tablespace_privilege`
DuckDB function has_tablespace_privilege

Description

DuckDB function `has_tablespace_privilege()`.

Arguments

`tablespace`	Unspecified.
`privilege`	Unspecified.
`user`	Unspecified.

Value

Unspecified.

Overloads

- `has_tablespace_privilege(tablespace, privilege)`
- `has_tablespace_privilege(user, tablespace, privilege)`

`hash` *DuckDB function hash*

Description

Returns a `UBIGINT` with the hash of the `value`. Note that this is not a cryptographic hash.

Usage

```
hash(value = ANY)
```

Arguments

`value`	`ANY`

Value

`UBIGINT`

SQL examples

```
hash(' ')
```

`hex` *DuckDB function hex*

Description

Converts the `string` to hexadecimal representation.

Converts the `value` to `VARCHAR` using hexadecimal representation.

Converts `blob` to `VARCHAR` using hexadecimal encoding.

Arguments

`string`	`VARCHAR`
`value`	`BIGNUM | BIGINT | UBIGINT | HUGEINT | UHUGEINT`
`blob`	`BLOB`

Value

`VARCHAR`

Overloads

- `hex(string = VARCHAR)`
- `hex(value = BIGNUM)`
- `hex(blob = BLOB)`
- `hex(value = BIGINT)`
- `hex(value = UBIGINT)`
- `hex(value = HUGEINT)`
- `hex(value = UHUGEINT)`

SQL examples

```
hex('Hello')
hex(42)
hex('\xAA\xBB'::BLOB)
```

`histogram` *DuckDB function histogram*

Description

Returns a LIST of STRUCTs with the fields bucket and count.

Arguments

`arg`	`ANY`
`col1`	`ANY[]`
`source`	Unspecified.
`col_name`	Unspecified.
`bin_count`	Unspecified.
`technique`	Unspecified.

Value

`MAP`

Overloads

- `histogram(arg = ANY, col1 = ANY[])`
- `histogram(arg = ANY)`
- `histogram(source, col_name, bin_count, technique)`

SQL examples

```
histogram(A)
```

`histogram_exact` *DuckDB function histogram_exact*

Description

Returns a LIST of STRUCTs with the fields bucket and count matching the buckets exactly.

Usage

```
histogram_exact(arg = ANY, bins = ANY[])
```

Arguments

`arg`	`ANY`
`bins`	`ANY[]`

Value

MAP

SQL examples

```
histogram_exact(A, [0, 1, 2])
```

---

histogram_values *DuckDB function histogram_values*

Description

DuckDB macro `histogram_values()`.

Usage

```
histogram_values(source, col_name, bin_count, technique)
```

Arguments

source	Unspecified.
col_name	Unspecified.
bin_count	Unspecified.
technique	Unspecified.

Value

Unspecified.

---

hour *DuckDB function hour*

Description

Extract the hour component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W

Value

BIGINT

Overloads

- `hour(ts = DATE)`
- `hour(ts = INTERVAL)`
- `hour(ts = TIME)`
- `hour(ts = TIMESTAMP)`
- `hour(ts = `TIME WITH TIME ZONE`)`
- `hour(ts = TIME_NS)`
- `hour(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
hour(timestamp '2021-08-03 11:59:44.123456')
```

---

ilike_escape *DuckDB function ilike_escape*

Description

Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.

Usage

```
ilike_escape(string = VARCHAR, like_specifier = VARCHAR, escape_character = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`like_specifier`	`VARCHAR`
`escape_character`	`VARCHAR`

Value

`BOOLEAN`

SQL examples

```
ilike_escape('A%c', 'a$%C', '$')
```

import_database *DuckDB function import_database*

Description

DuckDB function `import_database()`.

Usage

```
import_database(col0 = VARCHAR)
```

Arguments

col0	VARCHAR

Value

Unspecified.

in_search_path *DuckDB function in_search_path*

Description

Returns whether or not the database/schema are in the search path.

Usage

```
in_search_path(database_name = VARCHAR, schema_name = VARCHAR)
```

Arguments

database_name	VARCHAR
schema_name	VARCHAR

Value

BOOLEAN

SQL examples

```
in_search_path('memory', 'main')
```

`inet_client_addr` *DuckDB function inet_client_addr*

Description

DuckDB macro `inet_client_addr()`.

Usage

```
inet_client_addr()
```

Value

Unspecified.

`inet_client_port` *DuckDB function inet_client_port*

Description

DuckDB macro `inet_client_port()`.

Usage

```
inet_client_port()
```

Value

Unspecified.

`inet_server_addr` *DuckDB function inet_server_addr*

Description

DuckDB macro `inet_server_addr()`.

Usage

```
inet_server_addr()
```

Value

Unspecified.

`inet_server_port` *DuckDB function inet_server_port*

Description

DuckDB macro `inet_server_port()`.

Usage

```
inet_server_port()
```

Value

Unspecified.

`instr` *DuckDB function instr*

Description

Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.

Usage

```
instr(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
instr('test test', 'es')
```

`is_histogram_other_bin`
DuckDB function is_histogram_other_bin

Description

Whether or not the provided value is the histogram "other" bin (used for values not belonging to any provided bin).

Usage

```
is_histogram_other_bin(val = ANY)
```

Arguments

`val`	`ANY`

Value

`BOOLEAN`

SQL examples

```
is_histogram_other_bin(v)
```

`isfinite` *DuckDB function isfinite*

Description

Returns true if the floating point value is finite, false otherwise.

Arguments

`x`	`FLOAT | DOUBLE | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BOOLEAN`

Overloads

- `isfinite(x = FLOAT)`
- `isfinite(x = DOUBLE)`
- `isfinite(x = DATE)`
- `isfinite(x = TIMESTAMP)`
- `isfinite(x = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
isfinite(5.5)
```

`isinf` *DuckDB function isinf*

Description

Returns true if the floating point value is infinite, false otherwise.

Arguments

x	FLOAT | DOUBLE | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BOOLEAN

Overloads

- `isinf(x = FLOAT)`
- `isinf(x = DOUBLE)`
- `isinf(x = DATE)`
- `isinf(x = TIMESTAMP)`
- `isinf(x = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
isinf('Infinity'::float)
```

`isnan` *DuckDB function isnan*

Description

Returns true if the floating point value is not a number, false otherwise.

Arguments

x	FLOAT | DOUBLE

Value

BOOLEAN

Overloads

- `isnan(x = FLOAT)`
- `isnan(x = DOUBLE)`

SQL examples

```
isnan('NaN'::FLOAT)
```

`isodow` *DuckDB function isodow*

Description

Extract the isodow component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `isodow(ts = DATE)`
- `isodow(ts = INTERVAL)`
- `isodow(ts = TIMESTAMP)`
- `isodow(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
isodow(timestamp '2021-08-03 11:59:44.123456')
```

`isoyear` *DuckDB function isoyear*

Description

Extract the isoyear component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `isoyear(ts = DATE)`
- `isoyear(ts = INTERVAL)`
- `isoyear(ts = TIMESTAMP)`
- `isoyear(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
isoyear(timestamp '2021-08-03 11:59:44.123456')
```

`jaccard` *DuckDB function jaccard*

Description

The Jaccard similarity between two strings. Characters of different cases (e.g., `a` and `A`) are considered different. Returns a number between 0 and 1.

Usage

```
jaccard(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

`s1`	`VARCHAR`
`s2`	`VARCHAR`

Value

`DOUBLE`

SQL examples

```
jaccard('duck', 'luck')
```

`jaro_similarity` *DuckDB function jaro_similarity*

Description

The Jaro similarity between two strings. Characters of different cases (e.g., `a` and `A`) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.

Arguments

`s1`	`VARCHAR`
`s2`	`VARCHAR`
`score_cutoff`	`DOUBLE`

Value

`DOUBLE`

Overloads

- `jaro_similarity(s1 = VARCHAR, s2 = VARCHAR)`
- `jaro_similarity(s1 = VARCHAR, s2 = VARCHAR, score_cutoff = DOUBLE)`

SQL examples

```
jaro_similarity('duck', 'duckdb')
```

`jaro_winkler_similarity`
DuckDB function jaro_winkler_similarity

Description

The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., `a` and `A`) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.

Arguments

`s1`	`VARCHAR`
`s2`	`VARCHAR`
`score_cutoff`	`DOUBLE`

Value

DOUBLE

Overloads

- jaro_winkler_similarity(s1 = VARCHAR, s2 = VARCHAR)
- jaro_winkler_similarity(s1 = VARCHAR, s2 = VARCHAR, score_cutoff = DOUBLE)

SQL examples

```
jaro_winkler_similarity('duck', 'duckdb')
```

---

julian *DuckDB function julian*

Description

Extract the Julian Day number from a date or timestamp.

Arguments

ts	DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

DOUBLE

Overloads

- julian(ts = DATE)
- julian(ts = TIMESTAMP)
- julian(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
julian(timestamp '2006-01-01 12:00')
```

`kahan_sum` *DuckDB function kahan_sum*

Description

Calculates the sum using a more accurate floating point summation (Kahan Sum).

Usage

```
kahan_sum(arg = DOUBLE)
```

Arguments

`arg`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
kahan_sum(A)
```

`kurtosis` *DuckDB function kurtosis*

Description

Returns the excess kurtosis (Fisher's definition) of all input values, with a bias correction according to the sample size.

Usage

```
kurtosis(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

`kurtosis_pop` *DuckDB function kurtosis_pop*

Description

Returns the excess kurtosis (Fisher's definition) of all input values, without bias correction.

Usage

```
kurtosis_pop(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

`last` *DuckDB function last*

Description

Returns the last value of a column. This function is affected by ordering.

Arguments

`arg`	`DECIMAL | ANY`

Value

`DECIMAL | ANY`

Overloads

- `last(arg = DECIMAL)`
- `last(arg = ANY)`

SQL examples

```
last(A)
```

`last_day` *DuckDB function last_day*

Description

Returns the last day of the month.

Arguments

`ts`	`DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`DATE`

Overloads

- `last_day(ts = DATE)`
- `last_day(ts = TIMESTAMP)`
- `last_day(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
last_day(TIMESTAMP '1992-03-22 01:02:03.1234')
```

`lcase` *DuckDB function lcase*

Description

Converts `string` to lower case.

Usage

```
lcase(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
lcase('Hello')
```

`lcm` *DuckDB function lcm*

Description

Computes the least common multiple of x and y.

Arguments

x	BIGINT | HUGEINT
y	BIGINT | HUGEINT

Value

BIGINT | HUGEINT

Overloads

- `lcm(x = BIGINT, y = BIGINT)`
- `lcm(x = HUGEINT, y = HUGEINT)`

SQL examples

```
lcm(42, 57)
```

`least` *DuckDB function least*

Description

Returns the smallest value. For strings lexicographical ordering is used. Note that uppercase characters are considered "smaller" than lowercase characters, and collations are not supported.

Usage

```
least(arg1 = ANY)
```

Arguments

arg1	ANY

Value

ANY

SQL examples

```
least(42, 84)
least('abc', 'bcd', 'cde', 'EFG')
```

`least_common_multiple`
DuckDB function least_common_multiple

Description

Computes the least common multiple of x and y.

Arguments

`x`	`BIGINT | HUGEINT`
`y`	`BIGINT | HUGEINT`

Value

`BIGINT | HUGEINT`

Overloads

- `least_common_multiple(x = BIGINT, y = BIGINT)`
- `least_common_multiple(x = HUGEINT, y = HUGEINT)`

SQL examples

```
least_common_multiple(42, 57)
```

`left` *DuckDB function left*

Description

Extracts the left-most count characters.

Usage

```
left(string = VARCHAR, count = BIGINT)
```

Arguments

`string`	`VARCHAR`
`count`	`BIGINT`

Value

VARCHAR

SQL examples

```
left('Hello ', 2)
```

---

left_grapheme *DuckDB function left_grapheme*

Description

Extracts the left-most count grapheme clusters.

Usage

```
left_grapheme(string = VARCHAR, count = BIGINT)
```

Arguments

string	VARCHAR
count	BIGINT

Value

VARCHAR

SQL examples

```
left_grapheme('   ', 1)
```

---

len *DuckDB function len*

Description

Number of characters in `string`.

Returns the bit-length of the `bit` argument.

Returns the length of the `list`.

Arguments

string	VARCHAR
bit	BIT
list	ANY[]

Value

BIGINT

Overloads

- len(string = VARCHAR)
- len(bit = BIT)
- len(list = `ANY[]`)

SQL examples

```
length('Hello ')
length(42::TINYINT::BIT)
length([1,2,3])
```

---

length_grapheme *DuckDB function length_grapheme*

Description

Number of grapheme clusters in string.

Usage

```
length_grapheme(string = VARCHAR)
```

Arguments

string	VARCHAR

Value

BIGINT

SQL examples

```
length_grapheme('    ')
```

`levenshtein` *DuckDB function levenshtein*

Description

The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., `a` and `A`) are considered different.

Usage

```
levenshtein(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

`s1`	`VARCHAR`
`s2`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
levenshtein('duck', 'db')
```

`lgamma` *DuckDB function lgamma*

Description

Computes the log of the gamma function.

Usage

```
lgamma(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
lgamma(2)
```

`like_escape` *DuckDB function like_escape*

Description

Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.

Usage

```
like_escape(string = VARCHAR, like_specifier = VARCHAR, escape_character = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`like_specifier`	`VARCHAR`
`escape_character`	`VARCHAR`

Value

`BOOLEAN`

SQL examples

```
like_escape('a%c', 'a$%c', '$')
```

`list` *DuckDB function list*

Description

Returns a LIST containing all the values of a column.

Usage

```
list(arg = T)
```

Arguments

`arg`	`T`

Value

`T[]`

SQL examples

```
list(A)
```

`list_aggr` *DuckDB function list_aggr*

Description

Executes the aggregate function `function_name` on the elements of `list`.

Usage

```
list_aggr(list = `ANY[]`, function_name = VARCHAR)
```

Arguments

`list`	`ANY[]`
`function_name`	`VARCHAR`

Value

`ANY`

SQL examples

```
list_aggregate([1, 2, NULL], 'min')
```

`list_aggregate` *DuckDB function list_aggregate*

Description

Executes the aggregate function `function_name` on the elements of `list`.

Usage

```
list_aggregate(list = `ANY[]`, function_name = VARCHAR)
```

Arguments

`list`	`ANY[]`
`function_name`	`VARCHAR`

Value

`ANY`

SQL examples

```
list_aggregate([1, 2, NULL], 'min')
```

`list_any_value` *DuckDB function list_any_value*

Description

DuckDB macro `list_any_value()`.

Usage

```
list_any_value(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_append` *DuckDB function list_append*

Description

DuckDB macro `list_append()`.

Usage

```
list_append(l, e)
```

Arguments

`l`	Unspecified.
`e`	Unspecified.

Value

Unspecified.

`list_apply` *DuckDB function list_apply*

Description

Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function.

Usage

```
list_apply(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
list_apply([1, 2, 3], lambda x : x + 1)
```

`list_approx_count_distinct` *DuckDB function list_approx_count_distinct*

Description

DuckDB macro `list_approx_count_distinct()`.

Usage

```
list_approx_count_distinct(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_avg` *DuckDB function list_avg*

Description

DuckDB macro `list_avg()`.

Usage

```
list_avg(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_bit_and` *DuckDB function list_bit_and*

Description

DuckDB macro `list_bit_and()`.

Usage

```
list_bit_and(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_bit_or *DuckDB function list_bit_or*

Description

DuckDB macro `list_bit_or()`.

Usage

```
list_bit_or(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_bit_xor *DuckDB function list_bit_xor*

Description

DuckDB macro `list_bit_xor()`.

Usage

```
list_bit_xor(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_bool_and *DuckDB function list_bool_and*

Description

DuckDB macro `list_bool_and()`.

Usage

```
list_bool_and(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_bool_or *DuckDB function list_bool_or*

Description

DuckDB macro `list_bool_or()`.

Usage

```
list_bool_or(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_cat` *DuckDB function list_cat*

Description

Concatenates lists. `NULL` inputs are skipped. See also operator `||`.

Usage

```
list_cat()
```

Value

`ANY[]`

SQL examples

```
list_cat([2, 3], [4, 5, 6], [7])
```

`list_concat` *DuckDB function list_concat*

Description

Concatenates lists. `NULL` inputs are skipped. See also operator `||`.

Usage

```
list_concat()
```

Value

`ANY[]`

SQL examples

```
list_concat([2, 3], [4, 5, 6], [7])
```

`list_contains` *DuckDB function list_contains*

Description

Returns true if the list contains the element.

Usage

```
list_contains(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

`BOOLEAN`

SQL examples

```
list_contains([1, 2, NULL], 1)
```

`list_cosine_distance` *DuckDB function list_cosine_distance*

Description

Computes the cosine distance between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_cosine_distance(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_cosine_distance(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_cosine_distance([1, 2, 3], [1, 2, 3])
```

`list_cosine_similarity` *DuckDB function list_cosine_similarity*

Description

Computes the cosine similarity between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_cosine_similarity(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_cosine_similarity(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_cosine_similarity([1, 2, 3], [1, 2, 3])
```

`list_count` *DuckDB function list_count*

Description

DuckDB macro `list_count()`.

Usage

```
list_count(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_distance` *DuckDB function list_distance*

Description

Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_distance(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_distance(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_distance([1, 2, 3], [1, 2, 5])
```

`list_distinct` *DuckDB function list_distinct*

Description

Removes all duplicates and `NULL` values from a list. Does not preserve the original order.

Usage

```
list_distinct(list = `T[]`)
```

Arguments

`list`	`T[]`

Value

`T[]`

SQL examples

```
list_distinct([1, 1, NULL, -3, 1, 5])
```

`list_dot_product` *DuckDB function list_dot_product*

Description

Computes the inner product between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_dot_product(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_dot_product(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_dot_product([1, 2, 3], [1, 2, 3])
```

`list_element` *DuckDB function list_element*

Description

Extract the `index`th (1-based) value from the list.

Arguments

`list`	`T[] | VARCHAR`
`index`	`BIGINT`

Value

`T | VARCHAR`

Overloads

- `list_element(list = `T[]`, index = BIGINT)`
- `list_element(list = VARCHAR, index = BIGINT)`

SQL examples

```
list_element([4, 5, 6], 3)
```

`list_entropy` *DuckDB function list_entropy*

Description

DuckDB macro `list_entropy()`.

Usage

```
list_entropy(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_extract` *DuckDB function list_extract*

Description

Extract the `index`th (1-based) value from the list.

Arguments

`list`	`T[] | VARCHAR`
`index`	`BIGINT`

Value

`T | VARCHAR`

Overloads

- `list_extract(list = ``T[]``, index = BIGINT)`
- `list_extract(list = VARCHAR, index = BIGINT)`

SQL examples

```
list_extract([4, 5, 6], 3)
```

`list_filter` *DuckDB function list_filter*

Description

Constructs a list from those elements of the input `list` for which the `lambda` function returns `true`. DuckDB must be able to cast the `lambda` function's return type to `BOOL`. The return type of `list_filter` is the same as the input list's.

Usage

```
list_filter(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
list_filter([3, 4, 5], lambda x : x > 4)
```

`list_first` *DuckDB function list_first*

Description

DuckDB macro `list_first()`.

Usage

```
list_first(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_grade_up` *DuckDB function list_grade_up*

Description

Works like list_sort, but the results are the indexes that correspond to the position in the original list instead of the actual values.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`
`col2`	`VARCHAR`

Value

`ANY[]`

Overloads

- `list_grade_up(list = \`ANY[]\`)`
- `list_grade_up(list = \`ANY[]\`, col1 = VARCHAR)`
- `list_grade_up(list = \`ANY[]\`, col1 = VARCHAR, col2 = VARCHAR)`

SQL examples

```
list_grade_up([3, 6, 1, 2])
```

`list_has` *DuckDB function list_has*

Description

Returns true if the list contains the element.

Usage

```
list_has(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

BOOLEAN

SQL examples

```
list_has([1, 2, NULL], 1)
```

---

list_has_all *DuckDB function list_has_all*

Description

Returns true if all elements of list2 are in list1. NULLs are ignored.

Usage

```
list_has_all(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
list_has_all([1, 2, 3], [2, 3])
```

---

list_has_any *DuckDB function list_has_any*

Description

Returns true if the lists have any element in common. NULLs are ignored.

Usage

```
list_has_any(list1 = `T[]`, list2 = `T[]`)
```

Arguments

list1	T[]
list2	T[]

Value

BOOLEAN

SQL examples

```
list_has_any([1, 2, 3], [2, 3, 4])
```

---

list_histogram *DuckDB function list_histogram*

Description

DuckDB macro `list_histogram()`.

Usage

```
list_histogram(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

---

list_indexof *DuckDB function list_indexof*

Description

Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.

Usage

```
list_indexof(list = `T[]`, element = T)
```

Arguments

`list`	`T[]`
`element`	`T`

Value

INTEGER

SQL examples

```
list_indexof([1, 2, NULL], 2)
```

`list_inner_product` *DuckDB function list_inner_product*

Description

Computes the inner product between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_inner_product(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_inner_product(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_inner_product([1, 2, 3], [1, 2, 3])
```

`list_intersect` *DuckDB function list_intersect*

Description

DuckDB macro `list_intersect()`.

Usage

```
list_intersect(l1, l2)
```

Arguments

`l1`	Unspecified.
`l2`	Unspecified.

Value

Unspecified.

list_kurtosis *DuckDB function list_kurtosis*

Description

DuckDB macro `list_kurtosis()`.

Usage

```
list_kurtosis(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_kurtosis_pop *DuckDB function list_kurtosis_pop*

Description

DuckDB macro `list_kurtosis_pop()`.

Usage

```
list_kurtosis_pop(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_last *DuckDB function list_last*

Description

DuckDB macro `list_last()`.

Usage

```
list_last(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_mad *DuckDB function list_mad*

Description

DuckDB macro `list_mad()`.

Usage

```
list_mad(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_max *DuckDB function list_max*

Description

DuckDB macro `list_max()`.

Usage

```
list_max(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

list_median *DuckDB function list_median*

Description

DuckDB macro `list_median()`.

Usage

```
list_median(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_min` *DuckDB function list_min*

Description

DuckDB macro `list_min()`.

Usage

```
list_min(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_mode` *DuckDB function list_mode*

Description

DuckDB macro `list_mode()`.

Usage

```
list_mode(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_negative_dot_product`

DuckDB function list_negative_dot_product

Description

Computes the negative inner product between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_negative_dot_product(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_negative_dot_product(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_negative_dot_product([1, 2, 3], [1, 2, 3])
```

`list_negative_inner_product`

DuckDB function list_negative_inner_product

Description

Computes the negative inner product between two same-sized lists.

Arguments

`list1`	`FLOAT[] | DOUBLE[]`
`list2`	`FLOAT[] | DOUBLE[]`

Value

`FLOAT | DOUBLE`

Overloads

- `list_negative_inner_product(list1 = `FLOAT[]`, list2 = `FLOAT[]`)`
- `list_negative_inner_product(list1 = `DOUBLE[]`, list2 = `DOUBLE[]`)`

SQL examples

```
list_negative_inner_product([1, 2, 3], [1, 2, 3])
```

`list_pack` *DuckDB function list_pack*

Description

Creates a LIST containing the argument values.

Arguments

any	T

Value

`"NULL"[] | T[]`

Overloads

- `list_pack()`
- `list_pack(any = T)`

SQL examples

```
list_pack(4, 5, 6)
```

`list_position` *DuckDB function list_position*

Description

Returns the index of the `element` if the `list` contains the `element`. If the `element` is not found, it returns `NULL`.

Usage

```
list_position(list = `T[]`, element = T)
```

Arguments

list	T[]
element	T

Value

INTEGER

SQL examples

```
list_position([1, 2, NULL], 2)
```

---

list_prepend *DuckDB function list_prepend*

Description

DuckDB macro `list_prepend()`.

Usage

```
list_prepend(e, l)
```

Arguments

e	Unspecified.
l	Unspecified.

Value

Unspecified.

---

list_product *DuckDB function list_product*

Description

DuckDB macro `list_product()`.

Usage

```
list_product(l)
```

Arguments

l	Unspecified.

Value

Unspecified.

`list_reduce` *DuckDB function list_reduce*

Description

Reduces all elements of the input `list` into a single scalar value by executing the `lambda` function on a running result and the next list element. The `lambda` function has an optional `initial_value` argument.

Arguments

`list`	`ANY[]`
`initial_value`	`ANY`
`lambda(x, y)`	`LAMBDA`

Value

`ANY`

Overloads

- `list_reduce(list = `ANY[]`, `lambda(x,y)` = LAMBDA)`
- `list_reduce(list = `ANY[]`, `lambda(x,y)` = LAMBDA, initial_value = ANY)`

SQL examples

```
list_reduce([1, 2, 3], lambda x, y : x + y)
```

`list_resize` *DuckDB function list_resize*

Description

Resizes the `list` to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.

Arguments

`list`	`ANY[]`
`size[`	`ANY`
`value]`	`ANY`

Value

`ANY[]`

Overloads

- `list_resize(list = `ANY[]`, `size[` = ANY)`
- `list_resize(list = `ANY[]`, `size[` = ANY, `value]` = ANY)`

SQL examples

```
list_resize([1, 2, 3], 5, 0)
```

---

list_reverse *DuckDB function list_reverse*

Description

DuckDB macro `list_reverse()`.

Usage

```
list_reverse(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

---

list_reverse_sort *DuckDB function list_reverse_sort*

Description

Sorts the elements of the list in reverse order.

Arguments

`list`	`ANY[]`
`col1`	`VARCHAR`

Value

`ANY[]`

Overloads

- `list_reverse_sort(list = `ANY[]`)`
- `list_reverse_sort(list = `ANY[]`, col1 = VARCHAR)`

SQL examples

```
list_reverse_sort([3, 6, 1, 2])
```

`list_select` *DuckDB function list_select*

Description

Returns a list based on the elements selected by the `index_list`.

Usage

```
list_select(value_list = `T[]`, index_list = `BIGINT[]`)
```

Arguments

`value_list`	`T[]`
`index_list`	`BIGINT[]`

Value

`T[]`

SQL examples

```
list_select([10, 20, 30, 40], [1, 4])
```

`list_sem` *DuckDB function list_sem*

Description

DuckDB macro `list_sem()`.

Usage

```
list_sem(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_skewness` *DuckDB function list_skewness*

Description

DuckDB macro `list_skewness()`.

Usage

```
list_skewness(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_slice` *DuckDB function list_slice*

Description

Extracts a sublist or substring using slice conventions. Negative values are accepted. list_slice with added step feature.

Arguments

`list`	`ANY`
`begin`	`ANY`
`end`	`ANY`
`step`	`BIGINT`

Value

`ANY`

Overloads

- `list_slice(list = ANY, begin = ANY, end = ANY)`
- `list_slice(list = ANY, begin = ANY, end = ANY, step = BIGINT)`

SQL examples

```
list_slice([4, 5, 6], 2, 3)
list_slice([4, 5, 6], 1, 3, 2)
```

list_sort — *DuckDB function list_sort*

Description

Sorts the elements of the list.

Arguments

list	ANY[]
col1	VARCHAR
col2	VARCHAR

Value

ANY[]

Overloads

- list_sort(list = `ANY[]`)
- list_sort(list = `ANY[]`, col1 = VARCHAR)
- list_sort(list = `ANY[]`, col1 = VARCHAR, col2 = VARCHAR)

SQL examples

```
list_sort([3, 6, 1, 2])
```

list_stddev_pop — *DuckDB function list_stddev_pop*

Description

DuckDB macro list_stddev_pop().

Usage

```
list_stddev_pop(l)
```

Arguments

l	Unspecified.

Value

Unspecified.

`list_stddev_samp` *DuckDB function list_stddev_samp*

Description

DuckDB macro `list_stddev_samp()`.

Usage

```
list_stddev_samp(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_string_agg` *DuckDB function list_string_agg*

Description

DuckDB macro `list_string_agg()`.

Usage

```
list_string_agg(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_sum` *DuckDB function list_sum*

Description

DuckDB macro `list_sum()`.

Usage

```
list_sum(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_transform` *DuckDB function list_transform*

Description

Returns a list that is the result of applying the `lambda` function to each element of the input `list`. The return type is defined by the return type of the `lambda` function.

Usage

```
list_transform(list = `ANY[]`, `lambda(x)` = LAMBDA)
```

Arguments

`list`	`ANY[]`
`lambda(x)`	`LAMBDA`

Value

`ANY[]`

SQL examples

```
list_transform([1, 2, 3], lambda x : x + 1)
```

`list_unique` *DuckDB function list_unique*

Description

Counts the unique elements of a `list`.

Usage

```
list_unique(list = `ANY[]`)
```

Arguments

`list`	`ANY[]`

Value

`UBIGINT`

SQL examples

```
list_unique([1, 1, NULL, -3, 1, 5])
```

`list_value` *DuckDB function list_value*

Description

Creates a LIST containing the argument values.

Arguments

`any`	`T`

Value

`"NULL"[] | T[]`

Overloads

- `list_value()`
- `list_value(any = T)`

SQL examples

```
list_value(4, 5, 6)
```

`list_var_pop` *DuckDB function list_var_pop*

Description

DuckDB macro `list_var_pop()`.

Usage

```
list_var_pop(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_var_samp` *DuckDB function list_var_samp*

Description

DuckDB macro `list_var_samp()`.

Usage

```
list_var_samp(l)
```

Arguments

`l`	Unspecified.

Value

Unspecified.

`list_where` *DuckDB function list_where*

Description

Returns a list with the `BOOLEAN`s in `mask_list` applied as a mask to the `value_list`.

Usage

```
list_where(value_list = `T[]`, mask_list = `BOOLEAN[]`)
```

Arguments

`value_list`	`T[]`
`mask_list`	`BOOLEAN[]`

Value

`T[]`

SQL examples

```
list_where([10, 20, 30, 40], [true, false, false, true])
```

`list_zip` *DuckDB function list_zip*

Description

Zips n `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of n elements from each list `list_1`, …, `list_n`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.

Usage

```
list_zip()
```

Value

`STRUCT[]`

SQL examples

```
list_zip([1, 2], [3, 4], [5, 6])
list_zip([1, 2], [3, 4], [5, 6, 7])
list_zip([1, 2], [3, 4], [5, 6, 7], true)
```

`listagg` *DuckDB function listagg*

Description

Concatenates the column string values with an optional separator.

Arguments

`str`	`ANY`
`arg`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `listagg(str = ANY)`
- `listagg(str = ANY, arg = VARCHAR)`

SQL examples

```
listagg(A, '-')
```

`ln` *DuckDB function ln*

Description

Computes the natural logarithm of x.

Usage

```
ln(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
ln(2)
```

`log` *DuckDB function log*

Description

Computes the logarithm of x to base b. b may be omitted, in which case the default 10.

Arguments

`b`	`DOUBLE`
`x`	`DOUBLE`

Value

`DOUBLE`

Overloads

- `log(b = DOUBLE)`
- `log(b = DOUBLE, x = DOUBLE)`

SQL examples

```
log(2, 64)
```

`log10` *DuckDB function log10*

Description

Computes the 10-log of x.

Usage

```
log10(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
log10(1000)
```

`log2` *DuckDB function log2*

Description

Computes the 2-log of x.

Usage

```
log2(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
log2(8)
```

`lower` *DuckDB function lower*

Description

Converts `string` to lower case.

Usage

```
lower(string = VARCHAR)
```

Arguments

string	VARCHAR

Value

VARCHAR

SQL examples

```
lower('Hello')
```

`lpad` *DuckDB function lpad*

Description

Pads the `string` with the `character` on the left until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.

Usage

```
lpad(string = VARCHAR, count = INTEGER, character = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`count`	`INTEGER`
`character`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
lpad('hello', 8, '>')
```

`ltrim` *DuckDB function ltrim*

Description

Removes any occurrences of any of the `characters` from the left side of the `string`. `characters` defaults to `space`.

Arguments

`string`	`VARCHAR`
`characters`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `ltrim(string = VARCHAR)`
- `ltrim(string = VARCHAR, characters = VARCHAR)`

SQL examples

```
ltrim('    test  ')
ltrim('>>>>test<<', '><')
```

`mad` *DuckDB function mad*

Description

Returns the median absolute deviation for the values within x. NULL values are ignored. Temporal types return a positive INTERVAL. .

Arguments

x	DECIMAL | FLOAT | DOUBLE | DATE | TIMESTAMP | TIME | TIMESTAMP WITH TIME ZONE |

Value

DECIMAL | FLOAT | DOUBLE | INTERVAL

Overloads

- mad(x = DECIMAL)
- mad(x = FLOAT)
- mad(x = DOUBLE)
- mad(x = DATE)
- mad(x = TIMESTAMP)
- mad(x = TIME)
- mad(x = `TIMESTAMP WITH TIME ZONE`)
- mad(x = `TIME WITH TIME ZONE`)

SQL examples

```
mad(x)
```

`make_date` *DuckDB function make_date*

Description

The date for the given parts.

The date for the given struct.

Arguments

col0	INTEGER
year	BIGINT
month	BIGINT
day	BIGINT
date-struct	STRUCT("year" BIGINT, "month" BIGINT, "day" BIGINT)

Value

DATE

Overloads

- `make_date(col0 = INTEGER)`
- `make_date(year = BIGINT, month = BIGINT, day = BIGINT)`
- `make_date(`date-struct` = `STRUCT("year" BIGINT, "month" BIGINT, "day" BIGINT)`)`

SQL examples

```
make_date(1992, 9, 20)
make_date({'year': 2024, 'month': 11, 'day': 14})
```

`make_time` *DuckDB function make_time*

Description

The time for the given parts.

Usage

```
make_time(hour = BIGINT, minute = BIGINT, seconds = DOUBLE)
```

Arguments

hour	BIGINT
minute	BIGINT
seconds	DOUBLE

Value

TIME

SQL examples

```
make_time(13, 34, 27.123456)
```

---

make_timestamp *DuckDB function make_timestamp*

Description

The timestamp for the given parts.

Arguments

year	BIGINT
month	BIGINT
day	BIGINT
hour	BIGINT
minute	BIGINT
seconds	DOUBLE

Value

TIMESTAMP

Overloads

- make_timestamp(year = BIGINT, month = BIGINT, day = BIGINT, hour = BIGINT, minute = BIGINT, seconds = DOUBLE)
- make_timestamp(year = BIGINT)

SQL examples

```
make_timestamp(1992, 9, 20, 13, 34, 27.123456)
```

`make_timestamp_ms` *DuckDB function make_timestamp_ms*

Description

The timestamp for the given microseconds since the epoch.

Usage

```
make_timestamp_ms(nanos = BIGINT)
```

Arguments

nanos	BIGINT

Value

```
TIMESTAMP
```

SQL examples

```
make_timestamp_ms(1732117793000000)
```

`make_timestamp_ns` *DuckDB function make_timestamp_ns*

Description

The timestamp for the given nanoseconds since epoch.

Usage

```
make_timestamp_ns(nanos = BIGINT)
```

Arguments

nanos	BIGINT

Value

```
TIMESTAMP_NS
```

SQL examples

```
make_timestamp_ns(1732117793000000000)
```

`map` *DuckDB function map*

Description

Creates a map from a set of keys and values.

Arguments

`keys`	`K[]`
`values`	`V[]`

Value

`MAP("NULL", "NULL") | MAP(K, V)`

Overloads

- `map()`
- `map(keys = `K[]`, values = `V[]`)`

SQL examples

```
map(['key1', 'key2'], ['val1', 'val2'])
```

`map_concat` *DuckDB function map_concat*

Description

Returns a map created from merging the input maps, on key collision the value is taken from the last map with that key.

Usage

```
map_concat()
```

Value

`LIST`

SQL examples

```
map_concat(map([1, 2], ['a', 'b']), map([2, 3], ['c', 'd']));
```

map_contains *DuckDB function map_contains*

Description

Checks if a map contains a given key.

Usage

```
map_contains(map = `MAP(K, V)`, key = K)
```

Arguments

map	MAP(K, V)
key	K

Value

BOOLEAN

SQL examples

```
map_contains(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 'key2')
```

map_contains_entry *DuckDB function map_contains_entry*

Description

DuckDB macro `map_contains_entry()`.

Usage

```
map_contains_entry(map, key, value)
```

Arguments

map	Unspecified.
key	Unspecified.
value	Unspecified.

Value

Unspecified.

map_contains_value *DuckDB function map_contains_value*

Description

DuckDB macro `map_contains_value()`.

Usage

```
map_contains_value(map, value)
```

Arguments

`map`	Unspecified.
`value`	Unspecified.

Value

Unspecified.

map_entries *DuckDB function map_entries*

Description

Returns the map entries as a list of keys/values.

Usage

```
map_entries(map = `MAP(K, V)`)
```

Arguments

`map`	`MAP(K, V)`

Value

`STRUCT("key" K, "value" V)[]`

SQL examples

```
map_entries(map(['key'], ['val']))
```

`map_extract` *DuckDB function map_extract*

Description

Returns a list containing the value for a given key or an empty list if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.

Usage

```
map_extract(map = `MAP(K, V)`, key = K)
```

Arguments

map	MAP(K, V)
key	K

Value

`V[]`

SQL examples

```
map_extract(map(['key'], ['val']), 'key')
```

`map_extract_value` *DuckDB function map_extract_value*

Description

Returns the value for a given key or NULL if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.

Usage

```
map_extract_value(map = `MAP(K, V)`, key = K)
```

Arguments

map	MAP(K, V)
key	K

Value

`V`

SQL examples

```
map_extract_value(map(['key'], ['val']), 'key')
```

`map_from_entries` *DuckDB function map_from_entries*

Description

Returns a map created from the entries of the array.

Usage

```
map_from_entries(map = `STRUCT(K, V)[]`)
```

Arguments

map	STRUCT(K, V)[]

Value

```
MAP(K, V)
```

SQL examples

```
map_from_entries([{k: 5, v: 'val1'}, {k: 3, v: 'val2'}]);
```

`map_keys` *DuckDB function map_keys*

Description

Returns the keys of a map as a list.

Usage

```
map_keys(map = `MAP(K, V)`)
```

Arguments

map	MAP(K, V)

Value

```
K[]
```

SQL examples

```
map_keys(map(['key'], ['val']))
```

`map_to_pg_oid` *DuckDB function map_to_pg_oid*

Description

DuckDB macro `map_to_pg_oid()`.

Usage

```
map_to_pg_oid(type_name)
```

Arguments

`type_name`	Unspecified.

Value

Unspecified.

`map_values` *DuckDB function map_values*

Description

Returns the values of a map as a list.

Usage

```
map_values(map = `MAP(K, V)`)
```

Arguments

`map`	`MAP(K, V)`

Value

`V[]`

SQL examples

```
map_values(map(['key'], ['val']))
```

---

max *DuckDB function max*

Description

Returns the maximum value present in arg.

Arguments

arg	ANY
col1	BIGINT

Value

ANY | ANY[]

Overloads

- max(arg = ANY)
- max(arg = ANY, col1 = BIGINT)

SQL examples

```
max(A)
```

---

max_by *DuckDB function max_by*

Description

Finds the row with the maximum val. Calculates the non-NULL arg expression at that row.

Arguments

arg	INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON
val	INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT
col2	BIGINT

Value

INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC

Overloads

- max_by(arg = INTEGER, val = INTEGER)
- max_by(arg = INTEGER, val = BIGINT)
- max_by(arg = INTEGER, val = HUGEINT)
- max_by(arg = INTEGER, val = DOUBLE)
- max_by(arg = INTEGER, val = VARCHAR)
- max_by(arg = INTEGER, val = DATE)
- max_by(arg = INTEGER, val = TIMESTAMP)
- max_by(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = INTEGER, val = BLOB)
- max_by(arg = BIGINT, val = INTEGER)
- max_by(arg = BIGINT, val = BIGINT)
- max_by(arg = BIGINT, val = HUGEINT)
- max_by(arg = BIGINT, val = DOUBLE)
- max_by(arg = BIGINT, val = VARCHAR)
- max_by(arg = BIGINT, val = DATE)
- max_by(arg = BIGINT, val = TIMESTAMP)
- max_by(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = BIGINT, val = BLOB)
- max_by(arg = DOUBLE, val = INTEGER)
- max_by(arg = DOUBLE, val = BIGINT)
- max_by(arg = DOUBLE, val = HUGEINT)
- max_by(arg = DOUBLE, val = DOUBLE)
- max_by(arg = DOUBLE, val = VARCHAR)
- max_by(arg = DOUBLE, val = DATE)
- max_by(arg = DOUBLE, val = TIMESTAMP)
- max_by(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = DOUBLE, val = BLOB)
- max_by(arg = VARCHAR, val = INTEGER)
- max_by(arg = VARCHAR, val = BIGINT)
- max_by(arg = VARCHAR, val = HUGEINT)
- max_by(arg = VARCHAR, val = DOUBLE)
- max_by(arg = VARCHAR, val = VARCHAR)
- max_by(arg = VARCHAR, val = DATE)
- max_by(arg = VARCHAR, val = TIMESTAMP)
- max_by(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = VARCHAR, val = BLOB)

- max_by(arg = DATE, val = INTEGER)
- max_by(arg = DATE, val = BIGINT)
- max_by(arg = DATE, val = HUGEINT)
- max_by(arg = DATE, val = DOUBLE)
- max_by(arg = DATE, val = VARCHAR)
- max_by(arg = DATE, val = DATE)
- max_by(arg = DATE, val = TIMESTAMP)
- max_by(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = DATE, val = BLOB)
- max_by(arg = TIMESTAMP, val = INTEGER)
- max_by(arg = TIMESTAMP, val = BIGINT)
- max_by(arg = TIMESTAMP, val = HUGEINT)
- max_by(arg = TIMESTAMP, val = DOUBLE)
- max_by(arg = TIMESTAMP, val = VARCHAR)
- max_by(arg = TIMESTAMP, val = DATE)
- max_by(arg = TIMESTAMP, val = TIMESTAMP)
- max_by(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = TIMESTAMP, val = BLOB)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- max_by(arg = BLOB, val = INTEGER)
- max_by(arg = BLOB, val = BIGINT)
- max_by(arg = BLOB, val = HUGEINT)
- max_by(arg = BLOB, val = DOUBLE)
- max_by(arg = BLOB, val = VARCHAR)
- max_by(arg = BLOB, val = DATE)
- max_by(arg = BLOB, val = TIMESTAMP)
- max_by(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = BLOB, val = BLOB)
- max_by(arg = DECIMAL, val = INTEGER)

- max_by(arg = DECIMAL, val = BIGINT)
- max_by(arg = DECIMAL, val = HUGEINT)
- max_by(arg = DECIMAL, val = DOUBLE)
- max_by(arg = DECIMAL, val = VARCHAR)
- max_by(arg = DECIMAL, val = DATE)
- max_by(arg = DECIMAL, val = TIMESTAMP)
- max_by(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = DECIMAL, val = BLOB)
- max_by(arg = ANY, val = INTEGER)
- max_by(arg = ANY, val = BIGINT)
- max_by(arg = ANY, val = HUGEINT)
- max_by(arg = ANY, val = DOUBLE)
- max_by(arg = ANY, val = VARCHAR)
- max_by(arg = ANY, val = DATE)
- max_by(arg = ANY, val = TIMESTAMP)
- max_by(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- max_by(arg = ANY, val = BLOB)
- max_by(arg = ANY, val = ANY)
- max_by(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
max_by(A, B)
```

---

md5 *DuckDB function md5*

Description

Returns the MD5 hash of the `string` as a `VARCHAR`.

Returns the MD5 hash of the `blob` as a `VARCHAR`.

Arguments

`string`	`VARCHAR`
`blob`	`BLOB`

Value

`VARCHAR`

Overloads

- `md5(string = VARCHAR)`
- `md5(blob = BLOB)`

SQL examples

```
md5('abc')
md5('\xAA\xBB'::BLOB)
```

`md5_number` *DuckDB function md5_number*

Description

Returns the MD5 hash of the `string` as a `HUGEINT`.

Returns the MD5 hash of the `blob` as a `HUGEINT`.

Arguments

`string`	`VARCHAR`
`blob`	`BLOB`

Value

`UHUGEINT`

Overloads

- `md5_number(string = VARCHAR)`
- `md5_number(blob = BLOB)`

SQL examples

```
md5_number('abc')
md5_number('\xAA\xBB'::BLOB)
```

`md5_number_lower` *DuckDB function md5_number_lower*

Description

DuckDB macro `md5_number_lower()`.

Usage

```
md5_number_lower(param)
```

Arguments

`param`	Unspecified.

Value

Unspecified.

`md5_number_upper` *DuckDB function md5_number_upper*

Description

DuckDB macro `md5_number_upper()`.

Usage

```
md5_number_upper(param)
```

Arguments

`param`	Unspecified.

Value

Unspecified.

`mean` *DuckDB function mean*

Description

Calculates the average value for all tuples in x.

Arguments

x	DECIMAL | SMALLINT | INTEGER | BIGINT | HUGEINT | INTERVAL | DOUBLE | TIMESTAMP

Value

DECIMAL | DOUBLE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE | TIME | TIME WITH TIME ZONE

Overloads

- `mean(x = DECIMAL)`
- `mean(x = SMALLINT)`
- `mean(x = INTEGER)`
- `mean(x = BIGINT)`
- `mean(x = HUGEINT)`
- `mean(x = INTERVAL)`
- `mean(x = DOUBLE)`
- `mean(x = TIMESTAMP)`
- `mean(x = `TIMESTAMP WITH TIME ZONE`)`
- `mean(x = TIME)`
- `mean(x = `TIME WITH TIME ZONE`)`

SQL examples

```
SUM(x) / COUNT(*)
```

`median` *DuckDB function median*

Description

Returns the middle value of the set. NULL values are ignored. For even value counts, interpolate-able types (numeric, date/time) return the average of the two middle values. Non-interpolate-able types (everything else) return the lower of the two middle values.

Usage

```
median(x = ANY)
```

Arguments

x	ANY

Value

ANY

SQL examples

```
median(x)
```

---

metadata_info *DuckDB function metadata_info*

Description

DuckDB function `metadata_info()`.

Usage

```
metadata_info()
```

Value

Unspecified.

---

microsecond *DuckDB function microsecond*

Description

Extract the microsecond component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W

Value

BIGINT

Overloads

- microsecond(ts = DATE)
- microsecond(ts = INTERVAL)
- microsecond(ts = TIME)
- microsecond(ts = TIMESTAMP)
- microsecond(ts = `TIME WITH TIME ZONE`)
- microsecond(ts = TIME_NS)
- microsecond(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
microsecond(timestamp '2021-08-03 11:59:44.123456')
```

---

millennium *DuckDB function millennium*

Description

Extract the millennium component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- millennium(ts = DATE)
- millennium(ts = INTERVAL)
- millennium(ts = TIMESTAMP)
- millennium(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
millennium(timestamp '2021-08-03 11:59:44.123456')
```

`millisecond` *DuckDB function millisecond*

Description

Extract the millisecond component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W`

Value

`BIGINT`

Overloads

- `millisecond(ts = DATE)`
- `millisecond(ts = INTERVAL)`
- `millisecond(ts = TIME)`
- `millisecond(ts = TIMESTAMP)`
- `millisecond(ts = `TIME WITH TIME ZONE`)`
- `millisecond(ts = TIME_NS)`
- `millisecond(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
millisecond(timestamp '2021-08-03 11:59:44.123456')
```

`min` *DuckDB function min*

Description

Returns the minimum value present in arg.

Arguments

`arg`	`ANY`
`col1`	`BIGINT`

Value

`ANY | ANY[]`

Overloads

- `min(arg = ANY)`
- `min(arg = ANY, col1 = BIGINT)`

SQL examples

```
min(A)
```

`min_by` *DuckDB function min_by*

Description

Finds the row with the minimum val. Calculates the non-NULL arg expression at that row.

Arguments

`arg`	`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZON`
`val`	`INTEGER | BIGINT | HUGEINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WIT`
`col2`	`BIGINT`

Value

`INTEGER | BIGINT | DOUBLE | VARCHAR | DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE | BLOB | DEC`

Overloads

- `min_by(arg = INTEGER, val = INTEGER)`
- `min_by(arg = INTEGER, val = BIGINT)`
- `min_by(arg = INTEGER, val = HUGEINT)`
- `min_by(arg = INTEGER, val = DOUBLE)`
- `min_by(arg = INTEGER, val = VARCHAR)`
- `min_by(arg = INTEGER, val = DATE)`
- `min_by(arg = INTEGER, val = TIMESTAMP)`
- `min_by(arg = INTEGER, val = `TIMESTAMP WITH TIME ZONE`)`
- `min_by(arg = INTEGER, val = BLOB)`
- `min_by(arg = BIGINT, val = INTEGER)`
- `min_by(arg = BIGINT, val = BIGINT)`
- `min_by(arg = BIGINT, val = HUGEINT)`
- `min_by(arg = BIGINT, val = DOUBLE)`
- `min_by(arg = BIGINT, val = VARCHAR)`
- `min_by(arg = BIGINT, val = DATE)`

- min_by(arg = BIGINT, val = TIMESTAMP)
- min_by(arg = BIGINT, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = BIGINT, val = BLOB)
- min_by(arg = DOUBLE, val = INTEGER)
- min_by(arg = DOUBLE, val = BIGINT)
- min_by(arg = DOUBLE, val = HUGEINT)
- min_by(arg = DOUBLE, val = DOUBLE)
- min_by(arg = DOUBLE, val = VARCHAR)
- min_by(arg = DOUBLE, val = DATE)
- min_by(arg = DOUBLE, val = TIMESTAMP)
- min_by(arg = DOUBLE, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = DOUBLE, val = BLOB)
- min_by(arg = VARCHAR, val = INTEGER)
- min_by(arg = VARCHAR, val = BIGINT)
- min_by(arg = VARCHAR, val = HUGEINT)
- min_by(arg = VARCHAR, val = DOUBLE)
- min_by(arg = VARCHAR, val = VARCHAR)
- min_by(arg = VARCHAR, val = DATE)
- min_by(arg = VARCHAR, val = TIMESTAMP)
- min_by(arg = VARCHAR, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = VARCHAR, val = BLOB)
- min_by(arg = DATE, val = INTEGER)
- min_by(arg = DATE, val = BIGINT)
- min_by(arg = DATE, val = HUGEINT)
- min_by(arg = DATE, val = DOUBLE)
- min_by(arg = DATE, val = VARCHAR)
- min_by(arg = DATE, val = DATE)
- min_by(arg = DATE, val = TIMESTAMP)
- min_by(arg = DATE, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = DATE, val = BLOB)
- min_by(arg = TIMESTAMP, val = INTEGER)
- min_by(arg = TIMESTAMP, val = BIGINT)
- min_by(arg = TIMESTAMP, val = HUGEINT)
- min_by(arg = TIMESTAMP, val = DOUBLE)
- min_by(arg = TIMESTAMP, val = VARCHAR)
- min_by(arg = TIMESTAMP, val = DATE)
- min_by(arg = TIMESTAMP, val = TIMESTAMP)

- min_by(arg = TIMESTAMP, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = TIMESTAMP, val = BLOB)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = INTEGER)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = BIGINT)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = HUGEINT)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = DOUBLE)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = VARCHAR)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = DATE)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = TIMESTAMP)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = `TIMESTAMP WITH TIME ZONE`, val = BLOB)
- min_by(arg = BLOB, val = INTEGER)
- min_by(arg = BLOB, val = BIGINT)
- min_by(arg = BLOB, val = HUGEINT)
- min_by(arg = BLOB, val = DOUBLE)
- min_by(arg = BLOB, val = VARCHAR)
- min_by(arg = BLOB, val = DATE)
- min_by(arg = BLOB, val = TIMESTAMP)
- min_by(arg = BLOB, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = BLOB, val = BLOB)
- min_by(arg = DECIMAL, val = INTEGER)
- min_by(arg = DECIMAL, val = BIGINT)
- min_by(arg = DECIMAL, val = HUGEINT)
- min_by(arg = DECIMAL, val = DOUBLE)
- min_by(arg = DECIMAL, val = VARCHAR)
- min_by(arg = DECIMAL, val = DATE)
- min_by(arg = DECIMAL, val = TIMESTAMP)
- min_by(arg = DECIMAL, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = DECIMAL, val = BLOB)
- min_by(arg = ANY, val = INTEGER)
- min_by(arg = ANY, val = BIGINT)
- min_by(arg = ANY, val = HUGEINT)
- min_by(arg = ANY, val = DOUBLE)
- min_by(arg = ANY, val = VARCHAR)
- min_by(arg = ANY, val = DATE)
- min_by(arg = ANY, val = TIMESTAMP)
- min_by(arg = ANY, val = `TIMESTAMP WITH TIME ZONE`)
- min_by(arg = ANY, val = BLOB)
- min_by(arg = ANY, val = ANY)
- min_by(arg = ANY, val = ANY, col2 = BIGINT)

SQL examples

```
min_by(A, B)
```

`minute` *DuckDB function minute*

Description

Extract the minute component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W`

Value

`BIGINT`

Overloads

- `minute(ts = DATE)`
- `minute(ts = INTERVAL)`
- `minute(ts = TIME)`
- `minute(ts = TIMESTAMP)`
- `minute(ts = \`TIME WITH TIME ZONE\`)`
- `minute(ts = TIME_NS)`
- `minute(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
minute(timestamp '2021-08-03 11:59:44.123456')
```

`mismatches` *DuckDB function mismatches*

Description

The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., `a` and `A`) are considered different.

Usage

```
mismatches(s1 = VARCHAR, s2 = VARCHAR)
```

Arguments

s1	VARCHAR
s2	VARCHAR

Value

BIGINT

SQL examples

```
mismatches('duck', 'luck')
```

---

mod *DuckDB function mod*

Description

DuckDB function `mod()`.

Arguments

col0	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Overloads

- mod(col0 = TINYINT, col1 = TINYINT)
- mod(col0 = SMALLINT, col1 = SMALLINT)
- mod(col0 = INTEGER, col1 = INTEGER)
- mod(col0 = BIGINT, col1 = BIGINT)
- mod(col0 = HUGEINT, col1 = HUGEINT)
- mod(col0 = FLOAT, col1 = FLOAT)
- mod(col0 = DOUBLE, col1 = DOUBLE)
- mod(col0 = DECIMAL, col1 = DECIMAL)
- mod(col0 = UTINYINT, col1 = UTINYINT)
- mod(col0 = USMALLINT, col1 = USMALLINT)
- mod(col0 = UINTEGER, col1 = UINTEGER)
- mod(col0 = UBIGINT, col1 = UBIGINT)
- mod(col0 = UHUGEINT, col1 = UHUGEINT)

`mode` *DuckDB function mode*

Description

Returns the most frequent value for the values within x. NULL values are ignored.

Usage

```
mode(x = ANY)
```

Arguments

x	ANY

Value

ANY

`month` *DuckDB function month*

Description

Extract the month component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- `month(ts = DATE)`
- `month(ts = INTERVAL)`
- `month(ts = TIMESTAMP)`
- `month(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
month(timestamp '2021-08-03 11:59:44.123456')
```

`monthname` *DuckDB function monthname*

Description

The (English) name of the month.

Arguments

`ts`	`DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`VARCHAR`

Overloads

- `monthname(ts = DATE)`
- `monthname(ts = TIMESTAMP)`
- `monthname(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
monthname(TIMESTAMP '1992-09-20')
```

`multiply` *DuckDB function multiply*

Description

DuckDB function `multiply()`.

Arguments

`col0`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | INTERVAL`
`col1`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | INTERVAL`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | INTERVAL`

Overloads

- `multiply(col0 = TINYINT, col1 = TINYINT)`
- `multiply(col0 = SMALLINT, col1 = SMALLINT)`
- `multiply(col0 = INTEGER, col1 = INTEGER)`
- `multiply(col0 = BIGINT, col1 = BIGINT)`
- `multiply(col0 = HUGEINT, col1 = HUGEINT)`
- `multiply(col0 = FLOAT, col1 = FLOAT)`
- `multiply(col0 = DOUBLE, col1 = DOUBLE)`
- `multiply(col0 = DECIMAL, col1 = DECIMAL)`
- `multiply(col0 = UTINYINT, col1 = UTINYINT)`
- `multiply(col0 = USMALLINT, col1 = USMALLINT)`
- `multiply(col0 = UINTEGER, col1 = UINTEGER)`
- `multiply(col0 = UBIGINT, col1 = UBIGINT)`
- `multiply(col0 = UHUGEINT, col1 = UHUGEINT)`
- `multiply(col0 = INTERVAL, col1 = DOUBLE)`
- `multiply(col0 = DOUBLE, col1 = INTERVAL)`
- `multiply(col0 = BIGINT, col1 = INTERVAL)`
- `multiply(col0 = INTERVAL, col1 = BIGINT)`

`nanosecond` *DuckDB function nanosecond*

Description

Extract the nanosecond component from a date or timestamp.

Arguments

`tsns`	`DATE | TIMESTAMP | INTERVAL | TIME | TIME_NS | TIME WITH TIME ZONE | TIMESTAMP_N`

Value

`BIGINT`

Overloads

- `nanosecond(tsns = DATE)`
- `nanosecond(tsns = TIMESTAMP)`
- `nanosecond(tsns = INTERVAL)`
- `nanosecond(tsns = TIME)`
- `nanosecond(tsns = TIME_NS)`
- `nanosecond(tsns = `TIME WITH TIME ZONE`)`
- `nanosecond(tsns = TIMESTAMP_NS)`
- `nanosecond(tsns = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
nanosecond(timestamp_ns '2021-08-03 11:59:44.123456789')
```

---

nextafter *DuckDB function nextafter*

Description

Returns the next floating point value after x in the direction of y.

Arguments

x	DOUBLE | FLOAT
y	DOUBLE | FLOAT

Value

DOUBLE | FLOAT

Overloads

- nextafter(x = DOUBLE, y = DOUBLE)
- nextafter(x = FLOAT, y = FLOAT)

SQL examples

```
nextafter(1::float, 2::float)
```

---

nextval *DuckDB function nextval*

Description

Return the following value of the sequence.

Usage

```
nextval(`'sequence_name'` = VARCHAR)
```

Arguments

'sequence_name'	VARCHAR

Value

BIGINT

SQL examples

```
nextval('my_sequence_name')
```

`nfc_normalize` *DuckDB function nfc_normalize*

Description

Converts `string` to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.

Usage

```
nfc_normalize(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
nfc_normalize('ardèch')
```

`normalized_interval` *DuckDB function normalized_interval*

Description

Normalizes an INTERVAL to an equivalent interval.

Usage

```
normalized_interval(interval = INTERVAL)
```

Arguments

`interval`	`INTERVAL`

Value

`INTERVAL`

SQL examples

```
normalized_interval(INTERVAL '30 days')
```

`not-__postfix` *DuckDB function !__postfix*

Description

Factorial of x. Computes the product of the current integer and all integers below it.

Usage

```
`!__postfix`(x = INTEGER)
```

Arguments

x	INTEGER

Value

HUGEINT

SQL examples

```
4!
```

`not-~~*` *DuckDB function !~~**

Description

DuckDB function `!~~*()`.

Usage

```
`!~~*`(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

col0	VARCHAR
col1	VARCHAR

Value

BOOLEAN

`not-~~` *DuckDB function !~~*

Description

DuckDB function `!~~()`.

Usage

```
`!~~`(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`
`col1`	`VARCHAR`

Value

`BOOLEAN`

`not_ilike_escape` *DuckDB function not_ilike_escape*

Description

Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.

Usage

```
not_ilike_escape(string = VARCHAR, like_specifier = VARCHAR, escape_character = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`like_specifier`	`VARCHAR`
`escape_character`	`VARCHAR`

Value

`BOOLEAN`

SQL examples

```
not_ilike_escape('A%c', 'a$%C', '$')
```

`not_like_escape` *DuckDB function not_like_escape*

Description

Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.

Usage

```
not_like_escape(string = VARCHAR, like_specifier = VARCHAR, escape_character = VARCHAR)
```

Arguments

```
string          VARCHAR
like_specifier
                VARCHAR
escape_character
                VARCHAR
```

Value

`BOOLEAN`

SQL examples

```
not_like_escape('a%c', 'a$%c', '$')
```

`now` *DuckDB function now*

Description

Returns the current timestamp.

Usage

```
now()
```

Value

`TIMESTAMP WITH TIME ZONE`

SQL examples

```
now()
```

`nullif` *DuckDB function nullif*

Description

DuckDB macro `nullif()`.

Usage

```
nullif(a, b)
```

Arguments

`a`	Unspecified.
`b`	Unspecified.

Value

Unspecified.

`obj_description` *DuckDB function obj_description*

Description

DuckDB macro `obj_description()`.

Usage

```
obj_description(object_oid, catalog_name)
```

Arguments

`object_oid`	Unspecified.
`catalog_name`	Unspecified.

Value

Unspecified.

`octet_length` *DuckDB function octet_length*

Description

Number of bytes in `blob`.

Returns the number of bytes in the `bitstring`.

Arguments

`blob`	`BLOB`
`bitstring`	`BIT`

Value

`BIGINT`

Overloads

- `octet_length(blob = BLOB)`
- `octet_length(bitstring = BIT)`

SQL examples

```
octet_length('\xAA\xBB'::BLOB)
octet_length('1101011'::BITSTRING)
```

`or-` *DuckDB function |*

Description

Bitwise OR.

Arguments

`left`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`
`right`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Overloads

- `` `|`(left = TINYINT, right = TINYINT) ``
- `` `|`(left = SMALLINT, right = SMALLINT) ``
- `` `|`(left = INTEGER, right = INTEGER) ``
- `` `|`(left = BIGINT, right = BIGINT) ``
- `` `|`(left = HUGEINT, right = HUGEINT) ``
- `` `|`(left = UTINYINT, right = UTINYINT) ``
- `` `|`(left = USMALLINT, right = USMALLINT) ``
- `` `|`(left = UINTEGER, right = UINTEGER) ``
- `` `|`(left = UBIGINT, right = UBIGINT) ``
- `` `|`(left = UHUGEINT, right = UHUGEINT) ``
- `` `|`(left = BIT, right = BIT) ``

SQL examples

```
32 | 3
```

`or--or` *DuckDB function* `||`

Description

Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2, ...)`.

Usage

```
`||`(arg1 = ANY, arg2 = ANY)
```

Arguments

`arg1`	`ANY`
`arg2`	`ANY`

Value

`ANY`

SQL examples

```
'Duck' || 'DB'
[1, 2, 3] || [4, 5, 6]
'\xAA'::BLOB || '\xBB'::BLOB
```

`ord` *DuckDB function ord*

Description

Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.

Usage

```
ord(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`INTEGER`

SQL examples

```
[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]
```

`parquet_bloom_probe` *DuckDB function parquet_bloom_probe*

Description

DuckDB function `parquet_bloom_probe()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`
`col1`	`VARCHAR`
`col2`	`ANY`

Value

Unspecified.

Overloads

- `parquet_bloom_probe(col0 = VARCHAR, col1 = VARCHAR, col2 = ANY)`
- `parquet_bloom_probe(col0 = VARCHAR[], col1 = VARCHAR, col2 = ANY)`

`parquet_file_metadata`
DuckDB function parquet_file_metadata

Description

DuckDB function `parquet_file_metadata()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`

Value

Unspecified.

Overloads

- `parquet_file_metadata(col0 = VARCHAR)`
- `parquet_file_metadata(col0 = \`VARCHAR[]\`)`

`parquet_kv_metadata` *DuckDB function parquet_kv_metadata*

Description

DuckDB function `parquet_kv_metadata()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`

Value

Unspecified.

Overloads

- `parquet_kv_metadata(col0 = VARCHAR)`
- `parquet_kv_metadata(col0 = \`VARCHAR[]\`)`

parquet_metadata *DuckDB function parquet_metadata*

Description

DuckDB function `parquet_metadata()`.

Arguments

col0	VARCHAR | VARCHAR[]

Value

Unspecified.

Overloads

- `parquet_metadata(col0 = VARCHAR)`
- `parquet_metadata(col0 = \`VARCHAR[]\`)`

parquet_scan *DuckDB function parquet_scan*

Description

DuckDB function `parquet_scan()`.

Arguments

col0	VARCHAR | VARCHAR[]
can_have_nan	BOOLEAN
encryption_config	ANY
file_row_number	BOOLEAN
schema	ANY
parquet_version	VARCHAR
filename	ANY
binary_as_string	BOOLEAN
debug_use_openssl	BOOLEAN
union_by_name	BOOLEAN

```
explicit_cardinality
              UBIGINT
compression   VARCHAR
hive_types    ANY
hive_partitioning
              BOOLEAN
hive_types_autocast
              BOOLEAN
```

Value

Unspecified.

Overloads

- `parquet_scan(col0 = VARCHAR, can_have_nan = BOOLEAN, encryption_config = ANY, file_row_number = BOOLEAN, schema = ANY, parquet_version = VARCHAR, filename = ANY, binary_as_string = BOOLEAN, debug_use_openssl = BOOLEAN, union_by_name = BOOLEAN, explicit_cardinality = UBIGINT, compression = VARCHAR, hive_types = ANY, hive_partitioning = BOOLEAN, hive_types_autocast = BOOLEAN)`
- `parquet_scan(col0 = `VARCHAR[]`, hive_types_autocast = BOOLEAN, hive_partitioning = BOOLEAN, hive_types = ANY, compression = VARCHAR, explicit_cardinality = UBIGINT, union_by_name = BOOLEAN, debug_use_openssl = BOOLEAN, binary_as_string = BOOLEAN, filename = ANY, parquet_version = VARCHAR, schema = ANY, file_row_number = BOOLEAN, encryption_config = ANY, can_have_nan = BOOLEAN)`

---

parquet_schema *DuckDB function parquet_schema*

Description

DuckDB function `parquet_schema()`.

Arguments

```
col0          VARCHAR | VARCHAR[]
```

Value

Unspecified.

Overloads

- `parquet_schema(col0 = VARCHAR)`
- `parquet_schema(col0 = `VARCHAR[]`)`

`parse_dirname` *DuckDB function parse_dirname*

Description

Returns the top-level directory name from the given `path`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.

Arguments

`path`	`VARCHAR`
`separator`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `parse_dirname(path = VARCHAR)`
- `parse_dirname(path = VARCHAR, separator = VARCHAR)`

SQL examples

```
parse_dirname('path/to/file.csv', 'system')
```

`parse_dirpath` *DuckDB function parse_dirpath*

Description

Returns the head of the `path` (the pathname until the last slash) similarly to Python's `os.path.dirname`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.

Arguments

`path`	`VARCHAR`
`separator`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `parse_dirpath(path = VARCHAR)`
- `parse_dirpath(path = VARCHAR, separator = VARCHAR)`

SQL examples

```
parse_dirpath('path/to/file.csv', 'forward_slash')
```

`parse_duckdb_log_message` *DuckDB function parse_duckdb_log_message*

Description

Parse the message into the expected logical type.

Usage

```
parse_duckdb_log_message(type = VARCHAR, message = VARCHAR)
```

Arguments

`type`	`VARCHAR`
`message`	`VARCHAR`

Value

`ANY`

SQL examples

```
parse_duckdb_log_message('FileSystem', log_message)
```

`parse_filename` *DuckDB function parse_filename*

Description

Returns the last component of the `path` similarly to Python's `os.path.basename` function. If `trim_extension` is `true`, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.

Arguments

string	VARCHAR
trim_extension	VARCHAR | BOOLEAN
separator	VARCHAR

Value

VARCHAR

Overloads

- parse_filename(string = VARCHAR)
- parse_filename(string = VARCHAR, trim_extension = VARCHAR)
- parse_filename(string = VARCHAR, trim_extension = BOOLEAN)
- parse_filename(string = VARCHAR, trim_extension = BOOLEAN, separator = VARCHAR)

SQL examples

```
parse_filename('path/to/file.csv', true, 'forward_slash')
```

---

parse_path *DuckDB function parse_path*

Description

Returns a list of the components (directories and filename) in the path similarly to Python's pathlib.parts function. separator options: system, both_slash (default), forward_slash, backslash.

Arguments

path	VARCHAR
separator	VARCHAR

Value

VARCHAR[]

Overloads

- parse_path(path = VARCHAR)
- parse_path(path = VARCHAR, separator = VARCHAR)

SQL examples

```
parse_path('path/to/file.csv', 'system')
```

`pg_collation_is_visible` *DuckDB function pg_collation_is_visible*

Description

DuckDB macro `pg_collation_is_visible()`.

Usage

```
pg_collation_is_visible(collation_oid)
```

Arguments

`collation_oid` Unspecified.

Value

Unspecified.

`pg_conf_load_time` *DuckDB function pg_conf_load_time*

Description

DuckDB macro `pg_conf_load_time()`.

Usage

```
pg_conf_load_time()
```

Value

Unspecified.

`pg_conversion_is_visible`
DuckDB function pg_conversion_is_visible

Description

DuckDB macro `pg_conversion_is_visible()`.

Usage

```
pg_conversion_is_visible(conversion_oid)
```

Arguments

`conversion_oid`
Unspecified.

Value

Unspecified.

`pg_function_is_visible`
DuckDB function pg_function_is_visible

Description

DuckDB macro `pg_function_is_visible()`.

Usage

```
pg_function_is_visible(function_oid)
```

Arguments

`function_oid` Unspecified.

Value

Unspecified.

`pg_get_constraintdef` *DuckDB function pg_get_constraintdef*

Description

DuckDB function `pg_get_constraintdef()`.

Arguments

`constraint_oid`
: Unspecified.

`pretty_bool`
: Unspecified.

Value

Unspecified.

Overloads

- `pg_get_constraintdef(constraint_oid)`
- `pg_get_constraintdef(constraint_oid, pretty_bool)`

`pg_get_expr` *DuckDB function pg_get_expr*

Description

DuckDB macro `pg_get_expr()`.

Usage

```
pg_get_expr(pg_node_tree, relation_oid)
```

Arguments

`pg_node_tree`
: Unspecified.

`relation_oid`
: Unspecified.

Value

Unspecified.

pg_get_viewdef *DuckDB function pg_get_viewdef*

Description

DuckDB macro `pg_get_viewdef()`.

Usage

```
pg_get_viewdef(oid)
```

Arguments

`oid`	Unspecified.

Value

Unspecified.

pg_has_role *DuckDB function pg_has_role*

Description

DuckDB function `pg_has_role()`.

Arguments

`user`	Unspecified.
`role`	Unspecified.
`privilege`	Unspecified.

Value

Unspecified.

Overloads

- `pg_has_role(user, role, privilege)`
- `pg_has_role(role, privilege)`

`pg_is_other_temp_schema` *DuckDB function pg_is_other_temp_schema*

Description

DuckDB macro `pg_is_other_temp_schema()`.

Usage

```
pg_is_other_temp_schema(schema_id)
```

Arguments

`schema_id`	Unspecified.

Value

Unspecified.

`pg_my_temp_schema` *DuckDB function pg_my_temp_schema*

Description

DuckDB macro `pg_my_temp_schema()`.

Usage

```
pg_my_temp_schema()
```

Value

Unspecified.

`pg_opclass_is_visible`

DuckDB function pg_opclass_is_visible

Description

DuckDB macro `pg_opclass_is_visible()`.

Usage

```
pg_opclass_is_visible(opclass_oid)
```

Arguments

`opclass_oid`	Unspecified.

Value

Unspecified.

`pg_operator_is_visible`

DuckDB function pg_operator_is_visible

Description

DuckDB macro `pg_operator_is_visible()`.

Usage

```
pg_operator_is_visible(operator_oid)
```

Arguments

`operator_oid`	Unspecified.

Value

Unspecified.

`pg_opfamily_is_visible`
DuckDB function pg_opfamily_is_visible

Description

DuckDB macro `pg_opfamily_is_visible()`.

Usage

```
pg_opfamily_is_visible(opclass_oid)
```

Arguments

`opclass_oid`	Unspecified.

Value

Unspecified.

`pg_postmaster_start_time`
DuckDB function pg_postmaster_start_time

Description

DuckDB macro `pg_postmaster_start_time()`.

Usage

```
pg_postmaster_start_time()
```

Value

Unspecified.

`pg_size_pretty` *DuckDB function pg_size_pretty*

Description

DuckDB macro `pg_size_pretty()`.

Usage

```
pg_size_pretty(bytes)
```

Arguments

`bytes`	Unspecified.

Value

Unspecified.

`pg_table_is_visible` *DuckDB function pg_table_is_visible*

Description

DuckDB macro `pg_table_is_visible()`.

Usage

```
pg_table_is_visible(table_oid)
```

Arguments

`table_oid`	Unspecified.

Value

Unspecified.

`pg_ts_config_is_visible`
DuckDB function pg_ts_config_is_visible

Description

DuckDB macro `pg_ts_config_is_visible()`.

Usage

```
pg_ts_config_is_visible(config_oid)
```

Arguments

`config_oid`	Unspecified.

Value

Unspecified.

`pg_ts_dict_is_visible`
DuckDB function pg_ts_dict_is_visible

Description

DuckDB macro `pg_ts_dict_is_visible()`.

Usage

```
pg_ts_dict_is_visible(dict_oid)
```

Arguments

`dict_oid`	Unspecified.

Value

Unspecified.

`pg_ts_parser_is_visible`
DuckDB function pg_ts_parser_is_visible

Description

DuckDB macro `pg_ts_parser_is_visible()`.

Usage

```
pg_ts_parser_is_visible(parser_oid)
```

Arguments

`parser_oid`	Unspecified.

Value

Unspecified.

`pg_ts_template_is_visible`
DuckDB function pg_ts_template_is_visible

Description

DuckDB macro `pg_ts_template_is_visible()`.

Usage

```
pg_ts_template_is_visible(template_oid)
```

Arguments

`template_oid`	Unspecified.

Value

Unspecified.

pg_type_is_visible *DuckDB function pg_type_is_visible*

Description

DuckDB macro `pg_type_is_visible()`.

Usage

```
pg_type_is_visible(type_oid)
```

Arguments

`type_oid`	Unspecified.

Value

Unspecified.

pg_typeof *DuckDB function pg_typeof*

Description

DuckDB macro `pg_typeof()`.

Usage

```
pg_typeof(expression)
```

Arguments

`expression`	Unspecified.

Value

Unspecified.

`pi` *DuckDB function pi*

Description

Returns the value of pi.

Usage

```
pi()
```

Value

`DOUBLE`

SQL examples

```
pi()
```

`platform` *DuckDB function platform*

Description

DuckDB function `platform()`.

Usage

```
platform()
```

Value

Unspecified.

`position` *DuckDB function position*

Description

Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.

Usage

```
position(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
position('b' IN 'abc')
```

`power` *DuckDB function power*

Description

Computes x to the power of y.

Usage

```
power(x = DOUBLE, y = DOUBLE)
```

Arguments

`x`	`DOUBLE`
`y`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
power(2, 3)
```

`pragma_collations` *DuckDB function pragma_collations*

Description

DuckDB function `pragma_collations()`.

Usage

```
pragma_collations()
```

Value

Unspecified.

`pragma_database_size` *DuckDB function pragma_database_size*

Description

DuckDB function `pragma_database_size()`.

Usage

```
pragma_database_size()
```

Value

Unspecified.

`pragma_metadata_info` *DuckDB function pragma_metadata_info*

Description

DuckDB function `pragma_metadata_info()`.

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

Overloads

- `pragma_metadata_info()`
- `pragma_metadata_info(col0 = VARCHAR)`

`pragma_platform` *DuckDB function pragma_platform*

Description

DuckDB function `pragma_platform()`.

Usage

```
pragma_platform()
```

Value

Unspecified.

`pragma_show` *DuckDB function pragma_show*

Description

DuckDB function `pragma_show()`.

Usage

```
pragma_show(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

pragma_storage_info *DuckDB function pragma_storage_info*

Description

DuckDB function `pragma_storage_info()`.

Usage

```
pragma_storage_info(col0 = VARCHAR)
```

Arguments

col0	VARCHAR

Value

Unspecified.

pragma_table_info *DuckDB function pragma_table_info*

Description

DuckDB function `pragma_table_info()`.

Usage

```
pragma_table_info(col0 = VARCHAR)
```

Arguments

col0	VARCHAR

Value

Unspecified.

pragma_user_agent *DuckDB function pragma_user_agent*

Description

DuckDB function `pragma_user_agent()`.

Usage

```
pragma_user_agent()
```

Value

Unspecified.

pragma_version *DuckDB function pragma_version*

Description

DuckDB function `pragma_version()`.

Usage

```
pragma_version()
```

Value

Unspecified.

prefix *DuckDB function prefix*

Description

Returns `true` if `string` starts with `search_string`.

Usage

```
prefix(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

BOOLEAN

SQL examples

```
prefix('abc', 'ab')
```

---

printf *DuckDB function printf*

Description

Formats a `string` using printf syntax.

Usage

```
printf(format = VARCHAR)
```

Arguments

format	VARCHAR

Value

VARCHAR

SQL examples

```
printf('Benchmark "%s" took %d seconds', 'CSV', 42)
```

---

product *DuckDB function product*

Description

Calculates the product of all tuples in arg.

Usage

```
product(arg = DOUBLE)
```

Arguments

arg	DOUBLE

Value

DOUBLE

SQL examples

```
product(A)
```

`quantile` *DuckDB function quantile*

Description

Returns the exact quantile number between 0 and 1 . If pos is a LIST of FLOATs, then the result is a LIST of the corresponding exact quantiles.

Arguments

x	ANY
pos	DOUBLE | DOUBLE[]

Value

ANY

Overloads

- `quantile(x = ANY, pos = DOUBLE)`
- `quantile(x = ANY, pos = `DOUBLE[]`)`
- `quantile(x = ANY)`

SQL examples

```
quantile_disc(x, 0.5)
```

quantile_cont *DuckDB function quantile_cont*

Description

Returns the interpolated quantile number between 0 and 1 . If pos is a LIST of FLOATs, then the result is a LIST of the corresponding interpolated quantiles. .

Arguments

x	DECIMAL | TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DAT
pos	DOUBLE | DOUBLE[]

Value

DECIMAL | TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DATE | TIMESTAMP

Overloads

- quantile_cont(x = DECIMAL, pos = DOUBLE)
- quantile_cont(x = DECIMAL, pos = `DOUBLE[]`)
- quantile_cont(x = TINYINT, pos = DOUBLE)
- quantile_cont(x = TINYINT, pos = `DOUBLE[]`)
- quantile_cont(x = SMALLINT, pos = DOUBLE)
- quantile_cont(x = SMALLINT, pos = `DOUBLE[]`)
- quantile_cont(x = INTEGER, pos = DOUBLE)
- quantile_cont(x = INTEGER, pos = `DOUBLE[]`)
- quantile_cont(x = BIGINT, pos = DOUBLE)
- quantile_cont(x = BIGINT, pos = `DOUBLE[]`)
- quantile_cont(x = HUGEINT, pos = DOUBLE)
- quantile_cont(x = HUGEINT, pos = `DOUBLE[]`)
- quantile_cont(x = FLOAT, pos = DOUBLE)
- quantile_cont(x = FLOAT, pos = `DOUBLE[]`)
- quantile_cont(x = DOUBLE, pos = DOUBLE)
- quantile_cont(x = DOUBLE, pos = `DOUBLE[]`)
- quantile_cont(x = DATE, pos = DOUBLE)
- quantile_cont(x = DATE, pos = `DOUBLE[]`)
- quantile_cont(x = TIMESTAMP, pos = DOUBLE)
- quantile_cont(x = TIMESTAMP, pos = `DOUBLE[]`)
- quantile_cont(x = TIME, pos = DOUBLE)
- quantile_cont(x = TIME, pos = `DOUBLE[]`)

- `quantile_cont(x = `TIMESTAMP WITH TIME ZONE`, pos = DOUBLE)`
- `quantile_cont(x = `TIMESTAMP WITH TIME ZONE`, pos = `DOUBLE[]`)`
- `quantile_cont(x = `TIME WITH TIME ZONE`, pos = DOUBLE)`
- `quantile_cont(x = `TIME WITH TIME ZONE`, pos = `DOUBLE[]`)`

SQL examples

```
quantile_cont(x, 0.5)
```

`quantile_disc` *DuckDB function quantile_disc*

Description

Returns the exact quantile number between 0 and 1 . If pos is a LIST of FLOATs, then the result is a LIST of the corresponding exact quantiles.

Arguments

x	ANY
pos	DOUBLE | DOUBLE[]

Value

ANY

Overloads

- `quantile_disc(x = ANY, pos = DOUBLE)`
- `quantile_disc(x = ANY, pos = `DOUBLE[]`)`
- `quantile_disc(x = ANY)`

SQL examples

```
quantile_disc(x, 0.5)
```

---

quarter *DuckDB function quarter*

Description

Extract the quarter component from a date or timestamp.

Arguments

ts	DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE

Value

BIGINT

Overloads

- quarter(ts = DATE)
- quarter(ts = INTERVAL)
- quarter(ts = TIMESTAMP)
- quarter(ts = `TIMESTAMP WITH TIME ZONE`)

SQL examples

```
quarter(timestamp '2021-08-03 11:59:44.123456')
```

---

query *DuckDB function query*

Description

DuckDB function query().

Usage

```
query(col0 = VARCHAR)
```

Arguments

col0	VARCHAR

Value

Unspecified.

`query_table` *DuckDB function query_table*

Description

DuckDB function `query_table()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`
`col1`	`BOOLEAN`

Value

Unspecified.

Overloads

- `query_table(col0 = VARCHAR)`
- `query_table(col0 = \`VARCHAR[]\`)`
- `query_table(col0 = \`VARCHAR[]\`, col1 = BOOLEAN)`

`r_dataframe_scan` *DuckDB function r_dataframe_scan*

Description

DuckDB function `r_dataframe_scan()`.

Usage

```
r_dataframe_scan(col0 = POINTER, experimental = BOOLEAN, integer64 = BOOLEAN)
```

Arguments

`col0`	`POINTER`
`experimental`	`BOOLEAN`
`integer64`	`BOOLEAN`

Value

Unspecified.

`radians` *DuckDB function radians*

Description

Converts degrees to radians.

Usage

```
radians(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

`DOUBLE`

SQL examples

```
radians(90)
```

`random` *DuckDB function random*

Description

Returns a random number between 0 and 1.

Usage

```
random()
```

Value

`DOUBLE`

SQL examples

```
random()
```

`range` *DuckDB function range*

Description

Creates a list of values between `start` and `stop` - the stop parameter is exclusive.

Arguments

`col0`	`BIGINT | TIMESTAMP`
`col1`	`BIGINT | TIMESTAMP`
`col2`	`BIGINT | INTERVAL`
`start`	`BIGINT | TIMESTAMP | TIMESTAMP WITH TIME ZONE`
`stop`	`BIGINT | TIMESTAMP | TIMESTAMP WITH TIME ZONE`
`step`	`BIGINT | INTERVAL`

Value

`BIGINT[] | TIMESTAMP[] | TIMESTAMP WITH TIME ZONE[]`

Overloads

- `range(col0 = BIGINT)`
- `range(col0 = BIGINT, col1 = BIGINT)`
- `range(col0 = BIGINT, col1 = BIGINT, col2 = BIGINT)`
- `range(col0 = TIMESTAMP, col1 = TIMESTAMP, col2 = INTERVAL)`
- `range(start = BIGINT)`
- `range(start = BIGINT, stop = BIGINT)`
- `range(start = BIGINT, stop = BIGINT, step = BIGINT)`
- `range(start = TIMESTAMP, stop = TIMESTAMP, step = INTERVAL)`
- `range(start = ``TIMESTAMP WITH TIME ZONE``, stop = ``TIMESTAMP WITH TIME ZONE``, step = INTERVAL)`

SQL examples

```
range(2, 5, 3)
```

`read_blob` *DuckDB function read_blob*

Description

DuckDB function `read_blob()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`
`union_by_name`	`BOOLEAN`
`hive_partitioning`	`BOOLEAN`
`hive_types_autocast`	`BOOLEAN`
`hive_types`	`ANY`
`filename`	`ANY`

Value

Unspecified.

Overloads

- `read_blob(col0 = VARCHAR, union_by_name = BOOLEAN, hive_partitioning = BOOLEAN, hive_types_autocast = BOOLEAN, hive_types = ANY, filename = ANY)`
- `read_blob(col0 = \`VARCHAR[]\`, filename = ANY, hive_types = ANY, hive_types_autocast = BOOLEAN, hive_partitioning = BOOLEAN, union_by_name = BOOLEAN)`

`read_csv` *DuckDB function read_csv*

Description

DuckDB function `read_csv()`.

Arguments

`col0`	`VARCHAR | VARCHAR[]`
`hive_types_autocast`	`BOOLEAN`
`skip`	`BIGINT`
`types`	`ANY`
`nullstr`	`ANY`

```
encoding       VARCHAR
hive_types     ANY
filename       ANY
header         BOOLEAN
delim          VARCHAR
dateformat     VARCHAR
column_names   VARCHAR[]
union_by_name  BOOLEAN
new_line       VARCHAR
escape         VARCHAR
allow_quoted_nulls
               BOOLEAN
comment        VARCHAR
hive_partitioning
               BOOLEAN
sep            VARCHAR
columns        ANY
rejects_limit  BIGINT
force_not_null
               VARCHAR[]
auto_type_candidates
               ANY
sample_size    BIGINT
timestampformat
               VARCHAR
auto_detect    BOOLEAN
all_varchar    BOOLEAN
store_rejects  BOOLEAN
normalize_names
               BOOLEAN
rejects_table  VARCHAR
column_types   ANY
compression    VARCHAR
ignore_errors  BOOLEAN
names          VARCHAR[]
max_line_size  VARCHAR
quote          VARCHAR
maximum_line_size
               VARCHAR
rejects_scan   VARCHAR
```

```
buffer_size    UBIGINT
decimal_separator
               VARCHAR
parallel       BOOLEAN
null_padding   BOOLEAN
dtypes         ANY
strict_mode    BOOLEAN
thousands      VARCHAR
files_to_sniff
               BIGINT
```

Value

Unspecified.

Overloads

- `read_csv(col0 = VARCHAR, hive_types_autocast = BOOLEAN, skip = BIGINT, types = ANY, nullstr = ANY, encoding = VARCHAR, hive_types = ANY, filename = ANY, header = BOOLEAN, delim = VARCHAR, dateformat = VARCHAR, column_names = `VARCHAR[]`, union_by_name = BOOLEAN, new_line = VARCHAR, escape = VARCHAR, allow_quoted_nulls = BOOLEAN, comment = VARCHAR, hive_partitioning = BOOLEAN, sep = VARCHAR, columns = ANY, rejects_limit = BIGINT, force_not_null = `VARCHAR[]`, auto_type_candidates = ANY, sample_size = BIGINT, timestampformat = VARCHAR, auto_detect = BOOLEAN, all_varchar = BOOLEAN, store_rejects = BOOLEAN, normalize_names = BOOLEAN, rejects_table = VARCHAR, column_types = ANY, compression = VARCHAR, ignore_errors = BOOLEAN, names = `VARCHAR[]`, max_line_size = VARCHAR, quote = VARCHAR, maximum_line_size = VARCHAR, rejects_scan = VARCHAR, buffer_size = UBIGINT, decimal_separator = VARCHAR, parallel = BOOLEAN, null_padding = BOOLEAN, dtypes = ANY, strict_mode = BOOLEAN, thousands = VARCHAR, files_to_sniff = BIGINT)`
- `read_csv(col0 = `VARCHAR[]`, files_to_sniff = BIGINT, thousands = VARCHAR, strict_mode = BOOLEAN, dtypes = ANY, null_padding = BOOLEAN, parallel = BOOLEAN, decimal_separator = VARCHAR, buffer_size = UBIGINT, rejects_scan = VARCHAR, maximum_line_size = VARCHAR, quote = VARCHAR, max_line_size = VARCHAR, names = `VARCHAR[]`, ignore_errors = BOOLEAN, compression = VARCHAR, column_types = ANY, rejects_table = VARCHAR, normalize_names = BOOLEAN, store_rejects = BOOLEAN, all_varchar = BOOLEAN, auto_detect = BOOLEAN, timestampformat = VARCHAR, sample_size = BIGINT, auto_type_candidates = ANY, force_not_null = `VARCHAR[]`, rejects_limit = BIGINT, columns = ANY, sep = VARCHAR, hive_partitioning = BOOLEAN, comment = VARCHAR, allow_quoted_nulls = BOOLEAN, escape = VARCHAR, new_line = VARCHAR, union_by_name = BOOLEAN, column_names = `VARCHAR[]`, dateformat = VARCHAR, delim = VARCHAR, header = BOOLEAN, filename = ANY, hive_types = ANY, encoding = VARCHAR, nullstr = ANY, types = ANY, skip = BIGINT, hive_types_autocast = BOOLEAN)`

`read_csv_auto` *DuckDB function read_csv_auto*

Description

DuckDB function `read_csv_auto()`.

Arguments

col0	VARCHAR | VARCHAR[]
hive_types_autocast	BOOLEAN
skip	BIGINT
types	ANY
nullstr	ANY
encoding	VARCHAR
hive_types	ANY
filename	ANY
header	BOOLEAN
delim	VARCHAR
dateformat	VARCHAR
column_names	VARCHAR[]
union_by_name	BOOLEAN
new_line	VARCHAR
escape	VARCHAR
allow_quoted_nulls	BOOLEAN
comment	VARCHAR
hive_partitioning	BOOLEAN
sep	VARCHAR
columns	ANY
rejects_limit	BIGINT
force_not_null	VARCHAR[]
auto_type_candidates	ANY
sample_size	BIGINT
timestampformat	VARCHAR

```
auto_detect    BOOLEAN
all_varchar    BOOLEAN
store_rejects  BOOLEAN
normalize_names
               BOOLEAN
rejects_table  VARCHAR
column_types   ANY
compression    VARCHAR
ignore_errors  BOOLEAN
names          VARCHAR[]
max_line_size  VARCHAR
quote          VARCHAR
maximum_line_size
               VARCHAR
rejects_scan   VARCHAR
buffer_size    UBIGINT
decimal_separator
               VARCHAR
parallel       BOOLEAN
null_padding   BOOLEAN
dtypes         ANY
strict_mode    BOOLEAN
thousands      VARCHAR
files_to_sniff
               BIGINT
```

Value

Unspecified.

Overloads

- `read_csv_auto(col0 = VARCHAR, hive_types_autocast = BOOLEAN, skip = BIGINT, types = ANY, nullstr = ANY, encoding = VARCHAR, hive_types = ANY, filename = ANY, header = BOOLEAN, delim = VARCHAR, dateformat = VARCHAR, column_names = `VARCHAR[]`, union_by_name = BOOLEAN, new_line = VARCHAR, escape = VARCHAR, allow_quoted_nulls = BOOLEAN, comment = VARCHAR, hive_partitioning = BOOLEAN, sep = VARCHAR, columns = ANY, rejects_limit = BIGINT, force_not_null = `VARCHAR[]`, auto_type_candidates = ANY, sample_size = BIGINT, timestampformat = VARCHAR, auto_detect = BOOLEAN, all_varchar = BOOLEAN, store_rejects = BOOLEAN, normalize_names = BOOLEAN, rejects_table = VARCHAR, column_types = ANY, compression = VARCHAR, ignore_errors = BOOLEAN, names = `VARCHAR[]`, max_line_size = VARCHAR, quote = VARCHAR, maximum_line_size = VARCHAR, rejects_scan = VARCHAR, buffer_size = UBIGINT, decimal_separator = VARCHAR, parallel = BOOLEAN, null_padding = BOOLEAN, dtypes = ANY, strict_mode = BOOLEAN, thousands = VARCHAR, files_to_sniff = BIGINT)`

- `read_csv_auto(col0 = `VARCHAR[]`, files_to_sniff = BIGINT, thousands = VARCHAR, strict_mode = BOOLEAN, dtypes = ANY, null_padding = BOOLEAN, parallel = BOOLEAN, decimal_separator = VARCHAR, buffer_size = UBIGINT, rejects_scan = VARCHAR, maximum_line_size = VARCHAR, quote = VARCHAR, max_line_size = VARCHAR, names = `VARCHAR[]`, ignore_errors = BOOLEAN, compression = VARCHAR, column_types = ANY, rejects_table = VARCHAR, normalize_names = BOOLEAN, store_rejects = BOOLEAN, all_varchar = BOOLEAN, auto_detect = BOOLEAN, timestampformat = VARCHAR, sample_size = BIGINT, auto_type_candidates = ANY, force_not_null = `VARCHAR[]`, rejects_limit = BIGINT, columns = ANY, sep = VARCHAR, hive_partitioning = BOOLEAN, comment = VARCHAR, allow_quoted_nulls = BOOLEAN, escape = VARCHAR, new_line = VARCHAR, union_by_name = BOOLEAN, column_names = `VARCHAR[]`, dateformat = VARCHAR, delim = VARCHAR, header = BOOLEAN, filename = ANY, hive_types = ANY, encoding = VARCHAR, nullstr = ANY, types = ANY, skip = BIGINT, hive_types_autocast = BOOLEAN)`

`read_parquet` *DuckDB function read_parquet*

Description

DuckDB function `read_parquet()`.

Arguments

col0	VARCHAR | VARCHAR[]
can_have_nan	BOOLEAN
encryption_config	ANY
file_row_number	BOOLEAN
schema	ANY
parquet_version	VARCHAR
filename	ANY
binary_as_string	BOOLEAN
debug_use_openssl	BOOLEAN
union_by_name	BOOLEAN
explicit_cardinality	UBIGINT
compression	VARCHAR
hive_types	ANY
hive_partitioning	BOOLEAN
hive_types_autocast	BOOLEAN

Value

Unspecified.

Overloads

- `read_parquet(col0 = VARCHAR, can_have_nan = BOOLEAN, encryption_config = ANY, file_row_number = BOOLEAN, schema = ANY, parquet_version = VARCHAR, filename = ANY, binary_as_string = BOOLEAN, debug_use_openssl = BOOLEAN, union_by_name = BOOLEAN, explicit_cardinality = UBIGINT, compression = VARCHAR, hive_types = ANY, hive_partitioning = BOOLEAN, hive_types_autocast = BOOLEAN)`
- `read_parquet(col0 = \`VARCHAR[]\`, hive_types_autocast = BOOLEAN, hive_partitioning = BOOLEAN, hive_types = ANY, compression = VARCHAR, explicit_cardinality = UBIGINT, union_by_name = BOOLEAN, debug_use_openssl = BOOLEAN, binary_as_string = BOOLEAN, filename = ANY, parquet_version = VARCHAR, schema = ANY, file_row_number = BOOLEAN, encryption_config = ANY, can_have_nan = BOOLEAN)`

---

read_text *DuckDB function read_text*

Description

DuckDB function `read_text()`.

Arguments

col0	VARCHAR | VARCHAR[]
union_by_name	BOOLEAN
hive_partitioning	BOOLEAN
hive_types_autocast	BOOLEAN
hive_types	ANY
filename	ANY

Value

Unspecified.

Overloads

- `read_text(col0 = VARCHAR, union_by_name = BOOLEAN, hive_partitioning = BOOLEAN, hive_types_autocast = BOOLEAN, hive_types = ANY, filename = ANY)`
- `read_text(col0 = \`VARCHAR[]\`, filename = ANY, hive_types = ANY, hive_types_autocast = BOOLEAN, hive_partitioning = BOOLEAN, union_by_name = BOOLEAN)`

`reduce` *DuckDB function reduce*

Description

Reduces all elements of the input `list` into a single scalar value by executing the `lambda` function on a running result and the next list element. The `lambda` function has an optional `initial_value` argument.

Arguments

`list`	`ANY[]`
`initial_value`	`ANY`
`lambda(x, y)`	`LAMBDA`

Value

`ANY`

Overloads

- `reduce(list = \`ANY[]\`, \`lambda(x,y)\` = LAMBDA)`
- `reduce(list = \`ANY[]\`, \`lambda(x,y)\` = LAMBDA, initial_value = ANY)`

SQL examples

```
reduce([1, 2, 3], lambda x, y : x + y)
```

`regexp_escape` *DuckDB function regexp_escape*

Description

Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.

Usage

```
regexp_escape(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
regexp_escape('https://duckdb.org')
```

`regexp_extract` *DuckDB function regexp_extract*

Description

If `string` contains the `regex` pattern, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional regex `options` can be set.

If `string` contains the `regex` pattern, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`group`	`INTEGER`
`options`	`VARCHAR`
`name_list`	`VARCHAR[]`

Value

`VARCHAR`

Overloads

- `regexp_extract(string = VARCHAR, regex = VARCHAR)`
- `regexp_extract(string = VARCHAR, regex = VARCHAR, group = INTEGER)`
- `regexp_extract(string = VARCHAR, regex = VARCHAR, group = INTEGER, options = VARCHAR)`
- `regexp_extract(string = VARCHAR, regex = VARCHAR, name_list = ` ``VARCHAR[]`` `)`
- `regexp_extract(string = VARCHAR, regex = VARCHAR, name_list = ` ``VARCHAR[]`` `, options = VARCHAR)`

SQL examples

```
regexp_extract('abcde', '[a-z]{3}')
regexp_extract('abc', '([a-z])(b)', 1)
regexp_extract('ABC', '([a-z])(b)', 1, 'i')
regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])
regexp_extract('John Doe', '([a-z]+) ([a-z]+)', ['first_name', 'last_name'], 'i')
```

`regexp_extract_all` *DuckDB function regexp_extract_all*

Description

Finds non-overlapping occurrences of the `regex` in the `string` and returns the corresponding values of the capturing `group`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`group`	`INTEGER`
`options`	`VARCHAR`

Value

`VARCHAR[]`

Overloads

- `regexp_extract_all(string = VARCHAR, regex = VARCHAR)`
- `regexp_extract_all(string = VARCHAR, regex = VARCHAR, group = INTEGER)`
- `regexp_extract_all(string = VARCHAR, regex = VARCHAR, group = INTEGER, options = VARCHAR)`

SQL examples

```
regexp_extract_all('Peter: 33, Paul:14', '(\w+):\s*(\d+)', 2)
```

`regexp_full_match` *DuckDB function regexp_full_match*

Description

Returns `true` if the entire `string` matches the `regex`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`col2`	`VARCHAR`

Value

BOOLEAN

Overloads

- regexp_full_match(string = VARCHAR, regex = VARCHAR)
- regexp_full_match(string = VARCHAR, regex = VARCHAR, col2 = VARCHAR)

SQL examples

```
regexp_full_match('anabanana', '(an)*')
```

---

regexp_matches *DuckDB function regexp_matches*

Description

Returns `true` if `string` contains the `regex`, `false` otherwise. A set of optional regex `options` can be set.

Arguments

string	VARCHAR
regex	VARCHAR
options	VARCHAR

Value

BOOLEAN

Overloads

- regexp_matches(string = VARCHAR, regex = VARCHAR)
- regexp_matches(string = VARCHAR, regex = VARCHAR, options = VARCHAR)

SQL examples

```
regexp_matches('anabanana', '(an)*')
```

`regexp_replace` *DuckDB function regexp_replace*

Description

If `string` contains the `regex`, replaces the matching part with `replacement`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`replacement`	`VARCHAR`
`options`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `regexp_replace(string = VARCHAR, regex = VARCHAR, replacement = VARCHAR)`
- `regexp_replace(string = VARCHAR, regex = VARCHAR, replacement = VARCHAR, options = VARCHAR)`

SQL examples

```
regexp_replace('hello', '[lo]', '-')
```

`regexp_split_to_array` *DuckDB function regexp_split_to_array*

Description

Splits the `string` along the `regex`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`options`	`VARCHAR`

Value

VARCHAR[]

Overloads

- regexp_split_to_array(string = VARCHAR, regex = VARCHAR)
- regexp_split_to_array(string = VARCHAR, regex = VARCHAR, options = VARCHAR)

SQL examples

```
regexp_split_to_array('hello world; 42', ';? ')
```

---

regexp_split_to_table

DuckDB function regexp_split_to_table

Description

DuckDB macro regexp_split_to_table().

Usage

```
regexp_split_to_table(text, pattern)
```

Arguments

text	Unspecified.
pattern	Unspecified.

Value

Unspecified.

---

regr_avgx

DuckDB function regr_avgx

Description

Returns the average of the independent variable for non-NULL pairs in a group, where x is the independent variable and y is the dependent variable.

Usage

```
regr_avgx(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

`regr_avgy` *DuckDB function regr_avgy*

Description

Returns the average of the dependent variable for non-NULL pairs in a group, where x is the independent variable and y is the dependent variable.

Usage

```
regr_avgy(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

`regr_count` *DuckDB function regr_count*

Description

Returns the number of non-NULL number pairs in a group.

Usage

```
regr_count(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

UINTEGER

SQL examples

```
(SUM(x*y) - SUM(x) * SUM(y) / COUNT(*)) / COUNT(*)
```

---

regr_intercept *DuckDB function regr_intercept*

Description

Returns the intercept of the univariate linear regression line for non-NULL pairs in a group.

Usage

```
regr_intercept(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
AVG(y)-REGR_SLOPE(y, x)*AVG(x)
```

---

regr_r2 *DuckDB function regr_r2*

Description

Returns the coefficient of determination for non-NULL pairs in a group.

Usage

```
regr_r2(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

`regr_slope` *DuckDB function regr_slope*

Description

Returns the slope of the linear regression line for non-NULL pairs in a group.

Usage

```
regr_slope(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
COVAR_POP(x, y) / VAR_POP(x)
```

`regr_sxx` *DuckDB function regr_sxx*

Description

DuckDB function `regr_sxx()`.

Usage

```
regr_sxx(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
REGR_COUNT(y, x) * VAR_POP(x)
```

`regr_sxy` *DuckDB function regr_sxy*

Description

Returns the population covariance of input values.

Usage

```
regr_sxy(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
REGR_COUNT(y, x) * COVAR_POP(y, x)
```

`regr_syy` *DuckDB function regr_syy*

Description

DuckDB function `regr_syy()`.

Usage

```
regr_syy(y = DOUBLE, x = DOUBLE)
```

Arguments

y	DOUBLE
x	DOUBLE

Value

DOUBLE

SQL examples

```
REGR_COUNT(y, x) * VAR_POP(y)
```

`repeat` *DuckDB function repeat*

Description

Repeats the `string` `count` number of times.

Repeats the `blob` `count` number of times.

Arguments

`col0`	`ANY | T[]`
`col1`	`BIGINT`
`string`	`VARCHAR`
`count`	`BIGINT`
`blob`	`BLOB`

Value

`VARCHAR | BLOB | T[]`

Overloads

- `` `repeat`(col0 = ANY, col1 = BIGINT) ``
- `` `repeat`(string = VARCHAR, count = BIGINT) ``
- `` `repeat`(blob = BLOB, count = BIGINT) ``
- `` `repeat`(col0 = `T[]`, col1 = BIGINT) ``

SQL examples

```
repeat('A', 5)
repeat('\xAA\xBB'::BLOB, 5)
```

`repeat_row` *DuckDB function repeat_row*

Description

DuckDB function `repeat_row()`.

Usage

```
repeat_row(num_rows = BIGINT)
```

Arguments

`num_rows`	`BIGINT`

Value

Unspecified.

`replace` *DuckDB function replace*

Description

Replaces any occurrences of the `source` with `target` in `string`.

Usage

```
replace(string = VARCHAR, source = VARCHAR, target = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`source`	`VARCHAR`
`target`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
replace('hello', 'l', '-')
```

`replace_type` *DuckDB function replace_type*

Description

Casts all fields of type1 to type2.

Usage

```
replace_type(param = ANY, type1 = ANY, type2 = ANY)
```

Arguments

param	ANY
type1	ANY
type2	ANY

Value

ANY

SQL examples

```
replace_type({duck: 3.141592653589793::DOUBLE}, NULL::DOUBLE, NULL::DECIMAL(15,2))
```

`reservoir_quantile` *DuckDB function reservoir_quantile*

Description

Gives the approximate quantile using reservoir sampling, the sample size is optional and uses 8192 as a default size.

Arguments

x	DECIMAL | TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE
quantile	DOUBLE | DOUBLE[]
sample_size	INTEGER

Value

DECIMAL | DECIMAL[] | TINYINT | TINYINT[] | SMALLINT | SMALLINT[] | INTEGER | INTEGER[] | BIGINT | BIGINT[] | HUGEINT | HUGEINT[] | FLOAT | FLOAT[] | DOUBLE | DOUBLE[]

Overloads

- `reservoir_quantile(x = DECIMAL, quantile = DOUBLE)`
- `reservoir_quantile(x = DECIMAL, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = DECIMAL, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = DECIMAL, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = TINYINT, quantile = DOUBLE)`
- `reservoir_quantile(x = TINYINT, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = TINYINT, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = TINYINT, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = SMALLINT, quantile = DOUBLE)`
- `reservoir_quantile(x = SMALLINT, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = SMALLINT, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = SMALLINT, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = INTEGER, quantile = DOUBLE)`
- `reservoir_quantile(x = INTEGER, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = INTEGER, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = INTEGER, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = BIGINT, quantile = DOUBLE)`
- `reservoir_quantile(x = BIGINT, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = BIGINT, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = BIGINT, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = HUGEINT, quantile = DOUBLE)`
- `reservoir_quantile(x = HUGEINT, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = HUGEINT, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = HUGEINT, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = FLOAT, quantile = DOUBLE)`
- `reservoir_quantile(x = FLOAT, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = FLOAT, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = FLOAT, quantile = `DOUBLE[]`, sample_size = INTEGER)`
- `reservoir_quantile(x = DOUBLE, quantile = DOUBLE)`
- `reservoir_quantile(x = DOUBLE, quantile = DOUBLE, sample_size = INTEGER)`
- `reservoir_quantile(x = DOUBLE, quantile = `DOUBLE[]`)`
- `reservoir_quantile(x = DOUBLE, quantile = `DOUBLE[]`, sample_size = INTEGER)`

SQL examples

```
reservoir_quantile(A, 0.5, 1024)
```

`reverse` *DuckDB function reverse*

Description

Reverses the `string`.

Usage

```
reverse(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
reverse('hello')
```

`right` *DuckDB function right*

Description

Extract the right-most `count` characters.

Usage

```
right(string = VARCHAR, count = BIGINT)
```

Arguments

`string`	`VARCHAR`
`count`	`BIGINT`

Value

`VARCHAR`

SQL examples

```
right('Hello ', 3)
```

`right_grapheme` *DuckDB function right_grapheme*

Description

Extracts the right-most `count` grapheme clusters.

Usage

```
right_grapheme(string = VARCHAR, count = BIGINT)
```

Arguments

`string`	`VARCHAR`
`count`	`BIGINT`

Value

`VARCHAR`

SQL examples

```
right_grapheme('    ', 1)
```

`round` *DuckDB function round*

Description

Rounds x to s decimal places.

Arguments

`x`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL`
`precision`	`INTEGER`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL`

Overloads

- round(x = TINYINT)
- round(x = TINYINT, precision = INTEGER)
- round(x = SMALLINT)
- round(x = SMALLINT, precision = INTEGER)
- round(x = INTEGER)
- round(x = INTEGER, precision = INTEGER)
- round(x = BIGINT)
- round(x = BIGINT, precision = INTEGER)
- round(x = HUGEINT)
- round(x = HUGEINT, precision = INTEGER)
- round(x = FLOAT)
- round(x = FLOAT, precision = INTEGER)
- round(x = DOUBLE)
- round(x = DOUBLE, precision = INTEGER)
- round(x = DECIMAL)
- round(x = DECIMAL, precision = INTEGER)

SQL examples

```
round(42.4332, 2)
```

---

round_even *DuckDB function round_even*

Description

DuckDB macro round_even().

Usage

```
round_even(x, n)
```

Arguments

x	Unspecified.
n	Unspecified.

Value

Unspecified.

`roundbankers` *DuckDB function roundbankers*

Description

DuckDB macro `roundbankers()`.

Usage

```
roundbankers(x, n)
```

Arguments

`x`	Unspecified.
`n`	Unspecified.

Value

Unspecified.

`row` *DuckDB function row*

Description

Create an unnamed STRUCT (tuple) containing the argument values.

Usage

```
row()
```

Value

`STRUCT`

SQL examples

```
row(i, i % 4, i / 4)
```

`rpad` *DuckDB function rpad*

Description

Pads the `string` with the `character` on the right until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.

Usage

```
rpad(string = VARCHAR, count = INTEGER, character = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`count`	`INTEGER`
`character`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
rpad('hello', 10, '<')
```

`rtrim` *DuckDB function rtrim*

Description

Removes any occurrences of any of the `characters` from the right side of the `string`. `characters` defaults to `space`.

Arguments

`string`	`VARCHAR`
`characters`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `rtrim(string = VARCHAR)`
- `rtrim(string = VARCHAR, characters = VARCHAR)`

SQL examples

```
rtrim('    test  ')
rtrim('>>>>test<<', '><')
```

`second` *DuckDB function second*

Description

Extract the second component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIME | TIMESTAMP | TIME WITH TIME ZONE | TIME_NS | TIMESTAMP W`

Value

`BIGINT`

Overloads

- `second(ts = DATE)`
- `second(ts = INTERVAL)`
- `second(ts = TIME)`
- `second(ts = TIMESTAMP)`
- `second(ts = \`TIME WITH TIME ZONE\`)`
- `second(ts = TIME_NS)`
- `second(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
second(timestamp '2021-08-03 11:59:44.123456')
```

`sem` *DuckDB function sem*

Description

Returns the standard error of the mean.

Usage

```
sem(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

`seq_scan` *DuckDB function seq_scan*

Description

DuckDB function `seq_scan()`.

Usage

```
seq_scan()
```

Value

Unspecified.

`session_user` *DuckDB function session_user*

Description

DuckDB macro `session_user()`.

Usage

```
session_user()
```

Value

Unspecified.

`set_bit` *DuckDB function set_bit*

Description

Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.

Usage

```
set_bit(bitstring = BIT, index = INTEGER, new_value = INTEGER)
```

Arguments

bitstring	BIT
index	INTEGER
new_value	INTEGER

Value

```
BIT
```

SQL examples

```
set_bit('0110010'::BIT, 2, 0)
```

`setseed` *DuckDB function setseed*

Description

Sets the seed to be used for the random function.

Usage

```
setseed(col0 = DOUBLE)
```

Arguments

col0	DOUBLE

Value

```
"NULL"
```

SQL examples

```
setseed(0.42)
```

`sha1` *DuckDB function sha1*

Description

Returns a `VARCHAR` with the SHA-1 hash of the `value`.

Returns a `VARCHAR` with the SHA-1 hash of the `blob`.

Arguments

`value`	`VARCHAR`
`blob`	`BLOB`

Value

`VARCHAR`

Overloads

- `sha1(value = VARCHAR)`
- `sha1(blob = BLOB)`

SQL examples

```
sha1(' ')
sha1('\xAA\xBB'::BLOB)
```

`sha256` *DuckDB function sha256*

Description

Returns a `VARCHAR` with the SHA-256 hash of the `value`.

Returns a `VARCHAR` with the SHA-256 hash of the `blob`.

Arguments

`value`	`VARCHAR`
`blob`	`BLOB`

Value

`VARCHAR`

Overloads

- `sha256(value = VARCHAR)`
- `sha256(blob = BLOB)`

SQL examples

```
sha256(' ')
sha256('\xAA\xBB'::BLOB)
```

`shobj_description` *DuckDB function shobj_description*

Description

DuckDB macro `shobj_description()`.

Usage

```
shobj_description(object_oid, catalog_name)
```

Arguments

`object_oid`	Unspecified.
`catalog_name`	Unspecified.

Value

Unspecified.

`show` *DuckDB function show*

Description

DuckDB function `show()`.

Usage

```
show(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

show_databases *DuckDB function show_databases*

Description

DuckDB function `show_databases()`.

Usage

```
show_databases()
```

Value

Unspecified.

show_tables *DuckDB function show_tables*

Description

DuckDB function `show_tables()`.

Usage

```
show_tables()
```

Value

Unspecified.

show_tables_expanded *DuckDB function show_tables_expanded*

Description

DuckDB function `show_tables_expanded()`.

Usage

```
show_tables_expanded()
```

Value

Unspecified.

`sign` *DuckDB function sign*

Description

Returns the sign of x as -1, 0 or 1.

Arguments

x	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT

Value

TINYINT

Overloads

- sign(x = TINYINT)
- sign(x = SMALLINT)
- sign(x = INTEGER)
- sign(x = BIGINT)
- sign(x = HUGEINT)
- sign(x = FLOAT)
- sign(x = DOUBLE)
- sign(x = UTINYINT)
- sign(x = USMALLINT)
- sign(x = UINTEGER)
- sign(x = UBIGINT)
- sign(x = UHUGEINT)

SQL examples

```
sign(-349)
```

`signbit` *DuckDB function signbit*

Description

Returns whether the signbit is set or not.

Arguments

`x`	`FLOAT | DOUBLE`

Value

`BOOLEAN`

Overloads

- `signbit(x = FLOAT)`
- `signbit(x = DOUBLE)`

SQL examples

```
signbit(-0.0)
```

`sin` *DuckDB function sin*

Description

Computes the sin of x.

Usage

```
sin(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
sin(90)
```

`sinh` *DuckDB function sinh*

Description

Computes the hyperbolic sin of x.

Usage

```
sinh(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
sinh(1)
```

`skewness` *DuckDB function skewness*

Description

Returns the skewness of all input values.

Usage

```
skewness(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
skewness(A)
```

split *DuckDB function split*

Description

Splits the `string` along the `separator`.

Usage

```
split(string = VARCHAR, separator = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`separator`	`VARCHAR`

Value

`VARCHAR[]`

SQL examples

```
split('hello-world', '-')
```

split_part *DuckDB function split_part*

Description

DuckDB macro `split_part()`.

Usage

```
split_part(string, delimiter, position)
```

Arguments

`string`	Unspecified.
`delimiter`	Unspecified.
`position`	Unspecified.

Value

Unspecified.

`sqrt` *DuckDB function sqrt*

Description

Returns the square root of x.

Usage

```
sqrt(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
sqrt(4)
```

`starts_with` *DuckDB function starts_with*

Description

Returns `true` if `string` begins with `search_string`.

Usage

```
starts_with(string = VARCHAR, search_string = VARCHAR)
```

Arguments

string	VARCHAR
search_string	VARCHAR

Value

BOOLEAN

SQL examples

```
starts_with('abc', 'a')
```

`stats` *DuckDB function stats*

Description

Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.

Usage

```
stats(expression = ANY)
```

Arguments

`expression`	`ANY`

Value

`VARCHAR`

SQL examples

```
stats(5)
```

`stddev` *DuckDB function stddev*

Description

Returns the sample standard deviation.

Usage

```
stddev(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
sqrt(var_samp(x))
```

`stddev_pop` *DuckDB function stddev_pop*

Description

Returns the population standard deviation.

Usage

```
stddev_pop(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
sqrt(var_pop(x))
```

`stddev_samp` *DuckDB function stddev_samp*

Description

Returns the sample standard deviation.

Usage

```
stddev_samp(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
sqrt(var_samp(x))
```

storage_info — *DuckDB function storage_info*

Description

DuckDB function `storage_info()`.

Usage

```
storage_info(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

str_split — *DuckDB function str_split*

Description

Splits the `string` along the `separator`.

Usage

```
str_split(string = VARCHAR, separator = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`separator`	`VARCHAR`

Value

`VARCHAR[]`

SQL examples

```
str_split('hello-world', '-')
```

`str_split_regex` *DuckDB function str_split_regex*

Description

Splits the `string` along the `regex`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`options`	`VARCHAR`

Value

`VARCHAR[]`

Overloads

- `str_split_regex(string = VARCHAR, regex = VARCHAR)`
- `str_split_regex(string = VARCHAR, regex = VARCHAR, options = VARCHAR)`

SQL examples

```
str_split_regex('hello world; 42', ';? ')
```

`strftime` *DuckDB function strftime*

Description

Converts a `date` to a string according to the format string.

Arguments

`data`	`DATE | TIMESTAMP | TIMESTAMP_NS | VARCHAR`
`format`	`VARCHAR | DATE | TIMESTAMP | TIMESTAMP_NS`

Value

`VARCHAR`

Overloads

- `strftime(data = DATE, format = VARCHAR)`
- `strftime(data = TIMESTAMP, format = VARCHAR)`
- `strftime(data = TIMESTAMP_NS, format = VARCHAR)`
- `strftime(data = VARCHAR, format = DATE)`
- `strftime(data = VARCHAR, format = TIMESTAMP)`
- `strftime(data = VARCHAR, format = TIMESTAMP_NS)`

SQL examples

```
strftime(date '1992-01-01', '%a, %-d %B %Y')
```

`string_agg` *DuckDB function string_agg*

Description

Concatenates the column string values with an optional separator.

Arguments

`str`	`ANY`
`arg`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `string_agg(str = ANY)`
- `string_agg(str = ANY, arg = VARCHAR)`

SQL examples

```
string_agg(A, '-')
```

`string_split` *DuckDB function string_split*

Description

Splits the `string` along the `separator`.

Usage

```
string_split(string = VARCHAR, separator = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`separator`	`VARCHAR`

Value

`VARCHAR[]`

SQL examples

```
string_split('hello-world', '-')
```

`string_split_regex` *DuckDB function string_split_regex*

Description

Splits the `string` along the `regex`. A set of optional regex `options` can be set.

Arguments

`string`	`VARCHAR`
`regex`	`VARCHAR`
`options`	`VARCHAR`

Value

`VARCHAR[]`

Overloads

- `string_split_regex(string = VARCHAR, regex = VARCHAR)`
- `string_split_regex(string = VARCHAR, regex = VARCHAR, options = VARCHAR)`

SQL examples

```
string_split_regex('hello world; 42', ';? ')
```

`string_to_array` *DuckDB function string_to_array*

Description

Splits the `string` along the `separator`.

Usage

```
string_to_array(string = VARCHAR, separator = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`separator`	`VARCHAR`

Value

```
VARCHAR[]
```

SQL examples

```
string_to_array('hello-world', '-')
```

`strip_accents` *DuckDB function strip_accents*

Description

Strips accents from `string`.

Usage

```
strip_accents(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

```
VARCHAR
```

SQL examples

```
strip_accents('mühleisen')
```

`strlen` *DuckDB function strlen*

Description

Number of bytes in `string`.

Usage

```
strlen(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
strlen(' ')
```

`strpos` *DuckDB function strpos*

Description

Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.

Usage

```
strpos(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

`BIGINT`

SQL examples

```
strpos('test test', 'es')
```

`strptime` *DuckDB function strptime*

Description

Converts the `string` text to timestamp according to the format string. Throws an error on failure. To return `NULL` on failure, use try_strptime.

Converts the `string` text to timestamp applying the format strings in the list until one succeeds. Throws an error on failure. To return `NULL` on failure, use try_strptime.

Arguments

`text`	`VARCHAR`
`format`	`VARCHAR`
`format-list`	`VARCHAR[]`

Value

`TIMESTAMP`

Overloads

- `strptime(text = VARCHAR, format = VARCHAR)`
- `strptime(text = VARCHAR, `format-list` = `VARCHAR[]`)`

SQL examples

```
strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')
strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])
```

`struct_concat` *DuckDB function struct_concat*

Description

Merge the multiple STRUCTs into a single STRUCT.

Usage

```
struct_concat()
```

Value

`STRUCT`

SQL examples

```
struct_concat(struct_pack(i := 4), struct_pack(s := 'string'))
```

`struct_contains` *DuckDB function struct_contains*

Description

Check if an unnamed STRUCT contains the value.

Usage

```
struct_contains(struct = STRUCT, `'entry'` = ANY)
```

Arguments

struct	STRUCT
'entry'	ANY

Value

BOOLEAN

SQL examples

```
struct_contains(ROW(3, 3, 0), 3)
```

`struct_extract` *DuckDB function struct_extract*

Description

Extract the named entry from the STRUCT.

Arguments

struct	STRUCT
'entry'	VARCHAR | BIGINT

Value

ANY

Overloads

- `struct_extract(struct = STRUCT, `'entry'` = VARCHAR)`
- `struct_extract(struct = STRUCT, `'entry'` = BIGINT)`

SQL examples

```
struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')
```

`struct_has` *DuckDB function struct_has*

Description

Check if an unnamed STRUCT contains the value.

Usage

```
struct_has(struct = STRUCT, `'entry'` = ANY)
```

Arguments

`struct`	`STRUCT`
`'entry'`	`ANY`

Value

`BOOLEAN`

SQL examples

```
struct_has(ROW(3, 3, 0), 3)
```

`struct_indexof` *DuckDB function struct_indexof*

Description

Get the position of the entry in an unnamed STRUCT, starting at 1.

Usage

```
struct_indexof(struct = STRUCT, `'entry'` = ANY)
```

Arguments

`struct`	`STRUCT`
`'entry'`	`ANY`

Value

`INTEGER`

SQL examples

```
struct_indexof(ROW(3, 3, 0), 3)
```

`struct_insert` *DuckDB function struct_insert*

Description

Adds field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).

Usage

```
struct_insert()
```

Value

```
STRUCT
```

SQL examples

```
struct_insert({'a': 1}, b := 2)
```

`struct_pack` *DuckDB function struct_pack*

Description

Create a STRUCT containing the argument values. The entry name will be the bound variable name.

Usage

```
struct_pack()
```

Value

```
STRUCT
```

SQL examples

```
struct_pack(i := 4, s := 'string')
```

`struct_position` *DuckDB function struct_position*

Description

Get the position of the entry in an unnamed STRUCT, starting at 1.

Usage

```
struct_position(struct = STRUCT, `'entry'` = ANY)
```

Arguments

`struct`	`STRUCT`
`'entry'`	`ANY`

Value

`INTEGER`

SQL examples

```
struct_position(ROW(3, 3, 0), 3)
```

`struct_update` *DuckDB function struct_update*

Description

Changes field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).

Usage

```
struct_update()
```

Value

`STRUCT`

SQL examples

```
struct_update({'a': 1}, a := 2)
```

`substr` *DuckDB function substr*

Description

Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.

Arguments

`string`	`VARCHAR`
`start`	`BIGINT`
`length`	`BIGINT`

Value

`VARCHAR`

Overloads

- `substr(string = VARCHAR, start = BIGINT, length = BIGINT)`
- `substr(string = VARCHAR, start = BIGINT)`

SQL examples

```
substring('Hello', 2)
substring('Hello', 2, 2)
```

`substring` *DuckDB function substring*

Description

Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.

Arguments

`string`	`VARCHAR`
`start`	`BIGINT`
`length`	`BIGINT`

Value

VARCHAR

Overloads

- substring(string = VARCHAR, start = BIGINT, length = BIGINT)
- substring(string = VARCHAR, start = BIGINT)

SQL examples

```
substring('Hello', 2)
substring('Hello', 2, 2)
```

---

substring_grapheme *DuckDB function substring_grapheme*

Description

Extracts substring starting from grapheme clusters `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` grapheme clusters instead. Note that a `start` value of `1` refers to the `first` character of the `string`.

Arguments

string	VARCHAR
start	BIGINT
length	BIGINT

Value

VARCHAR

Overloads

- substring_grapheme(string = VARCHAR, start = BIGINT, length = BIGINT)
- substring_grapheme(string = VARCHAR, start = BIGINT)

SQL examples

```
substring_grapheme('    ', 3)
substring_grapheme('    ', 3, 2)
```

`subtract` *DuckDB function subtract*

Description

DuckDB function `subtract()`.

Arguments

col0	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTI
col1	TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIGNUM | DATE | TIMESTAMP | INTERVAL

Value

TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLI

Overloads

- `subtract(col0 = TINYINT)`
- `subtract(col0 = TINYINT, col1 = TINYINT)`
- `subtract(col0 = SMALLINT)`
- `subtract(col0 = SMALLINT, col1 = SMALLINT)`
- `subtract(col0 = INTEGER)`
- `subtract(col0 = INTEGER, col1 = INTEGER)`
- `subtract(col0 = BIGINT)`
- `subtract(col0 = BIGINT, col1 = BIGINT)`
- `subtract(col0 = HUGEINT)`
- `subtract(col0 = HUGEINT, col1 = HUGEINT)`
- `subtract(col0 = FLOAT)`
- `subtract(col0 = FLOAT, col1 = FLOAT)`
- `subtract(col0 = DOUBLE)`
- `subtract(col0 = DOUBLE, col1 = DOUBLE)`
- `subtract(col0 = DECIMAL)`
- `subtract(col0 = DECIMAL, col1 = DECIMAL)`
- `subtract(col0 = UTINYINT)`
- `subtract(col0 = UTINYINT, col1 = UTINYINT)`
- `subtract(col0 = USMALLINT)`
- `subtract(col0 = USMALLINT, col1 = USMALLINT)`
- `subtract(col0 = UINTEGER)`

- `subtract(col0 = UINTEGER, col1 = UINTEGER)`
- `subtract(col0 = UBIGINT)`
- `subtract(col0 = UBIGINT, col1 = UBIGINT)`
- `subtract(col0 = UHUGEINT)`
- `subtract(col0 = UHUGEINT, col1 = UHUGEINT)`
- `subtract(col0 = BIGNUM)`
- `subtract(col0 = BIGNUM, col1 = BIGNUM)`
- `subtract(col0 = DATE, col1 = DATE)`
- `subtract(col0 = DATE, col1 = INTEGER)`
- `subtract(col0 = TIMESTAMP, col1 = TIMESTAMP)`
- `subtract(col0 = INTERVAL, col1 = INTERVAL)`
- `subtract(col0 = DATE, col1 = INTERVAL)`
- `subtract(col0 = TIME, col1 = INTERVAL)`
- `subtract(col0 = TIMESTAMP, col1 = INTERVAL)`
- `subtract(col0 = `TIME WITH TIME ZONE`, col1 = INTERVAL)`
- `subtract(col0 = INTERVAL)`

`suffix` *DuckDB function suffix*

Description

Returns `true` if `string` ends with `search_string`.

Usage

```
suffix(string = VARCHAR, search_string = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`search_string`	`VARCHAR`

Value

`BOOLEAN`

SQL examples

```
suffix('abc', 'bc')
```

`sum` *DuckDB function sum*

Description

Calculates the sum value for all tuples in arg.

Arguments

`arg`	`DECIMAL | BOOLEAN | SMALLINT | INTEGER | BIGINT | HUGEINT | DOUBLE | BIGNUM`

Value

`DECIMAL | HUGEINT | DOUBLE | BIGNUM`

Overloads

- `sum(arg = DECIMAL)`
- `sum(arg = BOOLEAN)`
- `sum(arg = SMALLINT)`
- `sum(arg = INTEGER)`
- `sum(arg = BIGINT)`
- `sum(arg = HUGEINT)`
- `sum(arg = DOUBLE)`
- `sum(arg = BIGNUM)`

SQL examples

```
sum(A)
```

`sum_no_overflow` *DuckDB function sum_no_overflow*

Description

Internal only. Calculates the sum value for all tuples in arg without overflow checks.

Arguments

`arg`	`INTEGER | BIGINT | DECIMAL`

Value

`HUGEINT | DECIMAL`

Overloads

- `sum_no_overflow(arg = INTEGER)`
- `sum_no_overflow(arg = BIGINT)`
- `sum_no_overflow(arg = DECIMAL)`

SQL examples

```
sum_no_overflow(A)
```

`sumkahan` *DuckDB function sumkahan*

Description

Calculates the sum using a more accurate floating point summation (Kahan Sum).

Usage

```
sumkahan(arg = DOUBLE)
```

Arguments

`arg`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
sumkahan(A)
```

`summary` *DuckDB function summary*

Description

DuckDB function `summary()`.

Usage

```
summary(col0 = TABLE)
```

Arguments

`col0`	`TABLE`

Value

Unspecified.

`table_info` *DuckDB function table_info*

Description

DuckDB function `table_info()`.

Usage

```
table_info(col0 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`

Value

Unspecified.

`tan` *DuckDB function tan*

Description

Computes the tan of x.

Usage

```
tan(x = DOUBLE)
```

Arguments

`x`	`DOUBLE`

Value

`DOUBLE`

SQL examples

```
tan(90)
```

`tanh` *DuckDB function tanh*

Description

Computes the hyperbolic tan of x.

Usage

```
tanh(x = DOUBLE)
```

Arguments

```
x               DOUBLE
```

Value

```
DOUBLE
```

SQL examples

```
tanh(1)
```

`test_all_types` *DuckDB function test_all_types*

Description

DuckDB function `test_all_types()`.

Usage

```
test_all_types(use_large_enum = BOOLEAN, use_large_bignum = BOOLEAN)
```

Arguments

```
use_large_enum
                BOOLEAN
use_large_bignum
                BOOLEAN
```

Value

Unspecified.

`test_vector_types` *DuckDB function test_vector_types*

Description

DuckDB function `test_vector_types()`.

Usage

```
test_vector_types(col0 = ANY, all_flat = BOOLEAN)
```

Arguments

`col0`	`ANY`
`all_flat`	`BOOLEAN`

Value

Unspecified.

`time_bucket` *DuckDB function time_bucket*

Description

Truncate TIMESTAMPTZ by the specified interval bucket_width. Buckets are aligned relative to origin TIMESTAMPTZ. The origin defaults to 2000-01-03 00:00:00+00 for buckets that do not include a month or year interval, and to 2000-01-01 00:00:00+00 for month and year buckets.

Arguments

`bucket_width`	`INTERVAL`
`timestamp`	`DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`
`origin`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE | VARCHAR`

Value

`DATE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Overloads

- `time_bucket(bucket_width = INTERVAL, timestamp = DATE)`
- `time_bucket(bucket_width = INTERVAL, timestamp = DATE, origin = DATE)`
- `time_bucket(bucket_width = INTERVAL, timestamp = DATE, origin = INTERVAL)`
- `time_bucket(bucket_width = INTERVAL, timestamp = TIMESTAMP)`
- `time_bucket(bucket_width = INTERVAL, timestamp = TIMESTAMP, origin = INTERVAL)`
- `time_bucket(bucket_width = INTERVAL, timestamp = TIMESTAMP, origin = TIMESTAMP)`
- `time_bucket(bucket_width = INTERVAL, timestamp = `TIMESTAMP WITH TIME ZONE`)`
- `time_bucket(bucket_width = INTERVAL, timestamp = `TIMESTAMP WITH TIME ZONE`, origin = INTERVAL)`
- `time_bucket(bucket_width = INTERVAL, timestamp = `TIMESTAMP WITH TIME ZONE`, origin = `TIMESTAMP WITH TIME ZONE`)`
- `time_bucket(bucket_width = INTERVAL, timestamp = `TIMESTAMP WITH TIME ZONE`, origin = VARCHAR)`

SQL examples

```
time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00-07', TIMESTAMP '1992-04-01 00:0
```

---

timetz_byte_comparable

DuckDB function timetz_byte_comparable

Description

Converts a TIME WITH TIME ZONE to an integer sort key.

Usage

```
timetz_byte_comparable(time_tz = `TIME WITH TIME ZONE`)
```

Arguments

`time_tz`	`TIME WITH TIME ZONE`

Value

`UBIGINT`

SQL examples

```
timetz_byte_comparable('18:18:16.21-07:00'::TIMETZ)
```

`timezone` *DuckDB function timezone*

Description

Extract the timezone component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE | VARCHAR`
`col1`	`TIME WITH TIME ZONE | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT | TIME WITH TIME ZONE | TIMESTAMP WITH TIME ZONE | TIMESTAMP`

Overloads

- `timezone(ts = DATE)`
- `timezone(ts = INTERVAL)`
- `timezone(ts = INTERVAL, col1 = `TIME WITH TIME ZONE`)`
- `timezone(ts = TIMESTAMP)`
- `timezone(ts = `TIMESTAMP WITH TIME ZONE`)`
- `timezone(ts = VARCHAR, col1 = TIMESTAMP)`
- `timezone(ts = VARCHAR, col1 = `TIMESTAMP WITH TIME ZONE`)`
- `timezone(ts = VARCHAR, col1 = `TIME WITH TIME ZONE`)`

SQL examples

```
timezone(timestamp '2021-08-03 11:59:44.123456')
```

`timezone_hour` *DuckDB function timezone_hour*

Description

Extract the timezone_hour component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `timezone_hour(ts = DATE)`
- `timezone_hour(ts = INTERVAL)`
- `timezone_hour(ts = TIMESTAMP)`
- `timezone_hour(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
timezone_hour(timestamp '2021-08-03 11:59:44.123456')
```

`timezone_minute` *DuckDB function timezone_minute*

Description

Extract the timezone_minute component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `timezone_minute(ts = DATE)`
- `timezone_minute(ts = INTERVAL)`
- `timezone_minute(ts = TIMESTAMP)`
- `timezone_minute(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
timezone_minute(timestamp '2021-08-03 11:59:44.123456')
```

`to_base` *DuckDB function to_base*

Description

Converts `number` to a string in the given base `radix`, optionally padding with leading zeros to `min_length`.

Arguments

`number`	`BIGINT`
`radix`	`INTEGER`
`min_length`	`INTEGER`

Value

`VARCHAR`

Overloads

- `to_base(number = BIGINT, radix = INTEGER)`
- `to_base(number = BIGINT, radix = INTEGER, min_length = INTEGER)`

SQL examples

```
to_base(42, 16, 5)
```

`to_base64` *DuckDB function to_base64*

Description

Converts a `blob` to a base64 encoded string.

Usage

```
to_base64(blob = BLOB)
```

Arguments

`blob`	`BLOB`

Value

`VARCHAR`

SQL examples

```
to_base64('A'::BLOB)
```

`to_binary` *DuckDB function to_binary*

Description

Converts the `string` to binary representation.

Converts the `value` to binary representation.

Arguments

`string`	`VARCHAR`
`value`	`BIGNUM | UBIGINT | BIGINT | HUGEINT | UHUGEINT`

Value

`VARCHAR`

Overloads

- `to_binary(string = VARCHAR)`
- `to_binary(value = BIGNUM)`
- `to_binary(value = UBIGINT)`
- `to_binary(value = BIGINT)`
- `to_binary(value = HUGEINT)`
- `to_binary(value = UHUGEINT)`

SQL examples

```
to_binary('Aa')
to_binary(42)
```

`to_centuries` *DuckDB function to_centuries*

Description

Construct a century interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_centuries(integer = INTEGER)`
- `to_centuries(integer = BIGINT)`

SQL examples

```
to_centuries(5)
```

`to_days` *DuckDB function to_days*

Description

Construct a day interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_days(integer = INTEGER)`
- `to_days(integer = BIGINT)`

SQL examples

```
to_days(5)
```

`to_decades` *DuckDB function to_decades*

Description

Construct a decade interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_decades(integer = INTEGER)`
- `to_decades(integer = BIGINT)`

SQL examples

```
to_decades(5)
```

`to_hex` *DuckDB function to_hex*

Description

Converts the `string` to hexadecimal representation.

Converts the `value` to `VARCHAR` using hexadecimal representation.

Converts `blob` to `VARCHAR` using hexadecimal encoding.

Arguments

`string`	`VARCHAR`
`value`	`BIGNUM | BIGINT | UBIGINT | HUGEINT | UHUGEINT`
`blob`	`BLOB`

Value

`VARCHAR`

Overloads

- `to_hex(string = VARCHAR)`
- `to_hex(value = BIGNUM)`
- `to_hex(blob = BLOB)`
- `to_hex(value = BIGINT)`
- `to_hex(value = UBIGINT)`
- `to_hex(value = HUGEINT)`
- `to_hex(value = UHUGEINT)`

SQL examples

```
to_hex('Hello')
to_hex(42)
to_hex('\xAA\xBB'::BLOB)
```

`to_hours` *DuckDB function to_hours*

Description

Construct a hour interval.

Usage

```
to_hours(integer = BIGINT)
```

Arguments

```
integer        BIGINT
```

Value

```
INTERVAL
```

SQL examples

```
to_hours(5)
```

`to_microseconds` *DuckDB function to_microseconds*

Description

Construct a microsecond interval.

Usage

```
to_microseconds(integer = BIGINT)
```

Arguments

`integer`	`BIGINT`

Value

`INTERVAL`

SQL examples

```
to_microseconds(5)
```

`to_millennia` *DuckDB function to_millennia*

Description

Construct a millenium interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_millennia(integer = INTEGER)`
- `to_millennia(integer = BIGINT)`

SQL examples

```
to_millennia(1)
```

`to_milliseconds` *DuckDB function to_milliseconds*

Description

Construct a millisecond interval.

Usage

```
to_milliseconds(double = DOUBLE)
```

Arguments

`double`	`DOUBLE`

Value

`INTERVAL`

SQL examples

```
to_milliseconds(5.5)
```

`to_minutes` *DuckDB function to_minutes*

Description

Construct a minute interval.

Usage

```
to_minutes(integer = BIGINT)
```

Arguments

`integer`	`BIGINT`

Value

`INTERVAL`

SQL examples

```
to_minutes(5)
```

`to_months` *DuckDB function to_months*

Description

Construct a month interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_months(integer = INTEGER)`
- `to_months(integer = BIGINT)`

SQL examples

```
to_months(5)
```

`to_quarters` *DuckDB function to_quarters*

Description

Construct a quarter interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_quarters(integer = INTEGER)`
- `to_quarters(integer = BIGINT)`

SQL examples

```
to_quarters(5)
```

`to_seconds` *DuckDB function to_seconds*

Description

Construct a second interval.

Usage

```
to_seconds(double = DOUBLE)
```

Arguments

double	DOUBLE

Value

INTERVAL

SQL examples

```
to_seconds(5.5)
```

`to_timestamp` *DuckDB function to_timestamp*

Description

Converts secs since epoch to a timestamp with time zone.

Usage

```
to_timestamp(sec = DOUBLE)
```

Arguments

sec	DOUBLE

Value

TIMESTAMP WITH TIME ZONE

SQL examples

```
to_timestamp(1284352323.5)
```

`to_weeks` *DuckDB function to_weeks*

Description

Construct a week interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_weeks(integer = INTEGER)`
- `to_weeks(integer = BIGINT)`

SQL examples

```
to_weeks(5)
```

`to_years` *DuckDB function to_years*

Description

Construct a year interval.

Arguments

`integer`	`INTEGER | BIGINT`

Value

`INTERVAL`

Overloads

- `to_years(integer = INTEGER)`
- `to_years(integer = BIGINT)`

SQL examples

```
to_years(5)
```

`transaction_timestamp`
DuckDB function transaction_timestamp

Description

Returns the current timestamp.

Usage

```
transaction_timestamp()
```

Value

`TIMESTAMP WITH TIME ZONE`

SQL examples

```
transaction_timestamp()
```

`translate` *DuckDB function translate*

Description

Replaces each character in `string` that matches a character in the `from` set with the corresponding character in the `to` set. If `from` is longer than `to`, occurrences of the extra characters in `from` are deleted.

Usage

```
translate(string = VARCHAR, from = VARCHAR, to = VARCHAR)
```

Arguments

`string`	`VARCHAR`
`from`	`VARCHAR`
`to`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
translate('12345', '143', 'ax')
```

`trim` *DuckDB function trim*

Description

Removes any occurrences of any of the `characters` from either side of the `string`. `characters` defaults to `space`.

Arguments

`string`	`VARCHAR`
`characters`	`VARCHAR`

Value

`VARCHAR`

Overloads

- `trim(string = VARCHAR)`
- `trim(string = VARCHAR, characters = VARCHAR)`

SQL examples

```
trim('    test  ')
trim('>>>>test<<', '><')
```

`trunc` *DuckDB function trunc*

Description

Truncates the number.

Arguments

`x`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`
`col1`	`INTEGER`

Value

`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | FLOAT | DOUBLE | DECIMAL | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT`

Overloads

- `trunc(x = TINYINT)`
- `trunc(x = TINYINT, col1 = INTEGER)`
- `trunc(x = SMALLINT)`
- `trunc(x = SMALLINT, col1 = INTEGER)`
- `trunc(x = INTEGER)`
- `trunc(x = INTEGER, col1 = INTEGER)`
- `trunc(x = BIGINT)`
- `trunc(x = BIGINT, col1 = INTEGER)`
- `trunc(x = HUGEINT)`
- `trunc(x = HUGEINT, col1 = INTEGER)`
- `trunc(x = FLOAT)`
- `trunc(x = FLOAT, col1 = INTEGER)`
- `trunc(x = DOUBLE)`
- `trunc(x = DOUBLE, col1 = INTEGER)`
- `trunc(x = DECIMAL)`
- `trunc(x = DECIMAL, col1 = INTEGER)`
- `trunc(x = UTINYINT)`
- `trunc(x = UTINYINT, col1 = INTEGER)`
- `trunc(x = USMALLINT)`
- `trunc(x = USMALLINT, col1 = INTEGER)`
- `trunc(x = UINTEGER)`
- `trunc(x = UINTEGER, col1 = INTEGER)`
- `trunc(x = UBIGINT)`
- `trunc(x = UBIGINT, col1 = INTEGER)`
- `trunc(x = UHUGEINT)`
- `trunc(x = UHUGEINT, col1 = INTEGER)`

SQL examples

```
trunc(17.4)
```

`truncate_duckdb_logs` *DuckDB function truncate_duckdb_logs*

Description

DuckDB function `truncate_duckdb_logs()`.

Usage

```
truncate_duckdb_logs()
```

Value

Unspecified.

`try_strptime` *DuckDB function try_strptime*

Description

Converts the `string` text to timestamp according to the format string. Returns `NULL` on failure.

Arguments

`text`	`VARCHAR`
`format`	`VARCHAR | VARCHAR[]`

Value

`TIMESTAMP`

Overloads

- `try_strptime(text = VARCHAR, format = VARCHAR)`
- `try_strptime(text = VARCHAR, format = ` `VARCHAR[]` `)`

SQL examples

```
try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')
```

`txid_current` *DuckDB function txid_current*

Description

Returns the current transaction's ID (a BIGINT). It will assign a new one if the current transaction does not have one already.

Usage

```
txid_current()
```

Value

```
UBIGINT
```

SQL examples

```
txid_current()
```

`typeof` *DuckDB function typeof*

Description

Returns the name of the data type of the result of the expression.

Usage

```
typeof(expression = ANY)
```

Arguments

```
expression     ANY
```

Value

```
VARCHAR
```

SQL examples

```
typeof('abc')
```

`ucase` *DuckDB function ucase*

Description

Converts `string` to upper case.

Usage

```
ucase(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
ucase('Hello')
```

`unbin` *DuckDB function unbin*

Description

Converts a `value` from binary representation to a blob.

Usage

```
unbin(value = VARCHAR)
```

Arguments

`value`	`VARCHAR`

Value

`BLOB`

SQL examples

```
unbin('0110')
```

`unhex` *DuckDB function unhex*

Description

Converts a `value` from hexadecimal representation to a blob.

Usage

```
unhex(value = VARCHAR)
```

Arguments

`value`	`VARCHAR`

Value

`BLOB`

SQL examples

```
unhex('2A')
```

`unicode` *DuckDB function unicode*

Description

Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.

Usage

```
unicode(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`INTEGER`

SQL examples

```
[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]
```

`union_extract` *DuckDB function union_extract*

Description

Extract the value with the named tags from the union. NULL if the tag is not currently selected.

Usage

```
union_extract(union = UNION, tag = VARCHAR)
```

Arguments

union	UNION
tag	VARCHAR

Value

```
ANY
```

SQL examples

```
union_extract(s, 'k')
```

`union_tag` *DuckDB function union_tag*

Description

Retrieve the currently selected tag of the union as an ENUM.

Usage

```
union_tag(union = UNION)
```

Arguments

union	UNION

Value

```
ANY
```

SQL examples

```
union_tag(union_value(k := 'foo'))
```

`union_value` *DuckDB function union_value*

Description

Create a single member UNION containing the argument value. The tag of the value will be the bound variable name.

Usage

```
union_value()
```

Value

```
UNION
```

SQL examples

```
union_value(k := 'hello')
```

`unnest` *DuckDB function unnest*

Description

DuckDB function `unnest()`.

Usage

```
unnest(col0 = ANY)
```

Arguments

`col0`	`ANY`

Value

Unspecified.

`unpivot_list` *DuckDB function unpivot_list*

Description

Identical to list_value, but generated as part of unpivot for better error messages.

Usage

```
unpivot_list()
```

Value

```
LIST
```

SQL examples

```
unpivot_list(4, 5, 6)
```

`upper` *DuckDB function upper*

Description

Converts `string` to upper case.

Usage

```
upper(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

```
VARCHAR
```

SQL examples

```
upper('Hello')
```

`url_decode` *DuckDB function url_decode*

Description

Decodes a URL from a representation using Percent-Encoding.

Usage

```
url_decode(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
url_decode('https%3A%2F%2Fduckdb.org%2Fwhy_duckdb%23portable')
```

`url_encode` *DuckDB function url_encode*

Description

Encodes a URL to a representation using Percent-Encoding.

Usage

```
url_encode(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

`VARCHAR`

SQL examples

```
url_encode('this string has/ special+ characters>')
```

`user` *DuckDB function user*

Description

DuckDB macro `user()`.

Usage

```
user()
```

Value

Unspecified.

`user_agent` *DuckDB function user_agent*

Description

DuckDB function `user_agent()`.

Usage

```
user_agent()
```

Value

Unspecified.

`uuid` *DuckDB function uuid*

Description

Returns a random UUID v4 similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.

Usage

```
uuid()
```

Value

`UUID`

SQL examples

```
uuid()
```

`uuid_extract_timestamp`
DuckDB function uuid_extract_timestamp

Description

Extract the timestamp for the given UUID v7.

Usage

```
uuid_extract_timestamp(uuid = UUID)
```

Arguments

```
uuid           UUID
```

Value

```
TIMESTAMP WITH TIME ZONE
```

SQL examples

```
uuid_extract_timestamp('019482e4-1441-7aad-8127-eec99573b0a0')
```

`uuid_extract_version` *DuckDB function uuid_extract_version*

Description

Extract a version for the given UUID.

Usage

```
uuid_extract_version(uuid = UUID)
```

Arguments

```
uuid           UUID
```

Value

```
UINTEGER
```

SQL examples

```
uuid_extract_version('019482e4-1441-7aad-8127-eec99573b0a0')
```

`uuidv4` *DuckDB function uuidv4*

Description

Returns a random UUIDv4 similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.

Usage

```
uuidv4()
```

Value

`UUID`

SQL examples

```
uuidv4()
```

`uuidv7` *DuckDB function uuidv7*

Description

Returns a random UUID v7 similar to this: 019482e4-1441-7aad-8127-eec99573b0a0.

Usage

```
uuidv7()
```

Value

`UUID`

SQL examples

```
uuidv7()
```

`var_pop` *DuckDB function var_pop*

Description

Returns the population variance.

Usage

```
var_pop(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

`var_samp` *DuckDB function var_samp*

Description

Returns the sample variance of all input values.

Usage

```
var_samp(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
(SUM(x^2) - SUM(x)^2 / COUNT(x)) / (COUNT(x) - 1)
```

`variance` *DuckDB function variance*

Description

Returns the sample variance of all input values.

Usage

```
variance(x = DOUBLE)
```

Arguments

x	DOUBLE

Value

DOUBLE

SQL examples

```
(SUM(x^2) - SUM(x)^2 / COUNT(x)) / (COUNT(x) - 1)
```

`variant_extract` *DuckDB function variant_extract*

Description

DuckDB function `variant_extract()`.

Arguments

col0	VARIANT
col1	VARCHAR | UINTEGER

Value

VARIANT

Overloads

- `variant_extract(col0 = VARIANT, col1 = VARCHAR)`
- `variant_extract(col0 = VARIANT, col1 = UINTEGER)`

`variant_typeof` *DuckDB function variant_typeof*

Description

Returns the internal type of the `input_variant`.

Usage

```
variant_typeof(input_variant = VARIANT)
```

Arguments

```
input_variant  VARIANT
```

Value

```
VARCHAR
```

SQL examples

```
variant_typeof({'a': 42, 'b': [1,2,3]})::VARIANT)
```

`vector_type` *DuckDB function vector_type*

Description

Returns the VectorType of a given column.

Usage

```
vector_type(col = ANY)
```

Arguments

```
col            ANY
```

Value

```
VARCHAR
```

SQL examples

```
vector_type(col)
```

verify_external *DuckDB function verify_external*

Description

DuckDB function `verify_external()`.

Usage

```
verify_external()
```

Value

Unspecified.

verify_fetch_row *DuckDB function verify_fetch_row*

Description

DuckDB function `verify_fetch_row()`.

Usage

```
verify_fetch_row()
```

Value

Unspecified.

verify_parallelism *DuckDB function verify_parallelism*

Description

DuckDB function `verify_parallelism()`.

Usage

```
verify_parallelism()
```

Value

Unspecified.

`verify_serializer` *DuckDB function verify_serializer*

Description

DuckDB function `verify_serializer()`.

Usage

```
verify_serializer()
```

Value

Unspecified.

`version` *DuckDB function version*

Description

Returns the currently active version of DuckDB in this format: v0.3.2 .

Value

`VARCHAR`

Overloads

- `version()`
- `version()`

SQL examples

```
version()
```

wavg *DuckDB function wavg*

Description

DuckDB macro `wavg()`.

Usage

```
wavg(value, weight)
```

Arguments

`value`	Unspecified.
`weight`	Unspecified.

Value

Unspecified.

week *DuckDB function week*

Description

Extract the week component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `week(ts = DATE)`
- `week(ts = INTERVAL)`
- `week(ts = TIMESTAMP)`
- `week(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
week(timestamp '2021-08-03 11:59:44.123456')
```

`weekday` *DuckDB function weekday*

Description

Extract the weekday component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `weekday(ts = DATE)`
- `weekday(ts = INTERVAL)`
- `weekday(ts = TIMESTAMP)`
- `weekday(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
weekday(timestamp '2021-08-03 11:59:44.123456')
```

`weekofyear` *DuckDB function weekofyear*

Description

Extract the weekofyear component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `weekofyear(ts = DATE)`
- `weekofyear(ts = INTERVAL)`
- `weekofyear(ts = TIMESTAMP)`
- `weekofyear(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
weekofyear(timestamp '2021-08-03 11:59:44.123456')
```

`weighted_avg` *DuckDB function weighted_avg*

Description

DuckDB macro `weighted_avg()`.

Usage

```
weighted_avg(value, weight)
```

Arguments

`value`	Unspecified.
`weight`	Unspecified.

Value

Unspecified.

`which_secret` *DuckDB function which_secret*

Description

DuckDB function `which_secret()`.

Usage

```
which_secret(col0 = VARCHAR, col1 = VARCHAR)
```

Arguments

`col0`	`VARCHAR`
`col1`	`VARCHAR`

Value

Unspecified.

`write_log` *DuckDB function write_log*

Description

Writes to the logger.

Usage

```
write_log(string = VARCHAR)
```

Arguments

`string`	`VARCHAR`

Value

```
ANY
```

SQL examples

```
write_log('Hello')
```

`xor` *DuckDB function xor*

Description

Bitwise XOR.

Arguments

`left`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`
`right`	`TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER | UBIGINT | UHUGEINT | BIT`

Value

```
TINYINT | SMALLINT | INTEGER | BIGINT | HUGEINT | UTINYINT | USMALLINT | UINTEGER
| UBIGINT | UHUGEINT | BIT
```

Overloads

- `xor(left = TINYINT, right = TINYINT)`
- `xor(left = SMALLINT, right = SMALLINT)`
- `xor(left = INTEGER, right = INTEGER)`
- `xor(left = BIGINT, right = BIGINT)`
- `xor(left = HUGEINT, right = HUGEINT)`
- `xor(left = UTINYINT, right = UTINYINT)`
- `xor(left = USMALLINT, right = USMALLINT)`
- `xor(left = UINTEGER, right = UINTEGER)`
- `xor(left = UBIGINT, right = UBIGINT)`
- `xor(left = UHUGEINT, right = UHUGEINT)`
- `xor(left = BIT, right = BIT)`

SQL examples

```
xor(17, 5)
```

---

year *DuckDB function year*

Description

Extract the year component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `year(ts = DATE)`
- `year(ts = INTERVAL)`
- `year(ts = TIMESTAMP)`
- `year(ts = \`TIMESTAMP WITH TIME ZONE\`)`

SQL examples

```
year(timestamp '2021-08-03 11:59:44.123456')
```

`yearweek` *DuckDB function yearweek*

Description

Extract the yearweek component from a date or timestamp.

Arguments

`ts`	`DATE | INTERVAL | TIMESTAMP | TIMESTAMP WITH TIME ZONE`

Value

`BIGINT`

Overloads

- `yearweek(ts = DATE)`
- `yearweek(ts = INTERVAL)`
- `yearweek(ts = TIMESTAMP)`
- `yearweek(ts = `TIMESTAMP WITH TIME ZONE`)`

SQL examples

```
yearweek(timestamp '2021-08-03 11:59:44.123456')
```

Index