SAP HANA Cloud, SAP HANA Database SQL Reference Guide

Generated on: 2024-04-02 05:00:39 GMT+0000

SAP HANA Cloud, SAP HANA Database | QRC 1/2024

PUBLIC

Original content: https://help.sap.com/docs/HANA_CLOUD_DATABASE/c1d3f60099654ecfb3fe36ac93c121bb?locale=en-US&state=PRODUCTION&version=2024_1_QRC

Warning

SQL Functions

Documents the built-in SQL functions that are provided with SAP HANA.

Analytic functions are positioned in this guide as aggregate functions and window aggregate functions.

Alphabetical List Of Functions

Aggregate Functions

Aggregate functions are analytic functions that calculate an aggregate value based on a group of rows.

Array Functions

Array functions take arrays as input.

Data Type Conversion Functions

Data type conversion functions convert data from one data type to another data type.

Datetime Functions

Date and time functions perform operations on date and time data types or return date or time information.

Hierarchy Functions

Hierarchy functions allow you to work with hierarchical data such as tables with rows arranged in a tree or directed graph.

JSON Functions

JSON functions are functions that return or operate on JSON data.

Miscellaneous Functions

SAP HANA supports many functions that return system values and perform various operations on values, expressions, and return values of other functions.

Numeric Functions

Numeric functions perform mathematical operations on numerical data types or return numeric information.

Security Functions

Security functions provide special functionality for security purposes.

Series Data Functions

Series data functions provide special functionality for series data and series tables.

String Functions

String functions perform extraction and manipulation on strings, or return information about strings.

Vector Functions

Vector functions perform calculations on vectors.

Window Functions and the Window Specification

Window functions allow you to perform analytic operations over a set of input rows.

Alphabetical List Of Functions

ABAP_ALPHANUM Function (String)

Converts a string to what would result if the string was transformed into an ALPHANUM type and then converted back to a string.

Syntax

```
ABAP_ALPHANUM( <string>, <chars> )
```

Syntax Elements

<string>

Specifies the input string to convert.

<chars>

Specifies the length to left-pad *<string>* with (using 0s) if *<string>* is shorter than the length specified by *<chars>*.

Description

The input value, *<string>*, is checked to find out if it is numeric or a mixture of numeric and non-numeric. If *<string>* is numeric, then it is left-padded with zeroes up to the length specified by *<chars>*, if necessary. If *<string>* is a mixture or numeric and non-numeric, or is empty, then it is returned unchanged.

Strings that start with spaces and contain only digits after the whitespace prefix are regarded as numeric.

Example

The following example returns `012`. Since *<chars>* is set to 3, the left-most digit is padded with a 0.

```
SELECT ABAP_ALPHANUM('12', 3) FROM DUMMY;
```

The following example returns `1234`. The input value was numeric and was not shorter than *<chars>*, so no left-padding was required.

```
SELECT ABAP_ALPHANUM('1234', 3) FROM DUMMY;
```

The following example returns `12x`. Because the input value contained a mixture of numeric and non-numeric, it is returned unchanged.

```
SELECT ABAP_ALPHANUM('12x', 13) FROM DUMMY;
```

Related Information

Character String Data Types

ABAP_DF16RAW_TO_SMALLDECIMAL Function (String)

Converts from ABAP *DECFLOAT* type *DF16_RAW* to HANA *DECFLOAT* type *SMALLDECIMAL*.

Syntax

```
ABAP_DF16RAW_TO_SMALLDECIMAL(<binary_data_in_df16raw_format>)
```

Syntax Elements

<binary_data_in_df16raw_format>

The format to convert: Decimal Floating Point, 16 Digits, RAW on database

Description

Converts from ABAP *DECFLOAT* type *DF16_RAW* to HANA *DECFLOAT* type *SMALLDECIMAL*.

Please note that only one-way conversion from ABAP to HANA type is available. The reverse conversion must be performed in the ABAP layer if needed. For more information, see Decimal Floating Point Numbers in the ABAP Keyword Documentation.

Example

The following example converts the binary value to type decimal.

```
CREATE TABLE t1(b binary(16))
INSERT INTO t1 values(X'BF2C000000000000')
INSERT INTO t1 values(X'BF54000000000001')
INSERT INTO t1 values(X'3DE3E7F9FE7F9FE7')

SELECT ABAP_DF16RAW_TO_SMALLDECIMAL(b) FROM t1
/* returns
[ Decimal('0.1')
, Decimal('1.000000000000001')
, Decimal('-0.1')]
*/
```

ABAP_DF34RAW_TO_DECIMAL Function (String)

Converts from ABAP *DECFLOAT* type *DF34_RAW* to HANA *DECFLOAT* type *DECIMAL*.

Syntax

```
ABAP_DF34RAW_TO_DECIMAL(<binary_data_in_df34raw_format>)
```

Syntax Elements

<binary_data_in_df34raw_format>

The format to convert: Decimal Floating Point, 34 Digits, RAW on database

Description

Converts from ABAP *DECFLOAT* type *DF34_RAW* to HANA *DECFLOAT* type *DECIMAL*.

Please note that only one-way conversion from ABAP to HANA type is available. The reverse conversion must be performed in the ABAP layer if needed. For more information, see Decimal Floating Point Numbers in the ABAP Keyword Documentation.

Example

The following example converts the binary value to type *DECIMAL*.

```
CREATE TABLE t2(b binary(64));
INSERT INTO t2 values(X'BDBBC0000000000000000000000000000');
INSERT INTO t2 values(X'BDBE4000000000000000000000000001');
INSERT INTO t2 values(X'BDBEC8D9428D937193B9CEA0D7F45DF7');
```

```
SELECT ABAP_DF34RAW_TO_DECIMAL(b) FROM t2
/* returns
[ Decimal('0.1')
, Decimal('1.00000000000000000000000000000001')
, Decimal('3.14159265358979323846264338327950 3')]
*/
```

ABAP_LOWER Function (String)

Converts all characters in a specified string to lowercase.

Syntax

```
ABAP_LOWER(<string>)
```

Syntax Elements

<string>

The string to convert to lowercase.

Description

Converts all characters in the *<string>* parameter to lowercase.

Example

The following example converts the given string to lowercase and returns the value `ant`:

```
SELECT ABAP_LOWER ('AnT') "lower" FROM DUMMY;
```

Related Information

Character String Data Types

ABAP_NUMC Function (String)

Converts an input string to a string of a specified length, that contains only digits.

Syntax

```
ABAP_NUMC( <string>, <chars> )
```

Syntax Elements

<string>

Specifies the input string to convert.

<chars>

Specifies the length that the returned string should be.

Description

String and numeric values are left-padded with 0s if *<string>* is shorter than *<chars>*. If *<string>* is longer than *<chars>*, then characters are truncated from the left.

Numeric values are rounded to their integer part and the sign is discarded.

TIMESTAMP, SECONDDATE, DATE, and TIME values are padded and truncated from the right side, as necessary.

DECIMAL and SMALLDECIMAL inputs are never truncated but create an overflow error if they are found to be too long.

Example

The following example returns `012`.

```
SELECT ABAP_NUMC(12, 3) FROM DUMMY;
```

The following example returns `234`.

```
SELECT ABAP_NUMC(1234, 3) FROM DUMMY;
```

The following example returns `01230000000000000000`.

```
SELECT ABAP_NUMC(1.23e18, 20) FROM DUMMY;
```

The following example returns `001235`.

```
SELECT ABAP_NUMC(-1234.5, 6) FROM DUMMY;
```

Related Information

Character String Data Types

ABAP_UPPER Function (String)

Converts all characters in the specified string to uppercase.

Syntax

```
ABAP_UPPER(<string>)
```

Syntax Elements

<string>

The string to convert to uppercase.

Description

Converts all characters in the *<string>* parameter to uppercase.

Example

The following example converts the given string to uppercase and returns the value ANT:

```
SELECT ABAP_UPPER ('Ant') "uppercase" FROM DUMMY;
```

Related Information

Character String Data Types

ABS Function (Numeric)

Returns the absolute value of a numeric argument.

Syntax

```
ABS(<num>)
```

Syntax Elements

<num>

Specifies the numeric argument.

Description

Returns the absolute value of the numeric argument *<num>*.

Example

The following example returns the value `1` for `"absolute"`:

```
SELECT ABS (-1) "absolute" FROM DUMMY;
```

Related Information

Numeric Data Types

ACOS Function (Numeric)

Returns the arc-cosine, in radians, of a numeric argument between -1 and 1.

Syntax

```
ACOS(<num>)
```

Syntax Elements

<num>

Specifies a numeric argument between -1 and 1.

Description

Returns the arc-cosine, in radians, of the numeric argument *<n>* between -1 and 1.

Example

The following example returns the value `1.0471975511965979`:

```
SELECT ACOS (0.5) "acos" FROM DUMMY;
```

Related Information

Numeric Data Types

ADD_DAYS Function (Datetime)

Computes the specified date, plus or minus the specified number of days.

Syntax

```
ADD_DAYS(<date>, <num_days>)
```

Syntax Elements

<date>

Specifies the date to increment.

<num_days>

Specifies the number of days (integer). A positive integer increments the date by the specified amount; a negative integer decrements the date.

Description

Computes the specified date plus or minus the specified number of days.

Example

The following example increments the date value `2009-12-05` by 30 days and returns the value `2010-01-04`:

```
SELECT ADD_DAYS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 30) "add days" FROM DUMMY;
```

The following example decrements the date value `2009-12-05` by 30 days and returns the value `2009-11-05`:

```
SELECT ADD_DAYS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), -30) "subtract days" FROM DUMMY;
```

Related Information

Datetime Data Types

ADD_MONTHS Function (Datetime)

Computes the specified date plus the specified number of months.

Syntax

```
ADD_MONTHS(<date>, <num_months>)
```

Syntax Elements

<date>

The date that is incremented.

<num_months>

The number of months by which to increment the date.

Description

Computes the specified date plus the specified number of months.

To compute the date so that the output date is set to the last day of the month when the input date is the last day of the month, use the ADD_MONTHS_LAST function.

The parameter *<date>* must be implicitly or explicitly converted to one of the following SQL data types:

- DATE
- TIMESTAMP
- SECONDDATE

The SQL data type of the output parameters is the same as the SQL data type of the input parameters. For example, ADD_MONTHS_LAST(DATE) returns a date, while ADD_MONTHS_LAST(TIMESTAMP) returns a timestamp.

Example

The following example increments the date 2009-12-05 by one month, and returns the value 2010-01-05:

```
SELECT ADD_MONTHS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add months" FROM DUMMY;
```

Related Information

Datetime Data Types

ADD_MONTHS_LAST Function (Datetime)

Computes the specified date plus the specified number of months, with the output date being the last day of the month if the input date is the last day of the month, even if those two dates differ.

Syntax

```
ADD_MONTHS_LAST(<date>, <num_months>)
```

Syntax Elements

<date>

The date that is incremented.

<num_months>

The number of months by which to increment the date.

Description

Computes the specified date plus the specified number of months. If the input date is the last day of the input month, then the output date is set to the last day of the output month.

To compute the date plus months without using the last day of the month functionality, use the ADD_MONTHS function.

The parameter *<date>* must be implicitly or explicitly converted to one of the following SQL data types:

- DATE
- TIMESTAMP
- SECONDDATE

The SQL data type of the output parameters is the same as the SQL data type of the input parameters. For example, ADD_MONTHS_LAST(DATE) returns a date, while ADD_MONTHS_LAST(TIMESTAMP) returns a timestamp.

Example

The following example increments the date 2009-02-28 (the last day in February), by one month, and returns the value 2009-03-31, (the last day of March).

```
SELECT ADD_MONTHS_LAST (TO_DATE ('2009-02-28', 'YYYY-MM-DD'), 1) "add months last" FROM DUMMY;
```

Related Information

Datetime Data Types

ADD_NANO100 Function (Datetime)

Adds the specified number of 10^-7 unit of seconds to the specified TIMESTAMP value.

Syntax

```
ADD_NANO100( <time>, <num> )
```

Syntax Elements

<time>

The TIMESTAMP value to add the units of seconds to.

<num>

The number of 10^-7 unit of seconds to increment the TIMESTAMP value by.

Description

Computes the specified TIMESTAMP value plus the specified number of microseconds.

Example

The following example increments the TIMESTAMP value by `864000000000` microseconds and returns `1000-01-02 10:00:00.0000000`:

```
SELECT ADD_NANO100(TO_TIMESTAMP('1000-01-01 10:00:00.0000000'), 864000000000) FROM DUMMY;
```

The following example increments the TIMESTAMP value by `1` microsecond and returns `1000-01-01 10:00:00.0000001`:

```
SELECT ADD_NANO100(TO_TIMESTAMP('1000-01-01 10:00:00.0000000'), 1) FROM DUMMY;
```

Related Information

ADD_SECONDS Function (Datetime)

ADD_SECONDS Function (Datetime)

Computes the specified time plus the specified seconds.

Syntax

```
ADD_SECONDS(<time>, <num_seconds>)
```

Syntax Elements

<time>

The time that is incremented.

<num_seconds>

The number of seconds to increment the time by. Fractional values are supported for milliseconds and microseconds:

- 1/1000 – milliseconds
- 1/1000000 - microseconds

Description

Computes the specified time plus the number of specified seconds.

Example

The example increments the TIMESTAMP `2012-01-01 23:30:45` by `60*30` seconds, and returns the value `2012-01-02 00:00:45.0`:

```
SELECT ADD_SECONDS (TO_TIMESTAMP ('2012-01-01 23:30:45'), 60*30) "add seconds" FROM DUMMY;
```

Related Information

Datetime Data Types
ADD_NANO100 Function (Datetime)

ADD_WORKDAYS Function (Datetime)

Computes a date by adding a number of workdays to a starting date.

Syntax

```
ADD_WORKDAYS(
   <factory_calendar_id>,
   <start_date>,
   <workdays>
   [, <source_schema>
   [, <table_name_suffix>,
   [, <client> ] ] ] )
```

Syntax Elements

<factory_calendar_id>

Specifies the ID of the factory calendar.

```
<factory_calendar_id> ::= <string_literal>
```

<start_date>

Specifies the start date that the work days will be added to. You can use either a DATE type or a date format string (for example '20140101' or '2014-01-01') for this parameter.

```
<start_date> ::= <string_literal> | <date>
```

<workdays>

Specifies the number of working days to be added to the starting date.

```
<workdays> ::= <signed_integer>
```

When *<workdays>* is positive, the resulting calculated date is the next working day after the period defined by the number of *<workdays>*. When *<workdays>* is negative, the resulting calculated date is the previous working day before the period defined by the number of *<workdays>*.

<source_schema>

Specifies the schema containing the Factory Calendar table or the Factory And Holiday Calendar tables.

```
<source_schema> ::= <string_literal>
```

This parameter can be omitted if the schema is the same as the current schema.

<table_name_suffix>

Specifies the suffix appended to the standard Factory Calendar table name or the standard Factory And Holiday Calendar table names if using an alternative set of tables. For example, with suffix '_XYZ' the function accesses the alternative tables TFACS_XYZ or FHC_C_FCAL_XYZ instead of standard tables TFACS or FHC_C_FCAL. If omitted or empty, the standard tables are used.

```
<table_name_suffix> ::= <string_literal>
```

<client>

Specifies the client used for the client-dependent Factory And Holiday Calendar implementation. If omitted or empty, then the function tries to retrieve the client from the context (user parameter), which can be set in the context by ALTER USER *<user_name>* SET PARAMETER CLIENT = *<client>*. The client is ignored if the client-independent Factory Calendar implementation is active.

```
<client> ::= <string_literal>
```

Return Type

DATE

Description

There are two different implementations for workday calculation in ABAP based systems like SAP S/4HANA:

- The Factory Calendar that stores its data in a single table TFACS.
- The Factory And Holiday Calendar that stores its data in multiple tables (FHC*).

By default, the ADD_WORKDAYS function assumes that the Factory Calendar is active. Once table FHC_CONFIG (or FHC_CONFIG *<table_suffix>* where *<table_suffix>* is specified) is available in the system and contains an entry with KEY_FIELD = 'MIGRATION_COMPLETED' and VALUE = 'X' for the specified client, the ADD_WORKDAYS function assumes that the newer Factory And Holiday Calendar is active.

The respective tables must be available in the SAP HANA database to use the ADD_WORKDAYS function. In SAP BW, SAP CRM, and SAP ERP running on an SAP HANA database, these tables are located in the ABAP schema SAP*<SID>*. For other SAP HANA database systems, these tables can be replicated from an SAP Business Suite system.

When the Factory Calendar is active, ADD_WORKDAYS accesses the table TFACS (or TFACS*<table_suffix>* if *<table_suffix>* is specified). When the Factory And Holiday Calendar is active, ADD_WORKDAYS accesses the following tables, which must be available in the system with *<table_suffix>* appended to their respective names if *<table_suffix>* is specified:

- FHC_C_FCAL
- FHC_C_FCAL_EXC
- FHC_C_HCAL
- FHC_C_HCAL_ASSGN
- FHC_C_HOL
- FHC_C_HOL_EASTER

- FHC_C_HOL_FIXED
- FHC_C_HOL_FLOAT
- FHC_C_HOL_FLDATE
- FHC_C_HOL_WKDAY
- FHC_CONFIG

Examples

The following examples use the Factory Calendar table for the month of January 2014 exclusively. For examples using the Factory And Holiday Calendar, see Factory Calendar.

TFACS bitfield	Day of the month	Reason for not working
0	1	Public Holiday
1	2	
1	3	
0	4	Weekend
0	5	Weekend
0	6	Public Holiday
1	7	
1	8	
1	9	
1	10	
0	11	Weekend
0	12	Weekend
1	13	
1	14	
1	15	
1	16	
1	17	
0	18	Weekend
0	19	Weekend
1	20	
1	21	
1	22	
1	23	
1	24	
0	25	Weekend

TFACS bitfield	Day of the month	Reason for not working
0	26	Weekend
1	27	
1	28	
1	29	
1	30	
1	31	

The following example returns the value 10.01.2014. From this result you can see that a single workday was added to the start date of the ninth, with the result being the tenth.

```
SELECT ADD_WORKDAYS('01', '2014-01-09', 1, 'FCTEST') "result date" FROM DUMMY;
```

The following example returns the value 09.01.2014. From this result you can see that the tenth was considered to be a working day, producing a final result of the ninth.

```
SELECT ADD_WORKDAYS('01', '2014-01-10', -1, 'FCTEST') "result date" FROM DUMMY;
```

The following example takes positive workday input and returns the value 20.01.2014, a result after a non-working weekend period. From this result you can see that a single workday was added to the start date of the 17th. This produced the resulting day of the 20th, which allowed for the non-working period of the weekend.

```
SELECT ADD_WORKDAYS('01', '2014-01-17', 1, 'FCTEST') "result date" FROM DUMMY;
```

The following example takes negative workday input and returns the value 17.01.2014, showing a result after a non-working weekend period. From this result you can see that the 20th was considered to be the working day, producing a result of the 17th. This result takes into account the non-working time of the weekend.

```
SELECT ADD_WORKDAYS('01', '2014-01-20', -1, 'FCTEST') "result date" FROM DUMMY;
```

The following complex example returns the value 27.01.2014. The statement above adds 16 working days to the start date of the first. The system takes into account the weekends (4th, 5th, 11th, 12th, 18th and 19th) and public holidays (1st and 6th) during the working period, which gives the last working day as the 24th. The system then returns the next possible working day after this, which is the 27th.

```
SELECT ADD_WORKDAYS('01', '2014-01-01', 16, 'FCTEST') "result date" FROM DUMMY;
```

The following example uses a table for input and returns the values 4.02.2014, 14.05.2014, 05.08.2014, and 30.10.2014.

```
CREATE SCHEMA SAPABC;
 SET SCHEMA "SAPABC";
 CREATE ROW TABLE MY_DATES (FCID NVARCHAR(2), STARTDATE DATE, DURATION INTEGER);
 INSERT INTO MY_DATES VALUES ('01', '2014-01-01', 30);
 INSERT INTO MY_DATES VALUES ('01', '2014-04-01', 28);
 INSERT INTO MY_DATES VALUES ('01', '2014-07-01', 25);
 INSERT INTO MY_DATES VALUES ('01', '2014-10-01', 20);
```

```
SELECT ADD_WORKDAYS(FCID, STARTDATE, DURATION) "shipment date" FROM MY_DATES;
```

Related Information

Datetime Data Types
WORKDAYS_BETWEEN Function (Datetime)
ALTER USER Statement (Access Control)

ADD_YEARS Function (Datetime)

Computes the specified date plus the specified years.

Syntax

```
ADD_YEARS(<date>, <num_years>)
```

Syntax Elements

<date>

The date that is incremented.

<num_years>

The number of years by which to increment the date.

Description

Computes the specified date plus the specified number of years.

Example

The following example increments the specified date value `2009-12-05` by `1` year, and returns the value `2010-12-05`:

```
SELECT ADD_YEARS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add years" FROM DUMMY;
```

Related Information

Datetime Data Types

ASCII Function (String)

Returns the integer ASCII value of the first character in a specified string.

Syntax

```
ASCII(<string>)
```

Syntax Elements

<string>

Specifies the string to return the integer ASCII value from.

Description

Returns the integer ASCII value of the first character in a string *<string>*.

Example

This example converts the first character of the string `Ant` into a numeric ASCII value and returns the value `65`:

```
SELECT ASCII('Ant') "ascii" FROM DUMMY;
```

Related Information

Character String Data Types

ASIN Function (Numeric)

Returns the arc-sine, in radians, of a numeric argument.

Syntax

```
ASIN(<number>)
```

Syntax Elements

<number>

Specifies a numeric argument between -1 and 1.

Description

Returns the arc-sine, in radians, of the numeric argument *<number>* between -1 and 1.

Example

The following example returns the value `0.5235987755982989` for `"asin"`:

```
SELECT ASIN (0.5) "asin" FROM DUMMY;
```

Related Information

Numeric Data Types

ATAN Function (Numeric)

Returns the arc-tangent, in radians, of a numeric argument.

Syntax

```
ATAN(<number>)
```

Syntax Elements

<number>

Specifies a numeric argument.

Description

Returns the arc-tangent, in radians, of the numeric argument *<number>*. The range of *<number>* is unlimited.

Example

The following example returns the value `0.4636476090008061` for `"atan"`:

```
SELECT ATAN (0.5) "atan" FROM DUMMY;
```

Related Information

Numeric Data Types

ATAN2 Function (Numeric)

Returns the arc-tangent, in radians, of the ratio of two numbers.

Syntax

```
ATAN2(<number1>, <number2>)
```

Syntax Elements

<number1>

Specifies the first numeric argument.

<number2>

Specifies the second numeric argument.

Description

Returns the arc-tangent, in radians, of the ratio of two numbers *<number1>* and *<number2>*.

Example

The following example returns the value `0.4636476090008061` for `"atan2"`:

```
SELECT ATAN2 (1.0, 2.0) "atan2" FROM DUMMY;
```

Related Information

Numeric Data Types

AUTO_CORR Function (Aggregate)

Computes all autocorrelation coefficients for a given input expression and returns an array of values.

Syntax

```
AUTO_CORR( <expression>, <maxTimeLag> { <series_order_by_clause> | <order_by_clause> })
```

Syntax Elements

<expression>

Specifies the input expression. *<expression>* values can be of any numeric type.

<maxTimeLag>

The time frame size is limited by the maxTimeLag parameter. This parameter must be a positive integer. The result size is the minimum of maxTimeLag and column size - 2 for dense series data.

<series_orderby>

<series_orderby> can only be used with an equidistant series.

```
<series_orderby> ::= SERIES( <series_period> <series_equidistant_definition> )
```

The use of SQLScript variables in this clause is not supported. The series * must not contain missing elements. For more information about this clause, see the SERIES_GENERATE function.

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

Description

Computes all autocorrelation coefficients for a given input expression and returns an array of values.

Pairs that contain at least one null are removed. Even though AUTO_CORR can handle null input values, it is highly recommended to replace null values first (e.g. by using LINEAR_APPROX), which allows for much faster processing.

The output is empty if there are fewer than two rows.

Examples

The example below shows autocorrelation of dense series data and returns `[0.285714,-0.351351,-0.5625,-0.25,1,1,1,1]`.

```
CREATE COLUMN TABLE correlationTable (TS_ID NVARCHAR(10), DATE DAYDATE, VALUE DOUBLE);
INSERT INTO correlationTable VALUES ('A', '2014-10-01', 1);
INSERT INTO correlationTable VALUES ('A', '2014-10-02', 2);
INSERT INTO correlationTable VALUES ('A', '2014-10-03', 3);
INSERT INTO correlationTable VALUES ('A', '2014-10-04', 4);
INSERT INTO correlationTable VALUES ('A', '2014-10-05', 5);
INSERT INTO correlationTable VALUES ('A', '2014-10-06', 1);
INSERT INTO correlationTable VALUES ('A', '2014-10-07', 2);
INSERT INTO correlationTable VALUES ('A', '2014-10-08', 3);
INSERT INTO correlationTable VALUES ('A', '2014-10-09', 4);
INSERT INTO correlationTable VALUES ('A', '2014-10-10', 5);

SELECT TS_ID, AUTO_CORR(VALUE, 8 SERIES (PERIOD FOR SERIES(DATE)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY MISSING ELEMENTS NOT ALLOWED))
    FROM correlationTable
    GROUP BY TS_ID ORDER BY TS_ID;
```

The example below shows autocorrelation of sparse series data without considering missing entries, and returns `[1,1,1,1,1]`.

```
CREATE COLUMN TABLE correlationTable (ts_id NVARCHAR(20), date DAYDATE, val DOUBLE);
INSERT INTO correlationTable VALUES ('A', '2014-10-01', 1);
INSERT INTO correlationTable VALUES ('A', '2014-10-02', 2);
INSERT INTO correlationTable VALUES ('A', '2014-10-04', 3);
INSERT INTO correlationTable VALUES ('A', '2014-10-07', 4);
INSERT INTO correlationTable VALUES ('A', '2014-10-11', 5);
INSERT INTO correlationTable VALUES ('A', '2014-10-21', 6);
INSERT INTO correlationTable VALUES ('A', '2014-10-22', 7);

SELECT ts_id, AUTO_CORR(val, 999 SERIES (PERIOD FOR SERIES(DATE)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY MISSING ELEMENTS NOT ALLOWED))
    FROM correlationTable
    GROUP BY ts_id ORDER BY ts_id;
```

The correlationTable has missing entries, such as '2014-10-03', but the WITHIN GROUP clause considers the series to be equidistant with one day intervals, where missing elements are not allowed.

Since the series data is assumed to be dense, the autocorrelation of the data set [1..7] is calculated.

The example below shows autocorrelation of sparse series data considering the missing entries, and returns `[1.0, null, 1.0, null, null, null, null, null, null, * 1.0, null, null, null, null, null, null, null, null, null, 1.0]`
.

```
CREATE COLUMN TABLE correlationTable (ts_id NVARCHAR(20), date DAYDATE, val DOUBLE);
INSERT INTO correlationTable VALUES ('A', '2014-10-01', 1);
INSERT INTO correlationTable VALUES ('A', '2014-10-02', 2);
INSERT INTO correlationTable VALUES ('A', '2014-10-04', 3);
INSERT INTO correlationTable VALUES ('A', '2014-10-07', 4);
INSERT INTO correlationTable VALUES ('A', '2014-10-11', 5);
INSERT INTO correlationTable VALUES ('A', '2014-10-21', 6);
INSERT INTO correlationTable VALUES ('A', '2014-10-22', 7);

SELECT ts_id, AUTO_CORR(val, 999 SERIES (PERIOD FOR SERIES(DATE)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY MISSING ELEMENTS ALLOWED))
    FROM correlationTable
    GROUP BY ts_id ORDER BY ts_id;
```

Autocorrelation works as if there were nulls instead of the missing elements in the column.

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
CORR Function (Aggregate)
CORR_SPEARMAN Function (Aggregate)
CROSS_CORR Function (Aggregate)
Expressions

AVG Function (Aggregate)

Returns the arithmetical mean of the expression. This function can also be used as a window function.

Syntax

Aggregate function:

```
AVG( [ ALL | DISTINCT ] <expression> )
```

Window function:

```
AVG( <expression> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the input data for the function.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Result type based on input

TINYINT	SMALLINT	INTEGER	BIGINT	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE
DECIMAL(9,6)	DECIMAL(11,6)	DECIMAL(16,6)	DECIMAL(25,6)	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE

Example

The following statements create a table with data for example purposes:

```
DROP TABLE "MyProducts";
CREATE COLUMN TABLE "MyProducts"(
```

```
    "Product_ID" NVARCHAR(10),
    "Product_Name" NVARCHAR(100),
    "Category" NVARCHAR(100),
    "Quantity" INTEGER,
    "Price" DECIMAL(10,2),
    PRIMARY KEY ("Product_ID") );

  INSERT INTO "MyProducts" VALUES('P1','Shirts', 'Clothes', 32, 20.99);
  INSERT INTO "MyProducts" VALUES('P2','Jackets', 'Clothes', 16, 99.49);
  INSERT INTO "MyProducts" VALUES('P3','Trousers', 'Clothes', 30, 32.99);
  INSERT INTO "MyProducts" VALUES('P4','Coats', 'Clothes', 5, 129.99);
  INSERT INTO "MyProducts" VALUES('P5','Purse', 'Accessories', 3, 89.49);
```

The following example averages the prices of all products in the MyProducts table and returns the value `74.50`.:

```
SELECT AVG("Price") FROM "MyProducts";
```

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
Expressions

BINNING Function (Window)

Partitions an input set into disjoint subsets by assigning a bin number to each row.

Syntax

```
BINNING( <binning_param> => <expression> [ {, <binning_parameter> => <expression> } ... ] ) <window

<binning_param> ::= VALUE | BIN_COUNT | BIN_WIDTH | TILE_COUNT | STDDEV_COUNT
```

Syntax Elements

<binning_param>

VALUE is always required. It specifies the column that binning is applied to. When BIN_WIDTH is used, the input column must have a numeric data type. BIN_COUNT specifies the number of equal-width bins. BIN_WIDTH specifies the width of the bins. TILE_COUNT specifies the number of bins with equal number of records. STDDEV_COUNT specifies the number of standard deviations left and right from the mean.

The appropriate binning method is selected based on the parameter specified – exactly one of the last four parameters must be non-NULL. The value assigned to binning method parameter must be an integer expression.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

The OVER clause must not specify a *<window_order_by_clause>* nor any window frame since the binning function works on an entire partition.

Examples

The following example distributes the value of the input set into four bins that all have an equal width.

```
DROP TABLE weather;
CREATE ROW TABLE weather (station INT, ts DATE, temperature FLOAT);
INSERT INTO weather VALUES(1, '2014-01-01', 0);
INSERT INTO weather VALUES(1, '2014-01-02', 3);
INSERT INTO weather VALUES(1, '2014-01-03', 4.5);
INSERT INTO weather VALUES(1, '2014-01-04', 6);
INSERT INTO weather VALUES(1, '2014-01-05', 6.3);
INSERT INTO weather VALUES(1, '2014-01-06', 5.9);
INSERT INTO weather VALUES(1, '2015-01-01', 1);
INSERT INTO weather VALUES(1, '2015-01-02', 3.4);
INSERT INTO weather VALUES(1, '2015-01-03', 5);
INSERT INTO weather VALUES(1, '2015-01-04', 6.7);
INSERT INTO weather VALUES(1, '2015-01-05', 4.6);
INSERT INTO weather VALUES(1, '2015-01-06', 6.9);

SELECT *, BINNING(VALUE => temperature, BIN_COUNT => 4) OVER () AS bin_num FROM weather;
```

STATION	TS	TEMPERATURE	BIN_NUM
1	01.01.2014	0.0	1
1	02.01.2014	3.0	2
1	03.01.2014	4.5	3
1	04.01.2014	6.0	4
1	05.01.2014	6.3	4
1	06.01.2014	5.9	4
1	01.01.2014	1.0	1
1	02.01.2014	3.4	2
1	03.01.2014	5.0	3
1	04.01.2014	6.7	4
1	05.01.2014	4.6	3
1	06.01.2014	6.9	4

Related Information

Expressions

Window Functions and the Window Specification

BINTOHEX Function (String)

Converts a binary string to an NVARCHAR hexadecimal value.

Syntax

```
BINTOHEX(<expression>)
```

Syntax Elements

<expression>

The value to be converted to a hexadecimal value.

Description

Converts a binary value to a hexadecimal value as an NVARCHAR data type. If the input value is not a binary value, then it is first converted to a binary value.

Example

The following example converts the binary value AB to a hexadecimal NVARCHAR value 4142:

```
SELECT BINTOHEX('AB') "bintohex" FROM DUMMY;
```

Related Information

Expressions
Character String Data Types

BINTONHEX Function (String)

Converts a binary value to a hexadecimal value as an NVARCHAR data type.

Syntax

```
BINTONHEX(<expression>)
```

Syntax Elements

<expression>

The value to be converted to a hexadecimal value.

Description

Converts a binary value to a hexadecimal value as an NVARCHAR data type. The input value is converted to a binary value first if it is not a binary value.

Example

The following example converts the binary value AB to a hexadecimal NVARCHAR value 4142.

```
SELECT BINTONHEX('AB') "bintonhex" FROM DUMMY;
```

Related Information

Expressions
Character String Data Types

BINTOSTR Function (String)

Converts a VARBINARY string to a character string.

Syntax

```
BINTOSTR(<varbinary_string>)
```

Syntax Elements

<varbinary_string>

The VARBINARY string to be converted to a character string.

Description

Converts a VARBINARY string *<varbinary_string>* to a character string with CESU-8 encoding.

Example

This example converts the VARBINARY string `416E74` to a CESU-8 encoded character string, and returns the value `Ant`:

```
SELECT BINTOSTR ('416E74') "bintostr" FROM DUMMY;
```

Related Information

Character String Data Types

BITAND Function (Numeric)

Performs an AND operation on the bits of two arguments.

Syntax

```
BITAND(<value1>, <value2>)
```

Syntax Elements

<value1>

The argument must be an INTEGER or a VARBINARY value.

<value2>

The argument must be an INTEGER or a VARBINARY value.

Description

Performs an AND operation on the bits of the arguments *<value1>* and *<value2>*.

The BITAND function works a bit differently from the BITOR, BITXOR, and BITNOT functions. The BITAND function converts string arguments into BIGINT, whereas the other functions convert string arguments into INT.

The BITAND function returns a result along the argument's type.

Example

The following example returns the value 123:

```
SELECT BITAND (255, 123) "bitand" FROM DUMMY;
```

Related Information

Expressions

Numeric Data Types

BITCOUNT Function (Numeric)

Counts the number of set bits of an expression.

Syntax

```
BITCOUNT(<expression>)
```

Syntax Elements

<expression>

Specifies the expression that the function counts the number of set bits of. *<expression>* must be an INTEGER or a VARBINARY value.

Description

Counts the number of set bits of the argument *<expression>*.

The BITCOUNT function returns an INTEGER value.

Example

The following example counts the bits for 255, and returns the value 8:

```
SELECT BITCOUNT (255) "bitcount" FROM DUMMY;
```

Related Information

Expressions

BITNOT Function (Numeric)

Performs a bitwise NOT operation on the bits of an expression.

Syntax

```
BITNOT(<expression>)
```

Syntax Elements

<expression>

Specifies the expression that the function performs a bitwise NOT operation on. *<expression>* must be an INTEGER value.

Description

Performs a bitwise NOT operation on the bits of the argument *<expression>*.

The BITNOT functions works a bit differently than the BITAND function. The BITAND function converts string arguments into BIGINT, whereas the BITNOT function converts string arguments into INT.

The BITNOT function returns a result along the argument's type.

Example

The following example performs a BITNOT operation on 255, and returns the value `-256` for `"bitnot"`:

```
SELECT BITNOT (255) "bitnot" FROM DUMMY;
```

Related Information

Expressions
Numeric Data Types

BITOR Function (Numeric)

This function performs an OR operation on the bits of two arguments.

Syntax

```
BITOR(<expression1>, <expression2>)
```

Syntax Elements

<expression1>

An argument for the bitwise OR operation that must be a non-negative INTEGER or VARBINARY value.

<expression2>

An argument for the bitwise OR operation that must be a non-negative INTEGER or VARBINARY value.

Description

This function performs an OR operation on the bits of the arguments *<expression1>* and *<expression2>*.

The BITOR functions works a bit differently than the BITAND function. The BITAND function converts string arguments into BIGINT, whereas the BITOR function converts string arguments into INT.

The BITOR function returns a result along the argument's type.

Example

The following example performs a bitwise OR operation for the arguments 255 and 123, and returns the value 255 for `"bitor"`:

```
SELECT BITOR (255, 123) "bitor" FROM DUMMY;
```

Related Information

Numeric Data Types

BITSET Function (Numeric)

Sets a specific number of bits to 1 in a target number from a specified 1-based index position.

Syntax

```
BITSET(<target_num>, <start_bit>, <num_to_set>)
```

Syntax Elements

<target_num>

The VARBINARY number where the bits are to be set.

```
<target_num> ::= <string_literal>
```

<start_bit>

A 1-based index position where the first bit is to be set.

```
<start_bit> ::= <unsigned_integer>
```

<num_to_set>

The number of bits to be set in the target number.

```
<num_to_set> ::= <unsigned_integer>
```

Description

Sets *<num_to_set>* bits to 1 in *<target_num>* from the *<start_bit>* position.

Example

The following example returns the value `E000` for `"bitset"`:

```
SELECT BITSET ('0000', 1, 3) "bitset" FROM DUMMY;
```

Related Information

Numeric Data Types

BITUNSET Function (Numeric)

Sets a specified number of bits to 0 in a target number from a specified 1-based index position.

Syntax

```
BITUNSET(<target_num>, <start_bit>, <num_to_unset>)
```

Syntax Elements

<target_num>

The VARBINARY number where the bits are to be unset.

```
<target_num> ::= <string_literal>
```

<start_bit>

A 1-based index position where the first bit is to be unset.

```
<start_bit> ::= <unsigned_integer>
```

<num_to_set>

The number of bits to be unset in the target number.

```
<num_to_set> ::= <unsigned_integer>
```

Description

Sets *<num_to_unset>* bits to 0 in *<target_num>* from the *<start_bit>* position.

Example

The following example returns the value `1FFF` for `"bitunset"`:

```
SELECT BITUNSET ('ffff', 1, 3) "bitunset" FROM DUMMY;
```

Related Information

Numeric Data Types

BITXOR Function (Numeric)

Performs an XOR operation on the bits of two arguments.

Syntax

```
BITXOR(<expression1>, <expression2>)
```

Syntax Elements

<expression1>

An argument that must be an INTEGER or a VARBINARY value.

<expression2>

An argument for the bitwise OR operation that must be an INTEGER or a VARBINARY value.

Description

Performs an XOR operation on the bits of the arguments *<expression1>* and *<expression2>*.

The BITXOR functions works a bit differently than the BITAND function. The BITAND function converts string arguments into BIGINT, whereas the BITXOR function converts string arguments into INT.

The BITXOR function returns a result along the argument's type.

Example

The following example performs a bitwise XOR operation for the arguments `255` and `123`, and returns the value `132` for `"bitxor"`:

```
SELECT BITXOR (255, 123) "bitxor" FROM DUMMY;
```

Related Information

Numeric Data Types

CARDINALITY Function (Array)

Returns the number of elements in a specified array.

Syntax

```
CARDINALITY(<array_value_expression>)
```

Syntax Elements

<array_value_expression>

Specifies the array that the function returns the number of elements for.

Description

Returns the number of elements in *<array_value_expression>*.

Example

The following example returns the number of elements (3 and 4, respectively) contained in two arrays.

```
CREATE COLUMN TABLE ARRAY_TEST (IDX INT, VAL INT ARRAY);
INSERT INTO ARRAY_TEST VALUES (1, ARRAY(1, 2, 3));
INSERT INTO ARRAY_TEST VALUES (2, ARRAY(10, 20, 30, 40));

SELECT CARDINALITY(VAL) "cardinality" FROM ARRAY_TEST;
```

Related Information

Expressions

CAST Function (Data Type Conversion)

Returns the value of an expression converted to a supplied data type.

Syntax

```
CAST( <expression> AS <data_type>[ ( <length> ) ] )
```

Syntax Elements

<expression>

Specifies the expression to be converted. If you use a parameter, the parameter is bound as *<data_type>*, or as a String if *<data_type>* is LOB or GEOMETRY.

<data_type>

Specifies the target data type.

```
<data_type >::=
 BOOLEAN
 |TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | DECIMAL
 | SMALLDECIMAL
 | REAL
 | DOUBLE
 | NVARCHAR
 | DAYDATE
 | DATE
 | TIME
 | SECONDDATE
```

```
| TIMESTAMP
| LOB
| BINARY
| GEOMETRY
| REAL_VECTOR
```

<length>

If the target type is a string, *<length>* specifies the maximum length of the string in characters. If the target type is a vector, *<length>* specifies the dimension of the vector.

Description

Returns the value of an expression converted to a supplied data type.

Examples

Convert the value `7` to the NVARCHAR value `7`:

```
SELECT CAST (7 AS NVARCHAR) "cast" FROM DUMMY;
```

Converts the value `10.5` to the INTEGER value `10`, truncating the mantissa.

```
SELECT CAST (10.5 AS INTEGER) "cast" FROM DUMMY;
```

Bind a parameter as BIGINT. Without the CAST function, the data type of the parameter is ambiguous.

```
CREATE COLLECTION C1;
INSERT INTO C1 VALUES({A:1,B:2});
INSERT INTO C1 VALUES({A:'ABC'});
SELECT * FROM C1 WHERE A = CAST (? AS BIGINT);
```

Related Information

Expressions

Data Type Conversion

CEIL Function (Numeric)

Returns the first integer that is greater than or equal to the specified value.

Syntax

```
CEIL(<number>)
```

Syntax Elements

<number>

Specifies the number that the function returns the integer for.

Description

Returns the smallest decimal number greater than or equal to the specified *<number>*.

Example

The following example returns the value `15.0` for `"ceiling"`:

```
SELECT CEIL (14.5) "ceiling" FROM DUMMY;
```

Related Information

Numeric Data Types

CHAR Function (String)

Returns the character that has the ASCII value of the specified number.

Syntax

```
CHAR(<number>)
```

Syntax Elements

<number>

Specifies the number with the ASCII value that the function converts into a character.

Description

Returns the character that has the ASCII value of the specified number.

Example

This example converts three ASCII values into characters and concatenates the results, returning the string `Ant`:

```
SELECT CHAR (65) || CHAR (110) || CHAR (116) "character" FROM DUMMY;
```

Related Information

Character String Data Types

COALESCE Function (Miscellaneous)

Returns the first non-NULL expression from a specified list.

Syntax

```
COALESCE(<expression_list>)
```

Syntax Elements

<expression_list>

Specifies the expressions to return the first non-NULL expression from.

Description

Returns the first non-NULL expression from a list. At least two expressions must be contained in *<expression_list>*, and all expressions must be comparable. The result is NULL if all the expressions are NULL.

Example

```
CREATE ROW TABLE coalesce_example (ID INT PRIMARY KEY, A REAL, B REAL);
 INSERT INTO coalesce_example VALUES(1, 100, 80);
 INSERT INTO coalesce_example VALUES(2, NULL, 63);
 INSERT INTO coalesce_example VALUES(3, NULL, NULL);

 SELECT id, a, b, COALESCE (a, b*1.1, 50.0) "coalesce" FROM coalesce_example;
```

The example above returns the results below:

ID	A	B	coalesce
1	100.0	80.0	100.0
2	NULL	63.0	69.30000305175781
3	NULL	NULL	50.0

Related Information

Expressions

CONCAT Function (String)

Returns a combined string consisting of two specified strings.

Syntax

```
CONCAT(<string1>, <string2>)
```

Syntax Elements

<string1>

Specifies the first string to combine.

<string2>

Specifies the second string to combine.

Description

Returns a combined string consisting of *<string1>* followed by *<string2>*. The concatenation operator (||) is identical to this function.

The maximum length of the concatenated string is 8,388,607. If the string length is longer than the maximum length, an exception is thrown. Exceptionally, an implicit truncation is performed when converting an NCLOB typed value with a size greater than the maximum length of an NVARCHAR value.

If one ore both arguments are NULL, the function returns NULL.

Example

This example concatenates the specified string arguments and returns the value `Cat`:

```
SELECT CONCAT ('C', 'at') "concat" FROM DUMMY;
```

Related Information

Character String Data Types

CONCAT_NAZ Function (String)

Returns a combined, non-null value string consisting of two specified strings.

Syntax

```
CONCAT_NAZ(<string1>, <string2>)
```

Syntax Elements

<string1>

Specifies the first string to combine.

<string2>

Specifies the second string to combine.

Description

Returns a combined value string consisting of *<string1>* followed by *<string2>*. If one value is NULL, the other value is returned. If both values are NULL, NULL is returned.

The maximum length of the concatenated string is 8,388,607. If the string length is longer than the maximum length, then an exception is thrown. Exceptionally, an implicit truncation is performed when converting an NCLOB typed value with a size greater than the maximum length of an NVARCHAR typed value.

Example

The following example returns the AB:

```
SELECT CONCAT_NAZ ('A', 'B') "concat" FROM DUMMY;
```

concat
AB

The following example returns C:

```
SELECT CONCAT_NAZ ('C', null) "concat" FROM DUMMY;
```

concat
C

The following example returns the NULL value:

```
SELECT CONCAT_NAZ (null, null) "concat" FROM DUMMY;
```

concat
NULL

Related Information

Character String Data Types

CORR Function (Aggregate)

Computes the Pearson product momentum correlation coefficient between two columns. This function can also be used as a window function.

Syntax

Aggregate function:

```
CORR(<column1>, <column2>)
```

Window function:

```
CORR(<column1>, <column2>) <window_specification>
```

Syntax Elements

<column1>* and *<column2>

Specifies the columns providing the input data for the correlation.

The values of *<column1>* and *<column2>* can be of any numeric type.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Computes the Pearson product momentum correlation coefficient between two columns.

The result ranges from -1 to 1, depending on the correlation, or NULL if a correlation cannot be computed.

The result can return NULL for one of the following reasons:

- Less than two value pairs are correlated after NULLs have been removed
- There is zero variance in at least one of the two columns

Examples

The examples below assume that a correlation table has been created with the following values:

```
CREATE COLUMN TABLE correlationTable (
    ts_id NVARCHAR(20),
    DATE DAYDATE,
    value1 DOUBLE,
    value2 DOUBLE);

INSERT INTO correlationTable VALUES ('A', '2014-10-01', 1, 1);
INSERT INTO correlationTable VALUES ('A', '2014-10-02', 2, 2);
INSERT INTO correlationTable VALUES ('A', '2014-10-04', 3, 3);
INSERT INTO correlationTable VALUES ('B', '2014-10-07', 1, 3);
INSERT INTO correlationTable VALUES ('B', '2014-10-11', 2, 2);
INSERT INTO correlationTable VALUES ('B', '2014-10-21', 3, 1);
```

1. The following aggregate function example returns the correlation between the `ts_id` column and the columns `value1` and `value2`.

   ```
   SELECT ts_id, CORR(value1, value2) FROM correlationTable GROUP BY ts_id;
   ```

   The results are as follows:

TS_ID	CORR(VALUE1, VALUE2)
A	1
B	-1

2. The following WHERE clause example returns the correlation between the `ts_id` column and the columns `value1` and `value2` only for rows where `ts_id` equals A.

   ```
   SELECT ts_id, CORR(value1, value2) FROM correlationTable
       WHERE ts_id='A' GROUP BY ts_id;
   ```

   The results are as follows:

TS_ID	CORR(VALUE1, VALUE2)
A	1

3. The example below uses a window function.

   ```
   SELECT ts_id, CORR(value1, value2) OVER (PARTITION BY ts_id) FROM correlationTable;
   ```

The results are as follows:

TS_ID	CORR(VALUE1, VALUE2)OVER(PARTITIONBYTS_ID)
A	1

4. The example below uses a sliding window function.

```
SELECT ts_id, CORR(value1, value2) OVER (PARTITION BY ts_id ORDER BY date)
    FROM correlationTable ORDER BY ts_id;
```

The results are as follows:

TS_ID	CORR(VALUE1, VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATE)
A	?
A	0.9999999999999998
A	0.9999999999999998
B	?
B	-0.9999999999999998
B	-0.9999999999999998

5. The example below uses a ROWS BETWEEN clause.

```
SELECT ts_id, CORR(value1, value2) OVER (PARTITION BY ts_id ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) from correlationTable;
```

The results are as follows:

TS_ID	CORR(VALUE1, VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATEROWS)
A	?
A	0.9999999999999998
A	0.9999999999999998
B	?
B	-0.9999999999999998
B	-0.9999999999999998

Related Information

Window Functions and the Window Specification

Window Aggregate Functions

Aggregate Functions

AUTO_CORR Function (Aggregate)

CORR_SPEARMAN Function (Aggregate)

CORR_SPEARMAN Function (Aggregate)

Returns the Spearman's rank correlation coefficient of the values found in the corresponding rows of two columns. This function can also be used as a window function.

Syntax

Aggregate function:

```
CORR_SPEARMAN(<column1>, <column2>)
```

Window function:

```
CORR_SPEARMAN(<column1>, <column2>) <window_specification>
```

Syntax Elements

<column1>* and *<column2>

Specifies the columns providing the input data for the correlation.

The values of *<column1>* and *<column2>* can contain number or character types.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Returns the Spearman's rank correlation coefficient of the values found in the corresponding rows of *<column1>* and *<column2>*.

Although this function is grouped with the aggregate functions, its optional OVER clause positions it as a windows function as well.

The result ranges from -1 to 1, depending on the correlation, or NULL if a correlation could not be computed.

The result can return NULL for one of the following reasons:

- Less than two value pairs are correlated after NULLs have been removed
- There is zero variance in at least one of the two columns

Whenever a NULL value is found then both the NULL value and the corresponding value of the other input column are ignored.

Examples

The example below returns `-1`.

```
CREATE COLUMN TABLE A (date DAYDATE, val INT);
INSERT INTO A VALUES ('2014-10-01', 100);
```

```
INSERT INTO A VALUES ('2014-10-02', 200);
INSERT INTO A VALUES ('2014-10-03', 300);

CREATE COLUMN TABLE B (date DAYDATE, val INT);
INSERT INTO B VALUES ('2014-10-01', 300);
INSERT INTO B VALUES ('2014-10-02', 200);
INSERT INTO B VALUES ('2014-10-03', 100);

SELECT CORR_SPEARMAN(A.val, B.val) "corr" FROM A, B WHERE A.date = B.date;
```

The examples below assume that the correlation table has been created with the following values:

```
CREATE COLUMN TABLE correlationSpearmanTable (
    ts_id NVARCHAR(20),
    date DAYDATE,
    value1 DOUBLE,
    value2 DOUBLE);

INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-01', 34.345, 45.345);
INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-02', 27.145, 28.893);
INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-02', 48.312, 28.865);
INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-03', 94.213, 58.854);
INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-03', 16.567, 28.231);
INSERT INTO correlationSpearmanTable VALUES ('A', '2014-10-03', 38.894, 94.378);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-04', 45.643, 76.987);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-04', 53.345, 50.893);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-04', 66.342, 48.342);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-04', 76.432, 37.234);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-05', 88.432, 23.242);
INSERT INTO correlationSpearmanTable VALUES ('B', '2014-10-05', 93.234, 13.132);
```

1. Execute the aggregate function example below:

   ```
   SELECT ts_id, CORR_spearman(value1, value2) FROM correlationSpearmanTable GROUP BY ts_id;
   ```

   The results are as follows:

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)
A	0.542857
B	-1

2. Execute the window function below:

   ```
   SELECT ts_id, CORR_spearman(value1, value2) OVER (PARTITION BY ts_id) FROM correlationSpearma
   ```

   The results are as follows:

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_ID)
A	0.542857
A	0.542857
A	0.542857
A	0.542857
A	0.542857
A	0.542857

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_ID)
B	-1
B	-1
B	-1
B	-1
B	-1
B	-1

3. Execute the sliding window example below:

```
SELECT ts_id, CORR_spearman(value1, value2) OVER (PARTITION BY ts_id ORDER BY date) FROM corr
```

The results are as follows:

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATE)
A	?
A	-0.5
A	-0.5
A	0.5428571428571428
A	0.5428571428571428
A	0.5428571428571428
B	-1
B	-1
B	-1
B	-1
B	-1
B	-1

4. Execute the ROWS BETWEEN example below:

```
SELECT ts_id, CORR_spearman(value1, value2) OVER (PARTITION BY ts_id ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) from correlationSpearmanTable;
```

The results are as follows:

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATEROWS)
A	?
A	1
A	-0.5
A	0.4

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATEROWS)
A	0.7
A	0.5428571428571428
B	?
B	-1
B	-1
B	-1
B	-1
B	-1

5. Execute the group example below:

```
SELECT ts_id, CORR_spearman(value1, value2) OVER (PARTITION BY ts_id ORDER BY date GROUPS BET\
```

The results are as follows:

TS_ID	CORR_SPEARMAN(VALUE1,VALUE2)OVER(PARTITIONBYTS_IDORDERBYDATEGROUPS)
A	?
A	-0.5
A	-0.5
A	0.5428571428571428
A	0.5428571428571428
A	0.5428571428571428
B	-1
B	-1
B	-1
B	-1
B	-1
B	-1

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
AUTO_CORR Function (Aggregate)
CORR Function (Aggregate)
CROSS_CORR Function (Aggregate)

COS Function (Numeric)

Returns the cosine of the angle, in radians, for the specified argument.

Syntax

```
COS(<number>)
```

Description

Returns the cosine of the angle *<number>*, in radians.

Example

The following example returns the value `1.0` for `"cos"`:

```
SELECT COS (0.0) "cos" FROM DUMMY;
```

Related Information

Numeric Data Types

COSH Function (Numeric)

Computes the hyperbolic cosine of the specified argument.

Syntax

```
COSH(<number>)
```

Description

Computes the hyperbolic cosine of the numeric argument *<number>*.

Example

The following example returns the value `1.1276259652063807` for `"cosh"`:

```
SELECT COSH (0.5) "cosh" FROM DUMMY;
```

Related Information

Numeric Data Types

COT Function (Numeric)

Computes the cotangent of a specified number.

Syntax

```
COT(<number>)
```

Description

Computes the cotangent of a number *<number>*, where *<number>* is an angle expressed in radians.

Example

The following example returns the value `-0.8950829176379128` for `"cot"`:

```
SELECT COT (40) "cot" FROM DUMMY;
```

Related Information

Numeric Data Types

COUNT Function (Aggregate)

Counts the number of rows returned by a query. This function can also be used as a window function.

Syntax

Aggregate function:

```
COUNT (*)
  | COUNT ( [ ALL ] <expression> )
  | COUNT ( DISTINCT <expression_list>
```

Window function:

```
COUNT (*) [ <window_specification> ]
  | COUNT ( [ ALL ] <expression> ) [ <window_specification> ]
  | COUNT ( DISTINCT <expression> ) [ <window_specification> ]
```

Syntax Elements

*

Returns the number of rows returned by the query, regardless of the value of those rows, and including duplicate values.

ALL

Optional keyword reflecting the default behavior. That is, `COUNT(ALL <expression>)` and `COUNT(<expression>)` return the same result.

<expression>

Specifies the input data for the function. The function returns the number of non-NULL values for the specified expression returned by the query, including duplicate values (unless DISTINCT is also specified).

<expression_list>

Specifies the input data for the function as a comma-separated list of *<expression>*. *<expression_list>* is only supported for use with the DISTINCT clause.

```
<expression_list>::= <expression> [, <expression> [,…] ]
```

DISTINCT

Returns the number of distinct values returned by the query, excluding rows with all NULL values for that expression.

<window_specification>

Defines a window on the data over which the function operates. For syntax, see Window Functions and the Window Specification.

Description

Result type based on input

TINYINT	SMALLINT	INTEGER	BIGINT	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE
BIGINT	BIGINT	BIGINT	BIGINT	BIGINT	BIGINT	BIGINT	BIGINT

Example

The following statements create a table with data for example purposes:

```
DROP TABLE "MyProducts";
CREATE COLUMN TABLE "MyProducts"(
  "Product_ID" NVARCHAR(10),
  "Product_Name" NVARCHAR(100),
  "Category" NVARCHAR(100),
  "Quantity" INTEGER,
  "Price" DECIMAL(10,2),
  PRIMARY KEY ("Product_ID") );

INSERT INTO "MyProducts" VALUES('P1','Shirts', 'Clothes', 32, 20.99);
INSERT INTO "MyProducts" VALUES('P2','Jackets', 'Clothes', 16, 99.49);
INSERT INTO "MyProducts" VALUES('P3','Trousers', 'Clothes', 30, 32.99);
INSERT INTO "MyProducts" VALUES('P4','Coats', 'Clothes', 5, 129.99);
INSERT INTO "MyProducts" VALUES('P5','Purse', 'Accessories', 3, 89.49);
```

The following example returns 5, the number of rows in the MyProducts table:

```
SELECT COUNT(*) FROM "MyProducts";
```

The following example returns 2, the number of rows with distinct values in the Category column:

```
SELECT COUNT(DISTINCT "Category") FROM "MyProducts";
```

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions

CROSS_CORR Function (Aggregate)

Computes all cross-correlation coefficients between two given expressions.

Syntax

```
CROSS_CORR(<expression1>, <expression2>, <maxLag>
    { <series_orderby> | <order_by_clause> } ).{ POSITIVE_LAGS | NEGATIVE_LAGS | ZERO_LAG }
```

Syntax Elements

<expression1>* and *<expression2>

Numeric values between which the cross-correlation is calculated.

<maxLag>

The *<maxLag>* parameter must be a positive integer that defines the number of cross-correlation coefficients to be returned.

```
<maxLag> ::= INTEGER
```

<series_orderby>

The SERIES clause can only be used with an equidistant series. For more information about the SERIES clause, see the CREATE TABLE statement and the SERIES_GENERATE function.

```
<series_orderby> ::= SERIES (<series_period> <series_equidistant_definition>)
```

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

Description

Computes all cross-correlation coefficients between two given expressions.

The result is an array of cross-correlation coefficients of length *<maxLag>*.

If POSITIVE_LAGS is specified, then the cross-correlation coefficients with lags 1*<maxLag>* are returned.

If NEGATIVE_LAGS is specified, then the cross-correlation coefficients with lags -1 *<maxLag>* are returned.

If ZERO_LAG is specified, a single value associated with lag 0 is returned.

This function output can be non-deterministic among tie values.

Example

Example 1 - Cross correlation

Execute the cross correlation example below:

```
CREATE COLUMN TABLE table1 ( ts_id INTEGER, number1 DOUBLE, number2 DOUBLE );

INSERT INTO table1 VALUES ('1', 1, 2);
INSERT INTO table1 VALUES ('2', 2 ,1);
INSERT INTO table1 VALUES ('3', 1 ,2);

SELECT CROSS_CORR(number1, number2, 10 ORDER BY ts_id) FROM table1;
```

The results are as follows:

CROSS_CORR(NUMBER1,NUMBER2,10ORDERBYTS_ID)
1.0, -1.0, 1.0

Example 2 - Cross correlation using a series descriptor

Execute the example below:

```
CREATE COLUMN TABLE TSeries( key INTEGER, ts TIMESTAMP, val1 DOUBLE, val2 DOUBLE, PRIMARY KEY(key,
    SERIES( SERIES KEY (key) EQUIDISTANT INCREMENT BY INTERVAL 1 DAY PERIOD FOR SERIES(ts) );

INSERT INTO TSeries VALUES (1, '2014-1-1', 1, 3);
INSERT INTO TSeries VALUES (2, '2014-1-3', 2, 4);
INSERT INTO TSeries VALUES (3, '2014-1-4', 4, 2);
INSERT INTO TSeries VALUES (4, '2014-1-5', 3, 1);

SELECT CROSS_CORR(val1, val2, 10 ORDER BY ts) FROM TSeries;
```

The results are as follows:

CROSS_CORR(VAL1,VAL2,10ORDERBYTS)
-1.0, -0.928571, -0.6, 0.5, -1.0

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
AUTO_CORR Function (Aggregate)
CORR Function (Aggregate)
CORR_SPEARMAN Function (Aggregate)
Expressions
CREATE TABLE Statement (Data Definition)

CUBIC_SPLINE_APPROX Function (Window)

Replaces null values by interpolating the gaps based on calculated cubic splines and linearly extrapolating any leading or trailing null values.

Syntax

```
CUBIC_SPLINE_APPROX ( <expression> [, <BoundaryConditionArgument> [, <ExtrapolationModeArgument> [,
 [ OVER ( {
     SERIES(...) [ PARTITION BY <col1> [,...] [ <window_order_by_clause> ]
   | [ PARTITION BY <col1>[,...] <window_order_by_clause>
     } ) ]
```

Syntax Elements

<expression>

Specifies the input data expression

<BoundaryConditionArgument>

Specifies the boundary conditions that are used for the cubic spline calculation. The default value is SPLINE_TYPE_NATURAL. If this parameter is omitted, then natural boundary conditions are used to calculate the interpolating cubic splines.

```
<BoundaryConditionArgument> ::= SPLINE_TYPE_NATURAL | SPLINE_TYPE_NOT_A_KNOT
```

<ExtrapolationModeArgument>

Defines the applied extrapolation mode. The default value is EXTRAPOLATION_NONE, which indicates that extrapolation does not occur if this parameter is omitted.

```
<ExtrapolationModeArgument> ::= EXTRAPOLATION_NONE
 | EXTRAPOLATION_LINEAR
 | EXTRAPOLATION_CONSTANT
```

The definition, default value, and use of ExtrapolationModeArgument in conjunction with the Value1Argument and Value2Argument is identical to the LINEAR_APPROX function except from the calculation of the slope.

When using EXTRAPOLATION_LINEAR mode, the slope of the extrapolation line is set equal to the slope of the calculated cubic splines at the boundary points. For example, the slope of the calculated spline at the first non-null value is taken as the slope of the extrapolating line for leading nulls. Similarly, the slope of the calculated spline at the last non-null value is taken as the slope of the extrapolating line for trailing nulls.

Cubic spline interpolation can be applied to all number types that can be cast into long double type, such as INT, LONG, DOUBLE, or FLOAT.

<window_order_by_clause>

Determines the sort order. This expression must not contain any NULL values or duplicates. See the syntax of this clause located in the *Window Functions* topic.

Description

Replaces null values by interpolating the gaps based on calculated cubic splines and linearly extrapolating any leading or trailing null values.

Examples

Natural cubic spline interpolation

Perform natural cubic spline interpolation:

```
SELECT CUBIC_SPLINE_APPROX(temperature, 'SPLINE_TYPE_NATURAL') OVER (PARTITION BY station ORDER BY
  FROM WEATHER;
```

For natural boundary conditions, the BoundaryConditionArgument can be omitted. The following query returns identical results to the above one:

```
SELECT CUBIC_SPLINE_APPROX(temperature)
    OVER(PARTITION BY station ORDER BY ts) FROM WEATHER;
```

Perform cubic spline interpolation by using not-a-knot boundary conditions:

```
SELECT CUBIC_SPLINE_APPROX(temperature, 'SPLINE_TYPE_NOT_A_KNOT')
    OVER(PARTITION BY station ORDER BY ts) FROM WEATHER;
```

This example performs linear extrapolation on a series.

```
SELECT date, val, LINEAR_APPROX(val, 'EXTRAPOLATION_LINEAR') OVER(
    SERIES(SERIES KEY(ts_id) PERIOD FOR SERIES(date)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY MISSING ELEMENTS ALLOWED))
    FROM InterpolationTable;
```

Related Information

Expressions

Window Functions and the Window Specification

CUME_DIST Function (Window)

Returns the relative rank of a row.

Syntax

```
CUME_DIST() <window_specification>
```

Syntax Elements

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Returns the relative rank of a row.

The relative rank of a row R is defined as NP/NR, where:

- NP is defined to be the number of rows preceding or peer with R in the window ordering of the window partition of R.
- NR is defined to be the number of rows in the window ordering of the window.

Examples

```
CREATE ROW TABLE ProductSales (ProdName NVARCHAR(50), Description NVARCHAR(20), Sales INT);
INSERT INTO ProductSales VALUES('Tee Shirt','Plain',21);
INSERT INTO ProductSales VALUES ('Tee Shirt','Lettered',22);
INSERT INTO ProductSales VALUES ('Tee Shirt','Team logo',30);
INSERT INTO ProductSales VALUES('Hoodie','Plain',60);
INSERT INTO ProductSales VALUES ('Hoodie','Lettered',65);
INSERT INTO ProductSales VALUES ('Hoodie','Team logo',80);
INSERT INTO ProductSales VALUES('Ballcap','Plain',8);
INSERT INTO ProductSales VALUES ('Ballcap','Lettered',40);
INSERT INTO ProductSales VALUES ('Ballcap','Team logo',27);

SELECT ProdName, Description, Sales,
  PERCENT_RANK() OVER (ORDER BY Sales ASC) AS Percent_Rank,
  CUME_DIST() OVER (ORDER BY Sales ASC) AS Cume_Dist
FROM ProductSales
ORDER BY Sales DESC;
```

PRODNAME	DESCRIPTION	SALES	PERCENT_RANK	CUME_DIST
Hoodie	Team logo	80	1	1
Hoodie	Lettered	65	0.875	0.8888888888888888
Hoodie	Plain	60	0.75	0.7777777777777778
Ballcap	Lettered	40	0.625	0.6666666666666666
Tee Shirt	Team logo	30	0.5	0.5555555555555556
Ballcap	Team logo	27	0.375	0.4444444444444444
Tee Shirt	Lettered	22	0.25	0.3333333333333333
Tee Shirt	Plain	21	0.125	0.2222222222222222
Ballcap	Plain	8	0	0.1111111111111111

Related Information

Expressions

Window Functions and the Window Specification

CURRENT_DATE Function (Datetime)

Returns the current local system date.

Syntax

```
CURRENT_DATE
```

Description

Returns the current local system date.

It is recommended that you use UTC dates instead of local dates. See to the CURRENT_UTCDATE function for more information.

Example

The following example returns the local and UTC date of the local system:

```
SELECT CURRENT_DATE "Current Date", CURRENT_UTCDATE "Coordinated Universal Date" FROM DUMMY;
```

	Current Date	Coordinated Universal Date
1	May 1, 2019	May 1, 2019

```
SELECT CURRENT_DATE "Current Date", CURRENT_UTCDATE "Coordinated Universal Date" FROM DUMMY;
```

	Current Date	Coordinated Universal Date
1	May 1, 2019	May 1, 2019

Related Information

Datetime Data Types
CURRENT_UTCDATE Function (Datetime)

CURRENT_MVCC_SNAPSHOT_TIMESTAMP Function (Datetime)

Returns the timestamp of the current MVCC snapshot in SSSS format (seconds past midnight).

Syntax

```
CURRENT_MVCC_SNAPSHOT_TIMESTAMP()
```

Description

The CURRENT_MVCC_SNAPSHOT_TIMESTAMP function returns the timestamp of the current MVCC snapshot in SSSS format.

Example

The following example returns the timestamp of the current MVCC snapshot:

```
SELECT CURRENT_MVCC_SNAPSHOT_TIMESTAMP() FROM DUMMY;
```

CURRENT_TIME Function (Datetime)

Returns the local system time.

Syntax

```
CURRENT_TIME
```

Description

Returns the current local system time.

It is recommended that you use UTC times instead of local times. See to the CURRENT_UTCTIME function for more information.

Example

The following example returns the local and UTC time of the system:

```
SELECT CURRENT_TIME "Current Time", CURRENT_UTCTIME "Coordinated Universal Time" FROM DUMMY;
```

	Current Time	Coordinated Universal Time
1	7:56:51 AM	2:56:51 PM

Related Information

Datetime Data Types
CURRENT_UTCTIME Function (Datetime)

CURRENT_TIMESTAMP Function (Datetime)

Returns the current local system timestamp information.

Syntax

```
CURRENT_TIMESTAMP[(<precision>)]
```

Syntax Elements

<precision>

Specifies the precision of the sub-seconds displayed. *<precision>* is an integer datatype with a valid range of 0-7. If not specified, the default precision is three.

Description

Returns the current local system timestamp information.

It is recommended that you use UTC timestamps instead of local timestamps. See CURRENT_UTCTIMESTAMP function for more information.

Example

The following example returns the local timestamp of the system with zero, three, and seven precision values:

```
SELECT
   CURRENT_TIMESTAMP,
   CURRENT_TIMESTAMP(0),
   CURRENT_TIMESTAMP(3),
```

```
    CURRENT_TIMESTAMP(7),
FROM DUMMY;


    '2021-03-09 09:07:36.8110000',
    '2021-03-09 09:07:36.0000000',
    '2021-03-09 09:07:36.8110000',
    '2021-03-09 09:07:36.8118739'
```

Related Information

Datetime Data Types

CURRENT_UTCTIMESTAMP Function (Datetime)

CURRENT_UTCDATE Function (Datetime)

Returns the current UTC date.

Syntax

```
CURRENT_UTCDATE
```

Description

Returns the current UTC date.

In application code, it is recommended that you use UTC dates instead of local dates.

Example

The following example returns the local and UTC date of the local system:

```
SELECT CURRENT_DATE "Current Date", CURRENT_UTCDATE "Coordinated Universal Date" FROM DUMMY;
```

	Current Date	Coordinated Universal Date
1	May 1, 2019	May 1, 2019

Related Information

Datetime Data Types

CURRENT_UTCTIME Function (Datetime)

Returns the current UTC time.

Syntax

```
CURRENT_UTCTIME
```

Description

Returns the current UTC time.

Example

The following example returns the local and UTC time of the system:

```
SELECT CURRENT_TIME "Current Time", CURRENT_UTCTIME "Coordinated Universal Time" FROM DUMMY;
```

	Current Time	Coordinated Universal Time
1	7:56:51 AM	2:56:51 PM

Related Information

Datetime Data Types

CURRENT_UTCTIMESTAMP Function (Datetime)

Returns the current UTC timestamp.

Syntax

```
CURRENT_UTCTIMESTAMP[(<precision>)]
```

Syntax Elements

<precision>

Specifies the precision of the sub-seconds displayed. *<precision>* is an integer datatype with a valid range of 0-7. If not specified, the default precision is three.

Description

Returns the current UTC timestamp.

Example

The following example returns the UTC timestamp of the system with zero, three, and seven precision values:

```
SELECT
   CURRENT_UTCTIMESTAMP
   CURRENT_UTCTIMESTAMP(0)
   CURRENT_UTCTIMESTAMP(3)
   CURRENT_UTCTIMESTAMP(7)
FROM DUMMY;


   '2021-03-09 09:07:36.8110000',
   '2021-03-09 09:07:36.0000000',
   '2021-03-09 09:07:36.8110000',
   '2021-03-09 09:07:36.8118739'
```

Related Information

Datetime Data Types
CURRENT_TIMESTAMP Function (Datetime)

DAYNAME Function (Datetime)

Returns the weekday name for the specified date.

Syntax

```
DAYNAME(<date>)
```

Description

Returns the weekday in English for the specified date.

Example

The following example returns `Monday` as the weekday for the specified date:

```
SELECT DAYNAME ('2011-05-30') "dayname" FROM DUMMY;
```

Related Information

Datetime Data Types

DAYOFMONTH Function (Datetime)

Returns the day of the month for the specified date.

Syntax

```
DAYOFMONTH(<date>)
```

Description

Returns an integer for the day of the month for the specified date.

Example

The following example returns `30` as the number for the day of the month for the specified date:

```
SELECT DAYOFMONTH ('2011-05-30') "dayofmonth" FROM DUMMY;
```

Related Information

DAYOFYEAR Function (Datetime)

Returns an integer representation of the day of the year for the specified date.

Syntax

```
DAYOFYEAR(<date>)
```

Description

Returns an integer representation of the day of the year for the specified date.

Example

The following example returns the value `150` as the day of the year for the specified date:

```
SELECT DAYOFYEAR ('2011-05-30') "dayofyear" FROM DUMMY;
```

Related Information

Datetime Data Types

DAYS_BETWEEN Function (Datetime)

Computes the number of entire (24 hour) days between two dates.

Syntax

```
DAYS_BETWEEN(<date_1>, <date_2>)
```

Syntax Elements

<date_1>

Specifies the starting TIMESTAMP for the comparison.

<date_2>

Specifies the ending TIMESTAMP for the comparison.

Description

Computes the number of entire days between *<date_1>* and *<date_2>*.

Example

The following example returns the value `31` for days between the two dates specified:

```
SELECT DAYS_BETWEEN (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), TO_DATE('2010-01-05', 'YYYY-MM-DD')) "da
```

The following example returns the value `0` for days between the two specified dates:

```
SELECT DAYS_BETWEEN('2018-02-07 23:00:00',  '2018-02-08 01:00:00') AS sinceDays FROM dummy;
```

The following example returns the value `1` for days between the two specified dates:

```
SELECT DAYS_BETWEEN('2018-02-07 23:00:00',  '2018-02-08 23:00:00') AS sinceDays FROM dummy;
```

Related Information

Datetime Data Types
SAP Note 2573900 - Changed Behavior of the DAYS_BETWEEN function in HANA 2.0 SPS03

DENSE_RANK Function (Window)

Performs the same ranking operation as the RANK function, except that rank numbering does not skip when ties are found.

Syntax

```
DENSE_RANK() <window_specification>
```

Syntax Elements

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

The DENSE_RANK function performs the same ranking operation as the RANK function, except that rank numbering does not skip when ties are found.

Examples

In this example, the RANK function returns the rank 5 for the row `('A', 10, 0)` because there were two rows that returned the rank of 3. However, DENSE_RANK returns the rank 4 for the row `('A', 10, 0)`.

```
CREATE TABLE T (class NVARCHAR(10), val INT, offset INT);
 INSERT INTO T VALUES('A', 1, 1);
 INSERT INTO T VALUES('A', 3, 3);
 INSERT INTO T VALUES('A', 5, null);
 INSERT INTO T VALUES('A', 5, 2);
 INSERT INTO T VALUES('A', 10, 0);
 INSERT INTO T VALUES('B', 1, 3);
 INSERT INTO T VALUES('B', 1, 1);
 INSERT INTO T VALUES('B', 7, 1);

SELECT class,
  val,
  ROW_NUMBER() OVER (PARTITION BY class ORDER BY val) AS row_num,
```

```
  RANK() OVER (PARTITION BY class ORDER BY val) AS rank,
  DENSE_RANK() OVER (PARTITION BY class ORDER BY val) AS dense_rank
FROM T;
```

CLASS	VAL	ROW_NUM	RANK	DENSE_RANK
A	1	1	1	1
A	3	2	2	2
A	5	3	3	3
A	5	4	3	3
A	10	5	5	4
B	1	1	1	1
B	1	2	1	1
B	7	3	3	2

Related Information

Expressions

Window Functions and the Window Specification

DFT Function (Aggregate)

Computes expressions and returns an array with specific elements.

Syntax

```
DFT( <expression>, <N> { <series_orderby> | <order_by_clause> } ).{ REAL | IMAGINARY | AMPLITUDE |
```

Syntax Elements

<expression>

Specifies a value assumed to be a sample taken at a constant time interval. *<expression>* cannot contain any NULL values.

<N>

This parameter must be a power of 2. The input is padded with zeros if it contains less than *<N>* elements.

<series_orderby>

The SERIES definition can only be used with an equidistant series.

```
<series_orderby> ::= SERIES( <series_period> <series_equidistant_definition> )
```

The series must not contain missing elements. For more information about this clause, see the SERIES_GENERATE function.

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

Description

Computes the Discrete Fourier Transform of an expression for the first *<N>* values and returns an array with exactly *<N>* elements.

The returned values depend on the output parameter, which must be one of REAL, IMAGINARY, AMPLITUDE, or PHASE.

Examples

The example below computes the Discrete Fourier Transform of a column in an equidistant series.

```
SELECT DFT(FRACTION_OF_MIN_MAX_RANGE, 4 SERIES(EQUIDISTANT INCREMENT BY 1 PERIOD FOR SERIES(element
    FROM SERIES_GENERATE_INTEGER(1,0,10);
```

The example below uses the fictional MY_TABLE and returns an array with 4 numbers representing the real part of the result.

```
SELECT DFT(col, 4 ORDER BY DATE).REAL FROM MY_TABLE;
```

The example below uses the fictional MY_TABLE and returns an array with 8 numbers representing the imaginary part of the result.

```
SELECT DFT(col, 8 ORDER BY DATE).IMAGINARY FROM MY_TABLE;
```

The example below uses the fictional MY_TABLE and returns an array with 8 numbers representing the amplitude part (i.e. SQRT(REAL^2 + IMAGINARY^2)) of the result.

```
SELECT DFT(col, 8 ORDER BY DATE).AMPLITUDE FROM MY_TABLE;
```

The example below uses the fictional MY_TABLE returns an array with 8 numbers representing the phase part of the result and ranges between -PI and +PI.

```
SELECT DFT(col, 8 ORDER BY DATE).PHASE FROM MY_TABLE;
```

Related Information

Aggregate Functions
COLLATIONS System View
SERIES_GENERATE Function (Series Data)

ESCAPE_DOUBLE_QUOTES Function (Security)

Escapes double quotes in the specified string.

Syntax

```
ESCAPE_DOUBLE_QUOTES(<value>)
```

Description

Escapes double quotes in the *<value>* string, ensuring that a valid SQL identifier is used in dynamic SQL statements to prevent SQL injections. The function returns the input string with escaped double quotes.

Example

The following query escapes the double quotes and returns the value TAB""LE.

```
SELECT ESCAPE_DOUBLE_QUOTES('TAB"LE') "table_name" FROM DUMMY;
```

ESCAPE_SINGLE_QUOTES Function (Security)

Escapes single quotes in the specified string.

Syntax

```
ESCAPE_SINGLE_QUOTES(<value>)
```

Description

Escapes single quotes (apostrophes) in the given string *<value>*, ensuring a valid SQL string literal is used in dynamic SQL statements to prevent SQL injections. Returns the input string with escaped single quotes.

Examples

The following query escapes the parameter content `Str'ing` to `Str''ing`.

```
SELECT ESCAPE_SINGLE_QUOTES('Str''ing') "string_literal" FROM DUMMY;
```

The following query example shows the strings retrieved from a table `t`, both without and with ESCAPE_SINGLE_QUOTES applied. The column `col_txt` contains the two entries `Adam's` and `Eve`.

```
CREATE COLUMN TABLE txt(
col_txt NVARCHAR(5000) NOT NULL);

INSERT INTO txt VALUES ('Adam''s');
INSERT INTO txt VALUES ('Eve');

SELECT col_txt, escape_single_quotes(col_txt) FROM txt;
```

The results are as follows:

COL_TXT	ESCAPE_SINGLE_QUOTES(COL_TXT)
Adam's	Adam''s
Eve	Eve

EXP Function (Numeric)

Returns the result of the base of the natural logarithms e raised to the power of the specified argument.

Syntax

```
EXP(<number>)
```

Description

Returns the result of the base of the natural logarithms e raised to the power of the argument *<number>*.

Example

The following example returns the value `2.718281828459045` for `"exp"`:

```
SELECT EXP (1.0) "exp" FROM DUMMY;
```

Related Information

Numeric Data Types

EXTRACT Function (Datetime)

Returns the requested portion of a specified date.

Syntax

```
EXTRACT( {YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM <date> )
```

Description

Returns the requested portion from the specified date.

Example

The following example returns the value `2010` for the year extracted from the specified date:

```
SELECT EXTRACT (YEAR FROM TO_DATE ('2010-01-04', 'YYYY-MM-DD')) "extract" FROM DUMMY;
```

Related Information

FIRST_VALUE Function (Aggregate)

Returns the value of the first element of an expression. This function can also be used as a window function.

Syntax

Aggregate function:

```
FIRST_VALUE( <expression> <order_by_clause> )
```

Window function:

```
FIRST_VALUE( <expression> <order_by_clause> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the expression of data to operate over.

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Returns the value of the first element in *<expression>* as ordered by *<order_by_clause>*.

NULL is returned if the value is NULL or if *<expression>* is empty.

The output of the FIRST_VALUE function can be non-deterministic among tie values.

Example

The example below returns the first value in the COL1 column when the table is ordered by COL2:

```
CREATE ROW TABLE T (COL1 DOUBLE, COL2 DOUBLE);

INSERT INTO T VALUES(9, 1);
INSERT INTO T VALUES(4, 5);
INSERT INTO T VALUES(7, 3);

SELECT FIRST_VALUE (COL1 ORDER BY COL2) FROM T;
```

The query returns 9.

Example for Spatial Data Type (ST_Geometry, ST_Point)

The example below returns the first value, `POINT (1 1)`, in binary format, when the table is ordered by ID:

```
CREATE TABLE TAB (ID INT, SHAPE ST_Geometry(4326));

INSERT INTO TAB VALUES(1, ST_GeomFromText('POINT(1 1)', 4326));
INSERT INTO TAB VALUES(2, ST_GeomFromText('POINT(2 2)', 4326));
INSERT INTO TAB VALUES(3, ST_GeomFromText('POINT(3 3)', 4326));
INSERT INTO TAB VALUES(4, ST_GeomFromText('POINT(4 4)', 4326));

SELECT FIRST_VALUE(SHAPE ORDER BY ID) FROM TAB;
```

To return a result as data type NVARCHAR, use the following statement:

```
SELECT FIRST_VALUE(CAST(SHAPE AS NVARCHAR) ORDER BY ID) FROM TAB;
```

Related Information

Window Functions and the Window Specification

Window Aggregate Functions

Aggregate Functions

SAP HANA Cloud, SAP HANA Database Spatial Reference

FLOOR Function (Numeric)

Returns the largest integer that is not greater than the specified numeric argument.

Syntax

```
FLOOR(<number>)
```

Description

Returns the largest decimal value that is not greater than the numeric argument *<number>*.

Example

The following example returns the value `14.0` for "floor":

```
SELECT FLOOR (14.5) "floor" FROM DUMMY;
```

Related Information

Numeric Data Types

GENERATE_PASSWORD Function (Security)

Generates a password.

Syntax

```
GENERATE_PASSWORD( <password_length> [, <usergroup_name>] )
```

Syntax Elements

<password_length>

Specifies the length of the password to generate as an integer.

<usergroup_name>

Specifies a usergroup name. If there are password policy parameters defined for the usergroup, they are applied during password generation.

Description

Generates and returns a password of type NVARCHAR(128).

Note

The function call can fail displaying the following out of range error message if the password policy and exclude list are too restrictive:

```
password exclude list too restrictive
```

Depending on the strictness of the password exclude list, it might suffice to retry the command.

Example

The following example returns a generated password of 16 characters::

```
SELECT GENERATE_PASSWORD(16) FROM Dummy;
```

HAMMING_DISTANCE Function (String)

Performs a bitwise or bytewise comparison between two arguments and returns the hamming distance.

Syntax

```
HAMMING_DISTANCE(<left_hand_side>, <right_hand_side>)
```

Syntax Elements

<left_hand_side>

Specifies a left hand side argument against which *<right_hand_side>* is compared.

```
<lhs> ::= <string>
```

<right_hand_side>

Specifies a right hand side argument to compare against *<left_hand_side>*

```
<rhs> ::= <string>
```

Description

Hamming distance describes differences between two specified arguments, and reflects the number of corresponding positions that differ from each other across the two arguments. Integer and binary arguments are compared bitwise, whereas string arguments are compared byte-wise.

The HAMMING_DISTANCE function returns -1 if the length of the two arguments is different, and NULL if either of the arguments is NULL.

Examples

The following example returns `-1` because the arguments are of different length:

```
SELECT HAMMING_DISTANCE('abc', 'ca') "hamming_distance" FROM DUMMY;
```

The following example returns `0` because the arguments are identical:

```
SELECT HAMMING_DISTANCE('abc', 'abc') "hamming_distance" FROM DUMMY;
```

The following example returns `0` because the arguments are identical:

```
SELECT HAMMING_DISTANCE(4, 4) "hamming_distance" FROM DUMMY;
```

The following example returns 3 because all positions in the two arguments are different from the other string:

```
SELECT HAMMING_DISTANCE('abc', 'cab') "hamming_distance" FROM DUMMY;
```

The following example returns 4 to reflect the bitwise comparison of the arguments:

```
SELECT HAMMING_DISTANCE(TO_BINARY('abc'), TO_BINARY('cab')) "hamming_distance" FROM DUMMY;
```

The following example returns 3 to reflect the bitwise comparison of the arguments:

```
SELECT HAMMING_DISTANCE(4, 9) "hamming_distance" FROM DUMMY;
```

The following example returns `-1` because the binary arguments are of different length:

```
SELECT HAMMING_DISTANCE(to_binary('abc'), to_binary('ca')) "hamming_distance" FROM DUMMY;
```

Related Information

Character String Data Types

HEXTOBIN Function (String)

Converts a string of hexadecimal characters to a VARBINARY value.

Syntax

```
HEXTOBIN(<hexadecimal_string>)
```

Description

HEXTOBIN returns a VARBINARY value where each byte of the result corresponds to two characters of *<hexadecimal_string>*. If *<hexadecimal_string>* does not contain an even number of digits, an error is returned.

In SAP HANA, lowercase characters in *<hexadecimal_string>* are supported and are treated as uppercase.

Example

The following two examples return the VARBINARY value `608DA975`:

```
SELECT HEXTOBIN ('608da975') "Result" FROM DUMMY;
```

```
SELECT HEXTOBIN ('608DA975') "Result" FROM DUMMY;
```

Related Information

Character String Data Types

HEXTONUM Function (String)

Converts a hexadecimal value to a BIGINT string value.

Syntax

```
HEXTONUM(<string> [, -1])
```

Syntax Elements

<string>

Specifies the hexadecimal value to be converted to a string value.

-1

The result is interpreted as a negative number if a second parameter is provided.

Description

Converts a hexadecimal value to a string value as a BIGINT data type. The input value must be an NVARCHAR string type that is no longer than 16 characters.

Example

The following example converts the hexadecimal value `c2` to a BIGINT string value `12,345`:

```
CREATE TABLE t2 (c2 NVARCHAR(16));
INSERT INTO t2 VALUES('7FFFFFFFFFFFFFFF');
INSERT INTO t2 VALUES('8000000000000000');
INSERT INTO t2 VALUES ('FFFFFFFFFFFFFFFF');
INSERT INTO t2 VALUES ('0');
INSERT INTO t2 VALUES ('3039');
SELECT c2, HEXTONUM(c2) FROM t2;
```

Related Information

Character String Data Types

HOUR Function (Datetime)

Returns an integer representation of the hour portion of the specified time.

Syntax

```
HOUR(<time>)
```

Description

Returns an integer representation of the hour portion of the specified time.

Example

The following example returns the hour `12`:

```
SELECT HOUR ('12:34:56') "hour" FROM DUMMY;
```

Related Information

Datetime Data Types

INITCAP Function (String)

Converts the first character of each word in a specified string to uppercase and converts remaining characters to lowercase.

Syntax

```
INITCAP(<inputString>)
```

Syntax Elements

<inputString>

Specifies the NVARCHAR string to be converted.

```
<inputString> ::= <string>
```

Description

A word is delimited by any of the following characters:

- Blank space
- New line
- Form feed
- Carriage return
- Line feed
- Any of the following: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`

Example

The following query returns `The Example One`:

```
SELECT INITCAP('the EXAMPLE one') FROM DUMMY;
```

Related Information

Character String Data Types

IS_SQL_INJECTION_SAFE Function (Security)

Checks a specified SQL identifier for possible SQL injection risks.

Syntax

```
IS_SQL_INJECTION_SAFE(<value>[, <max_tokens>])
```

Syntax Elements

<value>

Specifies the string to be checked.

```
<value> ::= <string>
```

<max_tokens>

Specifies the maximum number of tokens expected in *<value>*. The default value is 1.

```
<max_tokens> ::= <integer>
```

Description

Use this function to ensure a string contains the specified number of tokens and SQL comments before using it in an SQL statement. A 1 (safe) is returned when the actual number of tokens found does not exceed *<max_tokens>* and no comments were found; otherwise, a 0 is returned (not safe).

For the purposes of counting, SAP HANA interprets whitespaces and the following characters as separators between tokens:

```
, ( ) [ ] . ; : + - * / % ^ < >  =
```

Example

The following query returns 0 (not safe) because two tokens were found and the default to check for was 1.

```
SELECT IS_SQL_INJECTION_SAFE('tab,le') "safe" FROM DUMMY;
```

The following query returns 0 (not safe) because comments are not allowed.

```
SELECT IS_SQL_INJECTION_SAFE('mytab /*', 4) "safe" FROM DUMMY;
```

ISOWEEK Function (Datetime)

Returns the ISO year and week number for a specified date.

Syntax

```
ISOWEEK(<date>)
```

Description

Returns the ISO year and week numbers of the date specified by *<date>*. The week number is prefixed by the letter W.

Both the WEEK and ISOWEEK functions return the week number for a specified date but the format of the result is quite different, and the two functions may handle the first week of the new year differently. For example, when supplied the date 2017-01-01, the WEEK function considers the date to be part of the ***first week of 2017*** and returns 1, whereas ISOWEEK considers the date to be part of the ***last week of 2016*** and returns 2016-W52.

ISOWEEK has either 52 or 53 full weeks, with the extra week considered to be a leap week

Example

The following example returns the value `2011-W22` for the ISO year and week numbers of the specified date:

```
SELECT ISOWEEK (TO_DATE('2011-05-30', 'YYYY-MM-DD')) "isoweek" FROM DUMMY;
```

Related Information

Datetime Data Types

JSON_QUERY Function (JSON)

Extracts JSON text from a JSON context item by using a SQL/JSON path expression.

Syntax

```
JSON_QUERY(
 <JSON_API_common_syntax>
 [ <JSON_output_clause> ]
 [ <JSON_query_wrapper_behavior> ]
 [ <JSON_query_empty_behavior> ON EMPTY ]
 [ <JSON_query_error_behavior> ON ERROR ]
 )
```

Syntax Elements

<JSON_API_common_syntax>

Specifies a JSON context item and a path to the context item, using common syntax for the JSON API.

```
<JSON_API_common_syntax> ::= <JSON_context_item>, <JSON_path_specification>
```

<JSON_context_item>

Specifies the JSON document to operate on, such as a table column, string, or collection.

<JSON_path_specification>

Specifies the path to *<JSON_context_item>*.

```
<JSON_path_specification> ::= <JSON_path_mode> <JSON_path_wff>

<JSON_path_mode> ::= STRICT | LAX
```

If a structural error occurs within a JSON filter expression and *<JSON_path_mode>* is set to STRICT, then the error handling of a JSON filter expression applies. Otherwise, a structural error is an unhandled error.

When the path is set to LAX, one of the following options occurs:

- If an operation requires an SQL/JSON array but the operand is not an SQL/JSON array, then the operand is wrapped in an SQL/JSON array prior to performing the operation.
- If an operation requires something other than an SQL/JSON array, but the operand is an SQL/JSON array, then the operand is unwrapped by converting its elements into an SQL/JSON sequence prior to performing the operation.

Array indexes start from 0, rather than 1 (the SQL standard).

If there is still a structural error after applying these resolutions, then the result is an empty SQL/JSON sequence.

<JSON_path_wff> indicates an actual JSON path (for example, `'$.item1'`). *<JSON_path_specification>* does not use double quotes.

<JSON_output_clause>

Specifies the output created by the JSON_QUERY function.

```
<JSON_output_clause> ::= RETURNING <data_type>
```

<data_type>

Specifies the data type to be set as the return type of the JSON QUERY function. Supported data types: NVARCHAR(*<length>*).

***<JSON_query_wrapper_behavior>* WRAPPER**

Specifies the wrapper behavior of the JSON query.

```
<JSON_query_wrapper_behavior> WRAPPER

<JSON_query_wrapper_behavior> ::=
WITHOUT [ ARRAY ]
 | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]
```

The default is WITHOUT ARRAY WRAPPER. However, if WITH is specified, then the default is UNCONDITIONAL ARRAY WRAPPER.

***<JSON_query_empty_behavior>* ON EMPTY**

Specifies the behavior of the function if the related data is not in the context item. The default is NULL ON EMPTY.

```
<JSON_query_empty_behavior> ON EMPTY

<JSON_query_empty_behavior> ::=
ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT
```

ERROR ON EMPTY returns an error if the related data is not in the context item. NULL ON EMPTY returns a NULL if the related data is not in the context item. EMPTY ARRAY ON EMPTY returns an empty array if the related data is not in the context item. EMPTY OBJECT ON EMPTY returns an empty JSON object if the related data is not in the context item.

***<JSON_query_error_behavior>* ON ERROR**

Specifies the behavior of the function when the query throws an error. The default is NULL ON ERROR.

```
<JSON_query_error_behavior> ON ERROR

<JSON_query_error_behavior> ::=
ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT
```

ERROR ON ERROR returns an error if the function result includes an error. NULL ON ERROR returns a NULL if the function result includes an error. EMPTY ARRAY ON ERROR returns an empty array if the function result includes an error. EMPTY OBJECT ON ERROR returns an empty JSON object if the function result includes an error.

Description

Extracts JSON text from a JSON context item using a SQL/JSON path expression.

The following tokens are supported in *<JSON_API_common_syntax>*:

Token	Description	Example
$	The context item (the first argument of the function).	'$'
.	The member of an object.	'$.item.description'
[	The array index specifier (open).	
]	The array index specifier (closed).	'$[1]' '$.item.list[1]'
to	The array index range.	'$[3 to 5]' = '$[3,4,5]'
*	The wild card.	'$.*.description' '$.item.list[*]'

Example

The following query returns the value `{"item1":1,"item2":2,"item3":3}`.

```
SELECT JSON_QUERY('{"item1":1, "item2":2, "item3":3}', '$') AS JSONQUERY FROM DUMMY;
```

The following query returns the value `[1]`.

```
SELECT JSON_QUERY('{"item1":1, "item2":2, "item3":3}', '$.item1' WITH WRAPPER ) AS JSONQUERY FROM D
```

Related Information

Expressions

JSON_TABLE Function (JSON)

Queries a JSON text and presents it as a relational table.

Syntax

```
JSON_TABLE(
 <JSON_API_common_syntax>
 <JSON_table_columns_clause>
 [ <JSON_table_error_behavior> ON ERROR ]
 )
```

Syntax Elements

<JSON_API_common_syntax>

Specifies a JSON context item and a path to the context item, using common syntax for the JSON API.

```
<JSON_API_common_syntax> ::= <JSON_context_item>, <JSON_path_specification>
```

<JSON_context_item>

Specifies the JSON document to operate on, such as a table column or string. Collections are not supported.

<JSON_path_specification>

Specifies the path to *<JSON_context_item>*.

```
<JSON_path_specification> ::= <JSON_path_mode> <JSON_path_wff>

<JSON_path_mode> ::= STRICT | LAX
```

Depending on the location in the grammar, there are three kinds of JSON path expressions:

Row pattern path expression

Used to produce an SQL/JSON sequence, with one SQL/JSON item for each row of the output table.

Column pattern path expression

Used to search for the column within the current SQL/JSON item produced by the row pattern.

Nested columns pattern path expression

Used for unnesting of (even deeply) nested JSON objects/arrays in one invocation rather than chaining several JSON_TABLE expressions in the SQL-statement.

For example:

Path Expression Type	Example Syntax
Row pattern path expression	'lax $'
Column pattern path expression	'lax $.name' 'lax $.type' 'lax $.number'
Nested pattern path expression	'lax $.phoneNumber[*]'

```
SELECT bookclub.id, jt.name, jt.type, jt.number
FROM bookclub,
    JSON_TABLE ( bookclub.jcol, 'lax $'
                 COLUMNS ( name VARCHAR(30) PATH 'lax $.Name',
                           NESTED PATH 'lax $.phoneNumber[*]'
                              COLUMNS ( type VARCHAR(10) PATH 'lax $.type',
                                        number CHAR(12) PATH 'lax $.number' )
                           )
                ) AS jt;
```

The error types generated by the *<JSON_path_specification>* are:

- an input conversion error (for example, invalid JSON document – cannot be parsed)
- an error returned by the PATH engine, which evaluates JSON path expressions
- a structural error meaning no matching path in the JSON document (for example, '$.ab' path expression against {"cd" : 1})
    - If a structural error occurs within a JSON filter expression and *<JSON_path_mode>* is set to STRICT, then the error handling of a JSON filter expression applies. Otherwise, a structural error is an unhandled error.

- When the path is set to LAX, one of the following options occurs:
  - If an operation requires an SQL/JSON array but the operand is not an SQL/JSON array, then the operand is wrapped in an SQL/JSON array prior to performing the operation.
  - If an operation requires something other than an SQL/JSON array, but the operand is an SQL/JSON array, then the operand is unwrapped by converting its elements into an SQL/JSON sequence prior to performing the operation.
- Array indexes start from 0, rather than 1 (the SQL standard).
- If there is still a structural error after applying these resolutions, then the result is an empty SQL/JSON sequence.

<JSON_path_wff> indicates an actual JSON path (for example, `'$.item1'`). *<JSON_path_specification>* does not use double quotes.

<JSON_table_columns_clause>

Specifies the columns that are created.

```
<JSON_table_columns_clause> ::=
 COLUMNS ( <JSON_table_column_definition> [,... ] )

<JSON_table_column_definition> ::=
<JSON_table_ordinality_column_definition>
 | <JSON_table_regular_column_definition>
 | <JSON_table_formatted_column_definition>
 | <JSON_table_nested_columns>
```

<JSON_table_column_definition>

Defines the columns generated.

<JSON_table_ordinality_column_definition>

Defines an ordinality column. An ordinality column is similar to a column defined using the ROW_NUMBER window function.

```
<JSON_table_ordinality_column_definition> ::= <column_name> FOR ORDINALITY
```

<JSON_table_regular_column_definition>

Defines the regular columns. Each result (row) of a regular JSON table column is equivalent to a JSON_VALUE function result.

```
<JSON_table_regular_column_definition> ::= <column_name> <data_type>
 PATH <JSON_table_column_path_specification>
 [ <JSON_table_column_empty_behavior> ON EMPTY ]
 [ <JSON_table_column_error_behavior> ON ERROR ]

<JSON_table_column_path_specification> ::= <JSON_path_specification>

<JSON_table_column_empty_behavior> ::=
ERROR
 | NULL
 | DEFAULT <value_expression>

<JSON_table_column_error_behavior> ::=
ERROR
 | NULL
 | DEFAULT <value_expression>
```

<column_name>

Specifies the name of the column.

<data_type>

Specifies the data type.

```
<data_type> ::=
BIGINT
 | DATE
 | DECIMAL
 | DOUBLE
 | INT
 | NVARCHAR (<int_const>)
 | SECONDDATE
 | SMALLDECIMAL
 | TIME
 | TIMESTAMP
```

<JSON_table_column_path_specification>

Specifies the JSON path that specifies which JSON value the JSON context item is extracted from.

***<JSON_table_column_empty_behavior>* ON EMPTY**

Specifies the behavior of the function when the created column is empty. The default is NULL ON EMPTY.

ERROR ON EMPTY returns an error when the created column is empty. NULL ON EMPTY returns a NULL when the created column is empty. DEFAULT *<value_expression>* ON EMPTY returns *<value_expression>* when the created column is empty.

***<JSON_table_column_error_behavior>* ON ERROR**

Specifies the behavior of the function when there is an error during column creation. The default is NULL ON ERROR.

ERROR ON ERROR results in an error being thrown if the function result includes an error. NULL ON ERROR returns a NULL if the function result includes an error. DEFAULT *<value_expression>* ON ERROR returns *<value_expression>* if the function result includes an error.

<JSON_table_formatted_column_definition>

Specifies the column definition for formatted columns where the records in the column are formatted in JSON syntax. Each result (row) of a regular JSON table column is equivalent to a JSON_QUERY function result.

```
<JSON_table_formatted_column_definition> ::=
 <column_name> <data_type>
 FORMAT <JSON_representation>
 PATH <JSON_table_column_path_specification>
 [ <JSON_table_formatted_column_wrapper_behavior> WRAPPER ]
 [ <JSON_table_formatted_column_empty_behavior> ON EMPTY ]
 [ <JSON_table_formatted_column_error_behavior> ON ERROR ]
```

<column_name>

Specifies the name of the column.

<data_type>

Specifies the data type to be set as the return type of the function. Supported types are: NVARCHAR.

<JSON_representation>

Specifies the JSON encoding to use.

```
<JSON_representation> ::=
JSON
 | JSON ENCODING { UTF8 }
 | JSON ENCODING { UTF16 }
 | JSON ENCODING { UTF32 }
```

<JSON_table_column_path_specification>

Specifies the JSON path that specifies which JSON value the JSON context item is extracted from.

<JSON_table_formatted_column_wrapper_behavior> **WRAPPER**

Specifies the wrapper behavior of the formatted column.

```
<JSON_table_formatted_column_wrapper_behavior> ::=
WITHOUT [ ARRAY ]
 | WITH [ CONDITIONAL | UNCONDITIONAL ] [ ARRAY ]
```

When WITHOUT [ARRAY] WRAPPER is specified, the formatted column is not represented as a JSON array. When WITH [CONDITIONAL | UNCONDITIONAL] [ARRAY] WRAPPER is specified, the formatted column is set with a conditional/unconditional array. With a conditional array, the result is formatted as a JSON array if the result is neither a JSON array nor a JSON object. With an unconditional array, the result is formatted as a JSON array if the result is not a JSON object. The difference between unconditional and conditional wrapper is that unconditional wrapper wraps the result as a JSON array once more when the result is a JSON array.

<JSON_table_formatted_column_empty_behavior> **ON EMPTY**

Specifies the JSON behavior if the related data is not in the context item.

```
<JSON_table_formatted_column_empty_behavior> ::=
ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT
```

ERROR ON EMPTY returns an error if the related data is not in the context item. NULL ON EMPTY returns a NULL if the related data is not in the context item. EMPTY ARRAY ON EMPTY returns an empty array if the related data is not in the context item. EMPTY OBJECT ON EMPTY returns an empty JSON object if the related data is not in the context item.

<JSON_table_formatted_column_error_behavior> **ON ERROR**

Specifies the behavior when the formatted column throws an error.

```
<JSON_table_formatted_column_error_behavior> ::=
ERROR
 | NULL
 | EMPTY ARRAY
 | EMPTY OBJECT
```

ERROR ON ERROR returns an error if the function returns no result. NULL ON ERROR returns a NULL if the function returns no result. EMPTY ARRAY ON ERROR returns an empty array if the result of the formatted column is empty. EMPTY OBJECT ON ERROR returns an empty JSON object if the result of the formatted column is empty.

<JSON_table_nested_columns>

Defines nested columns.

```
<JSON_table_nested_columns> ::= NESTED [ PATH ] <JSON_table_nested_path_specification>
 <JSON_table_columns_clause>

<JSON_table_nested_path_specification> ::= <JSON_path_specification>
```

- *<JSON_table_columns_clause>*: Specifies the columns that are created with the function.
- *<JSON_path_specification>*: The JSON path which specifies which JSON value from the JSON context item is extracted.

<JSON_table_error_behavior>

Specifies the behavior of the function when an error occurs. The default behavior is EMPTY ON ERROR.

```
<JSON_table_error_behavior> ::= { ERROR | EMPTY }
```

ERROR

If the function result includes an error, then the error is returned.

EMPTY

If the function result includes an error, then it returns an empty result.

Description

Queries a JSON text and presents it as a relational table.

All of the errors and empty handling that results from column pattern path expression evaluation, as well as output conversion errors are handled by *<JSON_table_column_empty_behavior>*, *<JSON_table_column_error_behavior>*, *<JSON_table_formatted_column_empty_behavior>* ON EMPTY, and *<JSON_table_formatted_column_error_behavior>* ON ERROR.

All of the errors resulting from row pattern expression evaluation and nested columns pattern path expression evaluation are handled by *<JSON_table_error behavior>*.

The following tokens are supported in *<JSON_API_common_syntax>*:

Token	Description	Example
$	The current context item.	`'$'`
.	The member of an object.	`'$.item.description'`
[	The array index specifier (open).	
]	The array index specifier (closed).	`'$[1]'` `'$.item.list[1]'`
to	The array index range.	`'$[3 to 5]'` `= '$[3,4,5]'`
*	The wild card.	`'$.*.description'` `'$.item.list[*]'`

Example

The following examples use the table created below:

```
CREATE ROW TABLE T1 (A INT, B NVARCHAR(5000));
INSERT INTO T1 VALUES (1, '
{
        "PONumber": 1,
        "Reference": "BSMITH-74635645",
        "Requestor": "Barb Smith",
        "User": "BSMITH",
        "CostCenter": "A50",
        "ShippingInstructions":
              {
                      "name": "Barb Smith",
                      "Address":
```

```
                                {
                                       "street": "100 Fairchild Ave",
                                       "city": "San Diego",
                                        "state": "CA",
                                       "zipCode": 23345,
                                       "country": "USA"
                                },
                          "Phone": [{"type": "Office", "number": "519-555-6310"}]
                },
          "SpecialInstructions": "Surface Mail",
          "LineItems": [
                {"ItemNumber": 1, "Part": {"Description": "Basic Kit", "UnitPrice": 19.95, "UPCCode"
                {"ItemNumber": 2, "Part": {"Description": "Base Kit 2", "UnitPrice": 29.95, "UPCCode
                {"ItemNumber": 3, "Part": {"Description": "Professional", "UnitPrice": 39.95, "UPCCo
                {"ItemNumber": 4, "Part": {"Description": "Enterprise", "UnitPrice": 49.95, "UPCCode
                {"ItemNumber": 5, "Part": {"Description": "Unlimited", "UnitPrice": 59.95, "UPCCode"
          ]
}
');
```

The following example selects from an ordinality column and a regular column:

```
SELECT JT.*
FROM JSON_TABLE(T1.B, '$.LineItems[*]'
COLUMNS
    (
        RN FOR ORDINALITY,
        ITEM_NUMBER INT PATH '$.ItemNumber',
        UPC_CODE BIGINT PATH '$.Part.UPCCode'
    )
) AS JT;
```

RN	ITEM_NUMBER	UPC_CODE
1	1	73649587162
2	2	83600229374
3	3	33298003521
4	4	91827739856
5	5	22983303876

The following example selects from a formatted column:

```
SELECT *
FROM JSON_TABLE(T1.B, '$.ShippingInstructions'
COLUMNS
    (
        PHONE NVARCHAR(50) FORMAT JSON PATH '$.Phone'
    )
) AS JT;
```

PHONE
[{"number":"519-555-6310","type":"Office"}]

The following example selects from a nested column:

```
SELECT *
FROM JSON_TABLE(T1.B, '$.ShippingInstructions'
COLUMNS
    (
        NESTED PATH '$.Address'
```

```
        COLUMNS
        (
            STREET NVARCHAR(50) PATH '$.street',
            CITY NVARCHAR(50) PATH '$.city'
        )
    )
) AS JT;
```

STREET	CITY
100 Fairchild Ave	San Diego

The following example selects from an ordinality column with nested columns:

```
SELECT *
FROM JSON_TABLE(T1.B, '$'
    COLUMNS
    (
        RN FOR ORDINALITY,
        USER_NAME NVARCHAR(20) PATH '$.User',
        NESTED PATH '$.LineItems[1,2]'
        COLUMNS
        (
            ORDER_NUMBER FOR ORDINALITY,
            ITEM_NUMBER INT PATH '$.ItemNumber',
            QUANTITY INT PATH '$.Quantity'
        )
    )
) AS JT;
```

RN	USER_NAME	ORDER_NUMBER	ITEM_NUMBER	QUANTITY
1	BSMITH	1	2	1
1	BSMITH	2	3	8

The following example demonstrates the difference when specifying CONDITIONAL ARRAY WRAPPER versus UNCONDITIONAL ARRAY WRAPPER when returning a JSON array:

```
CREATE ROW TABLE r1 ( a INT, b NVARCHAR(5000));
INSERT INTO r1 VALUES (1, '{"menu": {"header": "SVG Viewer","items": [{"id": "Open"},{"id": "OpenNe
```

SELECT statement	Value returned
`SELECT JSON_QUERY(B, '$.menu.items' WITH UNCONDITIONAL ARRAY WRAPPER) FROM r1 WITH HINT(IGNORE_PLAN_CACHE);`	`[[{"id":"Open"},{"id":"OpenNew","label":"Open New"},null,{"id":`
`SELECT JSON_QUERY(B, '$.menu.items' WITH CONDITIONAL ARRAY WRAPPER) FROM r1;`	`[{"id":"Open"},{"id":"OpenNew","label":"Open New"},null,{"id":"`

The following example demonstrates an example of ERROR ON ERROR, which results in an error stating that the data for '$.User' cannot be parsed to an integer data type.

```
SELECT *
FROM JSON_TABLE(T1.B, '$'
    COLUMNS
    (
        RN FOR ORDINALITY,
```

```
        USER_NAME INT PATH '$.User' ERROR ON ERROR
    )
ERROR ON ERROR
) AS JT;
```

The following example demonstrates an example of EMPTY ON ERROR. The default behavior of JSON_table_error_behavior is EMPTY ON ERROR. It returns an error result instead of throwing empty when JSON_table_column_error_behavior is ERROR ON ERROR or JSON_table_column_empty_behavior is ERROR ON EMPTY.

```
SELECT *
FROM JSON_TABLE(T1.B, '$'
    COLUMNS
    (
        RN FOR ORDINALITY,
        USER_NAME INT PATH '$.User' ERROR ON ERROR
    )
) AS JT;
```

Related Information

Expressions

JSON_VALUE Function (JSON)

Extracts an SQL value of a predefined type from a JSON value.

Syntax

```
JSON_VALUE(
 <JSON_API_common_syntax>
 [ <JSON_returning_clause> ]
 [ <JSON_value_empty_behavior> ON EMPTY ]
 [ <JSON_value_error_behavior> ON ERROR ]
 )
```

Syntax Elements

<JSON_API_common_syntax>

Specifies a JSON context item and a path to the context item, using common syntax for the JSON API.

```
<JSON_API_common_syntax> ::= <JSON_context_item>, <JSON_path_specification>
```

<JSON_context_item>

Specifies the JSON document to operate on, such as a table column, string, or collection.

<JSON_path_specification>

Specifies the path to *<JSON_context_item>*.

```
<JSON_path_specification> ::= <JSON_path_mode> <JSON_path_wff>

<JSON_path_mode> ::= STRICT | LAX
```

When *<JSON_path_mode>* is set to STRICT, if the structural error occurs within a JSON filter expression, then the error handling of a JSON filter expression applies. Otherwise, a structural error is an unhandled error.

When the path is set to LAX, one of the following options occurs:

- If an operation requires an SQL/JSON array but the operand is not an SQL/JSON array, then the operand is wrapped in an SQL/JSON array prior to performing the operation.
- If an operation requires something other than an SQL/JSON array, but the operand is an SQL/JSON array, then the operand is unwrapped by converting its elements into an SQL/JSON sequence prior to performing the operation.

Array indexes start from 0, rather than 1 (the SQL standard).

If there is still a structural error after applying these resolutions, then the result is an empty SQL/JSON sequence.

<JSON_path_wff> indicates an actual JSON path (for example, `'$.item1'`). *<JSON_path_specification>* does not use double quotes.

<JSON_returning_clause>

Defines the data type of the result.

```
<JSON_returning_clause> ::= RETURNING <data_type>

<data_type> ::=
INTEGER
 | BIGINT
 | DECIMAL
 | NVARCHAR(<integer>)
```

***<JSON_value_empty_behavior>* ON EMPTY**

Specifies the behavior of the function when the related data is not in the context item. The default is NULL ON EMPTY

```
<JSON_value_empty_behavior> ON EMPTY

<JSON_value_empty_behavior> ::=
ERROR
 | NULL
 | DEFAULT <value_expression>
```

ERROR ON EMPTY returns an error if the related data is not in the context item. NULL ON EMPTY returns a NULL if the related data is not in the context item. DEFAULT *<value_expression>* ON EMPTY returns *<value_expression>* if the related data is not in the context item.

***<JSON_value_error_behavior>* ON ERROR**

Specifies the behavior of the JSON_VALUE function when an error occurs. The default is NULL ON ERROR.

```
<JSON_value_error_behavior> ON ERROR

<JSON_value_error_behavior> ::=
ERROR
 | NULL
 | DEFAULT <value_expression>
```

ERROR ON ERROR returns an error if the function results include an error. NULL ON ERROR returns a NULL if the function results include an error. DEFAULT *<value_expression>* ON ERROR returns *<value_expression>* if the function results include an error.

Description

Extracts an SQL value of a predefined type from a JSON value.

The following tokens are supported in *<JSON_API_common_syntax>*:

Token	Description	Example
$	The context item (the first argument of the function).	'$'
.	The member of an object.	'$.item.description'
[	The array index specifier (open).	
]	Array index specifier (closed).	'$[1]' '$.item.list[1]'
to	The array index range.	'$[3 to 5]' = '$[3,4,5]'
*	The wild card.	'$.*.description' '$.item.list[*]'

Example

The following statement returns a value of 10:

```
SELECT JSON_VALUE('{"item1":10}', '$.item1') AS "value" FROM DUMMY;
```

The following statement returns a value of 5:

```
SELECT JSON_VALUE('{"item1":{"sub1":10}, "item2":{"sub2":5}, "item3":{"sub3":7}}', '$.*.sub2') AS "
```

The following statement returns a value of 0:

```
SELECT JSON_VALUE('[0, 1, 2, 3]', '$[0]') AS "value"  FROM DUMMY;
```

The following statement returns the value "No last name found":

```
SELECT JSON_VALUE('{"firstname":"John"}', '$.lastname' DEFAULT 'No last name found' ON EMPTY) AS "L
```

The following statement causes a type conversion error to demonstrate the behavior for ERROR ON ERROR:

```
SELECT JSON_VALUE('{"item":"string"}', '$.item' RETURNING DECIMAL ERROR ON ERROR) AS "Item" FROM DU
```

The following statement demonstrates what happens when there is no value (the object does not have the name "last name"):

```
SELECT JSON_VALUE('{"firstname":"John"}', 'strict $.lastname' ERROR ON ERROR) AS "Last Name" FROM D
```

Related Information

Expressions

LAG Function (Window)

Returns the value of the offset rows before the current row.

Syntax

```
LAG( <expression> [, <offset> [, <default_expr> ] ] ) <window_specification>
```

Syntax Elements

<offset>

The *<offset>* should be non-negative and the default is 1.

If the *<offset>* crosses boundaries of the partition, then *<default_expr>* value is returned. If the *<default_expr>* is not specified, then a null value is returned. The *<offset>* and *<default_expr>* are evaluated at current row.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

The output of the LAG function can be non-deterministic among tie values.

Examples

```
CREATE TABLE T (class NVARCHAR(10), val INT, offset INT);
 INSERT INTO T VALUES('A', 1, 1);
 INSERT INTO T VALUES('A', 3, 3);
 INSERT INTO T VALUES('A', 5, null);
 INSERT INTO T VALUES('A', 5, 2);
 INSERT INTO T VALUES('A', 10, 0);
 INSERT INTO T VALUES('B', 1, 3);
 INSERT INTO T VALUES('B', 1, 1);
 INSERT INTO T VALUES('B', 7, 1);

SELECT class,
  val,
  offset,
  LEAD(val) OVER (PARTITION BY class ORDER BY val) AS lead,
  LEAD(val,offset,-val) OVER (PARTITION BY class ORDER BY val) AS lead2,
  LAG(val) OVER (PARTITION BY class ORDER BY val) AS lag,
  LAG(val,offset,-val) OVER (PARTITION BY class ORDER BY val) AS lag2
 FROM T;
```

CLASS	VAL	OFFSET	LEAD	LEAD2	LAG	LAG2
A	1	1	3	3	null	-1
A	3	3	5	10	1	-3
A	5	null	5	-5	3	-5
A	5	2	10	-5	5	3
A	10	0	null	10	5	10
B	1	3	1	-1	null	-1
B	1	1	7	7	1	1
B	7	1	null	-7	1	1

Related Information

Expressions
Window Functions and the Window Specification

LAST_DAY Function (Datetime)

Returns the date of the last day of the month that contains the specified date.

Syntax

```
LAST_DAY(<date>)
```

Description

Returns the date of the last day of the month that contains the date *<date>*.

Example

The following example returns the value `2010-01-31` (or another format like `Jan 31, 2010`, depending on your date display settings):

```
SELECT LAST_DAY (TO_DATE('2010-01-04', 'YYYY-MM-DD')) "last day" FROM DUMMY;
```

Related Information

Datetime Data Types

LAST_VALUE Function (Aggregate)

Returns the value of the last element of an expression. This function can also be used as a window function.

Syntax

Aggregate function:

```
LAST_VALUE( <expression> <order_by_clause> )
```

Window function:

```
LAST_VALUE( <expression> <order_by_clause> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the expression of data to operate over.

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Returns the value of the last element in *<expression>* as ordered by *<order_by_clause>*.

NULL is returned if the value is NULL or if *<expression>* is empty.

The output of the LAST_VALUE function can be non-deterministic among tie values.

Example

The example below returns the last value in COL1 column when the table is ordered by COL2. The query returns 4.

```
CREATE ROW TABLE T (COL1 DOUBLE, COL2 DOUBLE);

INSERT INTO T VALUES(1, 1);
INSERT INTO T VALUES(4, 5);
INSERT INTO T VALUES(7, 3);

SELECT LAST_VALUE (COL1 ORDER BY COL2) FROM T;
```

Example for Spatial Data Type (ST_Geometry, ST_Point)

The example below returns the last value, `POINT (4 4)`, in binary format, when the table is ordered by ID:

```
CREATE TABLE TAB (ID INT, SHAPE ST_Geometry(4326));

INSERT INTO TAB VALUES(1, ST_GeomFromText('POINT(1 1)', 4326));
INSERT INTO TAB VALUES(2, ST_GeomFromText('POINT(2 2)', 4326));
INSERT INTO TAB VALUES(3, ST_GeomFromText('POINT(3 3)', 4326));
INSERT INTO TAB VALUES(4, ST_GeomFromText('POINT(4 4)', 4326));

SELECT LAST_VALUE(SHAPE ORDER BY ID) FROM TAB;
```

To return a result as data type NVARCHAR, use the following statement:

```
SELECT LAST_VALUE(CAST(SHAPE AS NVARCHAR) ORDER BY ID) FROM TAB;
```

Related Information

COLLATIONS System View

Window Functions and the Window Specification

Window Aggregate Functions

Aggregate Functions

SAP HANA Cloud, SAP HANA Database Spatial Reference

LCASE Function (String)

Converts all characters in a string to lowercase.

Syntax

```
LCASE(<string>)
```

Description

Converts all characters in string *<string>* to lowercase.

The LCASE function is identical to the LOWER function.

Example

This example converts all characters of the given string `TesT` to lowercase and returns the value `test`.

```
SELECT LCASE ('TesT') "lcase" FROM DUMMY;
```

Related Information

LOWER Function (String)

Character String Data Types

LEAD Function (Window)

Returns the offset of rows after the current row. The *<offset>* should be non-negative and its default is 1.

Syntax

```
LEAD( <expression> [, <offset> [, <default_expr> ] ] ) <window_specification>
```

Syntax Elements

<offset>

If the *<offset>* crosses boundaries of the partition, then the *<default_expr>* value is returned. If the *<default_expr>* is not specified, then a null value is returned. The *<offset>* and *<default_expr>* are evaluated at current row.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

The output of the LEAD function can be non-deterministic among tie values.

Examples

```
CREATE TABLE T (class NVARCHAR(10), val INT, offset INT);
 INSERT INTO T VALUES('A', 1, 1);
 INSERT INTO T VALUES('A', 3, 3);
 INSERT INTO T VALUES('A', 5, null);
 INSERT INTO T VALUES('A', 5, 2);
 INSERT INTO T VALUES('A', 10, 0);
 INSERT INTO T VALUES('B', 1, 3);
 INSERT INTO T VALUES('B', 1, 1);
 INSERT INTO T VALUES('B', 7, 1);

SELECT class,
  val,
  offset,
  LEAD(val) OVER (PARTITION BY class ORDER BY val) AS lead,
  LEAD(val, offset, -val) OVER (PARTITION BY class ORDER BY val) AS lead2,
  LAG(val) OVER (PARTITION BY class ORDER BY val) AS lag,
  LAG(val, offset, -val) OVER (PARTITION BY class ORDER BY val) AS lag2
 FROM T;
```

CLASS	VAL	OFFSET	LEAD	LEAD2	LAG	LAG2
A	1	1	3	3	null	-1
A	3	3	5	10	1	-3
A	5	null	5	-5	3	-5
A	5	2	10	-5	5	3
A	10	0	null	10	5	10
B	1	3	1	-1	null	-1
B	1	1	7	7	1	1
B	7	1	null	-7	1	1

Related Information

Expressions

Window Functions and the Window Specification

LEFT Function (String)

Returns the specified number of characters or bytes of a string, starting from the left side.

Syntax

```
LEFT(<string>, <number>)
```

Syntax Elements

<string>

Specifies the string to be operated on.

<number>

Specifies the length of characters or bytes to return, starting from the left side.

Description

Returns the first *<number>* of characters or bytes from the beginning of *<string>*.

Returns an empty string value if *<number>* is less than 1.

Returns *<string>* (without blank padding) if the value of *<number>* is greater than the length of *<string>*.

Example

The following example returns the leftmost three characters of the string `Hel`:

```
SELECT LEFT ('Hello', 3) "left" FROM DUMMY;
```

The following example returns the string `Hello` because the value 10 exceeds the string length:

```
SELECT LEFT ('Hello', 10) "left" FROM DUMMY;
```

LENGTH Function (String)

Returns the number of characters in a string.

Syntax

```
LENGTH(<string>)
```

Description

Returns the number of characters in string *<string>*.

Supplementary plane Unicode characters, each of which occupies 6 bytes in CESU-8 encoding, are counted as two characters.

Example

This example returns the number of characters (`14`) contained in the given string:

```
SELECT LENGTH ('length of this') "length" FROM DUMMY;
```

Related Information

Character String Data Types

LINEAR_APPROX Function (Window)

Operates on an entire series to produce a new series that replaces missing values by interpolating between adjacent non-NULL values and extrapolating any leading or trailing null values.

Syntax

```
LINEAR_APPROX(<expression> [, <ModeArgument> [, <Value1Argument> [,
    <Value2Argument>]]]) OVER ({ SERIES(...) [<window_partition_by_clause>]
    [<window_order_by_clause>] | [<window_partition_by_clause>] <window_order_by_clause>})

<expression> ::= <identifier>
```

Syntax Elements

<expression>

This parameter specifies the input data column.

<ModeArgument>

This parameter defines the applied extrapolation mode. The table below lists the possible values for *<ModeArgument>*.

```
<ModeArgument> ::= EXTRAPOLATION_NONE
    | EXTRAPOLATION_LINEAR
    | EXTRAPOLATION_CONSTANT
```

Mode	Description
EXTRAPOLATION_NONE	Prevents the extrapolation of leading and trailing nulls. Interpolation is performed on the values in the middle. Default value if *<ModeArgument>* is omitted. You do not need to specify the *<Value1Argument>* and *<Value2Argument>* parameters when using this mode. For example, an input of [null, null, 1, 2, null, null, 5, null, null] returns [null, null, 1, 2, 3, 4, 5, null, null].
EXTRAPOLATION_LINEAR	Performs linear extrapolation where *<Value1Argument>* is the minimum value and *<Value2Argument>* is the maximum value. During the extrapolation of leading and trailing nulls, this function checks that the extrapolation results are within range of the minimum and maximum values; otherwise, values are replaced by the minimum or maximum value, whichever is appropriate. An exception is thrown if *<Value1Argument>* is greater than *<Value2Argument>*.
EXTRAPOLATION_CONSTANT	Performs linear extrapolation where leading and trailing nulls values are replaced by the values specified by the *<Value1Argument>* and *<Value2Argument>* parameters, respectively. When *<Value1Argument>* is not specified or null, the first non-null value is used to replace leading null values. When *<Value2Argument>* is not specified or null, the last non-null is used to replace trailing null values.

Linear interpolation can be applied to all number types if they can be cast into a long double type, such as INT, LONG, DOUBLE, or FLOAT.

<Value1Argument>* and *<Value2Argument>

These parameters define arguments for the extrapolation and can be any numeric type.

```
<Value1Argument> ::= <identifier>
<Value2Argument> ::= <identifier>
```

<table_schema>

```
<table_schema> ::= [<schema_name>.]<table_name>
<schema_name> ::= <unicode_name>
<table_name> ::= <identifier>
```

<table_schema> must be a base table name and not a correlation name.

<window_order_by_clause>

Determines the sort order. This expression must not contain any NULL values or duplicates. See the syntax of this clause located in the *Window Functions* topic.

Description

Operates on an entire series to produce a new series that replaces missing values by interpolating between adjacent non-NULL values and extrapolating any leading or trailing null values.

When using this function, the input is assumed to be equidistant. The SERIES definition must be equidistant.

When only null values are contained in the input, only null values appear in the output unless the null values are replaced by an extrapolation via the EXTRAPOLATION_CONSTANT mode.

When using the SERIES syntax, this function uses the properties in the following table to perform linear approximation.

Property	Description
NON-EQUIDISTANT	An error occurs if the series is non-equidistant.
MISSING ELEMENTS ALLOWED	When specified, there must be exactly one ORDER BY column that is compatible with the type of the series period column and the INCREMENT BY value.
PARTITION BY	If not specified, the SERIES KEY property of the SERIES syntax is used to construct the default PARTITION BY.
ORDER BY	If not specified, the first PERIOD column is added as an ORDER BY.

Other SERIES properties, such as MINVALUE and MAXVALUE, are ignored.

Examples

Linear approximation without a series definition

If the SERIES definition were not specified in the example above, then the result would have been different. For the same SparseApproxTable, as populated above, the example below does not specify a series. The example below returns [-1, 0, 1, 2, 3, 4, 5, 6].

```
SELECT LINEAR_APPROX(val, 'EXTRAPOLATION_LINEAR') OVER (PARTITION BY ts_id ORDER BY DATE) AS approx
 FROM "SparseApproxTable";
```

In the example query above, the series is assumed to be dense and equidistant, so the date column is not taken into consideration when calculating the approximations.

Linear approximation of sparse series data

In the example below, the INCREMENT BY INTERVAL is set to 1 MONTH, meaning that at most one row is expected for each month. Apart from the null values, values are missing for several months in the table. The LINEAR_APPROX function always replaces the null values and does not insert new lines for missing values. However, if the SERIES definition is specified in the SELECT statement, then LINEAR_APPROX considers the missing months when calculating the slope between two non-null values.

```
CREATE COLUMN TABLE "SparseApproxTable" (ts_id NVARCHAR(20), date DAYDATE, val DOUBLE);

INSERT INTO "SparseApproxTable" VALUES('A','2013-11-01', null);
INSERT INTO "SparseApproxTable" VALUES('A','2014-01-01', null);
INSERT INTO "SparseApproxTable" VALUES('A','2014-02-05', 2);
INSERT INTO "SparseApproxTable" VALUES('A','2014-03-07', null);
INSERT INTO "SparseApproxTable" VALUES('A','2014-05-01', 5);
INSERT INTO "SparseApproxTable" VALUES('A','2014-07-27', 7);
INSERT INTO "SparseApproxTable" VALUES('A','2014-12-07', null);
INSERT INTO "SparseApproxTable" VALUES('A','2015-02-07', null);

SELECT LINEAR_APPROX(val, 'EXTRAPOLATION_LINEAR') OVER (SERIES (
    SERIES KEY(ts_id) EQUIDISTANT INCREMENT BY INTERVAL 1 MONTH
    MISSING ELEMENTS ALLOWED PERIOD FOR SERIES(date))
  PARTITION BY ts_id) AS approximated_value
 FROM "SparseApproxTable";
```

APPROXIMATED_VALUE
-1
1
2
3
5
7
9.666666666666666
11

Linear approximation of series data

```
CREATE COLUMN TABLE "InterpolationTable" (TS_ID NVARCHAR(20), date DAYDATE, val DOUBLE);
INSERT INTO "InterpolationTable" VALUES('A','2013-09-30', 1);
INSERT INTO "InterpolationTable" VALUES('A','2013-10-01', 2);
INSERT INTO "InterpolationTable" VALUES('A','2013-10-02', null);
INSERT INTO "InterpolationTable" VALUES('A','2013-10-03', 10);

SELECT date,
  val,
  LINEAR_APPROX (val, 'EXTRAPOLATION_LINEAR') OVER (PARTITION BY TS_ID ORDER BY date) AS LINEAR_APP
 FROM "InterpolationTable";
```

DATE	VAL	MYRESULT
Sep 30, 2013	1	1

DATE	VAL	MYRESULT
Oct 1, 2013	2	2
Oct 2, 2013	?	6
Oct 3, 2013	10	10

Related Information

Expressions
Window Functions and the Window Specification

LN Function (Numeric)

Returns the natural logarithm of a number.

Syntax

```
LN(<number>)
```

Description

Returns the natural logarithm of the specified number.

Example

This example returns the natural logarithm of the number 9, which is `2.1972245773362196`:

```
SELECT LN (9) "ln" FROM DUMMY;
```

Related Information

Numeric Data Types

LOCALTOUTC Function (Datetime)

A timestamp parameter holding the time to be converted between UTC and local time.

Syntax

```
LOCALTOUTC (<time> [, <timezone> [, <timezone_dataset>]])
```

Syntax Elements

<time>

Specifies the time to be converted between UTC and local time.

```
<time> ::= <timestamp>
```

<timezone>

Specifies the timezone defining the local time. For a list of available timezones, see the TIMEZONES system view. If the local timezone is not explicitly specified, then the local timezone of the SAP HANA system is used.

```
<timezone> ::= <string_literal>
```

<timezone_dataset>

Specifies the dataset in which to search for the given timezone. If specified, this must be set to `platform`, which searches in the dataset provided by the operating system.

```
<timezone_dataset> ::= platform
```

Description

Converts the local time *<time>* from a timezone to the UTC(GMT) time.

The usage of local timestamps is discouraged; use UTC times instead. The use of local times or conversion between local time zones might require additional handling in the application code.

Examples

The following example returns the value `2012-01-01 06:00:00.0` for the UTC date and time:

```
SELECT LOCALTOUTC (TO_TIMESTAMP('2012-01-01 01:00:00', 'YYYY-MM-DD HH24:MI:SS'), 'EST') "localtoutc
```

LOCATE Function (String)

Returns the position of a substring within a string.

Syntax

```
LOCATE( <haystack>, <needle>, [<start_position>] , [ <occurrences> ] )
```

Description

The LOCATE function returns the position of a substring *<needle>* within a string *<haystack>*.

- If *<needle>* is not found within *<haystack>*, or if *<occurrences>* is set to less than 1, then 0 is returned.
- If *<haystack>*, *<needle>*, or *<occurences>* is (explicitly) NULL, then NULL is returned.
- If *<occurrences>* is not specified, then the first matched position is returned. A setting of 1 for *<occurrences>* is the same as not specifying it, while a setting of ***n*** returns the ***n***th match (or 0 if there are no more matches found).
- If *<start_position>* is a positive integer, or 0, or not specified, then the matching direction proceeds from left to right.
- If *<start_position>* is negative, then matching starts at the end of *<haystack>* and proceeds in the reverse direction (right to left). For example:

Statement	Returns
`SELECT LOCATE('AAA', 'A', -1) FROM "DUMMY";`	3 - the position of the last A
`SELECT LOCATE('AAA', 'A', -2) FROM "DUMMY";`	2 - the position of the second to last A
`SELECT LOCATE('AAA', 'A', -3) FROM "DUMMY";`	1 - the position of the third to last A
`SELECT LOCATE('AAA', 'A', -4) FROM "DUMMY";`	0 - not found
`SELECT LOCATE('ABABAC', 'A', -2) FROM "DUMMY";`	5 - the position of the first A starting from the second last position

- If a match is found, then the match position returned is always a positive number, even when *<start_position>* is a negative number.

Examples

The following example returns `1` because *<needle>* is an empty string.

```
SELECT LOCATE ('length in char', '') "locate" FROM DUMMY;
```

The following example returns the starting position (`1`) of `length` in the string `length in char`:

```
SELECT LOCATE ('length in char', 'length') "locate" FROM DUMMY;
```

The following example returns `0` because the search pattern `zchar` cannot be found in the given string:

```
SELECT LOCATE ('length in char', 'zchar') "locate" FROM DUMMY;
```

Related Information

Character String Data Types

LOCATE_REGEXPR Function (String)

Searches a string for a regular expression pattern and returns an integer indicating the beginning position, or the ending position plus 1, of one occurrence of the matched substring.

Syntax

```
LOCATE_REGEXPR( [ <regex_position_start_or_after> ] <pattern>
   [  FLAG <flag> ]
   IN <regex_subject_string>
   [ FROM <start_position> ]
   [ OCCURRENCE <regex_occurrence> ]
   [ GROUP <regex_capture_group> ]
)
```

Syntax Elements

<regex_position_start_or_after>

Searches a string for a regular expression pattern and returns an integer indicating the beginning position, or the ending position plus 1, of one occurrence of the matched substring.

```
<regex_position_start_or_after> ::= START | AFTER
```

If *<regex_position_start_or_after>* is not specified, then START is implicit.

<pattern>

A search pattern based on Perl Compatible Regular Expression (PCRE).

<flag>

The matching behavior of the function can be defined by the *<flag>* literal. The following options are available:

```
<flag> ::= 'i' | 'm' | 's' | 'x'
```

Flag options

Flag option	Description
i	Enables case-insensitive matching
m	Enables multiline mode, where the *<subject_string>* will be treated as multiple lines and the expression ^ and $ match just after or just before, respectively, a line terminator or the end of the input sequence
s	Enables the expression *<.>* as a wildcard to match any character, including a line terminator
x	Permits whitespace and comments in the pattern
U	SAP HANA uses the Perl-compatible Regular Expressions (PCRE) library to process regular expressions. Specifying 'U' (short for "ungreedy") inverts the "greediness" of quantifiers so that they are not greedy by default but become greedy only when followed by "?". Ungreedy matching can often perform faster because it finds the shorter match at times when it is only interesting to know whether there is any match. For a full understanding of the "greedy" versus "ungreedy" matching behavior of the Perl-compatible Regular Expressions (PCRE) library, visit:https://www.pcre.org/original/doc/html/pcrematching.html .

<regex_subject_string>

```
<regex_subject_string> ::= <string>
```

Specifies a string in which to search for the regular expression pattern.

<start_position>

The *<start_position>* parameter is a positive integer and indicates the character of *<regex_subject_string>* where the search is started. If *<start_position>* is not a positive integer, then 0 is returned.

```
<start_position> ::= <positive_integer>
```

<regex_occurrence>

Specifies a positive integer to the occurrence of the *<pattern>* in *<regex_subject_string>*. The default is 1.

```
<regex_occurrence> ::= <integer>
```

If *<regex_occurrence>* is not a positive integer, then 0 is returned.

<regex_capture_group>

The *<regex_capture_group>* parameter is a non-negative integer and indicates the number of the captured substring's group by the regular expression. The default is 0.

```
<regex_capture_group> ::= <integer>
```

If *<regex_capture_group>* is a negative integer, then 0 is returned.

Description

Searches a string for a regular expression pattern and returns an integer indicating the beginning position, or the ending position plus 1, of one occurrence of the matched substring.

If any of the following parameters is `NULL`: *<pattern>*, *<flag>*, *<regex_subject_string>*, , *<start_position>*, *<regex_occurrence>* or *<regex_capture_group>*, then the function returns NULL.

Example

This example returns the start position of the day part from the date value 20140401; it returns 7:

```
SELECT LOCATE_REGEXPR(START '([[:digit:]]{4})([[:digit:]]{2})([[:digit:]]{2})' IN '20140401' GROUP
```

Related Information

Character String Data Types

LOG Function (Numeric)

Returns the natural logarithm of a specified number and base.

Syntax

```
LOG(<base>, <number>)
```

Description

Returns the natural logarithm of a number specified by *<number>* and a base specified by *<base>*, where *<base>* must be a positive value other than 1, and *<number>* must be any positive value.

Example

The following example returns the natural logarithm for 2 base 10, which is `0.30102999566398114`:

```
SELECT LOG (10, 2) "log" FROM DUMMY;
```

Related Information

Numeric Data Types

LOWER Function (String)

Converts all characters in a string to lowercase.

Syntax

```
LOWER(<string>)
```

Description

Converts all characters in *<string>* to lowercase.

The LOWER function is identical to the LCASE function.

Example

This example converts the given string AnT to lowercase, and returns the value `ant`:

```
SELECT LOWER ('AnT') "lower" FROM DUMMY;
```

Related Information

LCASE Function (String)
Character String Data Types

LPAD Function (String)

Left-pads a string with spaces, or a specified pattern, to make a string of a specified number of characters in length.

Syntax

```
LPAD(<string>, <number> [, <pattern>])
```

Syntax Elements

<string>

Specifies a string to be padded.

<number>

Specifies the length to which to pad *<string>*. *<number>* must be an integer.

<pattern>

Specifies a string of characters to use for padding instead of spaces.

Description

Left-pads the end of *<string>* with spaces to make a string of *<number>* characters. If *<pattern>* is specified, then *<string>* is padded using sequences of the given characters until the required length is met.

If the length of *<string>* is greater than *<number>*, then no padding is performed and the resulting value is truncated from the right side to the length specified in *<number>*.

LPAD returns an empty string value if *<number>* is less than 1.

Example

The following example left-pads the start of string `end` with the pattern `12345` to make a string of `15` characters in length, and returns the value `123451234512end`:

```
SELECT LPAD ('end', 15, '12345') "lpad" FROM DUMMY;
```

In the following example, *<string>* is longer than *<n>*, so no padding is performed and the result is *<string>* truncated to the length of *<number>* (that is, `en`):

```
SELECT LPAD ('end', 2, '12345') "lpad" FROM DUMMY;
```

Related Information

Character String Data Types

LTRIM Function (String)

Returns a string, trimmed of all leading spaces.

Syntax

```
LTRIM(<string> [, <remove_set>])
```

Description

Returns string *<string>*, trimmed of all leading spaces. If *<remove_set>* is specified, LTRIM removes all the characters contained in this set from the start of string *<string>*. This process continues until a character that is not in *<remove_set>* is reached.

<remove_set> is treated as a set of characters and not as a search string.

Example

This example removes all leading `a` and `b` characters from the given string and returns the value `Aabend`:

```
SELECT LTRIM ('bababAabend','ab') "ltrim" FROM DUMMY;
```

Related Information

Character String Data Types

MAX Function (Aggregate)

Returns the maximum value of the expression. This function can also be used as a window function.

Syntax

Aggregate function:

```
MAX( [ ALL | DISTINCT ] <expression> )
```

Window function:

```
MAX( <expression> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the input data for the function.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Result type based on input

TINYINT	SMALLINT	INTEGER	BIGINT	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE
TINYINT	SMALLINT	INTEGER	BIGINT	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE

Example

The following statements create a table with data for example purposes:

```
DROP TABLE "MyProducts";
CREATE COLUMN TABLE "MyProducts"(
  "Product_ID" NVARCHAR(10),
  "Product_Name" NVARCHAR(100),
  "Category" NVARCHAR(100),
  "Quantity" INTEGER,
  "Price" DECIMAL(10,2),
  PRIMARY KEY ("Product_ID") );

INSERT INTO "MyProducts" VALUES('P1','Shirts', 'Clothes', 32, 20.99);
INSERT INTO "MyProducts" VALUES('P2','Jackets', 'Clothes', 16, 99.49);
INSERT INTO "MyProducts" VALUES('P3','Trousers', 'Clothes', 30, 32.99);
INSERT INTO "MyProducts" VALUES('P4','Coats', 'Clothes', 5, 129.99);
INSERT INTO "MyProducts" VALUES('P5','Purse', 'Accessories', 3, 89.49);
```

The following example returns `129.99`, the maximum price of the products in the MyProducts table:

```
SELECT MAX("Price") FROM "MyProducts";
```

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
Expressions

MEDIAN Function (Aggregate)

Finds the statistical median of an input expression with a numeric data type. This function can also be used as a window function.

Syntax

Aggregate function:

```
MEDIAN( <expression> )
```

Window function:

```
MEDIAN( <expression> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the input data expression for the MEDIAN function.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

The MEDIAN function finds the statistical median of an input expression with a numeric data type. Although it is grouped with the aggregate functions, its optional OVER clause positions it as a windows function as well.

Null values are eliminated. If there is an even number of elements, then the average of the two middle elements is returned. Otherwise, the middle element is returned.

The result type is the type that is selected for the expression "x/2" for an x value of the input data type.

Examples

Median of integer input The following example returns a median value of 2.

```
CREATE TABLE T (class NVARCHAR(10), date DAYDATE, val INT);
INSERT INTO T VALUES('A', '01.01.1972', 1);
INSERT INTO T VALUES('A', '02.01.1972', 3);
INSERT INTO T VALUES('A', '03.01.1972', null);
INSERT INTO T VALUES('A', '04.01.1972', 2);

SELECT MEDIAN(val) "median value" FROM T;
```

If the number of non-null values is even, then the average of the two middle values is returned. For the following example, the average of 2 and 3 is returned. Since the input and output types are the same, the integer is rounded. The returned result is 3.

```
INSERT INTO T VALUES('A', '05.01.1972', 4);
SELECT MEDIAN(val) "median value" FROM T;
```

Median of double input The following example uses double values instead of integers. The returned result is `2.5`.

```
CREATE TABLE T (TS_ID NVARCHAR(10), date DAYDATE, val DOUBLE);
INSERT INTO T VALUES('A', '01.01.1972', 1.0);
INSERT INTO T VALUES('A', '02.01.1972', 3.0);
INSERT INTO T VALUES('A', '03.01.1972', null);
INSERT INTO T VALUES('A', '04.01.1972', 2.0);
INSERT INTO T VALUES('A', '05.01.1972', 4.0);

SELECT MEDIAN(val) "median value" FROM T;
```

Median as a window function The following example uses double values instead of integers.

```
CREATE TABLE T (TS_ID NVARCHAR(10), date DAYDATE, val DOUBLE);
INSERT INTO T VALUES('A', '01.01.1972', 1.0);
INSERT INTO T VALUES('A', '02.01.1972', 3.0);
INSERT INTO T VALUES('A', '03.01.1972', null);
INSERT INTO T VALUES('A', '04.01.1972', 2.0);
INSERT INTO T VALUES('A', '04.01.1972', 4.0);

SELECT MEDIAN(val) OVER (PARTITION BY TS_ID ) AS WF1 FROM T;
```

The returned result is:

WF1
2.5
2.5
2.5
2.5
2.5

Median of sliding window (GROUPS BETWEEN) Both of the SELECT statements in the following example produce identical results.

```
CREATE TABLE T (TS_ID NVARCHAR(10), date DAYDATE, val DOUBLE);
INSERT INTO T VALUES('A', '01.01.1972', 1.0);
INSERT INTO T VALUES('A', '02.01.1972', 3.0);
INSERT INTO T VALUES('A', '03.01.1972', null);
INSERT INTO T VALUES('A', '04.01.1972', 2.0);
INSERT INTO T VALUES('A', '04.01.1972', 4.0);

SELECT MEDIAN(val) OVER (PARTITION BY TS_ID ORDER BY date) AS WF2A FROM T;
SELECT MEDIAN(val) OVER (PARTITION BY TS_ID ORDER BY date GROUPS BETWEEN UNBOUNDED PRECEDING AND CU
```

The returned result is:

WF2B
1

WF2B
1
2
2
2
2
2.5
2.5
2.5
2.5

Median of sliding window (ROWS BETWEEN) The following example uses ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

```
CREATE TABLE T (TS_ID NVARCHAR(10), date DAYDATE, val DOUBLE);
INSERT INTO T VALUES('A', '01.01.1972', 1.0);
INSERT INTO T VALUES('A', '02.01.1972', 3.0);
INSERT INTO T VALUES('A', '03.01.1972', null);
INSERT INTO T VALUES('A', '04.01.1972', 2.0);
INSERT INTO T VALUES('A', '04.01.1972', 4.0);

SELECT MEDIAN(val) OVER (PARTITION BY TS_ID ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURR
```

The returned result is:

WF3
1
1
1
1
1
2
2
2
2
2
2.5
2
2.5
2
2.5

Related Information

Window Functions and the Window Specification
Window Aggregate Functions
Aggregate Functions
Expressions

MEMBER_AT Function (Array)

Returns values from a specified array position.

Syntax

```
MEMBER_AT(<array_value_expression>, <position> [, <default_value>])
```

Description

Accesses an array element at the specified ordinal position and returns the value.

<default_value> indicates the value to return if *<position>* is greater than the cardinality of the *<array_value_expression>*. If *<default_value>* is not specified, and *<position>* is greater than the cardinality of the *<array_value_expression>*. then NULL is returned.

Example

The following example creates a table and inserts arrays into it:

```
CREATE COLUMN TABLE ARRAY_TEST (IDX INT, VAL INT ARRAY);
INSERT INTO ARRAY_TEST VALUES (1, ARRAY(1, 2, 3));
INSERT INTO ARRAY_TEST VALUES (2, ARRAY(10, 20, 30, 40));
```

The following example returns the value in position 4 of each array in the table. The first array does not have a position 4, so NULL is returned:

```
SELECT MEMBER_AT(VAL, 4) "member_at" FROM ARRAY_TEST;
```

member_at
?
40

The following example returns the value in position 4 of each array in the table. The first array does not have a position 4, so 1 is returned *<position>* is greater than the cardinality of the array:

```
SELECT MEMBER_AT(VAL, 4, 1) "member_at" FROM ARRAY_TEST;
```

member_at
1

member_at
40

Related Information

Expressions

MIN Function (Aggregate)

Returns the minimum value of the expression. This function can also be used as a window function.

Syntax

Aggregate function:

```
MIN( [ ALL | DISTINCT ] <expression> )
```

Window function:

```
MIN( <expression> ) <window_specification>
```

Syntax Elements

<expression>

Specifies the input data for the function.

<window_specification>

Defines a window on the data over which the function operates. For *<window_specification>*, see Window Functions and the Window Specification.

Description

Result type based on input

TINYINT	**SMALLINT**	**INTEGER**	**BIGINT**	**DECIMAL(p,s)**	**DECIMAL**	**REAL**	**DOUBLE**
TINYINT	SMALLINT	INTEGER	BIGINT	DECIMAL(p,s)	DECIMAL	REAL	DOUBLE

Example

The following statements create a table with data for example purposes:

```
DROP TABLE "MyProducts";
CREATE COLUMN TABLE "MyProducts"(
  "Product_ID" NVARCHAR(10),
  "Product_Name" NVARCHAR(100),
  "Category" NVARCHAR(100),
  "Quantity" INTEGER,
  "Price" DECIMAL(10,2),
```

```
   PRIMARY KEY ("Product_ID") );

INSERT INTO "MyProducts" VALUES('P1','Shirts', 'Clothes', 32, 20.99);
INSERT INTO "MyProducts" VALUES('P2','Jackets', 'Clothes', 16, 99.49);
INSERT INTO "MyProducts" VALUES('P3','Trousers', 'Clothes', 30, 32.99);
INSERT INTO "MyProducts" VALUES('P4','Coats', 'Clothes', 5, 129.99);
INSERT INTO "MyProducts" VALUES('P5','Purse', 'Accessories', 3, 89.49);
```

The following example returns `20.99`, the minimum price of the products in the MyProducts table:

```
SELECT MIN("Price") FROM "MyProducts";
```

Related Information

Window Functions and the Window Specification

Window Aggregate Functions

Aggregate Functions

Expressions

MINUTE Function (Datetime)

Returns an integer representation of the minute for the specified time.

Syntax

```
MINUTE(<time>)
```

Description

Returns an integer representation of the minute for time *<time>*.

Example

The following example returns the value `34` as the minute for the specified time:

```
SELECT MINUTE ('12:34:56') "minute" FROM DUMMY;
```

Related Information

Datetime Data Types

MOD Function (Numeric)

Returns the remainder of a specified number divided by a specified divisor.

Syntax

```
MOD(<number>, <divisor>)
```

Description

Returns the remainder of a number *<number>* divided by a divisor *<divisor>*.

When *<number>* is negative, this function acts differently to the standard computational modulo operation. The following list shows examples of what the MOD function returns as the result:

- If *<divisor>* is zero, then *<number>* is returned.
- If *<number>* is greater than 0 and *<number>* is less than *<divisor>*, then *<number>* is returned.
- If *<number>* is less than 0 and *<number>* is greater than *<divisor>*, then *<number>* is returned.
- In cases other than those mentioned above, the remainder of the absolute value of *<number>* divided by the absolute value of *<divisor>* is used to calculate the remainder. If *<number>* is less than 0, then the returned remainder from MOD is a negative number, and if *<number>* is greater than 0, then the returned remainder from MOD is a positive number.

Example

The following example returns the value 3:

```
SELECT MOD (15, 4) "modulus" FROM DUMMY;
```

The following example returns the value -3:

```
SELECT MOD (-15, 4) "modulus" FROM DUMMY;
```

Related Information

Numeric Data Types

STRING_AGG Function (Aggregate)

Returns the concatenation string of the specified expression.

Syntax

```
STRING_AGG( <expression>[, <delimiter> ] [ <order_by_clause> ] )
```

Syntax Elements

<expression>

Specifies an NVARCHAR expression with values to be concatenated. If the input values are a different data type than NVARCHAR, then implicit casting is attempted.

For example, if the NUM column has five integer values (1, 2, 3, 4, 5), then STRING_AGG("NUM",0) returns 102030405.

<delimiter>

Specifies the character to use as a delimiter when aggregating values.

<order_by_clause>

Specifies the sort order of the input rows.

```
<order_by_clause> ::= ORDER BY <order_by_expression> [, <order_by_expression> [,...] ]

<order_by_expression> ::=
 <column_name> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <column_position> [ <collate_clause> ] [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<collate_clause> ::= COLLATE <collation_name>
```

<collate_clause> specifies the collation to use for ordering values in the results. *<collate_clause>* can only be used on columns defined as NVARCHAR (or VARCHAR, which is a synonym for NVARCHAR).*<collation_name>* is one of the supported collation names listed in the COLLATIONS system view.

Description

NULL values are treated as empty strings.

The default ordering is ASC NULLS FIRST. If DESC is specified, then the ORDER BY expression becomes DESC NULLS LAST.

Example

The example below creates table r1 and populate it with data.

```
CREATE ROW TABLE r1(a INT, str NVARCHAR(20), grp INT);

INSERT INTO r1 VALUES (3,'str2',0);
INSERT INTO r1 VALUES (0,'str1',0);
INSERT INTO r1 VALUES (NULL,'NULL',0);
INSERT INTO r1 VALUES (5,'str3',0);
INSERT INTO r1 VALUES (3,'val3',1);
INSERT INTO r1 VALUES (6,'val6',1);
INSERT INTO r1 VALUES (NULL,'NULL',1);
INSERT INTO r1 VALUES (1,'val1',1);
```

Execute the following statement to return the concatenation string of each record from table r1 in ascending order.

```
SELECT grp, STRING_AGG(str,','ORDER BY a)AS agg FROM R1 GROUP BY grp;
```

The statement above returns the following results.

GRP	AGG
0	NULL,str1,str2,str3
1	NULL,val1,val3,val6

Execute the following statement to return the concatenation string of each record from table r1 in descending order.

```
SELECT grp, STRING_AGG(str,','ORDER BY a DESC) AS agg FROM r1 GROUP BY grp;
```

The statement above returns the following results.

GRP	AGG
0	str3,str2,str1,NULL
1	val6,val3,val1,NULL

Related Information

Aggregate Functions
COLLATIONS System View

TO_DOUBLE Function (Data Type Conversion)

Converts a value to a DOUBLE data type.

Syntax

```
TO_DOUBLE(<value>)
```

Description

Converts a specified *<value>* to a DOUBLE (double precision) data type.

Example

The following example converts 15.12 to a DOUBLE, and then multiplies the value by 3, returning the DOUBLE value 45.36.

```
SELECT 3*TO_DOUBLE ('15.12') "to double" FROM DUMMY;
```

Related Information

Data Type Conversion