SQL Reference

AWS Clean Rooms

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Clean Rooms: SQL Reference

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Overview of SQL in AWS Clean Rooms

Welcome to the *AWS Clean Rooms SQL Reference.*

AWS Clean Rooms is built around industry-standard Structured Query Language (SQL), a query language that consists of commands and functions that you use to work with databases and database objects. SQL also enforces rules regarding the use of data types, expressions, and literals.

The following topics provide general information about the conventions, naming rules, and data types:

Topics

- SQL reference conventions
- SQL naming rules
- Data types

To gain an understanding of the SQL commands, types of SQL functions, and SQL conditions you can use in AWS Clean Rooms, review the following topics:

- SQL commands in AWS Clean Rooms
- SQL functions in AWS Clean Rooms
- SQL conditions in AWS Clean Rooms

For more information about AWS Clean Rooms, see the AWS Clean Rooms User Guide and the AWS Clean Rooms API Reference.

SQL reference conventions

This section explains the conventions that are used to write the syntax for the SQL expressions, commands, and functions.

Character	Description
CAPS	Words in capital letters are key words.

Character	Description
[ ]	Brackets denote optional arguments. Multiple arguments in brackets indicate that you can choose any number of the arguments. In addition, arguments in brackets on separate lines indicate that the parser expects the arguments to be in the order that they are listed in the syntax.
{ }	Braces indicate that you are required to choose one of the arguments inside the braces.
|	Pipes indicate that you can choose between the arguments.
italics	Words in italics indicate placeholders. You must insert the appropriate value in place of the word in italics.
...	An ellipsis indicates that you can repeat the preceding element.
'	Words in single quotation marks indicate that you must type the quotes.

SQL naming rules

The following sections explain the SQL naming rules in AWS Clean Rooms.

Configured table association names and columns

Members who can query use configured table association names as table names in queries. Configured table association names and configured table columns can be aliased in queries.

The following naming rules apply to configured table association names, configured table column names, and aliases:

- They must use only alphanumeric, underscore (_), or hyphen (-) characters but cannot start or end with a hyphen.

- (*Custom analysis rule only*) They can use the dollar sign ($) but cannot use a pattern that follows a dollar-quoted string constant.

  A dollar-quoted string constant consists of:

  - a dollar sign ($)
  - an optional "tag" of zero or more characters
  - another dollar sign
  - arbitrary sequence of characters that makes up the string content
  - a dollar sign ($)
  - the same tag that began the dollar quote
  - a dollar sign

  For example: `$$invalid$$`
- They cannot contain consecutive hyphen (-) characters.
- They cannot begin with any of the following prefixes:

  `padb_`, `pg_`, `stcs_`, `stl_`, `stll_`, `stv_`, `svcs_`, `svl_`, `svv_`, `sys_`, `systable_`
- They cannot contain backslash characters (\) , quotation marks ('), or spaces that are not double-quoted.
- If they start with a non-alphabetical character, they must be within double-quotes (" ").
- If they contain a hyphen (-) character, they must be within double-quotes (" ").
- They must be between 1 and 127 characters in length.
- Reserved words must be within double-quotes (" ").
- The following column names are reserved cannot be used in AWS Clean Rooms (even with quotes):
  - oid
  - tableoid
  - xmin
  - cmin
  - xmax
  - cmax
  - ctid

Literals

A literal or constant is a fixed data value, composed of a sequence of characters or a numeric constant.

The following naming rules are for literals in AWS Clean Rooms:

- Numeric, character and date, time, and timestamp literals are supported.
- Only TAB, CARRIAGE RETURN (CR), and LINE FEED (LF) Unicode control characters from the Unicode general category (Cc) are supported.
- Direct references to literals in the projection list are not supported in the SELECT statement.

  For example:

```
SELECT 'test', consumer.first_purchase_day
FROM consumer
INNER JOIN provider2
ON consumer.hashed_email = provider2.hashedemail
```

Reserved words

The following is a list of reserved words in AWS Clean Rooms.

AES128	DELTA32KDESC	LEADING	PRIMARY
AES256ALL	DISTINCT	LEFTLIKE	RAW
ALLOWOVER WRITEANALYSE	DO	LIMIT	READRATIO
ANALYZE	DISABLE	LOCALTIME	RECOVERRE FERENCES
AND	ELSE	LOCALTIMESTAMP	REJECTLOG
ANY	EMPTYASNU LLENABLE	LUN	RESORT
ARRAY	ENCODE	LUNS	RESPECT

AS	ENCRYPT	LZO	RESTORE
ASC	ENCRYPTIONEND	LZOP	RIGHTSELECT
AUTHORIZATION	EXCEPT	MINUS	SESSION_USER
AZ64	EXPLICITFALSE	MOSTLY16	SIMILAR
BACKUPBETWEEN	FOR	MOSTLY32	SNAPSHOT
BINARY	FOREIGN	MOSTLY8NATURAL	SOME
BLANKSASN ULLBOTH	FREEZE	NEW	SYSDATESYSTEM
BYTEDICT	FROM	NOT	TABLE
BZIP2CASE	FULL	NOTNULL	TAG
CAST	GLOBALDICT256	NULL	TDES
CHECK	GLOBALDIC T64KGRANT	NULLSOFF	TEXT255
COLLATE	GROUP	OFFLINEOFFSET	TEXT32KTHEN
COLUMN	GZIPHAVING	OID	TIMESTAMP
CONSTRAINT	IDENTITY	OLD	TO
CREATE	IGNOREILIKE	ON	TOPTRAILING
CREDENTIALSCROSS	IN	ONLY	TRUE
CURRENT_DATE	INITIALLY	OPEN	TRUNCATEC OLUMNSUNION
CURRENT_TIME	INNER	OR	UNIQUE
CURRENT_T IMESTAMP	INTERSECT	ORDER	UNNEST

CURRENT_USER	INTERVAL	OUTER	USING
CURRENT_USER_IDDEFAULT	INTO	OVERLAPS	VERBOSE
DEFERRABLE	IS	PARALLELPARTITION	WALLETWHEN
DEFLATE	ISNULL	PERCENT	WHERE
DEFRAG	JOIN	PERMISSIONS	WITH
DELTA	LANGUAGE	PIVOTPLACING	WITHOUT

Data types

Each value that AWS Clean Rooms stores or retrieves has a data type with a fixed set of associated properties. Data types are declared when tables are created. A data type constrains the set of values that a column or argument can contain.

The following table lists the data types that you can use in AWS Clean Rooms tables.

Data type	Aliases	Description
ARRAY	Not applicable	Array nested data type
BIGINT	Not applicable	Signed eight-byte integer
BOOLEAN	BOOL	Logical Boolean (true/false)
CHAR	CHARACTER	Fixed-length character string
DATE	Not applicable	Calendar date (year, month, day)
DECIMAL	NUMERIC	Exact numeric of selectable precision
DOUBLE PRECISION	FLOAT8, FLOAT	Double precision floating-point number

Data type	Aliases	Description
INTEGER	INT	Signed four-byte integer
MAP	Not applicable	Map nested data type
REAL	FLOAT4	Single precision floating-point number
SMALLINT	Not applicable	Signed two-byte integer
STRUCT	Not applicable	Struct nested data type
SUPER	Not applicable	Superset data type that encompasses all scalar types of AWS Clean Rooms including complex types such as ARRAY and STRUCTS.
TIME	Not applicable	Time of day
TIMETZ	Not applicable	Time of day with time zone
VARBYTE	VARBINARY, BINARY VARYING	Variable-length binary value
VARCHAR	CHARACTER VARYING	Variable-length character string with a user-defined limit

Note

The ARRAY, STRUCT, and MAP nested data types are currently only enabled for the custom analysis rule. For more information, see Nested type.

Multibyte characters

The VARCHAR data type supports UTF-8 multibyte characters up to a maximum of four bytes. Five-byte or longer characters are not supported. To calculate the size of a VARCHAR column that

contains multibyte characters, multiply the number of characters by the number of bytes per character. For example, if a string has four Chinese characters, and each character is three bytes long, then you will need a VARCHAR(12) column to store the string.

The VARCHAR data type doesn't support the following invalid UTF-8 codepoints:

`0xD800 – 0xDFFF` (Byte sequences: `ED A0 80` – `ED BF BF`)

The CHAR data type doesn't support multibyte characters.

Numeric types

Topics

Numeric data types include integers, decimals, and floating-point numbers.

Integer types

Use the SMALLINT, INTEGER, and BIGINT data types to store whole numbers of various ranges. You can't store values outside of the allowed range for each type.

Name	Storage	Range
SMALLINT	2 bytes	-32768 to +32767
INTEGER or INT	4 bytes	-2147483648 to +2147483647
BIGINT	8 bytes	-92233720 36854775808 to 922337203 6854775807

DECIMAL or NUMERIC type

Use the DECIMAL or NUMERIC data type to store values with a *user-defined precision*. The DECIMAL and NUMERIC keywords are interchangeable. In this document, *decimal* is the preferred term for this data type. The term *numeric* is used generically to refer to integer, decimal, and floating-point data types.

Storage	Range
Variable, up to 128 bits for uncompressed DECIMAL types.	128-bit signed integers with up to 38 digits of precision.

Define a DECIMAL column in a table by specifying a *precision* and *scale*:

```
decimal(precision, scale)
```

precision

The total number of significant digits in the whole value: the number of digits on both sides of the decimal point. For example, the number `48.2891` has a precision of 6 and a scale of 4. The default precision, if not specified, is 18. The maximum precision is 38.

If the number of digits to the left of the decimal point in an input value exceeds the precision of the column minus its scale, the value can't be copied into the column (or inserted or updated). This rule applies to any value that falls outside the range of the column definition. For example, the allowed range of values for a `numeric(5,2)` column is `-999.99` to `999.99`.

scale

The number of decimal digits in the fractional part of the value, to the right of the decimal point. Integers have a scale of zero. In a column specification, the scale value must be less than or equal to the precision value. The default scale, if not specified, is 0. The maximum scale is 37.

If the scale of an input value that is loaded into a table is greater than the scale of the column, the value is rounded to the specified scale. For example, the PRICEPAID column in the SALES table is a DECIMAL(8,2) column. If a DECIMAL(8,4) value is inserted into the PRICEPAID column, the value is rounded to a scale of 2.

```
insert into sales
```

```
values (0, 8, 1, 1, 2000, 14, 5, 4323.8951, 11.00, null);

select pricepaid, salesid from sales where salesid=0;

pricepaid | salesid
-----------+---------
4323.90 |       0
(1 row)
```

However, results of explicit casts of values selected from tables are not rounded.

Note

The maximum positive value that you can insert into a DECIMAL(19,0) column is 9223372036854775807 (2^{63} -1). The maximum negative value is -9223372036854775807. For example, an attempt to insert the value 9999999999999999999 (19 nines) will cause an overflow error. Regardless of the placement of the decimal point, the largest string that AWS Clean Rooms can represent as a DECIMAL number is 9223372036854775807. For example, the largest value that you can load into a DECIMAL(19,18) column is 9.223372036854775807.
These rules are because of the following:

- DECIMAL values with 19 or fewer significant digits of precision are stored internally as 8-byte integers.
- DECIMAL values with 20 to 38 significant digits of precision are stored as 16-byte integers.

Notes about using 128-bit DECIMAL or NUMERIC columns

Do not arbitrarily assign maximum precision to DECIMAL columns unless you are certain that your application requires that precision. 128-bit values use twice as much disk space as 64-bit values and can slow down query execution time.

Floating-point types

Use the REAL and DOUBLE PRECISION data types to store numeric values with *variable precision*. These types are *inexact* types, meaning that some values are stored as approximations, such that

storing and returning a specific value may result in slight discrepancies. If you require exact storage and calculations (such as for monetary amounts), use the DECIMAL data type.

REAL represents the single-precision floating point format, according to the IEEE Standard 754 for Floating-Point Arithmetic. It has a precision of about 6 digits, and a range of around 1E-37 to 1E +37. You can also specify this data type as FLOAT4.

DOUBLE PRECISION represents the double-precision floating point format, according to the IEEE Standard 754 for Binary Floating-Point Arithmetic. It has a precision of about 15 digits, and a range of around 1E-307 to 1E+308. You can also specify this data type as FLOAT or FLOAT8.

Computations with numeric values

In AWS Clean Rooms, *computation* refers to binary mathematical operations: addition, subtraction, multiplication, and division. This section describes the expected return types for these operations, as well as the specific formula that is applied to determine precision and scale when DECIMAL data types are involved.

When numeric values are computed during query processing, you might encounter cases where the computation is impossible and the query returns a numeric overflow error. You might also encounter cases where the scale of computed values varies or is unexpected. For some operations, you can use explicit casting (type promotion) or AWS Clean Rooms configuration parameters to work around these problems.

For information about the results of similar computations with SQL functions, see SQL functions in AWS Clean Rooms.

Return types for computations

Given the set of numeric data types supported in AWS Clean Rooms, the following table shows the expected return types for addition, subtraction, multiplication, and division operations. The first column on the left side of the table represents the first operand in the calculation, and the top row represents the second operand.

	SMALLINT	INTEGER	BIGINT	DECIMAL	FLOAT4	FLOAT8
SMALLINT	SMALLINT	INTEGER	BIGINT	DECIMAL	FLOAT8	FLOAT8
INTEGER	INTEGER	INTEGER	BIGINT	DECIMAL	FLOAT8	FLOAT8

BIGINT	BIGINT	BIGINT	BIGINT	DECIMAL	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT8	FLOAT8
FLOAT4	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT4	FLOAT8
FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8	FLOAT8

Precision and scale of computed DECIMAL results

The following table summarizes the rules for computing resulting precision and scale when mathematical operations return DECIMAL results. In this table, p1 and s1 represent the precision and scale of the first operand in a calculation and p2 and s2 represent the precision and scale of the second operand. (Regardless of these calculations, the maximum result precision is 38, and the maximum result scale is 38.)

Operation	Result precision and scale
+ or -	Scale = `max(s1,s2)` Precision = `max(p1-s1,p2-s2)+1+scale`
*	Scale = `s1+s2` Precision = `p1+p2+1`
/	Scale = `max(4,s1+p2-s2+1)` Precision = `p1-s1+ s2+scale`

For example, the PRICEPAID and COMMISSION columns in the SALES table are both DECIMAL(8,2) columns. If you divide PRICEPAID by COMMISSION (or vice versa), the formula is applied as follows:

```
Precision = 8-2 + 2 + max(4,2+8-2+1)
= 6 + 2 + 9 = 17

Scale = max(4,2+8-2+1) = 9
```

```
Result = DECIMAL(17,9)
```

The following calculation is the general rule for computing the resulting precision and scale for operations performed on DECIMAL values with set operators such as UNION, INTERSECT, and EXCEPT or functions such as COALESCE and DECODE:

```
Scale = max(s1,s2)
Precision = min(max(p1-s1,p2-s2)+scale,19)
```

For example, a DEC1 table with one DECIMAL(7,2) column is joined with a DEC2 table with one DECIMAL(15,3) column to create a DEC3 table. The schema of DEC3 shows that the column becomes a NUMERIC(15,3) column.

```
select * from dec1 union select * from dec2;
```

In the above example, the formula is applied as follows:

```
Precision = min(max(7-2,15-3) + max(2,3), 19)
= 12 + 3 = 15

Scale = max(2,3) = 3

Result = DECIMAL(15,3)
```

Notes on division operations

For division operations, divide-by-zero conditions return errors.

The scale limit of 100 is applied after the precision and scale are calculated. If the calculated result scale is greater than 100, division results are scaled as follows:

- Precision = `precision - (scale - max_scale)`
- Scale = `max_scale`

If the calculated precision is greater than the maximum precision (38), the precision is reduced to 38, and the scale becomes the result of: `max(38 + scale - precision), min(4, 100))`

Overflow conditions

Overflow is checked for all numeric computations. DECIMAL data with a precision of 19 or less is stored as 64-bit integers. DECIMAL data with a precision that is greater than 19 is stored as 128-bit integers. The maximum precision for all DECIMAL values is 38, and the maximum scale is 37. Overflow errors occur when a value exceeds these limits, which apply to both intermediate and final result sets:

- Explicit casting results in runtime overflow errors when specific data values do not fit the requested precision or scale specified by the cast function. For example, you can't cast all values from the PRICEPAID column in the SALES table (a DECIMAL(8,2) column) and return a DECIMAL(7,3) result:

  ```
  select pricepaid::decimal(7,3) from sales;
  ERROR:  Numeric data overflow (result precision)
  ```

  This error occurs because *some* of the larger values in the PRICEPAID column can't be cast.
- Multiplication operations produce results in which the result scale is the sum of the scale of each operand. If both operands have a scale of 4, for example, the result scale is 8, leaving only 10 digits for the left side of the decimal point. Therefore, it is relatively easy to run into overflow conditions when multiplying two large numbers that both have significant scale.

Numeric calculations with INTEGER and DECIMAL types

When one of the operands in a calculation has an INTEGER data type and the other operand is DECIMAL, the INTEGER operand is implicitly cast as a DECIMAL.

- SMALLINT is cast as DECIMAL(5,0)
- INTEGER is cast as DECIMAL(10,0)
- BIGINT is cast as DECIMAL(19,0)

For example, if you multiply SALES.COMMISSION, a DECIMAL(8,2) column, and SALES.QTYSOLD, a SMALLINT column, this calculation is cast as:

```
DECIMAL(8,2) * DECIMAL(5,0)
```

Character types

Character data types include CHAR (character) and VARCHAR (character varying).

Storage and ranges

CHAR and VARCHAR data types are defined in terms of bytes, not characters. A CHAR column can only contain single-byte characters, so a CHAR(10) column can contain a string with a maximum length of 10 bytes. A VARCHAR can contain multibyte characters, up to a maximum of four bytes per character. For example, a VARCHAR(12) column can contain 12 single-byte characters, 6 two-byte characters, 4 three-byte characters, or 3 four-byte characters.

Name	Storage	Range (width of column)
CHAR or CHARACTER	Length of string, including trailing blanks (if any)	4096 bytes
VARCHAR or CHARACTER VARYING	4 bytes + total bytes for character s, where each character can be 1 to 4 bytes.	65535 bytes (64K -1)

CHAR or CHARACTER

Use a CHAR or CHARACTER column to store fixed-length strings. These strings are padded with blanks, so a CHAR(10) column always occupies 10 bytes of storage.

```
char(10)
```

A CHAR column without a length specification results in a CHAR(1) column.

VARCHAR or CHARACTER VARYING

Use a VARCHAR or CHARACTER VARYING column to store variable-length strings with a fixed limit. These strings are not padded with blanks, so a VARCHAR(120) column consists of a maximum of

120 single-byte characters, 60 two-byte characters, 40 three-byte characters, or 30 four-byte characters.

```
varchar(120)
```

Significance of trailing blanks

Both CHAR and VARCHAR data types store strings up to *n* bytes in length. An attempt to store a longer string into a column of these types results in an error. However, if the extra characters are all spaces (blanks), the string is truncated to the maximum length. If the string is shorter than the maximum length, CHAR values are padded with blanks, but VARCHAR values store the string without blanks.

Trailing blanks in CHAR values are always semantically insignificant. They are disregarded when you compare two CHAR values, not included in LENGTH calculations, and removed when you convert a CHAR value to another string type.

Trailing spaces in VARCHAR and CHAR values are treated as semantically insignificant when values are compared.

Length calculations return the length of VARCHAR character strings with trailing spaces included in the length. Trailing blanks are not counted in the length for fixed-length character strings.

Datetime types

Datetime data types include DATE, TIME, TIMETZ, TIMESTAMP, and TIMESTAMPTZ.

Topics

Storage and ranges

Name	Storage	Range	Resolution
DATE	4 bytes	4713 BC to 294276 AD	1 day
TIME	8 bytes	00:00:00 to 24:00:00	1 microsecond
TIMETZ	8 bytes	00:00:00+1459 to 00:00:00+1459	1 microsecond
TIMESTAMP	8 bytes	4713 BC to 294276 AD	1 microsecond
TIMESTAMPTZ	8 bytes	4713 BC to 294276 AD	1 microsecond

DATE

Use the DATE data type to store simple calendar dates without timestamps.

TIME

Use the TIME data type to store the time of day.

TIME columns store values with up to a maximum of six digits of precision for fractional seconds.

By default, TIME values are Coordinated Universal Time (UTC) in both user tables and AWS Clean Rooms system tables.

TIMETZ

Use the TIMETZ data type to store the time of day with a time zone.

TIMETZ columns store values with up to a maximum of six digits of precision for fractional seconds.

By default, TIMETZ values are UTC in both user tables and AWS Clean Rooms system tables.

TIMESTAMP

Use the TIMESTAMP data type to store complete timestamp values that include the date and the time of day.

TIMESTAMP columns store values with up to a maximum of six digits of precision for fractional seconds.

If you insert a date into a TIMESTAMP column, or a date with a partial timestamp value, the value is implicitly converted into a full timestamp value. This full timestamp value has default values (00) for missing hours, minutes, and seconds. Time zone values in input strings are ignored.

By default, TIMESTAMP values are UTC in both user tables and AWS Clean Rooms system tables.

TIMESTAMPTZ

Use the TIMESTAMPTZ data type to input complete timestamp values that include the date, the time of day, and a time zone. When an input value includes a time zone, AWS Clean Rooms uses the time zone to convert the value to UTC and stores the UTC value.

To view a list of supported time zone names, run the following command.

```
select my_timezone_names();
```

To view a list of supported time zone abbreviations, run the following command.

```
select my_timezone_abbrevs();
```

You can also find current information about time zones in the IANA Time Zone Database.

The following table has examples of time zone formats.

Format	Example
dd mon hh:mi:ss yyyy tz	17 Dec 07:37:16 1997 PST
mm/dd/yyyy hh:mi:ss.ss tz	12/17/1997 07:37:16.00 PST
mm/dd/yyyy hh:mi:ss.ss tz	12/17/1997 07:37:16.00 US/Pacific
yyyy-mm-dd hh:mi:ss+/-tz	1997-12-17 07:37:16-08
dd.mm.yyyy hh:mi:ss tz	17.12.1997 07:37:16.00 PST

TIMESTAMPTZ columns store values with up to a maximum of six digits of precision for fractional seconds.

If you insert a date into a TIMESTAMPTZ column, or a date with a partial timestamp, the value is implicitly converted into a full timestamp value. This full timestamp value has default values (00) for missing hours, minutes, and seconds.

TIMESTAMPTZ values are UTC in user tables.

Examples with datetime types

The following examples show you how to work with datetime types that are supported by AWS Clean Rooms.

Date examples

The following examples insert dates that have different formats and display the output.

```
select * from datetable order by 1;

start_date |  end_date
-----------------------
2008-06-01 | 2008-12-31
2008-06-01 | 2008-12-31
```

If you insert a timestamp value into a DATE column, the time portion is ignored and only the date is loaded.

Time examples

The following examples insert TIME and TIMETZ values that have different formats and display the output.

```
select * from timetable order by 1;
start_time |  end_time
-----------------------
 19:11:19  | 20:41:19+00
 19:11:19  | 20:41:19+00
```

Time stamp examples

If you insert a date into a TIMESTAMP or TIMESTAMPTZ column, the time defaults to midnight. For example, if you insert the literal `20081231`, the stored value is `2008-12-31 00:00:00`.

The following examples insert timestamps that have different formats and display the output.

```
timeofday
--------------------
2008-06-01 09:59:59
2008-12-31 18:20:00
(2 rows)
```

Date, time, and timestamp literals

Following are rules for working with date, time, and timestamp literals that are supported by AWS Clean Rooms.

Dates

The following table shows input dates that are valid examples of literal date values that you can load into AWS Clean Rooms tables. The default `MDY DateStyle` mode is assumed to be in effect. This mode means that the month value precedes the day value in strings such as `1999-01-08` and `01/02/00`.

> **Note**
>
> A date or timestamp literal must be enclosed in quotation marks when you load it into a table.

Input date	Full date
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
01/02/00	January 2, 2000
2000-Jan-31	January 31, 2000
Jan-31-2000	January 31, 2000

Input date	Full date
31-Jan-2000	January 31, 2000
20080215	February 15, 2008
080215	February 15, 2008
2008.366	December 31, 2008 (the three-digit part of date must be between 001 and 366)

Times

The following table shows input times that are valid examples of literal time values that you can load into AWS Clean Rooms tables.

Input times	Description (of time part)
04:05:06.789	4:05 AM and 6.789 seconds
04:05:06	4:05 AM and 6 seconds
04:05	4:05 AM exactly
040506	4:05 AM and 6 seconds
04:05 AM	4:05 AM exactly; AM is optional
04:05 PM	4:05 PM exactly; the hour value must be less than 12
16:05	4:05 PM exactly

Timestamps

The following table shows input timestamps that are valid examples of literal time values that you can load into AWS Clean Rooms tables. All of the valid date literals can be combined with the following time literals.

Input timestamps (concatenated dates and times)	Description (of time part)
20080215 04:05:06.789	4:05 AM and 6.789 seconds
20080215 04:05:06	4:05 AM and 6 seconds
20080215 04:05	4:05 AM exactly
20080215 040506	4:05 AM and 6 seconds
20080215 04:05 AM	4:05 AM exactly; AM is optional
20080215 04:05 PM	4:05 PM exactly; the hour value must be less than 12
20080215 16:05	4:05 PM exactly
20080215	Midnight (by default)

Special datetime values

The following table shows special values that can be used as datetime literals and as arguments to date functions. They require single quotation marks and are converted to regular timestamp values during query processing.

Special value	Description
now	Evaluates to the start time of the current transaction and returns a timestamp with microsecond precision.
today	Evaluates to the appropriate date and returns a timestamp with zeroes for the time parts.
tomorrow	Evaluates to the appropriate date and returns a timestamp with zeroes for the time parts.

Special value	Description
yesterday	Evaluates to the appropriate date and returns a timestamp with zeroes for the time parts.

The following examples show how now and today work with the DATEADD function.

```
select dateadd(day,1,'today');

date_add
---------------------
2009-11-17 00:00:00
(1 row)

select dateadd(day,1,'now');

date_add
----------------------------
2009-11-17 10:45:32.021394
(1 row)
```

Interval literals

Following are rules for working with interval literals that are supported by AWS Clean Rooms.

Use an interval literal to identify specific periods of time, such as `12 hours` or `6 weeks`. You can use these interval literals in conditions and calculations that involve datetime expressions.

> **Note**
>
> You can't use the INTERVAL data type for columns in AWS Clean Rooms tables.

An interval is expressed as a combination of the INTERVAL keyword with a numeric quantity and a supported date part, for example `INTERVAL '7 days'` or `INTERVAL '59 minutes'`. You can connect several quantities and units to form a more precise interval, for example: `INTERVAL '7 days, 3 hours, 59 minutes'`. Abbreviations and plurals of each unit are also supported; for example: `5 s`, `5 second`, and `5 seconds` are equivalent intervals.

If you don't specify a date part, the interval value represents seconds. You can specify the quantity value as a fraction (for example: `0.5 days`).

Examples

The following examples show a series of calculations with different interval values.

The following example adds 1 second to the specified date.

```
select caldate + interval '1 second' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 00:00:01
(1 row)
```

The following example adds 1 minute to the specified date.

```
select caldate + interval '1 minute' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 00:01:00
(1 row)
```

The following example adds 3 hours and 35 minutes to the specified date.

```
select caldate + interval '3 hours, 35 minutes' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 03:35:00
(1 row)
```

The following example adds 52 weeks to the specified date.

```
select caldate + interval '52 weeks' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2009-12-30 00:00:00
(1 row)
```

The following example adds 1 week, 1 hour, 1 minute, and 1 second to the specified date.

```
select caldate + interval '1w, 1h, 1m, 1s' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2009-01-07 01:01:01
(1 row)
```

The following example adds 12 hours (half a day) to the specified date.

```
select caldate + interval '0.5 days' as dateplus from date
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 12:00:00
(1 row)
```

The following example subtracts 4 months from February 15, 2023 and the result is October 15, 2022.

```
select date '2023-02-15' - interval '4 months';

?column?
---------------------
2022-10-15 00:00:00
```

The following example subtracts 4 months from March 31, 2023 and the result is November 30, 2022. The calculation considers the number of days in a month.

```
select date '2023-03-31' - interval '4 months';

?column?
---------------------
2022-11-30 00:00:00
```

Boolean type

Use the BOOLEAN data type to store true and false values in a single-byte column. The following table describes the three possible states for a Boolean value and the literal values that result in

that state. Regardless of the input string, a Boolean column stores and outputs "t" for true and "f" for false.

State	Valid literal values	Storage
True	`TRUE 't' 'true' 'y' 'yes' '1'`	1 byte
False	`FALSE 'f' 'false' 'n' 'no' '0'`	1 byte
Unknown	`NULL`	1 byte

You can use an IS comparison to check a Boolean value only as a predicate in the WHERE clause. You can't use the IS comparison with a Boolean value in the SELECT list.

Examples

You can use a BOOLEAN column to store an "Active/Inactive" state for each customer in a CUSTOMER table.

```
select * from customer;
custid | active_flag
-------+-------------
   100 | t
```

In this example, the following query selects users from the USERS table who like sports but do not like theatre:

```
select firstname, lastname, likesports, liketheatre
from users
where likesports is true and liketheatre is false
order by userid limit 10;

firstname |  lastname  | likesports | liketheatre
```

```
----------+------------+------------+-------------
Alejandro | Rosalez    | t          | f
Akua      | Mansa      | t          | f
Arnav     | Desai      | t          | f
Carlos    | Salazar    | t          | f
Diego     | Ramirez    | t          | f
Efua      | Owusu      | t          | f
John      | Stiles     | t          | f
Jorge     | Souza      | t          | f
Kwaku     | Mensah     | t          | f
Kwesi     | Manu       | t          | f
(10 rows)
```

The following example selects users from the USERS table for whom is it unknown whether they like rock music.

```
select firstname, lastname, likerock
from users
where likerock is unknown
order by userid limit 10;

firstname | lastname | likerock
----------+----------+----------
Alejandro | Rosalez   |
Carlos    | Salazar   |
Diego     | Ramirez   |
John      | Stiles    |
Kwaku     | Mensah    |
Martha    | Rivera    |
Mateo     | Jackson   |
Paulo     | Santos    |
Richard   | Roe       |
Saanvi    | Sarkar    |
(10 rows)
```

The following example returns an error because it uses an IS comparison in the SELECT list.

```
select firstname, lastname, likerock is true as "check"
from users
order by userid limit 10;

[Amazon](500310) Invalid operation: Not implemented
```

The following example succeeds because it uses an equal comparison (=) in the SELECT list instead of the IS comparison.

```
select firstname, lastname, likerock = true as "check"
from users
order by userid limit 10;

firstname | lastname  | check
----------+-----------+------
Alejandro | Rosalez   |
Carlos    | Salazar   |
Diego     | Ramirez   | true
John      | Stiles    |
Kwaku     | Mensah    | true
Martha    | Rivera    | true
Mateo     | Jackson   |
Paulo     | Santos    | false
Richard   | Roe       |
Saanvi    | Sarkar    |
```

SUPER type

Use the SUPER data type to store semistructured data or documents as values.

Semistructured data doesn't conform to the rigid and tabular structure of the relational data model used in SQL databases. The SUPER data type contains tags that reference distinct entities within the data. SUPER data types can contain complex values such as arrays, nested structures, and other complex structures that are associated with serialization formats, such as JSON. The SUPER data type is a set of schemaless array and structure values that encompass all other scalar types of AWS Clean Rooms.

The SUPER data type supports up to 1 MB of data for an individual SUPER field or object.

The SUPER data type has the following properties:

- An AWS Clean Rooms scalar value:
  - A null
  - A boolean
  - A number, such as smallint, integer, bigint, decimal, or floating point (such as float4 or float8)

- A string value, such as varchar or char
- A complex value:
  - An array of values, including scalar or complex
  - A structure, also known as tuple or object, that is a map of attribute names and values (scalar or complex)

Any of the two types of complex values contain their own scalars or complex values without having any restrictions for regularity.

The SUPER data type supports the persistence of semistructured data in a schemaless form. Although hierarchical data model can change, the old versions of data can coexist in the same SUPER column.

Nested type

AWS Clean Rooms supports queries involving data with nested data types, specifically the AWS Glue struct, array, and map column types. Only the custom analysis rule supports nested data types.

Notably, nested data types don't conform to the rigid, tabular structure of the relational data model of SQL databases.

Nested data types contains tags that reference distinct entities within the data. They can contain complex values such as arrays, nested structures, and other complex structures that are associated with serialization formats, such as JSON. Nested data types support up to 1 MB of data for an individual nested data type field or object.

Examples of nested data types

For the `struct<given:varchar, family:varchar>` type, there are two attribute names: `given`, and `family`, each corresponding to a `varchar` value.

For the `array<varchar>` type, the array is specified as a list of `varchar`.

The `array<struct<shipdate:timestamp, price:double>>` type refers to a list of elements with `struct<shipdate:timestamp, price:double>` type.

The `map` data type behaves like an `array` of `structs`, where the attribute name for each element in the array is denoted by `key` and it maps to a `value`.

Example

For example, the `map<varchar(20), varchar(20)>` type is treated as `array<struct<key:varchar(20), value:varchar(20)>>`, where `key` and `value` refer to the attributes of the map in the underlying data.

For information about how AWS Clean Rooms enables navigation into arrays and structures, see Navigation.

For information about how AWS Clean Rooms enables iteration over arrays by navigating the array using the FROM clause of a query, see Unnesting queries.

VARBYTE type

Use a VARBYTE, VARBINARY, or BINARY VARYING column to store variable-length binary value with a fixed limit.

```
varbyte [ (n) ]
```

The maximum number of bytes (*n*) can range from 1 – 1,024,000. The default is 64,000.

Some examples where you might want to use a VARBYTE data type are as follows:

- Joining tables on VARBYTE columns.
- Creating materialized views that contain VARBYTE columns. Incremental refresh of materialized views that contain VARBYTE columns is supported. However, aggregate functions other than COUNT, MIN, and MAX and GROUP BY on VARBYTE columns don't support incremental refresh.

To ensure that all bytes are printable characters, AWS Clean Rooms uses the hex format to print VARBYTE values. For example, the following SQL converts the hexadecimal string 6162 into a binary value. Even though the returned value is a binary value, the results are printed as hexadecimal 6162.

```
select from_hex('6162');

 from_hex
----------
 6162
```

AWS Clean Rooms supports casting between VARBYTE and the following data types:

- CHAR
- VARCHAR
- SMALLINT
- INTEGER
- BIGINT

The following SQL statement casts a VARCHAR string to a VARBYTE. Even though the returned value is a binary value, the results are printed as hexadecimal 616263.

```
select 'abc'::varbyte;

 varbyte
---------
 616263
```

The following SQL statement casts a CHAR value in a column to a VARBYTE. This example creates a table with a CHAR(10) column (c), inserts character values that are shorter than the length of 10. The resulting cast pads the result with a space characters (hex'20') to the defined column size. Even though the returned value is a binary value, the results are printed as hexadecimal.

```
create table t (c char(10));
insert into t values ('aa'), ('abc');
select c::varbyte from t;
          c
----------------------
 6161202020202020202020
 6162632020202020202020
```

The following SQL statement casts a SMALLINT string to a VARBYTE. Even though the returned value is a binary value, the results are printed as hexadecimal `0005`, which is two bytes or four hexadecimal characters.

```
select 5::smallint::varbyte;

 varbyte
```

```
---------
 0005
```

The following SQL statement casts an INTEGER to a VARBYTE. Even though the returned value is a binary value, the results are printed as hexadecimal `00000005`, which is four bytes or eight hexadecimal characters.

```
select 5::int::varbyte;

 varbyte
----------
 00000005
```

The following SQL statement casts a BIGINT to a VARBYTE. Even though the returned value is a binary value, the results are printed as hexadecimal `0000000000000005`, which is eight bytes or 16 hexadecimal characters.

```
select 5::bigint::varbyte;

     varbyte
------------------
 0000000000000005
```

Limitations when using the VARBYTE data type with AWS Clean Rooms

The following are limitations when using the VARBYTE data type with AWS Clean Rooms:

- AWS Clean Rooms supports the VARBYTE data type only for Parquet and ORC files.
- AWS Clean Rooms query editor don't yet fully support VARBYTE data type. Therefore, use a different SQL client when working with VARBYTE expressions.

  As a workaround to use the query editor, if the length of your data is below 64 KB and the content is valid UTF-8, you can cast the VARBYTE values to VARCHAR, for example:

  ```
  select to_varbyte('6162', 'hex')::varchar;
  ```
- You can't use VARBYTE data types with Python or Lambda user-defined functions (UDFs).
- You can't create a HLLSKETCH column from a VARBYTE column or use APPROXIMATE COUNT DISTINCT on a VARBYTE column.

Type compatibility and conversion

The following discussion describes how type conversion rules and data type compatibility work in AWS Clean Rooms.

Compatibility

Data type matching and matching of literal values and constants to data types occurs during various database operations, including the following:

- Data manipulation language (DML) operations on tables
- UNION, INTERSECT, and EXCEPT queries
- CASE expressions
- Evaluation of predicates, such as LIKE and IN
- Evaluation of SQL functions that do comparisons or extractions of data
- Comparisons with mathematical operators

The results of these operations depend on type conversion rules and data type compatibility. *Compatibility* implies that a one-to-one matching of a certain value and a certain data type is not always required. Because some data types are *compatible*, an implicit conversion, or *coercion*, is possible. For more information, see Implicit conversion types. When data types are incompatible, you can sometimes convert a value from one data type to another by using an explicit conversion function.

General compatibility and conversion rules

Note the following compatibility and conversion rules:

- In general, data types that fall into the same type category (such as different numeric data types) are compatible and can be implicitly converted.

  For example, with implicit conversion you can insert a decimal value into an integer column. The decimal is rounded to produce a whole number. Or you can extract a numeric value, such as `2008`, from a date and insert that value into an integer column.
- Numeric data types enforce overflow conditions that occur when you attempt to insert out-of-range values. For example, a decimal value with a precision of 5 does not fit into a decimal column that was defined with a precision of 4. An integer or the whole part of a decimal is never

truncated. However, the fractional part of a decimal can be rounded up or down, as appropriate. However, results of explicit casts of values selected from tables are not rounded.

- Different types of character strings are compatible. VARCHAR column strings containing single-byte data and CHAR column strings are comparable and implicitly convertible. VARCHAR strings that contain multibyte data are not comparable. Also, you can convert a character string to a date, time, timestamp, or numeric value if the string is an appropriate literal value. Any leading or trailing spaces are ignored. Conversely, you can convert a date, time, timestamp, or numeric value to a fixed-length or variable-length character string.

  > **Note**
  >
  > A character string that you want to cast to a numeric type must contain a character representation of a number. For example, you can cast the strings `'1.0'` or `'5.9'` to decimal values, but you can't cast the string `'ABC'` to any numeric type.

- If you compare DECIMAL values with character strings, AWS Clean Rooms attempts to convert the character string to a DECIMAL value. When comparing all other numeric values with character strings, the numeric values are converted to character strings. To enforce the opposite conversion (for example, converting character strings to integers, or converting DECIMAL values to character strings), use an explicit function, such as CAST function.
- To convert 64-bit DECIMAL or NUMERIC values to a higher precision, you must use an explicit conversion function such as the CAST or CONVERT functions.
- When converting DATE or TIMESTAMP to TIMESTAMPTZ, or converting TIME to TIMETZ, the time zone is set to the current session time zone. The session time zone is UTC by default.

- Similarly, TIMESTAMPTZ is converted to DATE, TIME, or TIMESTAMP based on the current session time zone. The session time zone is UTC by default. After the conversion, time zone information is dropped.
- Character strings that represent a timestamp with time zone specified are converted to TIMESTAMPTZ using the current session time zone, which is UTC by default. Likewise, character strings that represent a time with time zone specified are converted to TIMETZ using the current session time zone, which is UTC by default.

Implicit conversion types

There are two types of implicit conversions:

- Implicit conversions in assignments, such as setting values in INSERT or UPDATE commands
- Implicit conversions in expressions, such as performing comparisons in the WHERE clause

The following table lists the data types that can be converted implicitly in assignments or expressions. You can also use an explicit conversion function to perform these conversions.

From type	To type
BIGINT	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER
	REAL (FLOAT4)
	SMALLINT
	VARCHAR
CHAR	VARCHAR
DATE	CHAR
	VARCHAR
	TIMESTAMP
	TIMESTAMPTZ
DECIMAL (NUMERIC)	BIGINT
	CHAR
	DOUBLE PRECISION (FLOAT8)
	INTEGER INT)

From type	To type
	REAL (FLOAT4)
	SMALLINT
	VARCHAR
DOUBLE PRECISION (FLOAT8)	BIGINT
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	REAL (FLOAT4)
	SMALLINT
	VARCHAR
INTEGER (INT)	BIGINT
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	SMALLINT
	VARCHAR
REAL (FLOAT4)	BIGINT
	CHAR

From type	To type
	DECIMAL (NUMERIC)
	INTEGER (INT)
	SMALLINT
	VARCHAR
SMALLINT	BIGINT
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER (INT)
	REAL (FLOAT4)
	VARCHAR
TIMESTAMP	CHAR
	DATE
	VARCHAR
	TIMESTAMPTZ
	TIME
TIMESTAMPTZ	CHAR
	DATE
	VARCHAR

From type	To type
	TIMESTAMP
	TIMETZ
TIME	VARCHAR
	TIMETZ
TIMETZ	VARCHAR
	TIME

Note

Implicit conversions between TIMESTAMPTZ, TIMESTAMP, DATE, TIME, TIMETZ, or character strings use the current session time zone.
The VARBYTE data type can't be implicitly converted to any other data type. For more information, see CAST function.

SQL commands in AWS Clean Rooms

The following SQL commands are supported in AWS Clean Rooms:

Topics

SELECT

The SELECT command returns rows from tables and user-defined functions.

The following SELECT SQL commands are supported in AWS Clean Rooms:

Topics

SELECT list

The SELECT list names the columns, functions, and expressions that you want the query to return. The list represents the output of the query.

Syntax

```
SELECT
```

```
[ TOP number ]
[ DISTINCT ] | expression [ AS column_alias ] [, ...]
```

Parameters

TOP *number*

TOP takes a positive integer as its argument, which defines the number of rows that are returned to the client. The behavior with the TOP clause is the same as the behavior with the LIMIT clause. The number of rows that is returned is fixed, but the set of rows is not fixed. To return a consistent set of rows, use TOP or LIMIT in conjunction with an ORDER BY clause.

DISTINCT

Option that eliminates duplicate rows from the result set, based on matching values in one or more columns.

expression

An expression formed from one or more columns that exist in the tables referenced by the query. An expression can contain SQL functions. For example:

```
coalesce(dimension, 'stringifnull') AS column_alias
```

AS column_alias

A temporary name for the column that is used in the final result set. The AS keyword is optional. For example:

```
coalesce(dimension, 'stringifnull') AS dimensioncomplete
```

If you don't specify an alias for an expression that isn't a simple column name, the result set applies a default name to that column.

> **Note**
>
> The alias is recognized right after it is defined in the target list. You cannot use an alias in other expressions defined after it in the same target list.

Usage notes

TOP is a SQL extension. TOP provides an alternative to the LIMIT behavior. You can't use TOP and LIMIT in the same query.

WITH clause

A WITH clause is an optional clause that precedes the SELECT list in a query. The WITH clause defines one or more *common_table_expressions*. Each common table expression (CTE) defines a temporary table, which is similar to a view definition. You can reference these temporary tables in the FROM clause. They're used only while the query they belong to runs. Each CTE in the WITH clause specifies a table name, an optional list of column names, and a query expression that evaluates to a table (a SELECT statement).

WITH clause subqueries are an efficient way of defining tables that can be used throughout the execution of a single query. In all cases, the same results can be achieved by using subqueries in the main body of the SELECT statement, but WITH clause subqueries may be simpler to write and read. Where possible, WITH clause subqueries that are referenced multiple times are optimized as common subexpressions; that is, it may be possible to evaluate a WITH subquery once and reuse its results. (Note that common subexpressions aren't limited to those defined in the WITH clause.)

Syntax

```
[ WITH  common_table_expression [, common_table_expression , ...] ]
```

where *common_table_expression* can be non-recursive. Following is the non-recursive form:

```
CTE_table_name AS ( query )
```

Parameters

common_table_expression

Defines a temporary table that you can reference in the FROM clause and is used only during the execution of the query to which it belongs.

CTE_table_name

A unique name for a temporary table that defines the results of a WITH clause subquery. You can't use duplicate names within a single WITH clause. Each subquery must be given a table name that can be referenced in the FROM clause.

query

Any SELECT query that AWS Clean Rooms supports. See SELECT.

Usage notes

You can use a WITH clause in the following SQL statement:

- SELECT, WITH, UNION, INTERSECT, and EXCEPT

If the FROM clause of a query that contains a WITH clause doesn't reference any of the tables defined by the WITH clause, the WITH clause is ignored and the query runs as normal.

A table defined by a WITH clause subquery can be referenced only in the scope of the SELECT query that the WITH clause begins. For example, you can reference such a table in the FROM clause of a subquery in the SELECT list, WHERE clause, or HAVING clause. You can't use a WITH clause in a subquery and reference its table in the FROM clause of the main query or another subquery. This query pattern results in an error message of the form `relation table_name doesn't exist` for the WITH clause table.

You can't specify another WITH clause inside a WITH clause subquery.

You can't make forward references to tables defined by WITH clause subqueries. For example, the following query returns an error because of the forward reference to table W2 in the definition of table W1:

```
with w1 as (select * from w2), w2 as (select * from w1)
select * from sales;
ERROR:  relation "w2" does not exist
```

Examples

The following example shows the simplest possible case of a query that contains a WITH clause. The WITH query named VENUECOPY selects all of the rows from the VENUE table. The main query in turn selects all of the rows from VENUECOPY. The VENUECOPY table exists only for the duration of this query.

```
with venuecopy as (select * from venue)
```

```
select * from venuecopy order by 1 limit 10;
```

```
 venueid |         venuename          |    venuecity    | venuestate | venueseats
---------+----------------------------+-----------------+------------+------------
1 | Toyota Park                | Bridgeview      | IL         |          0
2 | Columbus Crew Stadium      | Columbus        | OH         |          0
3 | RFK Stadium                | Washington      | DC         |          0
4 | CommunityAmerica Ballpark  | Kansas City     | KS         |          0
5 | Gillette Stadium           | Foxborough      | MA         |      68756
6 | New York Giants Stadium    | East Rutherford | NJ         |      80242
7 | BMO Field                  | Toronto         | ON         |          0
8 | The Home Depot Center      | Carson          | CA         |          0
9 | Dick's Sporting Goods Park | Commerce City   | CO         |          0
v     10 | Pizza Hut Park             | Frisco          | TX         |          0
(10 rows)
```

The following example shows a WITH clause that produces two tables, named VENUE_SALES and TOP_VENUES. The second WITH query table selects from the first. In turn, the WHERE clause of the main query block contains a subquery that constrains the TOP_VENUES table.

```
with venue_sales as
(select venuename, venuecity, sum(pricepaid) as venuename_sales
from sales, venue, event
where venue.venueid=event.venueid and event.eventid=sales.eventid
group by venuename, venuecity),

top_venues as
(select venuename
from venue_sales
where venuename_sales > 800000)

select venuename, venuecity, venuestate,
sum(qtysold) as venue_qty,
sum(pricepaid) as venue_sales
from sales, venue, event
where venue.venueid=event.venueid and event.eventid=sales.eventid
and venuename in(select venuename from top_venues)
group by venuename, venuecity, venuestate
order by venuename;
```

```
       venuename       |   venuecity   | venuestate | venue_qty | venue_sales
-----------------------+---------------+------------+-----------+-------------
```

```
August Wilson Theatre   | New York City | NY         |      3187 |  1032156.00
Biltmore Theatre        | New York City | NY         |      2629 |   828981.00
Charles Playhouse       | Boston        | MA         |      2502 |   857031.00
Ethel Barrymore Theatre | New York City | NY         |      2828 |   891172.00
Eugene O'Neill Theatre  | New York City | NY         |      2488 |   828950.00
Greek Theatre           | Los Angeles   | CA         |      2445 |   838918.00
Helen Hayes Theatre     | New York City | NY         |      2948 |   978765.00
Hilton Theatre          | New York City | NY         |      2999 |   885686.00
Imperial Theatre        | New York City | NY         |      2702 |   877993.00
Lunt-Fontanne Theatre   | New York City | NY         |      3326 |  1115182.00
Majestic Theatre        | New York City | NY         |      2549 |   894275.00
Nederlander Theatre     | New York City | NY         |      2934 |   936312.00
Pasadena Playhouse      | Pasadena      | CA         |      2739 |   820435.00
Winter Garden Theatre   | New York City | NY         |      2838 |   939257.00
(14 rows)
```

The following two examples demonstrate the rules for the scope of table references based on WITH clause subqueries. The first query runs, but the second fails with an expected error. The first query has WITH clause subquery inside the SELECT list of the main query. The table defined by the WITH clause (HOLIDAYS) is referenced in the FROM clause of the subquery in the SELECT list:

```
select caldate, sum(pricepaid) as daysales,
(with holidays as (select * from date where holiday ='t')
select sum(pricepaid)
from sales join holidays on sales.dateid=holidays.dateid
where caldate='2008-12-25') as dec25sales
from sales join date on sales.dateid=date.dateid
where caldate in('2008-12-25','2008-12-31')
group by caldate
order by caldate;

caldate   | daysales | dec25sales
----------+----------+------------
2008-12-25 | 70402.00 |   70402.00
2008-12-31 | 12678.00 |   70402.00
(2 rows)
```

The second query fails because it attempts to reference the HOLIDAYS table in the main query as well as in the SELECT list subquery. The main query references are out of scope.

```
select caldate, sum(pricepaid) as daysales,
(with holidays as (select * from date where holiday ='t')
```

```
select sum(pricepaid)
from sales join holidays on sales.dateid=holidays.dateid
where caldate='2008-12-25') as dec25sales
from sales join holidays on sales.dateid=holidays.dateid
where caldate in('2008-12-25','2008-12-31')
group by caldate
order by caldate;

ERROR:  relation "holidays" does not exist
```

FROM clause

The FROM clause in a query lists the table references (tables, views, and subqueries) that data is selected from. If multiple table references are listed, the tables must be joined, using appropriate syntax in either the FROM clause or the WHERE clause. If no join criteria are specified, the system processes the query as a cross-join (Cartesian product).

Topics

Syntax

```
FROM table_reference [, ...]
```

where *table_reference* is one of the following:

```
with_subquery_table_name | table_name | ( subquery ) [ [ AS ] alias ]
table_reference [ NATURAL ] join_type table_reference [ USING ( join_column [, ...] ) ]
table_reference [ INNER ] join_type table_reference ON expr
```

Parameters

with_subquery_table_name

A table defined by a subquery in the WITH clause.

table_name

Name of a table or view.

alias

Temporary alternative name for a table or view. An alias must be supplied for a table derived from a subquery. In other table references, aliases are optional. The AS keyword is always optional. Table aliases provide a convenient shortcut for identifying tables in other parts of a query, such as the WHERE clause.

For example:

```
select * from sales s, listing l
where s.listid=l.listid
```

If you define a table alias is defined, then the alias must be used to reference that table in the query.

For example, if the query is `SELECT "tbl"."col" FROM "tbl" AS "t"`, the query would fail because the table name is essentially overridden now. A valid query in this case would be `SELECT "t"."col" FROM "tbl" AS "t"`.

column_alias

Temporary alternative name for a column in a table or view.

subquery

A query expression that evaluates to a table. The table exists only for the duration of the query and is typically given a name or *alias*. However, an alias isn't required. You can also define column names for tables that derive from subqueries. Naming column aliases is important when you want to join the results of subqueries to other tables and when you want to select or constrain those columns elsewhere in the query.

A subquery may contain an ORDER BY clause, but this clause may have no effect if a LIMIT or OFFSET clause isn't also specified.

NATURAL

Defines a join that automatically uses all pairs of identically named columns in the two tables as the joining columns. No explicit join condition is required. For example, if the CATEGORY and EVENT tables both have columns named CATID, a natural join of those tables is a join over their CATID columns.

> **Note**
>
> If a NATURAL join is specified but no identically named pairs of columns exist in the tables to be joined, the query defaults to a cross-join.

join_type

Specify one of the following types of join:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

Cross-joins are unqualified joins; they return the Cartesian product of the two tables.

Inner and outer joins are qualified joins. They are qualified either implicitly (in natural joins); with the ON or USING syntax in the FROM clause; or with a WHERE clause condition.

An inner join returns matching rows only, based on the join condition or list of joining columns. An outer join returns all of the rows that the equivalent inner join would return plus non-matching rows from the "left" table, "right" table, or both tables. The left table is the first-listed table, and the right table is the second-listed table. The non-matching rows contain NULL values to fill the gaps in the output columns.

ON *join_condition*

Type of join specification where the joining columns are stated as a condition that follows the ON keyword. For example:

```
sales join listing
on sales.listid=listing.listid and sales.eventid=listing.eventid
```

USING (*join_column* [, ...])

Type of join specification where the joining columns are listed in parentheses. If multiple joining columns are specified, they are delimited by commas. The USING keyword must precede the list. For example:

```
sales join listing
using (listid,eventid)
```

Usage notes

Joining columns must have comparable data types.

A NATURAL or USING join retains only one of each pair of joining columns in the intermediate result set.

A join with the ON syntax retains both joining columns in its intermediate result set.

See also WITH clause.

JOIN examples

A SQL JOIN clause is used to combine the data from two or more tables based on common fields. The results might or might not change depending on the join method specified. For more information about the syntax of a JOIN clause, see Parameters.

The following query is an inner join (without the JOIN keyword) between the LISTING table and SALES table, where the LISTID from the LISTING table is between 1 and 5. This query matches LISTID column values in the LISTING table (the left table) and SALES table (the right table). The results show that LISTID 1, 4, and 5 match the criteria.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing, sales
where listing.listid = sales.listid
and listing.listid between 1 and 5
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     1 | 728.00 | 109.20
     4 |  76.00 |  11.40
     5 | 525.00 |  78.75
```

The following query is a left outer join. Left and right outer joins retain values from one of the joined tables when no match is found in the other table. The left and right tables are the first and second tables listed in the syntax. NULL values are used to fill the "gaps" in the result set. This

query matches LISTID column values in the LISTING table (the left table) and the SALES table (the right table). The results show that LISTIDs 2 and 3 did not result in any sales.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing left outer join sales on sales.listid = listing.listid
where listing.listid between 1 and 5
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     1 | 728.00 | 109.20
     2 | NULL   | NULL
     3 | NULL   | NULL
     4 |  76.00 |  11.40
     5 | 525.00 |  78.75
```

The following query is a right outer join. This query matches LISTID column values in the LISTING table (the left table) and the SALES table (the right table). The results show that LISTIDs 1, 4, and 5 match the criteria.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing right outer join sales on sales.listid = listing.listid
where listing.listid between 1 and 5
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     1 | 728.00 | 109.20
     4 |  76.00 |  11.40
     5 | 525.00 |  78.75
```

The following query is a full join. Full joins retain values from the joined tables when no match is found in the other table. The left and right tables are the first and second tables listed in the syntax. NULL values are used to fill the "gaps" in the result set. This query matches LISTID column values in the LISTING table (the left table) and the SALES table (the right table). The results show that LISTIDs 2 and 3 did not result in any sales.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing full join sales on sales.listid = listing.listid
```

```
where listing.listid between 1 and 5
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     1 | 728.00 | 109.20
     2 | NULL   | NULL
     3 | NULL   | NULL
     4 |  76.00 |  11.40
     5 | 525.00 |  78.75
```

The following query is a full join. This query matches LISTID column values in the LISTING table (the left table) and the SALES table (the right table). Only rows that do not result in any sales (LISTIDs 2 and 3) are in the results.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from listing full join sales on sales.listid = listing.listid
where listing.listid between 1 and 5
and (listing.listid IS NULL or sales.listid IS NULL)
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     2 | NULL   | NULL
     3 | NULL   | NULL
```

The following example is an inner join with the ON clause. In this case, NULL rows are not returned.

```
select listing.listid, sum(pricepaid) as price, sum(commission) as comm
from sales join listing
on sales.listid=listing.listid and sales.eventid=listing.eventid
where listing.listid between 1 and 5
group by 1
order by 1;

listid | price  |  comm
-------+--------+--------
     1 | 728.00 | 109.20
     4 |  76.00 |  11.40
     5 | 525.00 |  78.75
```

The following query is a cross join or Cartesian join of the LISTING table and the SALES table with a predicate to limit the results. This query matches LISTID column values in the SALES table and the LISTING table for LISTIDs 1, 2, 3, 4, and 5 in both tables. The results show that 20 rows match the criteria.

```
select sales.listid as sales_listid, listing.listid as listing_listid
from sales cross join listing
where sales.listid between 1 and 5
and listing.listid between 1 and 5
order by 1,2;

sales_listid | listing_listid
-------------+---------------
1            | 1
1            | 2
1            | 3
1            | 4
1            | 5
4            | 1
4            | 2
4            | 3
4            | 4
4            | 5
5            | 1
5            | 1
5            | 2
5            | 2
5            | 3
5            | 3
5            | 4
5            | 4
5            | 5
5            | 5
```

The following example is a natural join between two tables. In this case, the columns listid, sellerid, eventid, and dateid have identical names and data types in both tables and so are used as the join columns. The results are limited to five rows.

```
select listid, sellerid, eventid, dateid, numtickets
from listing natural join sales
order by 1
limit 5;
```

```
listid | sellerid  | eventid | dateid | numtickets
-------+-----------+---------+--------+-----------
113    | 29704     | 4699    | 2075   | 22
115    | 39115     | 3513    | 2062   | 14
116    | 43314     | 8675    | 1910   | 28
118    | 6079      | 1611    | 1862   | 9
163    | 24880     | 8253    | 1888   | 14
```

The following example is a join between two tables with the USING clause. In this case, the columns listid and eventid are used as the join columns. The results are limited to five rows.

```
select listid, listing.sellerid, eventid, listing.dateid, numtickets
from listing join sales
using (listid, eventid)
order by 1
limit 5;

listid | sellerid | eventid | dateid | numtickets
-------+----------+---------+--------+-----------
1      | 36861    | 7872    | 1850   | 10
4      | 8117     | 4337    | 1970   | 8
5      | 1616     | 8647    | 1963   | 4
5      | 1616     | 8647    | 1963   | 4
6      | 47402    | 8240    | 2053   | 18
```

The following query is an inner join of two subqueries in the FROM clause. The query finds the number of sold and unsold tickets for different categories of events (concerts and shows). The FROM clause subqueries are *table* subqueries; they can return multiple columns and rows.

```
select catgroup1, sold, unsold
from
(select catgroup, sum(qtysold) as sold
from category c, event e, sales s
where c.catid = e.catid and e.eventid = s.eventid
group by catgroup) as a(catgroup1, sold)
join
(select catgroup, sum(numtickets)-sum(qtysold) as unsold
from category c, event e, sales s, listing l
where c.catid = e.catid and e.eventid = s.eventid
and s.listid = l.listid
group by catgroup) as b(catgroup2, unsold)
```

```
on a.catgroup1 = b.catgroup2
order by 1;

catgroup1 |  sold  | unsold
----------+--------+--------
Concerts  | 195444 |1067199
Shows     | 149905 | 817736
```

WHERE clause

The WHERE clause contains conditions that either join tables or apply predicates to columns in tables. Tables can be inner-joined by using appropriate syntax in either the WHERE clause or the FROM clause. Outer join criteria must be specified in the FROM clause.

Syntax

```
[ WHERE condition ]
```

condition

Any search condition with a Boolean result, such as a join condition or a predicate on a table column. The following examples are valid join conditions:

```
sales.listid=listing.listid
sales.listid<>listing.listid
```

The following examples are valid conditions on columns in tables:

```
catgroup like 'S%'
venueseats between 20000 and 50000
eventname in('Jersey Boys','Spamalot')
year=2008
length(catdesc)>25
date_part(month, caldate)=6
```

Conditions can be simple or complex; for complex conditions, you can use parentheses to isolate logical units. In the following example, the join condition is enclosed by parentheses.

```
where (category.catid=event.catid) and category.catid in(6,7,8)
```

Usage notes

You can use aliases in the WHERE clause to reference select list expressions.

You can't restrict the results of aggregate functions in the WHERE clause; use the HAVING clause for this purpose.

Columns that are restricted in the WHERE clause must derive from table references in the FROM clause.

Example

The following query uses a combination of different WHERE clause restrictions, including a join condition for the SALES and EVENT tables, a predicate on the EVENTNAME column, and two predicates on the STARTTIME column.

```
select eventname, starttime, pricepaid/qtysold as costperticket, qtysold
from sales, event
where sales.eventid = event.eventid
and eventname='Hannah Montana'
and date_part(quarter, starttime) in(1,2)
and date_part(year, starttime) = 2008
order by 3 desc, 4, 2, 1 limit 10;

eventname    |      starttime      |   costperticket   | qtysold
---------------+---------------------+-------------------+---------
Hannah Montana | 2008-06-07 14:00:00 |     1706.00000000 |       2
Hannah Montana | 2008-05-01 19:00:00 |     1658.00000000 |       2
Hannah Montana | 2008-06-07 14:00:00 |     1479.00000000 |       1
Hannah Montana | 2008-06-07 14:00:00 |     1479.00000000 |       3
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       1
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       2
Hannah Montana | 2008-06-07 14:00:00 |     1163.00000000 |       4
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       1
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       2
Hannah Montana | 2008-05-01 19:00:00 |      497.00000000 |       4
(10 rows)
```

GROUP BY clause

The GROUP BY clause identifies the grouping columns for the query. Grouping columns must be declared when the query computes aggregates with standard functions such as SUM, AVG, and

COUNT. If an aggregate function is present in the SELECT expression, any column in the SELECT expression that is not in an aggregate function must be in the GROUP BY clause.

For more information, see SQL functions in AWS Clean Rooms.

Syntax

```
GROUP BY group_by_clause [, ...]

group_by_clause := {
    expr |
        ROLLUP ( expr [, ...] ) |
        }
```

Parameters

expr

The list of columns or expressions must match the list of non-aggregate expressions in the select list of the query. For example, consider the following simple query.

```
select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by listid, eventid
order by 3, 4, 2, 1
limit 5;

listid | eventid | revenue | numtix
-------+---------+---------+--------
89397  |      47 |   20.00 |      1
106590 |      76 |   20.00 |      1
124683 |     393 |   20.00 |      1
103037 |     403 |   20.00 |      1
147685 |     429 |   20.00 |      1
(5 rows)
```

In this query, the select list consists of two aggregate expressions. The first uses the SUM function and the second uses the COUNT function. The remaining two columns, LISTID and EVENTID, must be declared as grouping columns.

Expressions in the GROUP BY clause can also reference the select list by using ordinal numbers. For example, the previous example could be abbreviated as follows.

```
select listid, eventid, sum(pricepaid) as revenue,
count(qtysold) as numtix
from sales
group by 1,2
order by 3, 4, 2, 1
limit 5;

listid | eventid | revenue | numtix
-------+---------+---------+--------
89397  |      47 |   20.00 |      1
106590 |      76 |   20.00 |      1
124683 |     393 |   20.00 |      1
103037 |     403 |   20.00 |      1
147685 |     429 |   20.00 |      1
(5 rows)
```

ROLLUP

You can use the aggregation extension ROLLUP to perform the work of multiple GROUP BY operations in a single statement. For more information on aggregation extensions and related functions, see Aggregation extensions.

Aggregation extensions

AWS Clean Rooms supports aggregation extensions to do the work of multiple GROUP BY operations in a single statement.

GROUPING SETS

Computes one or more grouping sets in a single statement. A grouping set is the set of a single GROUP BY clause, a set of 0 or more columns by which you can group a query's result set. GROUP BY GROUPING SETS is equivalent to running a UNION ALL query on one result set grouped by different columns. For example, GROUP BY GROUPING SETS((a), (b)) is equivalent to GROUP BY a UNION ALL GROUP BY b.

The following example returns the cost of the order table's products grouped according to both the products' categories and the kind of products sold.

```
SELECT category, product, sum(cost) as total
FROM orders
GROUP BY GROUPING SETS(category, product);

       category       |       product        | total
----------------------+----------------------+-------
 computers            |                      |  2100
 cellphones           |                      |  1610
                      | laptop               |  2050
                      | smartphone           |  1610
                      | mouse                |    50

(5 rows)
```

ROLLUP

Assumes a hierarchy where preceding columns are considered the parents of subsequent columns. ROLLUP groups data by the provided columns, returning extra subtotal rows representing the totals throughout all levels of grouping columns, in addition to the grouped rows. For example, you can use GROUP BY ROLLUP((a), (b)) to return a result set grouped first by a, then by b while assuming that b is a subsection of a. ROLLUP also returns a row with the whole result set without grouping columns.

GROUP BY ROLLUP((a), (b)) is equivalent to GROUP BY GROUPING SETS((a,b), (a), ()).

The following example returns the cost of the order table's products grouped first by category and then product, with product as a subdivision of category.

```
SELECT category, product, sum(cost) as total
FROM orders
GROUP BY ROLLUP(category, product) ORDER BY 1,2;

       category       |       product        | total
----------------------+----------------------+-------
 cellphones           | smartphone           |  1610
 cellphones           |                      |  1610
 computers            | laptop               |  2050
 computers            | mouse                |    50
 computers            |                      |  2100
                      |                      |  3710
(6 rows)
```

CUBE

Groups data by the provided columns, returning extra subtotal rows representing the totals throughout all levels of grouping columns, in addition to the grouped rows. CUBE returns the same rows as ROLLUP, while adding additional subtotal rows for every combination of grouping column not covered by ROLLUP. For example, you can use GROUP BY CUBE ((a), (b)) to return a result set grouped first by a, then by b while assuming that b is a subsection of a, then by b alone. CUBE also returns a row with the whole result set without grouping columns.

GROUP BY CUBE((a), (b)) is equivalent to GROUP BY GROUPING SETS((a, b), (a), (b), ()).

The following example returns the cost of the order table's products grouped first by category and then product, with product as a subdivision of category. Unlike the preceding example for ROLLUP, the statement returns results for every combination of grouping column.

```
SELECT category, product, sum(cost) as total
FROM orders
GROUP BY CUBE(category, product) ORDER BY 1,2;

       category       |       product        | total
----------------------+----------------------+-------
 cellphones           | smartphone           |  1610
 cellphones           |                      |  1610
 computers            | laptop               |  2050
 computers            | mouse                |    50
 computers            |                      |  2100
                      | laptop               |  2050
                      | mouse                |    50
                      | smartphone           |  1610
                      |                      |  3710
(9 rows)
```

HAVING clause

The HAVING clause applies a condition to the intermediate grouped result set that a query returns.

Syntax

```
[ HAVING condition ]
```

For example, you can restrict the results of a SUM function:

```
having sum(pricepaid) >10000
```

The HAVING condition is applied after all WHERE clause conditions are applied and GROUP BY operations are completed.

The condition itself takes the same form as any WHERE clause condition.

Usage notes

- Any column that is referenced in a HAVING clause condition must be either a grouping column or a column that refers to the result of an aggregate function.
- In a HAVING clause, you can't specify:
  - An ordinal number that refers to a select list item. Only the GROUP BY and ORDER BY clauses accept ordinal numbers.

Examples

The following query calculates total ticket sales for all events by name, then eliminates events where the total sales were less than $800,000. The HAVING condition is applied to the results of the aggregate function in the select list: `sum(pricepaid)`.

```
select eventname, sum(pricepaid)
from sales join event on sales.eventid = event.eventid
group by 1
having sum(pricepaid) > 800000
order by 2 desc, 1;

eventname     |    sum
------------------+-----------
Mamma Mia!       | 1135454.00
Spring Awakening |  972855.00
The Country Girl |  910563.00
Macbeth          |  862580.00
Jersey Boys      |  811877.00
Legally Blonde   |  804583.00
(6 rows)
```

The following query calculates a similar result set. In this case, however, the HAVING condition is applied to an aggregate that isn't specified in the select list: `sum(qtysold)`. Events that did not sell more than 2,000 tickets are eliminated from the final result.

```
select eventname, sum(pricepaid)
from sales join event on sales.eventid = event.eventid
group by 1
having sum(qtysold) >2000
order by 2 desc, 1;

eventname     |    sum
------------------+-----------
Mamma Mia!       | 1135454.00
Spring Awakening |  972855.00
The Country Girl |  910563.00
Macbeth          |  862580.00
Jersey Boys      |  811877.00
Legally Blonde   |  804583.00
Chicago          |  790993.00
Spamalot         |  714307.00
(8 rows)
```

Set operators

The UNION, INTERSECT, and EXCEPT *set operators* are used to compare and merge the results of two separate query expressions. For example, if you want to know which users of a website are both buyers and sellers but their user names are stored in separate columns or tables, you can find the *intersection* of these two types of users. If you want to know which website users are buyers but not sellers, you can use the EXCEPT operator to find the *difference* between the two lists of users. If you want to build a list of all users, regardless of role, you can use the UNION operator.

> **Note**
>
> The ORDER BY, LIMIT, SELECT TOP, and OFFSET clauses cannot be used in the query expressions merged by the UNION, UNION ALL, INTERSECT, and EXCEPT set operators.

Topics

- Syntax
- Parameters

- Order of evaluation for set operators
- Usage notes
- Example UNION queries
- Example UNION ALL query
- Example INTERSECT queries
- Example EXCEPT query

Syntax

```
query
{ UNION [ ALL ] | INTERSECT | EXCEPT | MINUS }
query
```

Parameters

query

A query expression that corresponds, in the form of its select list, to a second query expression that follows the UNION, INTERSECT, or EXCEPT operator. The two expressions must contain the same number of output columns with compatible data types; otherwise, the two result sets can't be compared and merged. Set operations don't allow implicit conversion between different categories of data types; for more information, see Type compatibility and conversion.

You can build queries that contain an unlimited number of query expressions and link them with UNION, INTERSECT, and EXCEPT operators in any combination. For example, the following query structure is valid, assuming that the tables T1, T2, and T3 contain compatible sets of columns:

```
select * from t1
union
select * from t2
except
select * from t3
```

UNION

Set operation that returns rows from two query expressions, regardless of whether the rows derive from one or both expressions.

INTERSECT

Set operation that returns rows that derive from two query expressions. Rows that aren't returned by both expressions are discarded.

EXCEPT | MINUS

Set operation that returns rows that derive from one of two query expressions. To qualify for the result, rows must exist in the first result table but not the second. MINUS and EXCEPT are exact synonyms.

ALL

The ALL keyword retains any duplicate rows that are produced by UNION. The default behavior when the ALL keyword isn't used is to discard these duplicates. INTERSECT ALL, EXCEPT ALL, and MINUS ALL aren't supported.

Order of evaluation for set operators

The UNION and EXCEPT set operators are left-associative. If parentheses aren't specified to influence the order of precedence, a combination of these set operators is evaluated from left to right. For example, in the following query, the UNION of T1 and T2 is evaluated first, then the EXCEPT operation is performed on the UNION result:

```
select * from t1
union
select * from t2
except
select * from t3
```

The INTERSECT operator takes precedence over the UNION and EXCEPT operators when a combination of operators is used in the same query. For example, the following query evaluates the intersection of T2 and T3, then union the result with T1:

```
select * from t1
union
select * from t2
intersect
select * from t3
```

By adding parentheses, you can enforce a different order of evaluation. In the following case, the result of the union of T1 and T2 is intersected with T3, and the query is likely to produce a different result.

```
(select * from t1
union
select * from t2)
intersect
(select * from t3)
```

Usage notes

- The column names returned in the result of a set operation query are the column names (or aliases) from the tables in the first query expression. Because these column names are potentially misleading, in that the values in the column derive from tables on either side of the set operator, you might want to provide meaningful aliases for the result set.
- When set operator queries return decimal results, the corresponding result columns are promoted to return the same precision and scale. For example, in the following query, where T1.REVENUE is a DECIMAL(10,2) column and T2.REVENUE is a DECIMAL(8,4) column, the decimal result is promoted to DECIMAL(12,4):

  ```
  select t1.revenue union select t2.revenue;
  ```

  The scale is 4 because that is the maximum scale of the two columns. The precision is 12 because T1.REVENUE requires 8 digits to the left of the decimal point (12 - 4 = 8). This type promotion ensures that all values from both sides of the UNION fit in the result. For 64-bit values, the maximum result precision is 19 and the maximum result scale is 18. For 128-bit values, the maximum result precision is 38 and the maximum result scale is 37.

  If the resulting data type exceeds AWS Clean Rooms precision and scale limits, the query returns an error.
- For set operations, two rows are treated as identical if, for each corresponding pair of columns, the two data values are either *equal* or *both NULL*. For example, if tables T1 and T2 both contain one column and one row, and that row is NULL in both tables, an INTERSECT operation over those tables returns that row.

Example UNION queries

In the following UNION query, rows in the SALES table are merged with rows in the LISTING table. Three compatible columns are selected from each table; in this case, the corresponding columns have the same names and data types.

```
select listid, sellerid, eventid from listing
union select listid, sellerid, eventid from sales


listid | sellerid | eventid
--------+----------+---------
1 |    36861 |    7872
2 |    16002 |    4806
3 |    21461 |    4256
4 |     8117 |    4337
5 |     1616 |    8647
```

The following example shows how you can add a literal value to the output of a UNION query so you can see which query expression produced each row in the result set. The query identifies rows from the first query expression as "B" (for buyers) and rows from the second query expression as "S" (for sellers).

The query identifies buyers and sellers for ticket transactions that cost $10,000 or more. The only difference between the two query expressions on either side of the UNION operator is the joining column for the SALES table.

```
select listid, lastname, firstname, username,
pricepaid as price, 'S' as buyorsell
from sales, users
where sales.sellerid=users.userid
and pricepaid >=10000
union
select listid, lastname, firstname, username, pricepaid,
'B' as buyorsell
from sales, users
where sales.buyerid=users.userid
and pricepaid >=10000

listid | lastname | firstname | username |   price   | buyorsell
--------+----------+-----------+----------+-----------+-----------
```

```
209658 | Lamb     | Colette   | VOR15LYI |  10000.00 | B
209658 | West     | Kato      | ELU81XAA |  10000.00 | S
212395 | Greer    | Harlan    | GXO71KOC |  12624.00 | S
212395 | Perry    | Cora      | YWR73YNZ |  12624.00 | B
215156 | Banks    | Patrick   | ZNQ69CLT |  10000.00 | S
215156 | Hayden   | Malachi   | BBG56AKU |  10000.00 | B
```

The following example uses a UNION ALL operator because duplicate rows, if found, need to be retained in the result. For a specific series of event IDs, the query returns 0 or more rows for each sale associated with each event, and 0 or 1 row for each listing of that event. Event IDs are unique to each row in the LISTING and EVENT tables, but there might be multiple sales for the same combination of event and listing IDs in the SALES table.

The third column in the result set identifies the source of the row. If it comes from the SALES table, it is marked "Yes" in the SALESROW column. (SALESROW is an alias for SALES.LISTID.) If the row comes from the LISTING table, it is marked "No" in the SALESROW column.

In this case, the result set consists of three sales rows for listing 500, event 7787. In other words, three different transactions took place for this listing and event combination. The other two listings, 501 and 502, did not produce any sales, so the only row that the query produces for these list IDs comes from the LISTING table (SALESROW = 'No').

```
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union all
select eventid, listid, 'No'
from listing
where listid in(500,501,502)

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
7787 |    500 | Yes
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
```

If you run the same query without the ALL keyword, the result retains only one of the sales transactions.

```
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union
select eventid, listid, 'No'
from listing
where listid in(500,501,502)

eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
```

Example UNION ALL query

The following example uses a UNION ALL operator because duplicate rows, if found, need to be retained in the result. For a specific series of event IDs, the query returns 0 or more rows for each sale associated with each event, and 0 or 1 row for each listing of that event. Event IDs are unique to each row in the LISTING and EVENT tables, but there might be multiple sales for the same combination of event and listing IDs in the SALES table.

The third column in the result set identifies the source of the row. If it comes from the SALES table, it is marked "Yes" in the SALESROW column. (SALESROW is an alias for SALES.LISTID.) If the row comes from the LISTING table, it is marked "No" in the SALESROW column.

In this case, the result set consists of three sales rows for listing 500, event 7787. In other words, three different transactions took place for this listing and event combination. The other two listings, 501 and 502, did not produce any sales, so the only row that the query produces for these list IDs comes from the LISTING table (SALESROW = 'No').

```
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union all
select eventid, listid, 'No'
from listing
where listid in(500,501,502)

eventid | listid | salesrow
```

```
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
7787 |    500 | Yes
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
```

If you run the same query without the ALL keyword, the result retains only one of the sales transactions.

```
select eventid, listid, 'Yes' as salesrow
from sales
where listid in(500,501,502)
union
select eventid, listid, 'No'
from listing
where listid in(500,501,502)
eventid | listid | salesrow
---------+--------+----------
7787 |    500 | No
7787 |    500 | Yes
6473 |    501 | No
5108 |    502 | No
```

Example INTERSECT queries

Compare the following example with the first UNION example. The only difference between the two examples is the set operator that is used, but the results are very different. Only one of the rows is the same:

```
235494 |    23875 |    8771
```

This is the only row in the limited result of 5 rows that was found in both tables.

```
select listid, sellerid, eventid from listing
intersect
select listid, sellerid, eventid from sales

listid | sellerid | eventid
--------+----------+---------
235494 |    23875 |    8771
```

```
235482 |     1067 |    2667
235479 |     1589 |    7303
235476 |    15550 |     793
235475 |    22306 |    7848
```

The following query finds events (for which tickets were sold) that occurred at venues in both New York City and Los Angeles in March. The difference between the two query expressions is the constraint on the VENUECITY column.

```
select distinct eventname from event, sales, venue
where event.eventid=sales.eventid and event.venueid=venue.venueid
and date_part(month,starttime)=3 and venuecity='Los Angeles'
intersect
select distinct eventname from event, sales, venue
where event.eventid=sales.eventid and event.venueid=venue.venueid
and date_part(month,starttime)=3 and venuecity='New York City';

eventname
----------------------------
A Streetcar Named Desire
Dirty Dancing
Electra
Running with Annalise
Hairspray
Mary Poppins
November
Oliver!
Return To Forever
Rhinoceros
South Pacific
The 39 Steps
The Bacchae
The Caucasian Chalk Circle
The Country Girl
Wicked
Woyzeck
```

Example EXCEPT query

The CATEGORY table in the database contains the following 11 rows:

```
 catid | catgroup |  catname  |                 catdesc
```

```
-------+----------+-----------+--------------------------------------------
   1   | Sports   | MLB       | Major League Baseball
   2   | Sports   | NHL       | National Hockey League
   3   | Sports   | NFL       | National Football League
   4   | Sports   | NBA       | National Basketball Association
   5   | Sports   | MLS       | Major League Soccer
   6   | Shows    | Musicals  | Musical theatre
   7   | Shows    | Plays     | All non-musical theatre
   8   | Shows    | Opera     | All opera and light opera
   9   | Concerts | Pop       | All rock and pop music concerts
  10   | Concerts | Jazz      | All jazz singers and bands
  11   | Concerts | Classical | All symphony, concerto, and choir concerts
(11 rows)
```

Assume that a CATEGORY_STAGE table (a staging table) contains one additional row:

```
 catid | catgroup |  catname  |                 catdesc
-------+----------+-----------+--------------------------------------------
   1   | Sports   | MLB       | Major League Baseball
   2   | Sports   | NHL       | National Hockey League
   3   | Sports   | NFL       | National Football League
   4   | Sports   | NBA       | National Basketball Association
   5   | Sports   | MLS       | Major League Soccer
   6   | Shows    | Musicals  | Musical theatre
   7   | Shows    | Plays     | All non-musical theatre
   8   | Shows    | Opera     | All opera and light opera
   9   | Concerts | Pop       | All rock and pop music concerts
  10   | Concerts | Jazz      | All jazz singers and bands
  11   | Concerts | Classical | All symphony, concerto, and choir concerts
  12   | Concerts | Comedy    | All stand up comedy performances
(12 rows)
```

Return the difference between the two tables. In other words, return rows that are in the CATEGORY_STAGE table but not in the CATEGORY table:

```
select * from category_stage
except
select * from category;

catid | catgroup | catname |             catdesc
-------+----------+---------+----------------------------------
  12  | Concerts | Comedy  | All stand up comedy performances
```

```
(1 row)
```

The following equivalent query uses the synonym MINUS.

```
select * from category_stage
minus
select * from category;

catid | catgroup | catname |             catdesc
-------+----------+---------+----------------------------------
  12  | Concerts | Comedy  | All stand up comedy performances
(1 row)
```

If you reverse the order of the SELECT expressions, the query returns no rows.

ORDER BY clause

The ORDER BY clause sorts the result set of a query.

> **Note**
>
> The outermost ORDER BY expression must only have columns that are in the select list.

Topics

Syntax

```
[ ORDER BY expression [ ASC | DESC ] ]
[ NULLS FIRST | NULLS LAST ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
```

Parameters

expression

Expression that defines the sort order of the query result. It consists of one or more columns in the select list. Results are returned based on binary UTF-8 ordering. You can also specify the following:

- Ordinal numbers that represent the position of select list entries (or the position of columns in the table if no select list exists)
- Aliases that define select list entries

When the ORDER BY clause contains multiple expressions, the result set is sorted according to the first expression, then the second expression is applied to rows that have matching values from the first expression, and so on.

ASC | DESC

Option that defines the sort order for the expression, as follows:

- ASC: ascending (for example, low to high for numeric values and 'A' to 'Z' for character strings). If no option is specified, data is sorted in ascending order by default.
- DESC: descending (high to low for numeric values; 'Z' to 'A' for strings).

NULLS FIRST | NULLS LAST

Option that specifies whether NULL values should be ordered first, before non-null values, or last, after non-null values. By default, NULL values are sorted and ranked last in ASC ordering, and sorted and ranked first in DESC ordering.

LIMIT *number* | ALL

Option that controls the number of sorted rows that the query returns. The LIMIT number must be a positive integer; the maximum value is 2147483647.

LIMIT 0 returns no rows. You can use this syntax for testing purposes: to check that a query runs (without displaying any rows) or to return a column list from a table. An ORDER BY clause is redundant if you are using LIMIT 0 to return a column list. The default is LIMIT ALL.

OFFSET *start*

Option that specifies to skip the number of rows before *start* before beginning to return rows. The OFFSET number must be a positive integer; the maximum value is 2147483647. When used with the LIMIT option, OFFSET rows are skipped before starting to count the LIMIT rows that are returned. If the LIMIT option isn't used, the number of rows in the result set is reduced by the number of rows that are skipped. The rows skipped by an OFFSET clause still have to be scanned, so it might be inefficient to use a large OFFSET value.

Usage notes

Note the following expected behavior with ORDER BY clauses:

- NULL values are considered "higher" than all other values. With the default ascending sort order, NULL values sort at the end. To change this behavior, use the NULLS FIRST option.
- When a query doesn't contain an ORDER BY clause, the system returns result sets with no predictable ordering of the rows. The same query run twice might return the result set in a different order.
- The LIMIT and OFFSET options can be used without an ORDER BY clause; however, to return a consistent set of rows, use these options in conjunction with ORDER BY.
- In any parallel system like AWS Clean Rooms, when ORDER BY doesn't produce a unique ordering, the order of the rows is nondeterministic. That is, if the ORDER BY expression produces duplicate values, the return order of those rows might vary from other systems or from one run of AWS Clean Rooms to the next.
- AWS Clean Rooms doesn't support string literals in ORDER BY clauses.

Examples with ORDER BY

Return all 11 rows from the CATEGORY table, ordered by the second column, CATGROUP. For results that have the same CATGROUP value, order the CATDESC column values by the length of the character string. Then order by columns CATID and CATNAME.

```
select * from category order by 2, 1, 3;

catid | catgroup |  catname  |                 catdesc
-------+----------+-----------+--------------------------------------
10 | Concerts | Jazz      | All jazz singers and bands
```

```
9 | Concerts | Pop       | All rock and pop music concerts
11 | Concerts | Classical | All symphony, concerto, and choir conce
6 | Shows    | Musicals  | Musical theatre
7 | Shows    | Plays     | All non-musical theatre
8 | Shows    | Opera     | All opera and light opera
5 | Sports   | MLS       | Major League Soccer
1 | Sports   | MLB       | Major League Baseball
2 | Sports   | NHL       | National Hockey League
3 | Sports   | NFL       | National Football League
4 | Sports   | NBA       | National Basketball Association
(11 rows)
```

Return selected columns from the SALES table, ordered by the highest QTYSOLD values. Limit the result to the top 10 rows:

```
select salesid, qtysold, pricepaid, commission, saletime from sales
order by qtysold, pricepaid, commission, salesid, saletime desc

salesid | qtysold | pricepaid | commission |      saletime
---------+---------+-----------+------------+---------------------
15401 |       8 |    272.00 |      40.80 | 2008-03-18 06:54:56
61683 |       8 |    296.00 |      44.40 | 2008-11-26 04:00:23
90528 |       8 |    328.00 |      49.20 | 2008-06-11 02:38:09
74549 |       8 |    336.00 |      50.40 | 2008-01-19 12:01:21
130232 |       8 |    352.00 |      52.80 | 2008-05-02 05:52:31
55243 |       8 |    384.00 |      57.60 | 2008-07-12 02:19:53
16004 |       8 |    440.00 |      66.00 | 2008-11-04 07:22:31
489 |       8 |    496.00 |      74.40 | 2008-08-03 05:48:55
4197 |       8 |    512.00 |      76.80 | 2008-03-23 11:35:33
16929 |       8 |    568.00 |      85.20 | 2008-12-19 02:59:33
```

Return a column list and no rows by using LIMIT 0 syntax:

```
select * from venue limit 0;
venueid | venuename | venuecity | venuestate | venueseats
---------+-----------+-----------+------------+------------
(0 rows)
```

Subquery examples

The following examples show different ways in which subqueries fit into SELECT queries. See JOIN examples for another example of the use of subqueries.

SELECT list subquery

The following example contains a subquery in the SELECT list. This subquery is *scalar*: it returns only one column and one value, which is repeated in the result for each row that is returned from the outer query. The query compares the Q1SALES value that the subquery computes with sales values for two other quarters (2 and 3) in 2008, as defined by the outer query.

```
select qtr, sum(pricepaid) as qtrsales,
(select sum(pricepaid)
from sales join date on sales.dateid=date.dateid
where qtr='1' and year=2008) as q1sales
from sales join date on sales.dateid=date.dateid
where qtr in('2','3') and year=2008
group by qtr
order by qtr;

qtr  |  qtrsales   |   q1sales
-------+-------------+-------------
2     | 30560050.00 | 24742065.00
3     | 31170237.00 | 24742065.00
(2 rows)
```

WHERE clause subquery

The following example contains a table subquery in the WHERE clause. This subquery produces multiple rows. In this case, the rows contain only one column, but table subqueries can contain multiple columns and rows, just like any other table.

The query finds the top 10 sellers in terms of maximum tickets sold. The top 10 list is restricted by the subquery, which removes users who live in cities where there are ticket venues. This query can be written in different ways; for example, the subquery could be rewritten as a join within the main query.

```
select firstname, lastname, city, max(qtysold) as maxsold
from users join sales on users.userid=sales.sellerid
where users.city not in(select venuecity from venue)
group by firstname, lastname, city
order by maxsold desc, city desc
limit 10;

firstname | lastname  |      city      | maxsold
```

```
-----------+-----------+----------------+---------
Noah       | Guerrero | Worcester      |       8
Isadora    | Moss     | Winooski       |       8
Kieran     | Harrison | Westminster    |       8
Heidi      | Davis    | Warwick        |       8
Sara       | Anthony  | Waco           |       8
Bree       | Buck     | Valdez         |       8
Evangeline | Sampson  | Trenton        |       8
Kendall    | Keith    | Stillwater     |       8
Bertha     | Bishop   | Stevens Point  |       8
Patricia   | Anderson | South Portland |       8
(10 rows)
```

WITH clause subqueries

See WITH clause.

Correlated subqueries

The following example contains a *correlated subquery* in the WHERE clause; this kind of subquery contains one or more correlations between its columns and the columns produced by the outer query. In this case, the correlation is `where s.listid=l.listid`. For each row that the outer query produces, the subquery is run to qualify or disqualify the row.

```
select salesid, listid, sum(pricepaid) from sales s
where qtysold=
(select max(numtickets) from listing l
where s.listid=l.listid)
group by 1,2
order by 1,2
limit 5;

salesid | listid |   sum
--------+--------+----------
 27     |     28 | 111.00
 81     |    103 | 181.00
 142    |    149 | 240.00
 146    |    152 | 231.00
 194    |    210 | 144.00
(5 rows)
```

Correlated subquery patterns that are not supported

The query planner uses a query rewrite method called subquery decorrelation to optimize several patterns of correlated subqueries for execution in an MPP environment. A few types of correlated subqueries follow patterns that AWS Clean Rooms can't decorrelate and doesn't support. Queries that contain the following correlation references return errors:

- Correlation references that skip a query block, also known as "skip-level correlation references." For example, in the following query, the block containing the correlation reference and the skipped block are connected by a NOT EXISTS predicate:

  ```
  select event.eventname from event
  where not exists
  (select * from listing
  where not exists
  (select * from sales where event.eventid=sales.eventid));
  ```

  The skipped block in this case is the subquery against the LISTING table. The correlation reference correlates the EVENT and SALES tables.

- Correlation references from a subquery that is part of an ON clause in an outer query:

  ```
  select * from category
  left join event
  on category.catid=event.catid and eventid =
  (select max(eventid) from sales where sales.eventid=event.eventid);
  ```

  The ON clause contains a correlation reference from SALES in the subquery to EVENT in the outer query.

- Null-sensitive correlation references to an AWS Clean Rooms system table. For example:

  ```
  select attrelid
  from my_locks sl, my_attribute
  where sl.table_id=my_attribute.attrelid and 1 not in
  (select 1 from my_opclass where sl.lock_owner = opcowner);
  ```

- Correlation references from within a subquery that contains a window function.

  ```
  select listid, qtysold
  from sales s
  ```

```
where qtysold not in
(select sum(numtickets) over() from listing l where s.listid=l.listid);
```

- References in a GROUP BY column to the results of a correlated subquery. For example:

```
select listing.listid,
(select count (sales.listid) from sales where sales.listid=listing.listid) as list
from listing
group by list, listing.listid;
```

- Correlation references from a subquery with an aggregate function and a GROUP BY clause, connected to the outer query by an IN predicate. (This restriction doesn't apply to MIN and MAX aggregate functions.) For example:

```
select * from listing where listid in
(select sum(qtysold)
from sales
where numtickets>4
group by salesid);
```

SQL functions in AWS Clean Rooms

AWS Clean Rooms supports the following SQL functions:

Topics

Aggregate functions

AWS Clean Rooms supports the following aggregate functions:

Topics

- MAX function
- MEDIAN function
- MIN function
- PERCENTILE_CONT function
- STDDEV_SAMP and STDDEV_POP functions
- SUM and SUM DISTINCT functions
- VAR_SAMP and VAR_POP functions

ANY_VALUE function

The ANY_VALUE function returns any value from the input expression values nondeterministically. This function can return NULL if the input expression doesn't result in any rows being returned.

Syntax

```
ANY_VALUE ( [ DISTINCT | ALL ] expression )
```

Arguments

DISTINCT | ALL

Specify either DISTINCT or ALL to return any value from the input expression values. The DISTINCT argument has no effect and is ignored.

expression

The target column or expression on which the function operates. The *expression* is one of the following data types:

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- REAL
- DOUBLE PRECISON
- BOOLEAN

- CHAR
- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- TIME
- TIMETZ
- VARBYTE
- SUPER

Returns

Returns the same data type as *expression*.

Usage notes

If a statement that specifies the ANY_VALUE function for a column also includes a second column reference, the second column must appear in a GROUP BY clause or be included in an aggregate function.

Examples

The following example returns an instance of any `dateid` where the `eventname` is `Eagles`.

```
select any_value(dateid) as dateid, eventname from event where eventname ='Eagles'
 group by eventname;
```

Following are the results.

```
dateid | eventname
-------+---------------
 1878  | Eagles
```

The following example returns an instance of any `dateid` where the `eventname` is `Eagles` or `Cold War Kids`.

```
select any_value(dateid) as dateid, eventname from event where eventname in('Eagles',
 'Cold War Kids') group by eventname;
```

Following are the results.

```
dateid | eventname
-------+---------------
 1922  | Cold War Kids
 1878  | Eagles
```

APPROXIMATE PERCENTILE_DISC function

APPROXIMATE PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the given set. Approximation enables the function to run much faster, with a low relative error of around 0.5 percent.

For a given *percentile* value, APPROXIMATE PERCENTILE_DISC uses a quantile summary algorithm to approximate the discrete percentile of the expression in the ORDER BY clause. APPROXIMATE PERCENTILE_DISC returns the value with the smallest cumulative distribution value (with respect to the same sort specification) that is greater than or equal to *percentile*.

APPROXIMATE PERCENTILE_DISC is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
APPROXIMATE  PERCENTILE_DISC ( percentile )
WITHIN GROUP (ORDER BY expr)
```

Arguments

percentile

Numeric constant between 0 and 1. Nulls are ignored in the calculation.

WITHIN GROUP (ORDER BY *expr*)

Clause that specifies numeric or date/time values to sort and compute the percentile over.

Returns

The same data type as the ORDER BY expression in the WITHIN GROUP clause.

Usage notes

If the APPROXIMATE PERCENTILE_DISC statement includes a GROUP BY clause, the result set is limited. The limit varies based on node type and the number of nodes. If the limit is exceeded, the function fails and returns the following error.

```
GROUP BY limit for approximate percentile_disc exceeded.
```

If you need to evaluate more groups than the limit permits, consider using PERCENTILE_CONT function.

Examples

The following example returns the number of sales, total sales, and fiftieth percentile value for the top 10 dates.

```
select top 10 date.caldate,
count(totalprice), sum(totalprice),
approximate percentile_disc(0.5)
within group (order by totalprice)
from listing
join date on listing.dateid = date.dateid
group by date.caldate
order by 3 desc;

caldate    | count | sum        | percentile_disc
-----------+-------+------------+----------------
2008-01-07 |   658 | 2081400.00 |         2020.00
2008-01-02 |   614 | 2064840.00 |         2178.00
2008-07-22 |   593 | 1994256.00 |         2214.00
2008-01-26 |   595 | 1993188.00 |         2272.00
2008-02-24 |   655 | 1975345.00 |         2070.00
2008-02-04 |   616 | 1972491.00 |         1995.00
2008-02-14 |   628 | 1971759.00 |         2184.00
2008-09-01 |   600 | 1944976.00 |         2100.00
2008-07-29 |   597 | 1944488.00 |         2106.00
2008-07-23 |   592 | 1943265.00 |         1974.00
```

AVG function

The AVG function returns the average (arithmetic mean) of the input expression values. The AVG function works with numeric values and ignores NULL values.

Syntax

```
AVG (column)
```

Arguments

column

The target column that the function operates on. The column is one of the following data types:

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- DOUBLE

Data types

The argument types supported by the AVG function are SMALLINT, INTEGER, BIGINT, DECIMAL, and DOUBLE.

The return types supported by the AVG function are:

- BIGINT for any integer type argument
- DOUBLE for a floating point argument
- Returns the same data type as expression for any other argument type

The default precision for an AVG function result with a DECIMAL argument is 38. The scale of the result is the same as the scale of the argument. For example, an AVG of a DEC(5,2) column returns a DEC(38,2) data type.

Example

Find the average quantity sold per transaction from the SALES table.

```
select avg(qtysold)from sales;
```

BOOL_AND function

The BOOL_AND function operates on a single Boolean or integer column or expression. This function applies similar logic to the BIT_AND and BIT_OR functions. For this function, the return type is a Boolean value (`true` or `false`).

If all values in a set are true, the BOOL_AND function returns `true` (`t`). If any value is false, the function returns `false` (`f`).

Syntax

```
BOOL_AND ( [DISTINCT | ALL] expression )
```

Arguments

expression

The target column or expression that the function operates on. This expression must have a BOOLEAN or integer data type. The return type of the function is BOOLEAN.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default.

Examples

You can use the Boolean functions against either Boolean expressions or integer expressions.

For example, the following query return results from the standard USERS table in the TICKIT database, which has several Boolean columns.

The BOOL_AND function returns `false` for all five rows. Not all users in each of those states likes sports.

```
select state, bool_and(likesports) from users
group by state order by state limit 5;
```

```
state | bool_and
------+---------
AB    | f
AK    | f
AL    | f
AZ    | f
BC    | f
(5 rows)
```

BOOL_OR function

The BOOL_OR function operates on a single Boolean or integer column or expression. This function applies similar logic to the BIT_AND and BIT_OR functions. For this function, the return type is a Boolean value (`true`, `false`, or NULL).

If a value in a set is `true`, the BOOL_OR function returns `true` (t). If a value in a set is `false`, the function returns `false` (f). NULL can be returned if the value is unknown.

Syntax

```
BOOL_OR ( [DISTINCT | ALL] expression )
```

Arguments

expression

The target column or expression that the function operates on. This expression must have a BOOLEAN or integer data type. The return type of the function is BOOLEAN.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values for the specified expression before calculating the result. With the argument ALL, the function retains all duplicate values. ALL is the default.

Examples

You can use the Boolean functions with either Boolean expressions or integer expressions. For example, the following query return results from the standard USERS table in the TICKIT database, which has several Boolean columns.

The BOOL_OR function returns `true` for all five rows. At least one user in each of those states likes sports.

```
select state, bool_or(likesports) from users
group by state order by state limit 5;

state | bool_or
------+--------
AB    | t
AK    | t
AL    | t
AZ    | t
BC    | t
(5 rows)
```

The following example returns NULL.

```
SELECT BOOL_OR(NULL = '123')
               bool_or
------
NULL
```

COUNT and COUNT DISTINCT functions

The COUNT function counts the rows defined by the expression. The COUNT DISTINCT function computes the number of distinct non-NULL values in a column or expression. It eliminates all duplicate values from the specified expression before doing the count.

Syntax

```
COUNT (column)
```

```
COUNT (DISTINCT column)
```

Arguments

column

The target column that the function operates on.

Data types

The COUNT function and the COUNT DISTINCT function supports all argument data types.

The COUNT DISTINCT function returns BIGINT.

Examples

Count all of the users from the state of Florida.

```
select count (identifier) from users where state='FL';
```

Count all of the unique venue IDs from the EVENT table.

```
select count (distinct (venueid)) as venues from event;
```

COUNT function

The COUNT function counts the rows defined by the expression.

The COUNT function has the following variations.

- COUNT (*) counts all the rows in the target table whether they include nulls or not.
- COUNT (*expression*) computes the number of rows with non-NULL values in a specific column or expression.
- COUNT (DISTINCT *expression*) computes the number of distinct non-NULL values in a column or expression.
- APPROXIMATE COUNT DISTINCT approximates the number of distinct non-NULL values in a column or expression.

Syntax

```
COUNT( * | expression )
```

```
COUNT ( [ DISTINCT | ALL ] expression )
```

```
APPROXIMATE COUNT ( DISTINCT expression )
```

Arguments

expression

The target column or expression that the function operates on. The COUNT function supports all argument data types.

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before doing the count. With the argument ALL, the function retains all duplicate values from the expression for counting. ALL is the default.

APPROXIMATE

When used with APPROXIMATE, a COUNT DISTINCT function uses a HyperLogLog algorithm to approximate the number of distinct non-NULL values in a column or expression. Queries that use the APPROXIMATE keyword run much faster, with a low relative error of around 2%. Approximation is warranted for queries that return a large number of distinct values, in the millions or more per query, or per group, if there is a group by clause. For smaller sets of distinct values, in the thousands, approximation might be slower than a precise count. APPROXIMATE can only be used with COUNT DISTINCT.

Return type

The COUNT function returns BIGINT.

Examples

Count all of the users from the state of Florida:

```
select count(*) from users where state='FL';

count
-------
510
```

Count all of the event names from the EVENT table:

```
select count(eventname) from event;

count
```

```
-------
8798
```

Count all of the event names from the EVENT table:

```
select count(all eventname) from event;

count
-------
8798
```

Count all of the unique venue IDs from the EVENT table:

```
select count(distinct venueid) as venues from event;

venues
--------
204
```

Count the number of times each seller listed batches of more than four tickets for sale. Group the results by seller ID:

```
select count(*), sellerid from listing
where numtickets > 4
group by sellerid
order by 1 desc, 2;

count | sellerid
------+----------
12    |    6386
11    |    17304
11    |    20123
11    |    25428
...
```

The following examples compare the return values and execution times for COUNT and APPROXIMATE COUNT.

```
select  count(distinct pricepaid) from sales;

count
```

```
-------
  4528


Time: 48.048 ms


select approximate count(distinct pricepaid) from sales;

count
-------
  4553


Time: 21.728 ms
```

LISTAGG function

For each group in a query, the LISTAGG aggregate function orders the rows for that group according to the ORDER BY expression, then concatenates the values into a single string.

LISTAGG is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
LISTAGG( [DISTINCT] aggregate_expression [, 'delimiter' ] )
[ WITHIN GROUP (ORDER BY order_list) ]
```

Arguments

DISTINCT

(Optional) A clause that eliminates duplicate values from the specified expression before concatenating. Trailing spaces are ignored, so the strings `'a'` and `'a  '` are treated as duplicates. LISTAGG uses the first value encountered. For more information, see Significance of trailing blanks.

aggregate_expression

Any valid expression (such as a column name) that provides the values to aggregate. NULL values and empty strings are ignored.

delimiter

(Optional) The string constant to separate the concatenated values. The default is NULL.

AWS Clean Rooms supports any amount of leading or trailing whitespace around an optional comma or colon as well as an empty string or any number of spaces.

Examples of valid values are:

```
", "
```

```
": "
```

```
" "
```

WITHIN GROUP (ORDER BY order_list)

(Optional) A clause that specifies the sort order of the aggregated values.

Returns

VARCHAR(MAX). If the result set is larger than the maximum VARCHAR size (64K – 1, or 65535), then LISTAGG returns the following error:

```
Invalid operation: Result size exceeds LISTAGG limit
```

Usage notes

If a statement includes multiple LISTAGG functions that use WITHIN GROUP clauses, each WITHIN GROUP clause must use the same ORDER BY values.

For example, the following statement will return an error.

```
select listagg(sellerid)
within group (order by dateid) as sellers,
listagg(dateid)
within group (order by sellerid) as dates
from winsales;
```

The following statements will run successfully.

```
select listagg(sellerid)
```

```
within group (order by dateid) as sellers,
listagg(dateid)
within group (order by dateid) as dates
from winsales;

select listagg(sellerid)
within group (order by dateid) as sellers,
listagg(dateid) as dates
from winsales;
```

Examples

The following example aggregates seller IDs, ordered by seller ID.

```
select listagg(sellerid, ', ') within group (order by sellerid) from sales
where eventid = 4337;
listagg

------------------------------------------------------------------------------------------------
380, 380, 1178, 1178, 1178, 2731, 8117, 12905, 32043, 32043, 32043, 32432, 32432,
 38669, 38750, 41498, 45676, 46324, 47188, 47188, 48294
```

The following example uses DISTINCT to return a list of unique seller IDs.

```
select listagg(distinct sellerid, ', ') within group (order by sellerid) from sales
where eventid = 4337;

listagg

------------------------------------------------------------------------------------------------
380, 1178, 2731, 8117, 12905, 32043, 32432, 38669, 38750, 41498, 45676, 46324, 47188,
 48294
```

The following example aggregates seller IDs in date order.

```
select listagg(sellerid)
within group (order by dateid)
from winsales;

   listagg
-------------
 31141242333
```

The following example returns a pipe-separated list of sales dates for buyer B.

```
select listagg(dateid,'|')
within group (order by sellerid desc,salesid asc)
from winsales
where buyerid  = 'b';

             listagg
-------------------------------------
2003-08-02|2004-04-18|2004-04-18|2004-02-12
```

The following example returns a comma-separated list of sales IDs for each buyer ID.

```
select buyerid,
listagg(salesid,',')
within group (order by salesid) as sales_id
from winsales
group by buyerid
order by buyerid;

   buyerid | sales_id
-----------+------------------------
        a  |10005,40001,40005
        b  |20001,30001,30004,30003
        c  |10001,20002,30007,10006
```

MAX function

The MAX function returns the maximum value in a set of rows. DISTINCT or ALL might be used but do not affect the result.

Syntax

```
MAX ( [ DISTINCT | ALL ] expression )
```

Arguments

expression

The target column or expression that the function operates on. The *expression* is one of the following data types:

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- REAL
- DOUBLE PRECISON
- CHAR
- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- TIME
- TIMETZ
- VARBYTE
- SUPER

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before calculating the maximum. With the argument ALL, the function retains all duplicate values from the expression for calculating the maximum. ALL is the default.

Data types

Returns the same data type as *expression*.

Examples

Find the highest price paid from all sales:

```
select max(pricepaid) from sales;

max
----------
12624.00
(1 row)
```

Find the highest price paid per ticket from all sales:

```
select max(pricepaid/qtysold) as max_ticket_price
from sales;

max_ticket_price
-----------------
2500.00000000
(1 row)
```

MEDIAN function

Calculates the median value for the range of values. NULL values in the range are ignored.

MEDIAN is an inverse distribution function that assumes a continuous distribution model.

MEDIAN is a special case of PERCENTILE_CONT(.5).

MEDIAN is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
MEDIAN ( median_expression )
```

Arguments

median_expression

The target column or expression that the function operates on.

Data types

The return type is determined by the data type of *median_expression*. The following table shows the return type for each *median_expression* data type.

Input type	Return type
NUMERIC, DECIMAL	DECIMAL
FLOAT, DOUBLE	DOUBLE

Input type	Return type
DATE	DATE
TIMESTAMP	TIMESTAMP
TIMESTAMPTZ	TIMESTAMPTZ

Usage notes

If the *median_expression* argument is a DECIMAL data type defined with the maximum precision of 38 digits, it is possible that MEDIAN will return either an inaccurate result or an error. If the return value of the MEDIAN function exceeds 38 digits, the result is truncated to fit, which causes a loss of precision. If, during interpolation, an intermediate result exceeds the maximum precision, a numeric overflow occurs and the function returns an error. To avoid these conditions, we recommend either using a data type with lower precision or casting the *median_expression* argument to a lower precision.

If a statement includes multiple calls to sort-based aggregate functions (LISTAGG, PERCENTILE_CONT, or MEDIAN), they must all use the same ORDER BY values. Note that MEDIAN applies an implicit order by on the expression value.

For example, the following statement returns an error.

```
select top 10 salesid, sum(pricepaid),
percentile_cont(0.6) within group (order by salesid),
median (pricepaid)
from sales group by salesid, pricepaid;

An error occurred when executing the SQL command:
select top 10 salesid, sum(pricepaid),
percentile_cont(0.6) within group (order by salesid),
median (pricepaid)
from sales group by salesid, pricepai...

ERROR: within group ORDER BY clauses for aggregate functions must be the same
```

The following statement runs successfully.

```
select top 10 salesid, sum(pricepaid),
```

```
percentile_cont(0.6) within group (order by salesid),
median (salesid)
from sales group by salesid, pricepaid;
```

Examples

The following example shows that MEDIAN produces the same results as PERCENTILE_CONT(0.5).

```
select top 10  distinct sellerid, qtysold,
percentile_cont(0.5) within group (order by qtysold),
median (qtysold)
from sales
group by sellerid, qtysold;

sellerid | qtysold | percentile_cont | median
---------+---------+-----------------+-------
       1 |       1 |             1.0 |    1.0
       2 |       3 |             3.0 |    3.0
       5 |       2 |             2.0 |    2.0
       9 |       4 |             4.0 |    4.0
      12 |       1 |             1.0 |    1.0
      16 |       1 |             1.0 |    1.0
      19 |       2 |             2.0 |    2.0
      19 |       3 |             3.0 |    3.0
      22 |       2 |             2.0 |    2.0
      25 |       2 |             2.0 |    2.0
```

MIN function

The MIN function returns the minimum value in a set of rows. DISTINCT or ALL might be used but do not affect the result.

Syntax

```
MIN ( [ DISTINCT | ALL ] expression )
```

Arguments

expression

The target column or expression that the function operates on. The *expression* is one of the following data types:

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- REAL
- DOUBLE PRECISON
- CHAR
- VARCHAR
- DATE
- TIMESTAMP
- TIMESTAMPTZ
- TIME
- TIMETZ
- VARBYTE
- SUPER

DISTINCT | ALL

With the argument DISTINCT, the function eliminates all duplicate values from the specified expression before calculating the minimum. With the argument ALL, the function retains all duplicate values from the expression for calculating the minimum. ALL is the default.

Data types

Returns the same data type as *expression*.

Examples

Find the lowest price paid from all sales:

```
select min(pricepaid) from sales;

min
-------
20.00
```

```
(1 row)
```

Find the lowest price paid per ticket from all sales:

```
select min(pricepaid/qtysold)as min_ticket_price
from sales;

min_ticket_price
------------------
20.00000000
(1 row)
```

PERCENTILE_CONT function

PERCENTILE_CONT is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into the given percentile value with respect to the sort specification.

PERCENTILE_CONT computes a linear interpolation between values after ordering them. Using the percentile value `(P)` and the number of not null rows `(N)` in the aggregation group, the function computes the row number after ordering the rows according to the sort specification. This row number `(RN)` is computed according to the formula `RN = (1+ (P*(N-1))`. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN = CEILING(RN)` and `FRN = FLOOR(RN)`.

The final result will be as follows.

If `(CRN = FRN = RN)` then the result is `(value of expression from row at RN)`

Otherwise the result is as follows:

`(CRN - RN) * (value of expression for row at FRN) + (RN - FRN) * (value of expression for row at CRN)`.

PERCENTILE_CONT is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
PERCENTILE_CONT ( percentile )
```

```
WITHIN GROUP (ORDER BY expr)
```

Arguments

percentile

Numeric constant between 0 and 1. Nulls are ignored in the calculation.

WITHIN GROUP (ORDER BY *expr*)

Specifies numeric or date/time values to sort and compute the percentile over.

Returns

The return type is determined by the data type of the ORDER BY expression in the WITHIN GROUP clause. The following table shows the return type for each ORDER BY expression data type.

Input type	Return type
SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL	DECIMAL
FLOAT, DOUBLE	DOUBLE
DATE	DATE
TIMESTAMP	TIMESTAMP
TIMESTAMPTZ	TIMESTAMPTZ

Usage notes

If the ORDER BY expression is a DECIMAL data type defined with the maximum precision of 38 digits, it is possible that PERCENTILE_CONT will return either an inaccurate result or an error. If the return value of the PERCENTILE_CONT function exceeds 38 digits, the result is truncated to fit, which causes a loss of precision.. If, during interpolation, an intermediate result exceeds the maximum precision, a numeric overflow occurs and the function returns an error. To avoid these conditions, we recommend either using a data type with lower precision or casting the ORDER BY expression to a lower precision.

If a statement includes multiple calls to sort-based aggregate functions (LISTAGG, PERCENTILE_CONT, or MEDIAN), they must all use the same ORDER BY values. Note that MEDIAN applies an implicit order by on the expression value.

For example, the following statement returns an error.

```
select top 10 salesid, sum(pricepaid),
percentile_cont(0.6) within group (order by salesid),
median (pricepaid)
from sales group by salesid, pricepaid;

An error occurred when executing the SQL command:
select top 10 salesid, sum(pricepaid),
percentile_cont(0.6) within group (order by salesid),
median (pricepaid)
from sales group by salesid, pricepai...

ERROR: within group ORDER BY clauses for aggregate functions must be the same
```

The following statement runs successfully.

```
select top 10 salesid, sum(pricepaid),
percentile_cont(0.6) within group (order by salesid),
median (salesid)
from sales group by salesid, pricepaid;
```

Examples

The following example shows that MEDIAN produces the same results as PERCENTILE_CONT(0.5).

```
select top 10  distinct sellerid, qtysold,
percentile_cont(0.5) within group (order by qtysold),
median (qtysold)
from sales
group by sellerid, qtysold;

sellerid | qtysold | percentile_cont | median
---------+---------+-----------------+-------
       1 |       1 |             1.0 |    1.0
       2 |       3 |             3.0 |    3.0
       5 |       2 |             2.0 |    2.0
       9 |       4 |             4.0 |    4.0
```

```
 12 |       1 |             1.0 |    1.0
 16 |       1 |             1.0 |    1.0
 19 |       2 |             2.0 |    2.0
 19 |       3 |             3.0 |    3.0
 22 |       2 |             2.0 |    2.0
 25 |       2 |             2.0 |    2.0
```

STDDEV_SAMP and STDDEV_POP functions

The STDDEV_SAMP and STDDEV_POP functions return the sample and population standard deviation of a set of numeric values (integer, decimal, or floating-point). The result of the STDDEV_SAMP function is equivalent to the square root of the sample variance of the same set of values.

STDDEV_SAMP and STDDEV are synonyms for the same function.

Syntax

```
STDDEV_SAMP | STDDEV ( [ DISTINCT | ALL ] expression)
STDDEV_POP ( [ DISTINCT | ALL ] expression)
```

The expression must have an integer, decimal, or floating point data type. Regardless of the data type of the expression, the return type of this function is a double precision number.

> **Note**
>
> Standard deviation is calculated using floating point arithmetic, which might result in slight imprecision.

Usage notes

When the sample standard deviation (STDDEV or STDDEV_SAMP) is calculated for an expression that consists of a single value, the result of the function is NULL not 0.

Examples

The following query returns the average of the values in the VENUESEATS column of the VENUE table, followed by the sample standard deviation and population standard deviation of the same set of values. VENUESEATS is an INTEGER column. The scale of the result is reduced to 2 digits.

```
select avg(venueseats),
cast(stddev_samp(venueseats) as dec(14,2)) stddevsamp,
cast(stddev_pop(venueseats) as dec(14,2)) stddevpop
from venue;

avg  | stddevsamp | stddevpop
-------+------------+-----------
17503 |   27847.76 |  27773.20
(1 row)
```

The following query returns the sample standard deviation for the COMMISSION column in the SALES table. COMMISSION is a DECIMAL column. The scale of the result is reduced to 10 digits.

```
select cast(stddev(commission) as dec(18,10))
from sales;

stddev
----------------
130.3912659086
(1 row)
```

The following query casts the sample standard deviation for the COMMISSION column as an integer.

```
select cast(stddev(commission) as integer)
from sales;

stddev
--------
130
(1 row)
```

The following query returns both the sample standard deviation and the square root of the sample variance for the COMMISSION column. The results of these calculations are the same.

```
select
cast(stddev_samp(commission) as dec(18,10)) stddevsamp,
cast(sqrt(var_samp(commission)) as dec(18,10)) sqrtvarsamp
from sales;

stddevsamp   |  sqrtvarsamp
```

```
----------------+----------------
130.3912659086 | 130.3912659086
(1 row)
```

SUM and SUM DISTINCT functions

The SUM function returns the sum of the input column or expression values. The SUM function works with numeric values and ignores NULL values.

The SUM DISTINCT function eliminates all duplicate values from the specified expression before calculating the sum.

Syntax

```
SUM (column)
```

```
SUM (DISTINCT column )
```

Arguments

column

The target column that the function operates on. The column is one of the following data types:

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- DOUBLE

Data types

The argument types supported by the SUM function are SMALLINT, INTEGER, BIGINT, DECIMAL, and DOUBLE.

The SUM function supports the following return types:

- BIGINT for BIGINT, SMALLINT, and INTEGER arguments

- DOUBLE for floating point arguments
- Returns the same data type as expression for any other argument type

The default precision for a SUM function result with a DECIMAL argument is 38. The scale of the result is the same as the scale of the argument. For example, a SUM of a DEC(5,2) column returns a DEC(38,2) data type.

Examples

Find the sum of all commissions paid from the SALES table.

```
select sum(commission) from sales
```

Find the sum of all distinct commissions paid from the SALES table.

```
select sum (distinct (commission)) from sales
```

VAR_SAMP and VAR_POP functions

The VAR_SAMP and VAR_POP functions return the sample and population variance of a set of numeric values (integer, decimal, or floating-point). The result of the VAR_SAMP function is equivalent to the squared sample standard deviation of the same set of values.

VAR_SAMP and VARIANCE are synonyms for the same function.

Syntax

```
VAR_SAMP | VARIANCE ( [ DISTINCT | ALL ] expression)
VAR_POP ( [ DISTINCT | ALL ] expression)
```

The expression must have an integer, decimal, or floating-point data type. Regardless of the data type of the expression, the return type of this function is a double precision number.

> **Note**
>
> The results of these functions might vary across data warehouse clusters, depending on the configuration of the cluster in each case.

Usage notes

When the sample variance (VARIANCE or VAR_SAMP) is calculated for an expression that consists of a single value, the result of the function is NULL not 0.

Examples

The following query returns the rounded sample and population variance of the NUMTICKETS column in the LISTING table.

```
select avg(numtickets),
round(var_samp(numtickets)) varsamp,
round(var_pop(numtickets)) varpop
from listing;

avg | varsamp | varpop
-----+---------+--------
10 |      54 |     54
(1 row)
```

The following query runs the same calculations but casts the results to decimal values.

```
select avg(numtickets),
cast(var_samp(numtickets) as dec(10,4)) varsamp,
cast(var_pop(numtickets) as dec(10,4)) varpop
from listing;

avg | varsamp | varpop
-----+---------+---------
10 | 53.6291 | 53.6288
(1 row)
```

Array functions

This section describes the array functions for SQL supported in AWS Clean Rooms.

Topics

- array function
- array_concat function
- array_flatten function

- get_array_length function
- split_to_array function
- subarray function

array function

Creates an array of the SUPER data type.

Syntax

```
ARRAY( [ expr1 ] [ , expr2 [ , ... ] ] )
```

Argument

expr1, *expr2*

Expressions of any data type except date and time types. The arguments don't need to be of the same data type.

Return type

The array function returns the SUPER data type.

Example

The following example shows an array of numeric values and an array of different data types.

```
--an array of numeric values
select array(1,50,null,100);
      array
------------------
 [1,50,null,100]
(1 row)

--an array of different data types
select array(1,'abc',true,3.14);
        array
----------------------
 [1,"abc",true,3.14]
```

```
(1 row)
```

array_concat function

The array_concat function concatenates two arrays to create an array that contains all the elements in the first array followed by all the elements in the second array. The two arguments must be valid arrays.

Syntax

```
array_concat( super_expr1,  super_expr2 )
```

Arguments

super_expr1

The value that specifies the first of the two arrays to concatenate.

super_expr2

The value that specifies the second of the two arrays to concatenate.

Return type

The array_concat function returns a SUPER data value.

Example

The following example shows concatenation of two arrays of the same type and concatenation of two arrays of different types.

```
-- concatenating two arrays
SELECT ARRAY_CONCAT(ARRAY(10001,10002),ARRAY(10003,10004));
              array_concat
-----------------------------------
 [10001,10002,10003,10004]
(1 row)

-- concatenating two arrays of different types
SELECT ARRAY_CONCAT(ARRAY(10001,10002),ARRAY('ab','cd'));
          array_concat
```

```
-----------------------------
 [10001,10002,"ab","cd"]
(1 row)
```

array_flatten function

Merges multiple arrays into a single array of SUPER type.

Syntax

```
array_flatten( super_expr1,super_expr2,.. )
```

Arguments

super_expr1,*super_expr2*

A valid SUPER expression of array form.

Return type

The array_flatten function returns a SUPER data value.

Example

The following example shows an array_flatten function.

```
SELECT ARRAY_FLATTEN(ARRAY(ARRAY(1,2,3,4),ARRAY(5,6,7,8),ARRAY(9,10)));
     array_flatten
-----------------------
 [1,2,3,4,5,6,7,8,9,10]
(1 row)
```

get_array_length function

Returns the length of the specified array. The GET_ARRAY_LENGTH function returns the length of a SUPER array given an object or array path.

Syntax

```
get_array_length( super_expr )
```

Arguments

super_expr

A valid SUPER expression of array form.

Return type

The get_array_length function returns a BIGINT.

Example

The following example shows a get_array_length function.

```
SELECT GET_ARRAY_LENGTH(ARRAY(1,2,3,4,5,6,7,8,9,10));
 get_array_length
----------------------
            10
(1 row)
```

split_to_array function

Uses a delimiter as an optional parameter. If no delimiter is present, then the default is a comma.

Syntax

```
split_to_array( string,delimiter )
```

Arguments

string

The input string to be split.

delimiter

An optional value on which the input string will be split. The default is a comma.

Return type

The split_to_array function returns a SUPER data value.

Example

The following example show a split_to_array function.

```
SELECT SPLIT_TO_ARRAY('12|345|6789', '|');
     split_to_array
------------------------
 ["12","345","6789"]
(1 row)
```

subarray function

Manipulates arrays to return a subset of the input arrays.

Syntax

```
SUBARRAY( super_expr, start_position, length )
```

Arguments

super_expr

A valid SUPER expression in array form.

start_position

The position within the array to begin the extraction, starting at index position 0. A negative position counts backward from the end of the array.

length

The number of elements to extract (the length of the substring).

Return type

The subarray function returns a SUPER data value.

Example

The following is an example of a subarray function.

```
 SELECT SUBARRAY(ARRAY('a', 'b', 'c', 'd', 'e', 'f'), 2, 3);
   subarray
---------------
 ["c","d","e"]
(1 row)
```

Conditional expressions

AWS Clean Rooms supports the following conditional expressions:

Topics

- CASE conditional expression
- COALESCE expression
- GREATEST and LEAST functions
- NVL and COALESCE functions
- NVL2 function
- NULLIF function

CASE conditional expression

The CASE expression is a conditional expression, similar to if/then/else statements found in other languages. CASE is used to specify a result when there are multiple conditions. Use CASE where a SQL expression is valid, such as in a SELECT command.

There are two types of CASE expressions: simple and searched.

- In simple CASE expressions, an expression is compared with a value. When a match is found, the specified action in the THEN clause is applied. If no match is found, the action in the ELSE clause is applied.
- In searched CASE expressions, each CASE is evaluated based on a Boolean expression, and the CASE statement returns the first matching CASE. If no match is found among the WHEN clauses, the action in the ELSE clause is returned.

Syntax

Simple CASE statement used to match conditions:

```
CASE expression
  WHEN value THEN result
  [WHEN...]
  [ELSE result]
END
```

Searched CASE statement used to evaluate each condition:

```
CASE
  WHEN condition THEN result
  [WHEN ...]
  [ELSE result]
END
```

Arguments

expression

A column name or any valid expression.

value

Value that the expression is compared with, such as a numeric constant or a character string.

result

The target value or expression that is returned when an expression or Boolean condition is evaluated. The data types of all the result expressions must be convertible to a single output type.

condition

A Boolean expression that evaluates to true or false. If *condition* is true, the value of the CASE expression is the result that follows the condition, and the remainder of the CASE expression is not processed. If *condition* is false, any subsequent WHEN clauses are evaluated. If no WHEN condition results are true, the value of the CASE expression is the result of the ELSE clause. If the ELSE clause is omitted and no condition is true, the result is null.

Examples

Use a simple CASE expression to replace `New York City` with `Big Apple` in a query against the VENUE table. Replace all other city names with `other`.

```
select venuecity,
  case venuecity
    when 'New York City'
    then 'Big Apple' else 'other'
  end
from venue
order by venueid desc;

venuecity        |   case
-----------------+-----------
Los Angeles      | other
New York City    | Big Apple
San Francisco    | other
Baltimore        | other
...
```

Use a searched CASE expression to assign group numbers based on the PRICEPAID value for individual ticket sales:

```
select pricepaid,
  case when pricepaid <10000 then 'group 1'
    when pricepaid >10000 then 'group 2'
    else 'group 3'
  end
from sales
order by 1 desc;

pricepaid |  case
----------+---------
12624     | group 2
10000     | group 3
10000     | group 3
9996      | group 1
9988      | group 1
...
```

COALESCE expression

A COALESCE expression returns the value of the first expression in the list that is not null. If all expressions are null, the result is null. When a non-null value is found, the remaining expressions in the list are not evaluated.

This type of expression is useful when you want to return a backup value for something when the preferred value is missing or null. For example, a query might return one of three phone numbers (cell, home, or work, in that order), whichever is found first in the table (not null).

Syntax

```
COALESCE (expression, expression, ... )
```

Examples

Apply COALESCE expression to two columns.

```
select coalesce(start_date, end_date)
from datetable
order by 1;
```

The default column name for an NVL expression is COALESCE. The following query returns the same results.

```
select coalesce(start_date, end_date) from datetable order by 1;
```

GREATEST and LEAST functions

Returns the largest or smallest value from a list of any number of expressions.

Syntax

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

Parameters

expression_list

A comma-separated list of expressions, such as column names. The expressions must all be convertible to a common data type. NULL values in the list are ignored. If all of the expressions evaluate to NULL, the result is NULL.

Returns

Returns the greatest (for GREATEST) or least (for LEAST) value from the provided list of expressions.

Example

The following example returns the highest value alphabetically for `firstname` or `lastname`.

```
select firstname, lastname, greatest(firstname,lastname) from users
where userid < 10
order by 3;

 firstname | lastname  | greatest
-----------+-----------+-----------
 Alejandro | Rosalez   | Ratliff
 Carlos    | Salazar   | Carlos
 Jane      | Doe       | Doe
 John      | Doe       | Doe
 John      | Stiles    | John
 Shirley   | Rodriguez | Rodriguez
 Terry     | Whitlock  | Terry
 Richard   | Roe       | Richard
 Xiulan    | Wang      | Wang
(9 rows)
```

NVL and COALESCE functions

Returns the value of the first expression that isn't null in a series of expressions. When a non-null value is found, the remaining expressions in the list aren't evaluated.

NVL is identical to COALESCE. They are synonyms. This topic explains the syntax and contains examples for both.

Syntax

```
NVL( expression, expression, ... )
```

The syntax for COALESCE is the same:

```
COALESCE( expression, expression, ... )
```

If all expressions are null, the result is null.

These functions are useful when you want to return a secondary value when a primary value is missing or null. For example, a query might return the first of three available phone numbers: cell, home, or work. The order of the expressions in the function determines the order of evaluation.

Arguments

expression

An expression, such as a column name, to be evaluated for null status.

Return type

AWS Clean Rooms determines the data type of the returned value based on the input expressions. If the data types of the input expressions don't have a common type, then an error is returned.

Examples

If the list contains integer expressions, the function returns an integer.

```
SELECT COALESCE(NULL, 12, NULL);

coalesce
--------------
12
```

This example, which is the same as the previous example, except that it uses NVL, returns the same result.

```
SELECT NVL(NULL, 12, NULL);

coalesce
--------------
12
```

The following example returns a string type.

```
SELECT COALESCE(NULL, 'AWS Clean Rooms', NULL);
```

```
coalesce
--------------
AWS Clean Rooms
```

The following example results in an error because the data types vary in the expression list. In this case, there is both a string type and a number type in the list.

```
SELECT COALESCE(NULL, 'AWS Clean Rooms', 12);
ERROR: invalid input syntax for integer: "AWS Clean Rooms"
```

NVL2 function

Returns one of two values based on whether a specified expression evaluates to NULL or NOT NULL.

Syntax

```
NVL2 ( expression, not_null_return_value, null_return_value )
```

Arguments

expression

An expression, such as a column name, to be evaluated for null status.

not_null_return_value

The value returned if *expression* evaluates to NOT NULL. The *not_null_return_value* value must either have the same data type as *expression* or be implicitly convertible to that data type.

null_return_value

The value returned if *expression* evaluates to NULL. The *null_return_value* value must either have the same data type as *expression* or be implicitly convertible to that data type.

Return type

The NVL2 return type is determined as follows:

- If either *not_null_return_value* or *null_return_value* is null, the data type of the not-null expression is returned.

If both *not_null_return_value* and *null_return_value* are not null:

- If *not_null_return_value* and *null_return_value* have the same data type, that data type is returned.
- If *not_null_return_value* and *null_return_value* have different numeric data types, the smallest compatible numeric data type is returned.
- If *not_null_return_value* and *null_return_value* have different datetime data types, a timestamp data type is returned.
- If *not_null_return_value* and *null_return_value* have different character data types, the data type of *not_null_return_value* is returned.
- If *not_null_return_value* and *null_return_value* have mixed numeric and non-numeric data types, the data type of *not_null_return_value* is returned.

> **Important**
>
> In the last two cases where the data type of *not_null_return_value* is returned, *null_return_value* is implicitly cast to that data type. If the data types are incompatible, the function fails.

Usage notes

For NVL2, the return will have the value of either the *not_null_return_value* or *null_return_value* parameter, whichever is selected by the function, but will have the data type of *not_null_return_value*.

For example, assuming column1 is NULL, the following queries will return the same value. However, the DECODE return value data type will be INTEGER and the NVL2 return value data type will be VARCHAR.

```
select decode(column1, null, 1234, '2345');
select nvl2(column1, '2345', 1234);
```

Example

The following example modifies some sample data, then evaluates two fields to provide appropriate contact information for users:

```
update users set email = null where firstname = 'Aphrodite' and lastname = 'Acevedo';

select (firstname + ' ' + lastname) as name,
nvl2(email, email, phone) AS contact_info
from users
where state = 'WA'
and lastname  like 'A%'
order by lastname, firstname;

name        contact_info
--------------------+--------------------------------------------
Aphrodite Acevedo (555) 555-0100
Caldwell Acevedo  Nunc.sollicitudin@example.ca
Quinn Adams     vel@example.com
Kamal Aguilar   quis@example.com
Samson Alexander  hendrerit.neque@example.com
Hall Alford     ac.mattis@example.com
Lane Allen      et.netus@example.com
Xander Allison    ac.facilisis.facilisis@example.com
Amaya Alvarado    dui.nec.tempus@example.com
Vera Alvarez    at.arcu.Vestibulum@example.com
Yetta Anthony   enim.sit@example.com
Violet Arnold   ad.litora@example.comm
August Ashley   consectetuer.euismod@example.com
Karyn Austin    ipsum.primis.in@example.com
Lucas Ayers     at@example.com
```

NULLIF function

Syntax

The NULLIF expression compares two arguments and returns null if the arguments are equal. If they are not equal, the first argument is returned. This expression is the inverse of the NVL or COALESCE expression.

```
NULLIF ( expression1, expression2 )
```

Arguments

expression1, expression2

The target columns or expressions that are compared. The return type is the same as the type of the first expression. The default column name of the NULLIF result is the column name of the first expression.

Examples

In the following example, the query returns the string `first` because the arguments are not equal.

```
SELECT NULLIF('first', 'second');

case
-------
first
```

In the following example, the query returns NULL because the string literal arguments are equal.

```
SELECT NULLIF('first', 'first');

case
-------
NULL
```

In the following example, the query returns 1 because the integer arguments are not equal.

```
SELECT NULLIF(1, 2);

case
-------
1
```

In the following example, the query returns NULL because the integer arguments are equal.

```
SELECT NULLIF(1, 1);

case
```

```
-------
NULL
```

In the following example, the query returns null when the LISTID and SALESID values match:

```
select nullif(listid,salesid), salesid
from sales where salesid<10 order by 1, 2 desc;

listid  | salesid
--------+---------
     4  |       2
     5  |       4
     5  |       3
     6  |       5
     10 |       9
     10 |       8
     10 |       7
     10 |       6
        |       1
(9 rows)
```

Data type formatting functions

Using a data type formatting function, you can convert values from one data type to another. For each of these functions, the first argument is always the value to be formatted and the second argument contains the template for the new format. AWS Clean Rooms supports several data type formatting functions.

Topics

- CAST function
- CONVERT function
- TO_CHAR
- TO_DATE function
- TO_NUMBER
- Datetime format strings
- Numeric format strings
- Teradata-style formatting characters for numeric data

CAST function

The CAST function converts one data type to another compatible data type. For instance, you can convert a string to a date, or a numeric type to a string. CAST performs a runtime conversion, which means that the conversion doesn't change a value's data type in a source table. It's changed only in the context of the query.

The CAST function is very similar to the section called “CONVERT”, in that they both convert one data type to another, but they are called differently.

Certain data types require an explicit conversion to other data types using the CAST or CONVERT function. Other data types can be converted implicitly, as part of another command, without using CAST or CONVERT. See Type compatibility and conversion.

Syntax

Use either of these two equivalent syntax forms to cast expressions from one data type to another.

```
CAST ( expression AS type )
expression :: type
```

Arguments

expression

An expression that evaluates to one or more values, such as a column name or a literal. Converting null values returns nulls. The expression can't contain blank or empty strings.

type

One of the supported Data types, except for VARBYTE, BINARY, and BINARY VARYING data types.

Return type

CAST returns the data type specified by the *type* argument.

> **Note**
>
> AWS Clean Rooms returns an error if you try to perform a problematic conversion, such as a DECIMAL conversion that loses precision, like the following:

```
select 123.456::decimal(2,1);
```

or an INTEGER conversion that causes an overflow:

```
select 12345678::smallint;
```

Examples

The following two queries are equivalent. They both cast a decimal value to an integer:

```
select cast(pricepaid as integer)
from sales where salesid=100;

pricepaid
-----------
162
(1 row)
```

```
select pricepaid::integer
from sales where salesid=100;

pricepaid
-----------
162
(1 row)
```

The following produces a similar result. It doesn't require sample data to run:

```
select cast(162.00 as integer) as pricepaid;

pricepaid
-----------
162
(1 row)
```

In this example, the values in a timestamp column are cast as dates, which results in removing the time from each result:

```
select cast(saletime as date), salesid
from sales order by salesid limit 10;

 saletime  | salesid
-----------+---------
2008-02-18 |       1
2008-06-06 |       2
2008-06-06 |       3
2008-06-09 |       4
2008-08-31 |       5
2008-07-16 |       6
2008-06-26 |       7
2008-07-10 |       8
2008-07-22 |       9
2008-08-06 |      10

(10 rows)
```

If you didn't use CAST as illustrated in the previous sample, the results would include the time: *2008-02-18 02:36:48*.

The following query casts variable character data to a date. It doesn't require sample data to run.

```
select cast('2008-02-18 02:36:48' as date) as mysaletime;

mysaletime
--------------------
2008-02-18
(1 row)
```

In this example, the values in a date column are cast as timestamps:

```
select cast(caldate as timestamp), dateid
from date order by dateid limit 10;

      caldate       | dateid
--------------------+--------
2008-01-01 00:00:00 |   1827
2008-01-02 00:00:00 |   1828
2008-01-03 00:00:00 |   1829
2008-01-04 00:00:00 |   1830
2008-01-05 00:00:00 |   1831
```

```
2008-01-06 00:00:00 |   1832
2008-01-07 00:00:00 |   1833
2008-01-08 00:00:00 |   1834
2008-01-09 00:00:00 |   1835
2008-01-10 00:00:00 |   1836

(10 rows)
```

In a case like the previous sample, you can gain additional control over output formatting by using TO_CHAR.

In this example, an integer is cast as a character string:

```
select cast(2008 as char(4));

bpchar
--------
2008
```

In this example, a DECIMAL(6,3) value is cast as a DECIMAL(4,1) value:

```
select cast(109.652 as decimal(4,1));

numeric
---------
109.7
```

This example shows a more complex expression. The PRICEPAID column (a DECIMAL(8,2) column) in the SALES table is converted to a DECIMAL(38,2) column and the values are multiplied by 100000000000000000000:

```
select salesid, pricepaid::decimal(38,2)*100000000000000000000
as value from sales where salesid<10 order by salesid;


 salesid |           value
---------+----------------------------
       1 | 728000000000000000000000.00
       2 |  760000000000000000000.00
       3 | 350000000000000000000000.00
       4 | 175000000000000000000000.00
```

```
       5 | 154000000000000000000000.00
       6 | 394000000000000000000000.00
       7 | 788000000000000000000000.00
       8 | 197000000000000000000000.00
       9 | 591000000000000000000000.00

(9 rows)
```

CONVERT function

Like the CAST function, the CONVERT function converts one data type to another compatible data type. For instance, you can convert a string to a date, or a numeric type to a string. CONVERT performs a runtime conversion, which means that the conversion doesn't change a value's data type in a source table. It's changed only in the context of the query.

Certain data types require an explicit conversion to other data types using the CONVERT function. Other data types can be converted implicitly, as part of another command, without using CAST or CONVERT. See Type compatibility and conversion.

Syntax

```
CONVERT ( type, expression )
```

Arguments

type

One of the supported Data types, except for VARBYTE, BINARY, and BINARY VARYING data types.

expression

An expression that evaluates to one or more values, such as a column name or a literal. Converting null values returns nulls. The expression can't contain blank or empty strings.

Return type

CONVERT returns the data type specified by the *type* argument.

Note

AWS Clean Rooms returns an error if you try to perform a problematic conversion, such as a DECIMAL conversion that loses precision, like the following:

```
SELECT CONVERT(decimal(2,1), 123.456);
```

or an INTEGER conversion that causes an overflow:

```
SELECT CONVERT(smallint, 12345678);
```

Examples

The following query uses the CONVERT function to convert a column of decimals into integers

```
SELECT CONVERT(integer, pricepaid)
FROM sales WHERE salesid=100;
```

This example converts an integer into a character string.

```
SELECT CONVERT(char(4), 2008);
```

In this example, the current date and time is converted to a variable character data type:

```
SELECT CONVERT(VARCHAR(30), GETDATE());

getdate
---------
2023-02-02 04:31:16
```

This example converts the saletime column into just the time, removing the dates from each row.

```
SELECT CONVERT(time, saletime), salesid
FROM sales order by salesid limit 10;
```

The following example converts variable character data into a datetime object.

```
SELECT CONVERT(datetime, '2008-02-18 02:36:48') as mysaletime;
```

TO_CHAR

TO_CHAR converts a timestamp or numeric expression to a character-string data format.

Syntax

```
TO_CHAR (timestamp_expression | numeric_expression , 'format')
```

Arguments

timestamp_expression

An expression that results in a TIMESTAMP or TIMESTAMPTZ type value or a value that can implicitly be coerced to a timestamp.

numeric_expression

An expression that results in a numeric data type value or a value that can implicitly be coerced to a numeric type. For more information, see Numeric types. TO_CHAR inserts a space to the left of the numeral string.

> **Note**
>
> TO_CHAR doesn't support 128-bit DECIMAL values.

format

The format for the new value. For valid formats, see Datetime format strings and Numeric format strings.

Return type

VARCHAR

Examples

The following example converts a timestamp to a value with the date and time in a format with the name of the month padded to nine characters, the name of the day of the week, and the day number of the month.

```
select to_char(timestamp '2009-12-31 23:15:59', 'MONTH-DY-DD-YYYY HH12:MIPM');
to_char
-------------------------
DECEMBER -THU-31-2009 11:15PM
```

The following example converts a timestamp to a value with day number of the year.

```
select to_char(timestamp '2009-12-31 23:15:59', 'DDD');

to_char
-------------------------
365
```

The following example converts a timestamp to an ISO day number of the week.

```
select to_char(timestamp '2022-05-16 23:15:59', 'ID');

to_char
-------------------------
1
```

The following example extracts the month name from a date.

```
select to_char(date '2009-12-31', 'MONTH');

to_char
-------------------------
DECEMBER
```

The following example converts each STARTTIME value in the EVENT table to a string that consists of hours, minutes, and seconds.

```
select to_char(starttime, 'HH12:MI:SS')
from event where eventid between 1 and 5
order by eventid;

to_char
----------
02:30:00
08:00:00
02:30:00
```

```
02:30:00
07:00:00
(5 rows)
```

The following example converts an entire timestamp value into a different format.

```
select starttime, to_char(starttime, 'MON-DD-YYYY HH12:MIPM')
from event where eventid=1;

      starttime      |       to_char
---------------------+---------------------
 2008-01-25 14:30:00 | JAN-25-2008 02:30PM
(1 row)
```

The following example converts a timestamp literal to a character string.

```
select to_char(timestamp '2009-12-31 23:15:59','HH24:MI:SS');
to_char
----------
23:15:59
(1 row)
```

The following example converts a number to a character string with the negative sign at the end.

```
select to_char(-125.8, '999D99S');
to_char
---------
125.80-
(1 row)
```

The following example converts a number to a character string with the currency symbol.

```
select to_char(-125.88, '$S999D99');
to_char
---------
$-125.88
(1 row)
```

The following example converts a number to a character string using angle brackets for negative numbers.

```
select to_char(-125.88, '$999D99PR');
to_char
---------
$<125.88>
(1 row)
```

The following example converts a number to a Roman numeral string.

```
select to_char(125, 'RN');
to_char
---------
CXXV
(1 row)
```

The following example displays the day of the week.

```
SELECT to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS');
               to_char
-----------------------
Wednesday, 31 09:34:26
```

The following example displays the ordinal number suffix for a number.

```
SELECT to_char(482, '999th');
               to_char
-----------------------
 482nd
```

The following example subtracts the commission from the price paid in the sales table. The difference is then rounded up and converted to a roman numeral, shown in the `to_char` column:

```
select salesid, pricepaid, commission, (pricepaid - commission)
as difference, to_char(pricepaid - commission, 'rn') from sales
group by sales.pricepaid, sales.commission, salesid
order by salesid limit 10;

 salesid | pricepaid | commission | difference |     to_char
---------+-----------+------------+------------+-----------------
       1 |    728.00 |     109.20 |     618.80 |           dcxix
       2 |     76.00 |      11.40 |      64.60 |             lxv
```

```
       3 |    350.00 |      52.50 |     297.50 |        ccxcviii
       4 |    175.00 |      26.25 |     148.75 |           cxlix
       5 |    154.00 |      23.10 |     130.90 |           cxxxi
       6 |    394.00 |      59.10 |     334.90 |         cccxxxv
       7 |    788.00 |     118.20 |     669.80 |           dclxx
       8 |    197.00 |      29.55 |     167.45 |          clxvii
       9 |    591.00 |      88.65 |     502.35 |             dii
      10 |     65.00 |       9.75 |      55.25 |              lv
(10 rows)
```

The following example adds the currency symbol to the difference values shown in the `to_char` column:

```
select salesid, pricepaid, commission, (pricepaid - commission)
as difference, to_char(pricepaid - commission, 'l99999D99') from sales
group by sales.pricepaid, sales.commission, salesid
order by salesid limit 10;

salesid | pricepaid | commission | difference |  to_char
--------+-----------+------------+------------+------------
      1 |    728.00 |     109.20 |     618.80 | $   618.80
      2 |     76.00 |      11.40 |      64.60 | $    64.60
      3 |    350.00 |      52.50 |     297.50 | $   297.50
      4 |    175.00 |      26.25 |     148.75 | $   148.75
      5 |    154.00 |      23.10 |     130.90 | $   130.90
      6 |    394.00 |      59.10 |     334.90 | $   334.90
      7 |    788.00 |     118.20 |     669.80 | $   669.80
      8 |    197.00 |      29.55 |     167.45 | $   167.45
      9 |    591.00 |      88.65 |     502.35 | $   502.35
     10 |     65.00 |       9.75 |      55.25 | $    55.25
(10 rows)
```

The following example lists the century in which each sale was made.

```
select salesid, saletime, to_char(saletime, 'cc') from sales
order by salesid limit 10;

 salesid |      saletime       | to_char
---------+---------------------+---------
       1 | 2008-02-18 02:36:48 | 21
       2 | 2008-06-06 05:00:16 | 21
       3 | 2008-06-06 08:26:17 | 21
       4 | 2008-06-09 08:38:52 | 21
```

```
       5 | 2008-08-31 09:17:02 | 21
       6 | 2008-07-16 11:59:24 | 21
       7 | 2008-06-26 12:56:06 | 21
       8 | 2008-07-10 02:12:36 | 21
       9 | 2008-07-22 02:23:17 | 21
      10 | 2008-08-06 02:51:55 | 21
(10 rows)
```

The following example converts each STARTTIME value in the EVENT table to a string that consists of hours, minutes, seconds, and time zone.

```
select to_char(starttime, 'HH12:MI:SS TZ')
from event where eventid between 1 and 5
order by eventid;

to_char
----------
02:30:00 UTC
08:00:00 UTC
02:30:00 UTC
02:30:00 UTC
07:00:00 UTC
(5 rows)

(10 rows)
```

The following example shows formatting for seconds, milliseconds, and microseconds.

```
select sysdate,
to_char(sysdate, 'HH24:MI:SS') as seconds,
to_char(sysdate, 'HH24:MI:SS.MS') as milliseconds,
to_char(sysdate, 'HH24:MI:SS:US') as microseconds;

timestamp           | seconds  | milliseconds | microseconds
--------------------+----------+--------------+----------------
2015-04-10 18:45:09 | 18:45:09 | 18:45:09.325 | 18:45:09:325143
```

TO_DATE function

TO_DATE converts a date represented by a character string to a DATE data type.

Syntax

```
TO_DATE(string, format)
```

```
TO_DATE(string, format, is_strict)
```

Arguments

string

A string to be converted.

format

A string literal that defines the format of the input *string*, in terms of its date parts. For a list of valid day, month, and year formats, see Datetime format strings.

is_strict

An optional Boolean value that specifies whether an error is returned if an input date value is out of range. When *is_strict* is set to TRUE, an error is returned if there is an out of range value. When *is_strict* is set to FALSE, which is the default, then overflow values are accepted.

Return type

TO_DATE returns a DATE, depending on the *format* value.

If the conversion to *format* fails, then an error is returned.

Examples

The following SQL statement converts the date `02 Oct 2001` into a date data type.

```
select to_date('02 Oct 2001', 'DD Mon YYYY');

to_date
------------
2001-10-02
(1 row)
```

The following SQL statement converts the string 20010631 to a date.

```
select to_date('20010631', 'YYYYMMDD', FALSE);
```

The result is July 1, 2001, because there are only 30 days in June.

```
to_date
------------
2001-07-01
```

The following SQL statement converts the string 20010631 to a date:

```
to_date('20010631', 'YYYYMMDD', TRUE);
```

The result is an error because there are only 30 days in June.

```
ERROR:  date/time field date value out of range: 2001-6-31
```

TO_NUMBER

TO_NUMBER converts a string to a numeric (decimal) value.

Syntax

```
to_number(string, format)
```

Arguments

string

String to be converted. The format must be a literal value.

format

The second argument is a format string that indicates how the character string should be parsed to create the numeric value. For example, the format `'99D999'` specifies that the string to be converted consists of five digits with the decimal point in the third position. For example, `to_number('12.345','99D999')` returns `12.345` as a numeric value. For a list of valid formats, see Numeric format strings.

Return type

TO_NUMBER returns a DECIMAL number.

If the conversion to *format* fails, then an error is returned.

Examples

The following example converts the string `12,454.8-` to a number:

```
select to_number('12,454.8-', '99G999D9S');

to_number
-----------
-12454.8
```

The following example converts the string `$ 12,454.88` to a number:

```
select to_number('$ 12,454.88', 'L 99G999D99');

to_number
-----------
12454.88
```

The following example converts the string `$ 2,012,454.88` to a number:

```
select to_number('$ 2,012,454.88', 'L 9,999,999.99');

to_number
-----------
2012454.88
```

Datetime format strings

The following datetime format strings apply to functions such as TO_CHAR. These strings can contain datetime separators (such as '`-`', '`/`', or '`:`') and the following "dateparts" and "timeparts".

For examples of formatting dates as strings, see TO_CHAR.

Datepart or timepart	Meaning
BC or B.C., AD or A.D., b.c. or bc, ad or a.d.	Upper and lowercase era indicators
CC	Two-digit century number
YYYY, YYY, YY, Y	4-digit, 3-digit, 2-digit, 1-digit year number
Y,YYY	4-digit year number with comma
IYYY, IYY, IY, I	4-digit, 3-digit, 2-digit, 1-digit International Organization for Standardization (ISO) year number
Q	Quarter number (1 to 4)
MONTH, Month, month	Month name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)
MON, Mon, mon	Abbreviated month name (uppercase, mixed-case, lowercase, blank-padded to 3 characters)
MM	Month number (01-12)
RM, rm	Month number in Roman numerals (I–XII, with I being January, uppercase or lowercase)
W	Week of month (1–5; the first week starts on the first day of the month.)
WW	Week number of year (1–53; the first week starts on the first day of the year.)
IW	ISO week number of year (the first Thursday of the new year is in week 1.)
DAY, Day, day	Day name (uppercase, mixed-case, lowercase, blank-padded to 9 characters)

Datepart or timepart	Meaning
DY, Dy, dy	Abbreviated day name (uppercase, mixed-case, lowercase, blank-padded to 3 characters)
DDD	Day of year (001–366)
IDDD	Day of ISO 8601 week-numbering year (001-371; day 1 of the year is Monday of the first ISO week)
DD	Day of month as a number (01–31)
D	Day of week (1–7; Sunday is 1) **Note** The D datepart behaves differently from the day of week (DOW) datepart used for the datetime functions DATE_PART and EXTRACT. DOW is based on integers 0–6, where Sunday is 0. For more information, see Date parts for date or timestamp functions.
ID	ISO 8601 day of the week, Monday (1) to Sunday (7)
J	Julian day (days since January 1, 4712 BC)
HH24	Hour (24-hour clock, 00–23)
HH or HH12	Hour (12-hour clock, 01–12)
MI	Minutes (00–59)
SS	Seconds (00–59)
MS	Milliseconds (.000)

Datepart or timepart	Meaning
US	Microseconds (.000000)
AM or PM, A.M. or P.M., a.m. or p.m., am or pm	Upper and lowercase meridian indicators (for 12-hour clock)
TZ, tz	Upper and lowercase time zone abbreviation; valid for TIMESTAMPTZ only
OF	Offset from UTC; valid for TIMESTAMPTZ only

Note

You must surround datetime separators (such as '-', '/' or ':') with single quotation marks, but you must surround the "dateparts" and "timeparts" listed in the preceding table with double quotation marks.

Numeric format strings

The following numeric format strings apply to functions such as TO_NUMBER and TO_CHAR.

- For examples of formatting strings as numbers, see TO_NUMBER.
- For examples of formatting numbers as strings, see TO_CHAR.

Format	Description
9	Numeric value with the specified number of digits.
0	Numeric value with leading zeros.
. (period), D	Decimal point.
, (comma)	Thousands separator.

Format	Description
CC	Century code. For example, the 21st century started on 2001-01-01 (supported for TO_CHAR only).
FM	Fill mode. Suppress padding blanks and zeroes.
PR	Negative value in angle brackets.
S	Sign anchored to a number.
L	Currency symbol in the specified position.
G	Group separator.
MI	Minus sign in the specified position for numbers that are less than 0.
PL	Plus sign in the specified position for numbers that are greater than 0.
SG	Plus or minus sign in the specified position.
RN	Roman numeral between 1 and 3999 (supported for TO_CHAR only).
TH or th	Ordinal number suffix. Does not convert fractional numbers or values that are less than zero.

Teradata-style formatting characters for numeric data

This topic shows you how the TEXT_TO_INT_ALT and TEXT_TO_NUMERIC_ALT functions interpret the characters in the input *expression* string. In the following table, you can also find a list of the characters that you can specify in the *format* phrase. In addition, you can find a description of the differences between Teradata-style formatting and AWS Clean Rooms for the *format* option.

Format	Description
G	Not supported as a group separator in the input *expression* string. You can't specify this character in the *format* phrase.
D	Radix symbol. You can specify this character in the *format* phrase. This character is equivalent to the . (period). The radix symbol can't appear in a *format* phrase that contains any of the following characters: • . (period) • S (uppercase 's') • V (uppercase 'v')
/ , : %	Insertion characters / (forward slash), comma (,), : (colon), and % (percent sign). You can't include these characters in the *format* phrase. AWS Clean Rooms ignores these characters in the input *expression* string.
.	Period as a radix character, that is, a decimal point. This character can't appear in a *format* phrase that contains any of the following characters: • D (uppercase 'd') • S (uppercase 's') • V (uppercase 'v')
B	You can't include the blank space character (B) in the *format* phrase. In the input *expression*

Format	Description
	string, leading and trailing spaces are ignored and spaces between digits aren't allowed.
+ -	You can't include the plus sign (+) or minus sign (-) in the *format* phrase. However, the plus sign (+) and minus sign (-) are parsed implicitly as part of numeric value if they appear in the input *expression* string.
V	Decimal point position indicator. This character can't appear in a *format* phrase that contains any of the following characters: • D (uppercase 'd') • . (period)
Z	Zero-suppressed decimal digit. AWS Clean Rooms trims leading zeros. The Z character can't follow a 9 character. The Z character must be to the left of the radix character if the fraction part contains the 9 character.
9	Decimal digit.
CHAR(*n*)	For this format, you can specify the following: • CHAR consists of Z or 9 characters. AWS Clean Rooms doesn't support a + (plus) or - (minus) in the CHAR value. • *n* is an integer constant, I, or F. For I, this is the number of characters necessary to display the integer portion of numeric or integer data. For F, this is the number of characters necessary to display the fractiona l portion of numeric data.

Format	Description
-	Hyphen (-) character. You can't include this character in the *format* phrase. AWS Clean Rooms ignores this character in the input *expression* string.
S	Signed zoned decimal. The S character must follow the last decimal digit in the *format* phrase. The last character of the input *expression* string and the correspon ding numeric conversion are listed in Data formatting characters for signed zoned decimal, Teradata–style numeric data formatting . The S character can't appear in a *format* phrase that contains any of the following characters: • + (plus sign) • . (period) • D (uppercase 'd') • Z (uppercase 'z') • F (uppercase 'f') • E (uppercase 'e')
E	Exponential notation. The input *expression* string can include the exponent character. You can't specify E as an exponent character in *format* phrase.
FN9	Not supported in AWS Clean Rooms.
FNE	Not supported in AWS Clean Rooms.

Format	Description
$, USD, US Dollars	Dollar sign ($), ISO currency symbol (USD), and the currency name US Dollars. The ISO currency symbol USD and the currency name US Dollars are case-sensitive. AWS Clean Rooms supports only the USD currency. The input *expression* string can include spaces between the USD currency symbol and the numeric value, for example '$ 123E2' or '123E2 $'.
L	Currency symbol. This currency symbol character can only appear once in the *format* phrase. You can't specify repeated currency symbol characters.
C	ISO currency symbol. This currency symbol character can only appear once in the *format* phrase. You can't specify repeated currency symbol characters.
N	Full currency name. This currency symbol character can only appear once in the *format* phrase. You can't specify repeated currency symbol characters.
O	Dual currency symbol. You can't specify this character in the *format* phrase.
U	Dual ISO currency symbol. You can't specify this character in the *format* phrase.
A	Full dual currency name. You can't specify this character in the *format* phrase.

Data formatting characters for signed zoned decimal, Teradata–style numeric data formatting

You can use the following characters in the *format* phrase of the TEXT_TO_INT_ALT and TEXT_TO_NUMERIC_ALT functions for a signed zoned decimal value.

Last character of the input string	Numeric conversion
{ or 0	*n* … 0
A or 1	*n* … 1
B or 2	*n* … 2
C or 3	*n* … 3
D or 4	*n* … 4
E or 5	*n* … 5
F or 6	*n* … 6
G or 7	*n* … 7
H or 8	*n* … 8
I or 9	*n* … 9
}	-*n* … 0
J	-*n* … 1
K	-*n* … 2
L	-*n* … 3
M	-*n* … 4
N	-*n* … 5
O	-*n* … 6

Last character of the input string	Numeric conversion
P	*-n* … 7
Q	*-n* … 8
R	*-n* … 9

Date and time functions

AWS Clean Rooms supports the following date and time functions:

Topics

Summary of date and time functions

The following table provides a summary of date and time functions that are used in AWS Clean Rooms.

Function	Syntax	Returns
+ (Concatenation) operator Concatenates a date to a time on either side of the + symbol and returns a TIMESTAMP or TIMESTAMPTZ.	*date* + *time*	TIMESTAMP or TIMESTAMPTZ
ADD_MONTHS Adds the specified number of months to a date or timestamp.	ADD_MONTHS ({*date*|*timestamp*}, *integer*)	TIMESTAMP
CURRENT_DATE function Returns a date in the current session time zone (UTC by default) for the start of the current transaction.	CURRENT_DATE	DATE
DATEADD Increments a date or time by a specified interval.	DATEADD (*datepart*, *interval*, {*date*|*time*|*timetz*|*timestamp*})	TIMESTAMP or TIME or TIMETZ
DATEDIFF Returns the difference between two dates or times for a given date part, such as a day or month.	DATEDIFF (*datepart*, {*date*|*time*|*timetz*|*timestamp*}, {*date*|*time*|*timetz*|*timestamp*})	BIGINT
DATE_PART Extracts a date part value from a date or time.	DATE_PART (*datepart*, {*date*|*timestamp*})	DOUBLE
DATE_TRUNC	DATE_TRUNC ('*datepart*', *timestamp*)	TIMESTAMP

Function	Syntax	Returns
Truncates a timestamp based on a date part.		
EXTRACT Extracts a date or time part from a timestamp, timestamptz, time, or timetz.	EXTRACT (*datepart* FROM *source*)	`INTEGER or DOUBLE`
GETDATE function Returns the current date and time in the current session time zone (UTC by default). The parentheses are required.	GETDATE()	`TIMESTAMP`
SYSDATE Returns the date and time in UTC for the start of the current transaction.	SYSDATE	`TIMESTAMP`
TIMEOFDAY Returns the current weekday, date, and time in the current session time zone (UTC by default) as a string value.	TIMEOFDAY()	`VARCHAR`
TO_TIMESTAMP Returns a timestamp with time zone for the specified timestamp and time zone format.	TO_TIMESTAMP ('*timestamp*', '*format*')	`TIMESTAMP TZ`

Note

Leap seconds are not considered in elapsed-time calculations.

Date and time functions in transactions

When you run the following functions within a transaction block (BEGIN … END), the function returns the start date or time of the current transaction, not the start of the current statement.

- SYSDATE
- TIMESTAMP
- CURRENT_DATE

The following functions always return the start date or time of the current statement, even when they are within a transaction block.

- GETDATE
- TIMEOFDAY

+ (Concatenation) operator

Concatenates numeric literals, string literals, and/or datetime and interval literals. They are on either side of the + symbol and return different types based on the inputs on either side of the + symbol.

Syntax

```
numeric + string
```

```
date + time
```

```
date + timetz
```

The order of the arguments can be reversed.

Arguments

numeric literals

Literals or constants that represent numbers can be integer or floating-point.

string literals

Strings, character strings, or character constants

date

A DATE column or an expression that implicitly converts to a DATE.

time

A TIME column or an expression that implicitly converts to a TIME.

timetz

A TIMETZ column or an expression that implicitly converts to a TIMETZ.

Example

The following example table TIME_TEST has a column TIME_VAL (type TIME) with three values inserted.

```
select date '2000-01-02' + time_val as ts from time_test;
```

ADD_MONTHS function

ADD_MONTHS adds the specified number of months to a date or timestamp value or expression. The DATEADD function provides similar functionality.

Syntax

```
ADD_MONTHS( {date | timestamp}, integer)
```

Arguments

date | *timestamp*

A date or timestamp column or an expression that implicitly converts to a date or timestamp. If the date is the last day of the month, or if the resulting month is shorter, the function returns the last day of the month in the result. For other dates, the result contains the same day number as the date expression.

integer

A positive or negative integer. Use a negative number to subtract months from dates.

Return type

TIMESTAMP

Example

The following query uses the ADD_MONTHS function inside a TRUNC function. The TRUNC function removes the time of day from the result of ADD_MONTHS. The ADD_MONTHS function adds 12 months to each value from the CALDATE column.

```
select distinct trunc(add_months(caldate, 12)) as calplus12,
trunc(caldate) as cal
from date
order by 1 asc;

 calplus12  |    cal
------------+------------
 2009-01-01 | 2008-01-01
 2009-01-02 | 2008-01-02
 2009-01-03 | 2008-01-03
...
(365 rows)
```

The following examples demonstrate the behavior when the ADD_MONTHS function operates on dates with months that have different numbers of days.

```
select add_months('2008-03-31',1);

add_months
---------------------
2008-04-30 00:00:00
(1 row)

select add_months('2008-04-30',1);

add_months
---------------------
```

```
2008-05-31 00:00:00
(1 row)
```

CONVERT_TIMEZONE function

CONVERT_TIMEZONE converts a timestamp from one time zone to another. The function automatically adjusts for daylight saving time.

Syntax

```
CONVERT_TIMEZONE ( ['source_timezone',] 'target_timezone', 'timestamp')
```

Arguments

source_timezone

(Optional) The time zone of the current timestamp. The default is UTC.

target_timezone

The time zone for the new timestamp.

timestamp

A timestamp column or an expression that implicitly converts to a timestamp.

Return type

TIMESTAMP

Examples

The following example converts the timestamp value from the default UTC time zone to PST.

```
select convert_timezone('PST', '2008-08-21 07:23:54');

 convert_timezone
-----------------------
2008-08-20 23:23:54
```

The following example converts the timestamp value in the LISTTIME column from the default UTC time zone to PST. Though the timestamp is within the daylight time period, it's converted to standard time because the target time zone is specified as an abbreviation (PST).

```
select listtime, convert_timezone('PST', listtime) from listing
where listid = 16;

     listtime       |   convert_timezone
--------------------+-------------------
2008-08-24 09:36:12     2008-08-24 01:36:12
```

The following example converts a timestamp LISTTIME column from the default UTC time zone to US/Pacific time zone. The target time zone uses a time zone name, and the timestamp is within the daylight time period, so the function returns the daylight time.

```
select listtime, convert_timezone('US/Pacific', listtime) from listing
where listid = 16;

     listtime       |   convert_timezone
--------------------+---------------------
2008-08-24 09:36:12 | 2008-08-24 02:36:12
```

The following example converts a timestamp string from EST to PST:

```
select convert_timezone('EST', 'PST', '20080305 12:25:29');

 convert_timezone
-------------------
2008-03-05 09:25:29
```

The following example converts a timestamp to US Eastern Standard Time because the target time zone uses a time zone name (America/New_York) and the timestamp is within the standard time period.

```
select convert_timezone('America/New_York', '2013-02-01 08:00:00');

 convert_timezone
---------------------
2013-02-01 03:00:00
(1 row)
```

The following example converts the timestamp to US Eastern Daylight Time because the target time zone uses a time zone name (America/New_York) and the timestamp is within the daylight time period.

```
select convert_timezone('America/New_York', '2013-06-01 08:00:00');

 convert_timezone
---------------------
2013-06-01 04:00:00
(1 row)
```

The following example demonstrates the use of offsets.

```
SELECT CONVERT_TIMEZONE('GMT','NEWZONE +2','2014-05-17 12:00:00') as newzone_plus_2,
CONVERT_TIMEZONE('GMT','NEWZONE-2:15','2014-05-17 12:00:00') as newzone_minus_2_15,
CONVERT_TIMEZONE('GMT','America/Los_Angeles+2','2014-05-17 12:00:00') as la_plus_2,
CONVERT_TIMEZONE('GMT','GMT+2','2014-05-17 12:00:00') as gmt_plus_2;

   newzone_plus_2    | newzone_minus_2_15  |      la_plus_2      |     gmt_plus_2
---------------------+---------------------+---------------------+---------------------
2014-05-17 10:00:00 | 2014-05-17 14:15:00 | 2014-05-17 10:00:00 | 2014-05-17 10:00:00
(1 row)
```

CURRENT_DATE function

CURRENT_DATE returns a date in the current session time zone (UTC by default) in the default format: YYYY-MM-DD.

> **Note**
>
> CURRENT_DATE returns the start date for the current transaction, not for the start of the current statement. Consider the scenario where you start a transaction containing multiple statements on 10/01/08 23:59, and the statement containing CURRENT_DATE runs at 10/02/08 00:00. CURRENT_DATE returns 10/01/08, not 10/02/08.

Syntax

```
CURRENT_DATE
```

Return type

DATE

Example

The following example returns the current date (in the AWS Region where the function runs).

```
select current_date;

   date
------------
2008-10-01
```

DATEADD function

Increments a DATE, TIME, TIMETZ, or TIMESTAMP value by a specified interval.

Syntax

```
DATEADD( datepart, interval, {date|time|timetz|timestamp} )
```

Arguments

datepart

The date part (year, month, day, or hour, for example) that the function operates on. For more information, see Date parts for date or timestamp functions.

interval

An integer that specified the interval (number of days, for example) to add to the target expression. A negative integer subtracts the interval.

date|*time*|*timetz*|*timestamp*

A DATE, TIME, TIMETZ, or TIMESTAMP column or an expression that implicitly converts to a DATE, TIME, TIMETZ, or TIMESTAMP. The DATE, TIME, TIMETZ, or TIMESTAMP expression must contain the specified date part.

Return type

TIMESTAMP or TIME or TIMETZ depending on the input data type.

Examples with a DATE column

The following example adds 30 days to each date in November that exists in the DATE table.

```
select dateadd(day,30,caldate) as novplus30
from date
where month='NOV'
order by dateid;

novplus30
---------------------
2008-12-01 00:00:00
2008-12-02 00:00:00
2008-12-03 00:00:00
...
(30 rows)
```

The following example adds 18 months to a literal date value.

```
select dateadd(month,18,'2008-02-28');

date_add
---------------------
2009-08-28 00:00:00
(1 row)
```

The default column name for a DATEADD function is DATE_ADD. The default timestamp for a date value is `00:00:00`.

The following example adds 30 minutes to a date value that doesn't specify a timestamp.

```
select dateadd(m,30,'2008-02-28');

date_add
---------------------
2008-02-28 00:30:00
(1 row)
```

You can name date parts in full or abbreviate them. In this case, *m* stands for minutes, not months.

Examples with a TIME column

The following example table TIME_TEST has a column TIME_VAL (type TIME) with three values inserted.

```
select time_val from time_test;

time_val
--------------------
20:00:00
00:00:00.5550
00:58:00
```

The following example adds 5 minutes to each TIME_VAL in the TIME_TEST table.

```
select dateadd(minute,5,time_val) as minplus5 from time_test;

minplus5
---------------
20:05:00
00:05:00.5550
01:03:00
```

The following example adds 8 hours to a literal time value.

```
select dateadd(hour, 8, time '13:24:55');

date_add
---------------
21:24:55
```

The following example shows when a time goes over 24:00:00 or under 00:00:00.

```
select dateadd(hour, 12, time '13:24:55');

date_add
---------------
01:24:55
```

Examples with a TIMETZ column

The output values in these examples are in UTC which is the default timezone.

The following example table TIMETZ_TEST has a column TIMETZ_VAL (type TIMETZ) with three values inserted.

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

The following example adds 5 minutes to each TIMETZ_VAL in TIMETZ_TEST table.

```
select dateadd(minute,5,timetz_val) as minplus5_tz from timetz_test;

minplus5_tz
---------------
04:05:00+00
00:05:00.5550+00
06:03:00+00
```

The following example adds 2 hours to a literal timetz value.

```
select dateadd(hour, 2, timetz '13:24:55 PST');

date_add
---------------
23:24:55+00
```

Examples with a TIMESTAMP column

The output values in these examples are in UTC which is the default timezone.

The following example table TIMESTAMP_TEST has a column TIMESTAMP_VAL (type TIMESTAMP) with three values inserted.

```
SELECT timestamp_val FROM timestamp_test;
```

```
timestamp_val
------------------
1988-05-15 10:23:31
2021-03-18 17:20:41
2023-06-02 18:11:12
```

The following example adds 20 years only to the TIMESTAMP_VAL values in TIMESTAMP_TEST from before the year 2000.

```
SELECT dateadd(year,20,timestamp_val)
FROM timestamp_test
WHERE timestamp_val < to_timestamp('2000-01-01 00:00:00', 'YYYY-MM-DD HH:MI:SS');

date_add
---------------
2008-05-15 10:23:31
```

The following example adds 5 seconds to a literal timestamp value written without a seconds indicator.

```
SELECT dateadd(second, 5, timestamp '2001-06-06');

date_add
---------------
2001-06-06 00:00:05
```

Usage notes

The DATEADD(month, ...) and ADD_MONTHS functions handle dates that fall at the ends of months differently:

- ADD_MONTHS: If the date you are adding to is the last day of the month, the result is always the last day of the result month, regardless of the length of the month. For example, April 30 + 1 month is May 31.

  ```
  select add_months('2008-04-30',1);

  add_months
  ----------------------
  ```

```
2008-05-31 00:00:00
(1 row)
```

- DATEADD: If there are fewer days in the date you are adding to than in the result month, the result is the corresponding day of the result month, not the last day of that month. For example, April 30 + 1 month is May 30.

```
select dateadd(month,1,'2008-04-30');

date_add
---------------------
2008-05-30 00:00:00
(1 row)
```

The DATEADD function handles the leap year date 02-29 differently when using dateadd(month, 12,...) or dateadd(year, 1, ...).

```
select dateadd(month,12,'2016-02-29');

date_add
---------------------
2017-02-28 00:00:00

select dateadd(year, 1, '2016-02-29');

date_add
---------------------
2017-03-01 00:00:00
```

DATEDIFF function

DATEDIFF returns the difference between the date parts of two date or time expressions.

Syntax

```
DATEDIFF ( datepart, {date|time|timetz|timestamp}, {date|time|timetz|timestamp} )
```

Arguments

datepart

The specific part of the date or time value (year, month, or day, hour, minute, second, millisecond, or microsecond) that the function operates on. For more information, see Date parts for date or timestamp functions.

Specifically, DATEDIFF determines the number of date part boundaries that are crossed between two expressions. For example, suppose that you're calculating the difference in years between two dates, `12-31-2008` and `01-01-2009`. In this case, the function returns 1 year despite the fact that these dates are only one day apart. If you are finding the difference in hours between two timestamps, `01-01-2009 8:30:00` and `01-01-2009 10:00:00`, the result is 2 hours. If you are finding the difference in hours between two timestamps, `8:30:00` and `10:00:00`, the result is 2 hours.

date|*time*|*timetz*|*timestamp*

A DATE, TIME, TIMETZ, or TIMESTAMP column or expressions that implicitly convert to a DATE, TIME, TIMETZ, or TIMESTAMP. The expressions must both contain the specified date or time part. If the second date or time is later than the first date or time, the result is positive. If the second date or time is earlier than the first date or time, the result is negative.

Return type

BIGINT

Examples with a DATE column

The following example finds the difference, in number of weeks, between two literal date values.

```
select datediff(week,'2009-01-01','2009-12-31') as numweeks;

numweeks
----------
52
(1 row)
```

The following example finds the difference, in hours, between two literal date values. When you don't provide the time value for a date, it defaults to 00:00:00.

```
select datediff(hour, '2023-01-01', '2023-01-03 05:04:03');

date_diff
----------
53
(1 row)
```

The following example finds the difference, in days, between two literal TIMESTAMETZ values.

```
Select datediff(days, 'Jun 1,2008  09:59:59 EST', 'Jul 4,2008  09:59:59 EST')

date_diff
----------
33
```

The following example finds the difference, in days, between two dates in the same row of a table.

```
select * from date_table;

start_date |   end_date
-----------+-----------
2009-01-01 | 2009-03-23
2023-01-04 | 2024-05-04
(2 rows)

select datediff(day, start_date, end_date) as duration from date_table;

duration
---------
      81
     486
(2 rows)
```

The following example finds the difference, in number of quarters, between a literal value in the past and today's date. This example assumes that the current date is June 5, 2008. You can name date parts in full or abbreviate them. The default column name for the DATEDIFF function is DATE_DIFF.

```
select datediff(qtr, '1998-07-01', current_date);

date_diff
```

```
-----------
40
(1 row)
```

The following example joins the SALES and LISTING tables to calculate how many days after they were listed any tickets were sold for listings 1000 through 1005. The longest wait for sales of these listings was 15 days, and the shortest was less than one day (0 days).

```
select priceperticket,
datediff(day, listtime, saletime) as wait
from sales, listing where sales.listid = listing.listid
and sales.listid between 1000 and 1005
order by wait desc, priceperticket desc;

priceperticket | wait
---------------+------
 96.00         |   15
 123.00        |   11
 131.00        |    9
 123.00        |    6
 129.00        |    4
 96.00         |    4
 96.00         |    0
(7 rows)
```

This example calculates the average number of hours sellers waited for all ticket sales.

```
select avg(datediff(hours, listtime, saletime)) as avgwait
from sales, listing
where sales.listid = listing.listid;

avgwait
---------
465
(1 row)
```

Examples with a TIME column

The following example table TIME_TEST has a column TIME_VAL (type TIME) with three values inserted.

```
select time_val from time_test;
```

```
time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

The following example finds the difference in number of hours between the TIME_VAL column and a time literal.

```
select datediff(hour, time_val, time '15:24:45') from time_test;

 date_diff
-----------
        -5
        15
        15
```

The following example finds the difference in number of minutes between two literal time values.

```
select datediff(minute, time '20:00:00', time '21:00:00') as nummins;

nummins
----------
60
```

Examples with a TIMETZ column

The following example table TIMETZ_TEST has a column TIMETZ_VAL (type TIMETZ) with three values inserted.

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

The following example finds the differences in number of hours, between a TIMETZ literal and timetz_val.

```
select datediff(hours, timetz '20:00:00 PST', timetz_val) as numhours from timetz_test;

numhours
----------
0
-4
1
```

The following example finds the difference in number of hours, between two literal TIMETZ values.

```
select datediff(hours, timetz '20:00:00 PST', timetz '00:58:00 EST') as numhours;

numhours
----------
1
```

DATE_PART function

DATE_PART extracts date part values from an expression. DATE_PART is a synonym of the PGDATE_PART function.

Syntax

```
DATE_PART(datepart, {date|timestamp})
```

Arguments

datepart

An identifier literal or string of the specific part of the date value (for example, year, month, or day) that the function operates on. For more information, see Date parts for date or timestamp functions.

{*date*|*timestamp*}

A date column, timestamp column, or an expression that implicitly converts to a date or timestamp. The column or expression in *date* or *timestamp* must contain the date part specified in *datepart*.

Return type

DOUBLE

Examples

The default column name for the DATE_PART function is `pgdate_part`.

The following example finds the minute from a timestamp literal.

```
SELECT DATE_PART(minute, timestamp '20230104 04:05:06.789');

pgdate_part
-----------
          5
```

The following example finds the week number from a timestamp literal. The week number calculation follows the ISO 8601 standard. For more information, see ISO 8601 in Wikipedia.

```
SELECT DATE_PART(week, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
         18
```

The following example finds the day of the month from a timestamp literal.

```
SELECT DATE_PART(day, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
          2
```

The following example finds the day of the week from a timestamp literal. The week number calculation follows the ISO 8601 standard. For more information, see ISO 8601 in Wikipedia.

```
SELECT DATE_PART(dayofweek, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
          1
```

The following example finds the century from a timestamp literal. The century calculation follows the ISO 8601 standard. For more information, see ISO 8601 in Wikipedia.

```
SELECT DATE_PART(century, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
         21
```

The following example finds the millennium from a timestamp literal. The millennium calculation follows the ISO 8601 standard. For more information, see ISO 8601 in Wikipedia.

```
SELECT DATE_PART(millennium, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
         3
```

The following example finds the microseconds from a timestamp literal. The microseconds calculation follows the ISO 8601 standard. For more information, see ISO 8601 in Wikipedia.

```
SELECT DATE_PART(microsecond, timestamp '20220502 04:05:06.789');

pgdate_part
-----------
    789000
```

The following example finds the month from a date literal.

```
SELECT DATE_PART(month, date '20220502');

pgdate_part
-----------
         5
```

The following example applies the DATE_PART function to a column in a table.

```
SELECT date_part(w, listtime) AS weeks, listtime
FROM listing
WHERE listid=10
```

```
weeks |      listtime
------+---------------------
 25   | 2008-06-17 09:44:54
(1 row)
```

You can name date parts in full or abbreviate them; in this case, *w* stands for weeks.

The day of week date part returns an integer from 0-6, starting with Sunday. Use DATE_PART with dow (DAYOFWEEK) to view events on a Saturday.

```
SELECT date_part(dow, starttime) AS dow, starttime
FROM event
WHERE date_part(dow, starttime)=6
ORDER BY 2,1;

 dow |      starttime
-----+---------------------
   6 | 2008-01-05 14:00:00
   6 | 2008-01-05 14:00:00
   6 | 2008-01-05 14:00:00
   6 | 2008-01-05 14:00:00
...
(1147 rows)
```

DATE_TRUNC function

The DATE_TRUNC function truncates a timestamp expression or literal based on the date part that you specify, such as hour, day, or month.

Syntax

```
DATE_TRUNC('datepart', timestamp)
```

Arguments

datepart

The date part to which to truncate the timestamp value. The input *timestamp* is truncated to the precision of the input *datepart*. For example, `month` truncates to the first day of the month. Valid formats are as follows:

- microsecond, microseconds
- millisecond, milliseconds
- second, seconds
- minute, minutes
- hour, hours
- day, days
- week, weeks
- month, months
- quarter, quarters
- year, years
- decade, decades
- century, centuries
- millennium, millennia

For more information about abbreviations of some formats, see Date parts for date or timestamp functions

timestamp

A timestamp column or an expression that implicitly converts to a timestamp.

Return type

TIMESTAMP

Examples

Truncate the input timestamp to the second.

```
SELECT DATE_TRUNC('second', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-04-30 04:05:06
```

Truncate the input timestamp to the minute.

```
SELECT DATE_TRUNC('minute', TIMESTAMP '20200430 04:05:06.789');
date_trunc
```

```
2020-04-30 04:05:00
```

Truncate the input timestamp to the hour.

```
SELECT DATE_TRUNC('hour', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-04-30 04:00:00
```

Truncate the input timestamp to the day.

```
SELECT DATE_TRUNC('day', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-04-30 00:00:00
```

Truncate the input timestamp to the first day of a month.

```
SELECT DATE_TRUNC('month', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-04-01 00:00:00
```

Truncate the input timestamp to the first day of a quarter.

```
SELECT DATE_TRUNC('quarter', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-04-01 00:00:00
```

Truncate the input timestamp to the first day of a year.

```
SELECT DATE_TRUNC('year', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2020-01-01 00:00:00
```

Truncate the input timestamp to the first day of a century.

```
SELECT DATE_TRUNC('millennium', TIMESTAMP '20200430 04:05:06.789');
date_trunc
2001-01-01 00:00:00
```

Truncate the input timestamp to the Monday of a week.

```
select date_trunc('week', TIMESTAMP '20220430 04:05:06.789');
date_trunc
2022-04-25 00:00:00
```

In the following example, the DATE_TRUNC function uses the 'week' date part to return the date for the Monday of each week.

```
select date_trunc('week', saletime), sum(pricepaid) from sales where
saletime like '2008-09%' group by date_trunc('week', saletime) order by 1;

date_trunc  |    sum
------------+-------------
2008-09-01  | 2474899
2008-09-08  | 2412354
2008-09-15  | 2364707
2008-09-22  | 2359351
2008-09-29  |  705249
```

EXTRACT function

The EXTRACT function returns a date or time part from a TIMESTAMP, TIMESTAMPTZ, TIME, or TIMETZ value. Examples include a day, month, year, hour, minute, second, millisecond, or microsecond from a timestamp.

Syntax

```
EXTRACT(datepart FROM source)
```

Arguments

datepart

The subfield of a date or time to extract, such as a day, month, year, hour, minute, second, millisecond, or microsecond. For possible values, see Date parts for date or timestamp functions.

source

A column or expression that evaluates to a data type of TIMESTAMP, TIMESTAMPTZ, TIME, or TIMETZ.

Return type

INTEGER if the *source* value evaluates to data type TIMESTAMP, TIME, or TIMETZ.

DOUBLE PRECISION if the *source* value evaluates to data type TIMESTAMPTZ.

Examples with TIMESTAMP

The following example determines the week numbers for sales in which the price paid was $10,000 or more.

```
select salesid, extract(week from saletime) as weeknum
from sales
where pricepaid > 9999
order by 2;

salesid | weeknum
--------+---------
 159073 |       6
 160318 |       8
 161723 |      26
```

The following example returns the minute value from a literal timestamp value.

```
select extract(minute from timestamp '2009-09-09 12:08:43');

date_part
--
```

The following example returns the millisecond value from a literal timestamp value.

```
select extract(ms from timestamp '2009-09-09 12:08:43.101');

date_part
-----------
101
```

Examples with TIMESTAMPTZ

The following example returns the year value from a literal timestamptz value.

```
select extract(year from timestamptz '1.12.1997 07:37:16.00 PST');
```

```
date_part
-----------
1997
```

Examples with TIME

The following example table TIME_TEST has a column TIME_VAL (type TIME) with three values inserted.

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

The following example extracts the minutes from each time_val.

```
select extract(minute from time_val) as minutes from time_test;

minutes
-----------
         0
         0
         58
```

The following example extracts the hours from each time_val.

```
select extract(hour from time_val) as hours from time_test;

hours
-----------
         20
         0
         0
```

The following example extracts milliseconds from a literal value.

```
select extract(ms from time '18:25:33.123456');
```

```
 date_part
-----------
     123
```

Examples with TIMETZ

The following example table TIMETZ_TEST has a column TIMETZ_VAL (type TIMETZ) with three values inserted.

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

The following example extracts the hours from each timetz_val.

```
select extract(hour from timetz_val) as hours from time_test;

hours
-----------
         4
         0
         5
```

The following example extracts milliseconds from a literal value. Literals aren't converted to UTC before the extraction is processed.

```
select extract(ms from timetz '18:25:33.123456 EST');

 date_part
-----------
     123
```

The following example returns the timezone offset hour from UTC from a literal timetz value.

```
select extract(timezone_hour from timetz '1.12.1997 07:37:16.00 PDT');

date_part
-----------
```

```
-7
```

GETDATE function

The GETDATE function returns the current date and time in the current session time zone (UTC by default).

It returns the start date or time of the current statement, even when it is within a transaction block.

Syntax

```
GETDATE()
```

The parentheses are required.

Return type

TIMESTAMP

Example

The following example uses the GETDATE function to return the full timestamp for the current date.

```
select getdate();
```

SYSDATE function

SYSDATE returns the current date and time in the current session time zone (UTC by default).

> **Note**
>
> SYSDATE returns the start date and time for the current transaction, not for the start of the current statement.

Syntax

```
SYSDATE
```

This function requires no arguments.

Return type

TIMESTAMP

Examples

The following example uses the SYSDATE function to return the full timestamp for the current date.

```
select sysdate;

timestamp
----------------------------
2008-12-04 16:10:43.976353
(1 row)
```

The following example uses the SYSDATE function inside the TRUNC function to return the current date without the time.

```
select trunc(sysdate);

trunc
------------
2008-12-04
(1 row)
```

The following query returns sales information for dates that fall between the date when the query is issued and whatever date is 120 days earlier.

```
select salesid, pricepaid, trunc(saletime) as saletime, trunc(sysdate) as now
from sales
where saletime between trunc(sysdate)-120 and trunc(sysdate)
order by saletime asc;

salesid | pricepaid |  saletime  |    now
---------+-----------+------------+------------
91535 |    670.00 | 2008-08-07 | 2008-12-05
91635 |    365.00 | 2008-08-07 | 2008-12-05
91901 |   1002.00 | 2008-08-07 | 2008-12-05
...
```

TIMEOFDAY function

TIMEOFDAY is a special alias used to return the weekday, date, and time as a string value. It returns the time of day string for the current statement, even when it is within a transaction block.

Syntax

```
TIMEOFDAY()
```

Return type

VARCHAR

Examples

The following example returns the current date and time by using the TIMEOFDAY function.

```
select timeofday();
timeofday
------------
Thu Sep 19 22:53:50.333525 2013 UTC
(1 row)
```

TO_TIMESTAMP function

TO_TIMESTAMP converts a TIMESTAMP string to TIMESTAMPTZ.

Syntax

```
to_timestamp (timestamp, format)
```

```
to_timestamp (timestamp, format, is_strict)
```

Arguments

timestamp

A string that represents a timestamp value in the format specified by *format*. If this argument is left as empty, the timestamp value defaults to `0001-01-01 00:00:00`.

format

A string literal that defines the format of the *timestamp* value. Formats that include a time zone (**TZ**, **tz**, or **OF**) are not supported as input. For valid timestamp formats, see Datetime format strings.

is_strict

An optional Boolean value that specifies whether an error is returned if an input timestamp value is out of range. When *is_strict* is set to TRUE, an error is returned if there is an out of range value. When *is_strict* is set to FALSE, which is the default, then overflow values are accepted.

Return type

TIMESTAMPTZ

Examples

The following example demonstrates using the TO_TIMESTAMP function to convert a TIMESTAMP string to a TIMESTAMPTZ.

```
select sysdate, to_timestamp(sysdate, 'YYYY-MM-DD HH24:MI:SS') as second;

timestamp                  | second
---------------------------   ----------------------
2021-04-05 19:27:53.281812 | 2021-04-05 19:27:53+00
```

It's possible to pass TO_TIMESTAMP part of a date. The remaining date parts are set to default values. The time is included in the output:

```
SELECT TO_TIMESTAMP('2017','YYYY');

to_timestamp
---------------------------
2017-01-01 00:00:00+00
```

The following SQL statement converts the string '2011-12-18 24:38:15' to a TIMESTAMPTZ. The result is a TIMESTAMPTZ that falls on the next day because the number of hours is more than 24 hours:

```
SELECT TO_TIMESTAMP('2011-12-18 24:38:15', 'YYYY-MM-DD HH24:MI:SS');

to_timestamp
---------------------
2011-12-19 00:38:15+00
```

The following SQL statement converts the string '2011-12-18 24:38:15' to a TIMESTAMPTZ. The result is an error because the time value in the timestamp is more than 24 hours:

```
SELECT TO_TIMESTAMP('2011-12-18 24:38:15', 'YYYY-MM-DD HH24:MI:SS', TRUE);

ERROR:  date/time field time value out of range: 24:38:15.0
```

Date parts for date or timestamp functions

The following table identifies the date part and time part names and abbreviations that are accepted as arguments to the following functions:

- DATEADD
- DATEDIFF
- DATE_PART
- EXTRACT

Date part or time part	Abbreviations
millennium, millennia	mil, mils
century, centuries	c, cent, cents
decade, decades	dec, decs
epoch	epoch (supported by the EXTRACT)
year, years	y, yr, yrs
quarter, quarters	qtr, qtrs
month, months	mon, mons

Date part or time part	Abbreviations
week, weeks	w
day of week	dayofweek, dow, dw, weekday (supported by the DATE_PART and the EXTRACT function) Returns an integer from 0–6, starting with Sunday. **Note** The DOW date part behaves differently from the day of week (D) date part used for datetime format strings. D is based on integers 1–7, where Sunday is 1. For more information, see Datetime format strings.
day of year	dayofyear, doy, dy, yearday (supported by the EXTRACT)
day, days	d
hour, hours	h, hr, hrs
minute, minutes	m, min, mins
second, seconds	s, sec, secs
millisecond, milliseconds	ms, msec, msecs, msecond, mseconds, millisec, millisecs, millisecon
microsecond, microseconds	microsec, microsecs, microsecond, usecond, useconds, us, usec, usecs
timezone, timezone_hour, timezone_minute	Supported by the EXTRACT for timestamp with time zone (TIMESTAMPTZ) only.

Variations in results with seconds, milliseconds, and microseconds

Minor differences in query results occur when different date functions specify seconds, milliseconds, or microseconds as date parts:

- The EXTRACT function return integers for the specified date part only, ignoring higher- and lower-level date parts. If the specified date part is seconds, milliseconds and microseconds are not included in the result. If the specified date part is milliseconds, seconds and microseconds are not included. If the specified date part is microseconds, seconds and milliseconds are not included.
- The DATE_PART function returns the complete seconds portion of the timestamp, regardless of the specified date part, returning either a decimal value or an integer as required.

CENTURY, EPOCH, DECADE, and MIL notes

CENTURY or CENTURIES

AWS Clean Rooms interprets a CENTURY to start with year *###1* and end with year ###0:

```
select extract (century from timestamp '2000-12-16 12:21:13');
date_part
-----------
20
(1 row)

select extract (century from timestamp '2001-12-16 12:21:13');
date_part
-----------
21
(1 row)
```

EPOCH

The AWS Clean Rooms implementation of EPOCH is relative to 1970-01-01 00:00:00.000000 independent of the time zone where the cluster resides. You might need to offset the results by the difference in hours depending on the time zone where the cluster is located.

DECADE or DECADES

AWS Clean Rooms interprets the DECADE or DECADES DATEPART based on the common calendar. For example, because the common calendar starts from the year 1, the first decade (decade 1) is 0001-01-01 through 0009-12-31, and the second decade (decade 2) is 0010-01-01 through 0019-12-31. For example, decade 201 spans from 2000-01-01 to 2009-12-31:

```
select extract(decade from timestamp '1999-02-16 20:38:40');
```

```
date_part
-----------
200
(1 row)

select extract(decade from timestamp '2000-02-16 20:38:40');
date_part
-----------
201
(1 row)

select extract(decade from timestamp '2010-02-16 20:38:40');
date_part
-----------
202
(1 row)
```

MIL or MILS

AWS Clean Rooms interprets a MIL to start with the first day of year *#001* and end with the last day of year #000:

```
select extract (mil from timestamp '2000-12-16 12:21:13');
date_part
-----------
2
(1 row)

select extract (mil from timestamp '2001-12-16 12:21:13');
date_part
-----------
3
(1 row)
```

Hash functions

A hash function is a mathematical function that converts a numerical input value into another value. AWS Clean Rooms supports the following hash functions:

Topics

- SHA function
- SHA1 function
- SHA2 function
- MURMUR3_32_HASH

MD5 function

Uses the MD5 cryptographic hash function to convert a variable-length string into a 32-character string that is a text representation of the hexadecimal value of a 128-bit checksum.

Syntax

```
MD5(string)
```

Arguments

string

A variable-length string.

Return type

The MD5 function returns a 32-character string that is a text representation of the hexadecimal value of a 128-bit checksum.

Examples

The following example shows the 128-bit value for the string 'AWS Clean Rooms':

```
select md5('AWS Clean Rooms');
md5
----------------------------------
f7415e33f972c03abd4f3fed36748f7a
(1 row)
```

SHA function

Synonym of SHA1 function.

See SHA1 function.

SHA1 function

The SHA1 function uses the SHA1 cryptographic hash function to convert a variable-length string into a 40-character string that is a text representation of the hexadecimal value of a 160-bit checksum.

Syntax

SHA1 is a synonym of SHA function.

```
SHA1(string)
```

Arguments

string

A variable-length string.

Return type

The SHA1 function returns a 40-character string that is a text representation of the hexadecimal value of a 160-bit checksum.

Example

The following example returns the 160-bit value for the word 'AWS Clean Rooms':

```
select sha1('AWS Clean Rooms');
```

SHA2 function

The SHA2 function uses the SHA2 cryptographic hash function to convert a variable-length string into a character string. The character string is a text representation of the hexadecimal value of the checksum with the specified number of bits.

Syntax

```
SHA2(string, bits)
```

Arguments

string

A variable-length string.

integer

The number of bits in the hash functions. Valid values are 0 (same as 256), 224, 256, 384, and 512.

Return type

The SHA2 function returns a character string that is a text representation of the hexadecimal value of the checksum or an empty string if the number of bits is invalid.

Example

The following example returns the 256-bit value for the word 'AWS Clean Rooms':

```
select sha2('AWS Clean Rooms', 256);
```

MURMUR3_32_HASH

The MURMUR3_32_HASH function computes the 32-bit Murmur3A non-cryptographic hash for all common data types including numeric and string types.

Syntax

```
MURMUR3_32_HASH(value [, seed])
```

Arguments

value

The input value to hash. AWS Clean Rooms hashes the binary representation of the input value. This behavior is similar to FNV_HASH, but the value is converted to the binary representation specified by the Apache Iceberg 32-bit Murmur3 hash specification.

seed

The INT seed of the hash function. This argument is optional. If not given, AWS Clean Rooms uses the default seed of 0. This enables combining the hash of multiple columns without any conversions or concatenations.

Return type

The function returns an INT.

Example

The following examples return the Murmur3 hash of a number, the string 'AWS Clean Rooms', and the concatenation of the two.

```
select MURMUR3_32_HASH(1);

    MURMUR3_32_HASH
----------------------
 -5968735742475085980
(1 row)
```

```
select MURMUR3_32_HASH('AWS Clean Rooms');

    MURMUR3_32_HASH
----------------------
 7783490368944507294
(1 row)
```

```
select MURMUR3_32_HASH('AWS Clean Rooms', MURMUR3_32_HASH(1));

    MURMUR3_32_HASH
----------------------
 -2202602717770968555
(1 row)
```

Usage notes

To compute the hash of a table with multiple columns, you can compute the Murmur3 hash of the first column and pass it as a seed to the hash of the second column. Then, it passes the Murmur3 hash of the second column as a seed to the hash of the third column.

The following example creates seeds to hash a table with multiple columns.

```
select MURMUR3_32_HASH(column_3, MURMUR3_32_HASH(column_2, MURMUR3_32_HASH(column_1)))
 from sample_table;
```

The same property can be used to compute the hash of a concatenation of strings.

```
select MURMUR3_32_HASH('abcd');

   MURMUR3_32_HASH
----------------------
 -281581062704388899
(1 row)
```

```
select MURMUR3_32_HASH('cd', MURMUR3_32_HASH('ab'));

   MURMUR3_32_HASH
----------------------
 -281581062704388899
(1 row)
```

The hash function uses the type of the input to determine the number of bytes to hash. Use casting to enforce a specific type, if necessary.

The following examples use different input types to produce different results.

```
select MURMUR3_32_HASH(1::smallint);

   MURMUR3_32_HASH
--------------------
 589727492704079044
(1 row)
```

```
select MURMUR3_32_HASH(1);
```

```
   MURMUR3_32_HASH
----------------------
 -5968735742475085980
(1 row)
```

```
select MURMUR3_32_HASH(1::bigint);

   MURMUR3_32_HASH
----------------------
 -8517097267634966620
(1 row)
```

JSON functions

When you need to store a relatively small set of key-value pairs, you might save space by storing the data in JSON format. Because JSON strings can be stored in a single column, using JSON might be more efficient than storing your data in tabular format.

Example

For example, suppose you have a sparse table, where you need to have many columns to fully represent all possible attributes. However, most of the column values are NULL for any given row or any given column. By using JSON for storage, you might be able to store the data for a row in key-value pairs in a single JSON string and eliminate the sparsely-populated table columns.

In addition, you can easily modify JSON strings to store additional key:value pairs without needing to add columns to a table.

We recommend using JSON sparingly. JSON isn't a good choice for storing larger datasets because, by storing disparate data in a single column, JSON doesn't use the AWS Clean Rooms column store architecture.

JSON uses UTF-8 encoded text strings, so JSON strings can be stored as CHAR or VARCHAR data types. Use VARCHAR if the strings include multi-byte characters.

JSON strings must be properly formatted JSON, according to the following rules:

- The root level JSON can either be a JSON object or a JSON array. A JSON object is an unordered set of comma-separated key:value pairs enclosed by curly braces.

For example, `{"one":1, "two":2}`

- A JSON array is an ordered set of comma-separated values enclosed by brackets.

  An example is the following: `["first", {"one":1}, "second", 3, null]`
- JSON arrays use a zero-based index; the first element in an array is at position 0. In a JSON key:value pair, the key is a string in double quotation marks.
- A JSON value can be any of the following:
  - JSON object
  - JSON array
  - String in double quotation marks
  - Number (integer and float)
  - Boolean
  - Null
- Empty objects and empty arrays are valid JSON values.
- JSON fields are case-sensitive.
- White space between JSON structural elements (such as `{ }, [ ]`) is ignored.

The AWS Clean Rooms JSON functions and the AWS Clean Rooms COPY command use the same methods to work with JSON-formatted data.

Topics

CAN_JSON_PARSE function

The CAN_JSON_PARSE function parses data in JSON format and returns `true` if the result can be converted to a SUPER value using the JSON_PARSE function.

Syntax

```
CAN_JSON_PARSE(json_string)
```

Arguments

json_string

An expression that returns serialized JSON in the VARBYTE or VARCHAR form.

Return type

BOOLEAN

Example

To see if the JSON array `[10001,10002,"abc"]` can be converted into the SUPER data type, use the following example.

```
SELECT CAN_JSON_PARSE('[10001,10002,"abc"]');

+----------------+
| can_json_parse |
+----------------+
| true           |
+----------------+
```

JSON_EXTRACT_ARRAY_ELEMENT_TEXT function

The JSON_EXTRACT_ARRAY_ELEMENT_TEXT function returns a JSON array element in the outermost array of a JSON string, using a zero-based index. The first element in an array is at position 0. If the index is negative or out of bound, JSON_EXTRACT_ARRAY_ELEMENT_TEXT returns empty string. If the *null_if_invalid* argument is set to `true` and the JSON string is invalid, the function returns NULL instead of returning an error.

For more information, see JSON functions.

Syntax

```
json_extract_array_element_text('json string', pos [, null_if_invalid ] )
```

Arguments

json_string

A properly formatted JSON string.

pos

An integer representing the index of the array element to be returned, using a zero-based array index.

null_if_invalid

A Boolean value that specifies whether to return NULL if the input JSON string is invalid instead of returning an error. To return NULL if the JSON is invalid, specify `true` (`t`). To return an error if the JSON is invalid, specify `false` (`f`). The default is `false`.

Return type

A VARCHAR string representing the JSON array element referenced by *pos*.

Example

The following example returns array element at position 2, which is the third element of a zero-based array index:

```
select json_extract_array_element_text('[111,112,113]', 2);

json_extract_array_element_text
-------------------------------
113
```

The following example returns an error because the JSON is invalid.

```
select json_extract_array_element_text('["a",["b",1,["c",2,3,null,]]]',1);

An error occurred when executing the SQL command:
select json_extract_array_element_text('["a",["b",1,["c",2,3,null,]]]',1)
```

The following example sets *null_if_invalid* to *true*, so the statement returns NULL instead of returning an error for invalid JSON.

```
select json_extract_array_element_text('["a",["b",1,["c",2,3,null,]]]',1,true);

json_extract_array_element_text
-------------------------------
```

JSON_EXTRACT_PATH_TEXT function

The JSON_EXTRACT_PATH_TEXT function returns the value for the *key:value* pair referenced by a series of path elements in a JSON string. The JSON path can be nested up to five levels deep. Path elements are case-sensitive. If a path element does not exist in the JSON string, JSON_EXTRACT_PATH_TEXT returns an empty string. If the `null_if_invalid` argument is set to `true` and the JSON string is invalid, the function returns NULL instead of returning an error.

For information about additional JSON functions, see JSON functions.

Syntax

```
json_extract_path_text('json_string', 'path_elem' [,'path_elem'[, …] ]
 [, null_if_invalid ] )
```

Arguments

json_string

A properly formatted JSON string.

path_elem

A path element in a JSON string. One path element is required. Additional path elements can be specified, up to five levels deep.

null_if_invalid

A Boolean value that specifies whether to return NULL if the input JSON string is invalid instead of returning an error. To return NULL if the JSON is invalid, specify `true` (`t`). To return an error if the JSON is invalid, specify `false` (`f`). The default is `false`.

In a JSON string, AWS Clean Rooms recognizes `\n` as a newline character and `\t` as a tab character. To load a backslash, escape it with a backslash (`\\`).

Return type

VARCHAR string representing the JSON value referenced by the path elements.

Example

The following example returns the value for the path `'f4', 'f6'`.

```
select json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"star"}}','f4', 'f6');


json_extract_path_text
----------------------
star
```

The following example returns an error because the JSON is invalid.

```
select json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"star"}','f4', 'f6');

An error occurred when executing the SQL command:
select json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"star"}','f4', 'f6')
```

The following example sets *null_if_invalid* to *true*, so the statement returns NULL for invalid JSON instead of returning an error.

```
select json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"star"}','f4',
 'f6',true);

json_extract_path_text
------------------------------
NULL
```

The following example returns the value for the path `'farm', 'barn', 'color'`, where the value retrieved is at the third level. This sample is formatted with a JSON lint tool, to make it easier to read.

```
select json_extract_path_text('{
    "farm": {
        "barn": {
            "color": "red",
            "feed stocked": true
        }
```

```
    }
}', 'farm', 'barn', 'color');

json_extract_path_text
----------------------
red
```

The following example returns NULL because the `'color'` element is missing. This sample is formatted with a JSON lint tool.

```
select json_extract_path_text('{
    "farm": {
        "barn": {}
    }
}', 'farm', 'barn', 'color');

json_extract_path_text
----------------------
NULL
```

If the JSON is valid, trying to extract an element that's missing returns NULL.

The following example returns the value for the path `'house', 'appliances', 'washing machine', 'brand'`.

```
select json_extract_path_text('{
  "house": {
    "address": {
      "street": "123 Any St.",
      "city": "Any Town",
      "state": "FL",
      "zip": "32830"
    },
    "bathroom": {
      "color": "green",
      "shower": true
    },
    "appliances": {
      "washing machine": {
        "brand": "Any Brand",
        "color": "beige"
      },
      "dryer": {
```

```
        "brand": "Any Brand",
        "color": "white"
      }
    }
  }
}', 'house', 'appliances', 'washing machine', 'brand');

json_extract_path_text
----------------------
Any Brand
```

JSON_PARSE function

The JSON_PARSE function parses data in JSON format and converts it into the SUPER representation.

To ingest into SUPER data type using the INSERT or UPDATE command, use the JSON_PARSE function. When you use JSON_PARSE() to parse JSON strings into SUPER values, certain restrictions apply.

Syntax

```
JSON_PARSE(json_string)
```

Arguments

json_string

An expression that returns serialized JSON as a varbyte or varchar type.

Return type

SUPER

Example

The following example is an example of the JSON_PARSE function.

```
SELECT JSON_PARSE('[10001,10002,"abc"]');
     json_parse
-------------------------
```

```
 [10001,10002,"abc"]
(1 row)
```

```
SELECT JSON_TYPEOF(JSON_PARSE('[10001,10002,"abc"]'));
 json_typeof
----------------
 array
(1 row)
```

JSON_SERIALIZE function

The JSON_SERIALIZE function serializes a SUPER expression into textual JSON representation to follow RFC 8259. For more information on that RFC, see The JavaScript Object Notation (JSON) Data Interchange Format.

The SUPER size limit is approximately the same as the block limit, and the varchar limit is smaller than the SUPER size limit. Therefore, the JSON_SERIALIZE function returns an error when the JSON format exceeds the varchar limit of the system.

Syntax

```
JSON_SERIALIZE(super_expression)
```

Arguments

super_expression

A super expression or column.

Return type

varchar

Example

The following example serializes a SUPER value to a string.

```
 SELECT JSON_SERIALIZE(JSON_PARSE('[10001,10002,"abc"]'));
   json_serialize
---------------------
```

```
 [10001,10002,"abc"]
(1 row)
```

JSON_SERIALIZE_TO_VARBYTE function

The JSON_SERIALIZE_TO_VARBYTE function converts a SUPER value to a JSON string similar to JSON_SERIALIZE(), but stored in a VARBYTE value instead.

Syntax

```
JSON_SERIALIZE_TO_VARBYTE(super_expression)
```

Arguments

super_expression

A super expression or column.

Return type

varbyte

Example

The following example serializes a SUPER value and returns the result in VARBYTE format.

```
SELECT JSON_SERIALIZE_TO_VARBYTE(JSON_PARSE('[10001,10002,"abc"]'));
```

```
       json_serialize_to_varbyte
----------------------------------------
 5b31303030312c31303030322c22616263225d
```

The following example serializes a SUPER value and casts the result to VARCHAR format.

```
SELECT JSON_SERIALIZE_TO_VARBYTE(JSON_PARSE('[10001,10002,"abc"]'))::VARCHAR;
```

```
  json_serialize_to_varbyte
---------------------------
 [10001,10002,"abc"]
```

Math functions

This section describes the mathematical operators and functions supported in AWS Clean Rooms.

Topics

- SQRT function
- TRUNC function

Mathematical operator symbols

The following table lists the supported mathematical operators.

Supported operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division	4 / 2	2
%	modulo	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
|/	square root	| / 25.0	5
||/	cube root	|| / 27.0	3
@	absolute value	@ -5.0	5

Examples

Calculate the commission paid plus a $2.00 handling fee for a given transaction:

```
select commission, (commission + 2.00) as comm
from sales where salesid=10000;
```

```
commission | comm
-----------+-------
28.05      | 30.05
(1 row)
```

Calculate 20 percent of the sales price for a given transaction:

```
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;

pricepaid | twentypct
----------+-----------
187.00    |    37.400
(1 row)
```

Forecast ticket sales based on a continuous growth pattern. In this example, the subquery returns the number of tickets sold in 2008. That result is multiplied exponentially by a continuous growth rate of 5 percent over 10 years.

```
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid and year=2008)
^ ((5::float/100)*10) as qty10years;

qty10years
------------------
587.664019657491
(1 row)
```

Find the total price paid and commission for sales with a date ID that is greater than or equal to 2,000. Then subtract the total commission from the total price paid.

```
select sum (pricepaid) as sum_price, dateid,
sum (commission) as sum_comm, (sum (pricepaid) - sum (commission)) as value
from sales where dateid >= 2000
group by dateid order by dateid limit 10;

 sum_price | dateid | sum_comm |   value
-----------+--------+----------+-----------
 364445.00 |   2044 | 54666.75 | 309778.25
 349344.00 |   2112 | 52401.60 | 296942.40
 343756.00 |   2124 | 51563.40 | 292192.60
 378595.00 |   2116 | 56789.25 | 321805.75
```

```
 328725.00 |   2080 | 49308.75 | 279416.25
 349554.00 |   2028 | 52433.10 | 297120.90
 249207.00 |   2164 | 37381.05 | 211825.95
 285202.00 |   2064 | 42780.30 | 242421.70
 320945.00 |   2012 | 48141.75 | 272803.25
 321096.00 |   2016 | 48164.40 | 272931.60
(10 rows)
```

ABS function

ABS calculates the absolute value of a number, where that number can be a literal or an expression that evaluates to a number.

Syntax

```
ABS (number)
```

Arguments

number

Number or expression that evaluates to a number. It can be the SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4, or FLOAT8 type.

Return type

ABS returns the same data type as its argument.

Examples

Calculate the absolute value of -38:

```
select abs (-38);
abs
-------
38
(1 row)
```

Calculate the absolute value of (14-76):

```
select abs (14-76);
```

```
abs
-------
62
(1 row)
```

ACOS function

ACOS is a trigonometric function that returns the arc cosine of a number. The return value is in radians and is between `0` and `PI`.

Syntax

```
ACOS(number)
```

Arguments

number

The input parameter is a `DOUBLE PRECISION` number.

Return type

`DOUBLE PRECISION`

Examples

To return the arc cosine of `-1`, use the following example.

```
SELECT ACOS(-1);

+-------------------+
|       acos        |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ASIN function

ASIN is a trigonometric function that returns the arc sine of a number. The return value is in radians and is between `PI/2` and `-PI/2`.

Syntax

```
ASIN(number)
```

Arguments

number

The input parameter is a DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Examples

To return the arc sine of 1, use the following example.

```
SELECT ASIN(1) AS halfpi;

+--------------------+
|       halfpi       |
+--------------------+
| 1.5707963267948966 |
+--------------------+
```

ATAN function

ATAN is a trigonometric function that returns the arc tangent of a number. The return value is in radians and is between -PI and PI.

Syntax

```
ATAN(number)
```

Arguments

number

The input parameter is a DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Examples

To return the arc tangent of 1 and multiply it by 4, use the following example.

```
SELECT ATAN(1) * 4 AS pi;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ATAN2 function

ATAN2 is a trigonometric function that returns the arc tangent of one number divided by another number. The return value is in radians and is between PI/2 and -PI/2.

Syntax

```
ATAN2(number1, number2)
```

Arguments

number1

A DOUBLE PRECISION number.

number2

A DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Examples

To return the arc tangent of 2/2 and multiply it by 4, use the following example.

```
SELECT ATAN2(2,2) * 4 AS PI;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

CBRT function

The CBRT function is a mathematical function that calculates the cube root of a number.

Syntax

```
CBRT (number)
```

Argument

CBRT takes a DOUBLE PRECISION number as an argument.

Return type

CBRT returns a DOUBLE PRECISION number.

Examples

Calculate the cube root of the commission paid for a given transaction:

```
select cbrt(commission) from sales where salesid=10000;

cbrt
------------------
3.03839539048843
(1 row)
```

CEILING (or CEIL) function

The CEILING or CEIL function is used to round a number up to the next whole number. (The FLOOR function rounds a number down to the next whole number.)

Syntax

```
CEIL | CEILING(number)
```

Arguments

number

The number or expression that evaluates to a number. It can be the SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4, or FLOAT8 type.

Return type

CEILING and CEIL return the same data type as its argument.

Example

Calculate the ceiling of the commission paid for a given sales transaction:

```
select ceiling(commission) from sales
where salesid=10000;

ceiling
---------
29
(1 row)
```

COS function

COS is a trigonometric function that returns the cosine of a number. The return value is in radians and is between -1 and 1, inclusive.

Syntax

```
COS(double_precision)
```

Argument

number

The input parameter is a double precision number.

Return type

The COS function returns a double precision number.

Examples

The following example returns cosine of 0:

```
select cos(0);
cos
-----
1
(1 row)
```

The following example returns the cosine of PI:

```
select cos(pi());
cos
-----
-1
(1 row)
```

COT function

COT is a trigonometric function that returns the cotangent of a number. The input parameter must be nonzero.

Syntax

```
COT(number)
```

Argument

number

The input parameter is a `DOUBLE PRECISION` number.

Return type

`DOUBLE PRECISION`

Examples

To return the cotangent of 1, use the following example.

```
SELECT COT(1);

+--------------------+
|        cot         |
+--------------------+
| 0.6420926159343306 |
+--------------------+
```

DEGREES function

Converts an angle in radians to its equivalent in degrees.

Syntax

```
DEGREES(number)
```

Argument

number

The input parameter is a DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Example

To return the degree equivalent of .5 radians, use the following example.

```
SELECT DEGREES(.5);

+-------------------+
|      degrees      |
+-------------------+
| 28.64788975654116 |
```

```
+-------------------+
```

To convert PI radians to degrees, use the following example.

```
SELECT DEGREES(pi());

+---------+
| degrees |
+---------+
|     180 |
+---------+
```

DEXP function

The DEXP function returns the exponential value in scientific notation for a double precision number. The only difference between the DEXP and EXP functions is that the parameter for DEXP must be a DOUBLE PRECISION.

Syntax

```
DEXP(number)
```

Argument

number

The input parameter is a DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Example

```
SELECT (SELECT SUM(qtysold)
FROM sales, date
WHERE sales.dateid=date.dateid
AND year=2008) * DEXP((7::FLOAT/100)*10) qty2010;

+-------------------+
```

```
|      qty2010      |
+-------------------+
| 695447.4837722216 |
+-------------------+
```

DLOG1 function

The DLOG1 function returns the natural logarithm of the input parameter.

The DLOG1 function is a synonym of the LN function.

DLOG10 function

The DLOG10 returns the base 10 logarithm of the input parameter.

The DLOG10 function is a synonym of the LOG function.

Syntax

```
DLOG10(number)
```

Argument

number

The input parameter is a double precision number.

Return type

The DLOG10 function returns a double precision number.

Example

The following example returns the base 10 logarithm of the number 100:

```
select dlog10(100);

dlog10
--------
2
```

```
(1 row)
```

EXP function

The EXP function implements the exponential function for a numeric expression, or the base of the natural logarithm, e, raised to the power of expression. The EXP function is the inverse of LN function.

Syntax

```
EXP (expression)
```

Argument

expression

The expression must be an INTEGER, DECIMAL, or DOUBLE PRECISION data type.

Return type

EXP returns a DOUBLE PRECISION number.

Example

Use the EXP function to forecast ticket sales based on a continuous growth pattern. In this example, the subquery returns the number of tickets sold in 2008. That result is multiplied by the result of the EXP function, which specifies a continuous growth rate of 7% over 10 years.

```
select (select sum(qtysold) from sales, date
where sales.dateid=date.dateid
and year=2008) * exp((7::float/100)*10) qty2018;

qty2018
------------------
695447.483772222
(1 row)
```

FLOOR function

The FLOOR function rounds a number down to the next whole number.

Syntax

```
FLOOR (number)
```

Argument

number

The number or expression that evaluates to a number. It can be the SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4, or FLOAT8 type.

Return type

FLOOR returns the same data type as its argument.

Example

The example shows the value of the commission paid for a given sales transaction before and after using the FLOOR function.

```
select commission from sales
where salesid=10000;

floor
-------
28.05
(1 row)

select floor(commission) from sales
where salesid=10000;

floor
-------
28
(1 row)
```

LN function

The LN function returns the natural logarithm of the input parameter.

The LN function is a synonym of the DLOG1 function.

Syntax

```
LN(expression)
```

Argument

expression

The target column or expression that the function operates on.

> **Note**
>
> This function returns an error for some data types if the expression references an AWS Clean Rooms user-created table or an AWS Clean Rooms STL or STV system table.

Expressions with the following data types produce an error if they reference a user-created or system table.

- BOOLEAN
- CHAR
- DATE
- DECIMAL or NUMERIC
- TIMESTAMP
- VARCHAR

Expressions with the following data types run successfully on user-created tables and STL or STV system tables:

- BIGINT
- DOUBLE PRECISION
- INTEGER
- REAL
- SMALLINT

Return type

The LN function returns the same type as the expression.

Example

The following example returns the natural logarithm, or base e logarithm, of the number 2.718281828:

```
select ln(2.718281828);
ln
--------------------
0.9999999998311267
(1 row)
```

Note that the answer is nearly equal to 1.

This example returns the natural logarithm of the values in the USERID column in the USERS table:

```
select username, ln(userid) from users order by userid limit 10;

 username |        ln
----------+-------------------
 JSG99FHE |                 0
 PGL08LJI | 0.693147180559945
 IFT66TXU |  1.09861228866811
 XDZ38RDD |  1.38629436111989
 AEB55QTM |   1.6094379124341
 NDQ15VBM |  1.79175946922805
 OWY35QYB |  1.94591014905531
 AZG78YIP |  2.07944154167984
 MSD36KVR |  2.19722457733622
 WKW41AIW |  2.30258509299405
(10 rows)
```

LOG function

Returns the base 10 logarithm of a number.

Synonym of DLOG10 function.

Syntax

```
LOG(number)
```

Argument

number

The input parameter is a double precision number.

Return type

The LOG function returns a double precision number.

Example

The following example returns the base 10 logarithm of the number 100:

```
select log(100);
dlog10
--------
2
(1 row)
```

MOD function

Returns the remainder of two numbers, otherwise known as a *modulo* operation. To calculate the result, the first parameter is divided by the second.

Syntax

```
MOD(number1, number2)
```

Arguments

number1

The first input parameter is an INTEGER, SMALLINT, BIGINT, or DECIMAL number. If either parameter is a DECIMAL type, the other parameter must also be a DECIMAL type. If either parameter is an INTEGER, the other parameter can be an INTEGER, SMALLINT, or BIGINT. Both parameters can also be SMALLINT or BIGINT, but one parameter cannot be a SMALLINT if the other is a BIGINT.

number2

The second parameter is an INTEGER, SMALLINT, BIGINT, or DECIMAL number. The same data type rules apply to *number2* as to *number1*.

Return type

Valid return types are DECIMAL, INT, SMALLINT, and BIGINT. The return type of the MOD function is the same numeric type as the input parameters, if both input parameters are the same type. If either input parameter is an INTEGER, however, the return type will also be an INTEGER.

Usage notes

You can use % as a modulo operator.

Examples

The following example return the remainder when a number is divided by another:

```
SELECT MOD(10, 4);

 mod
------
 2
```

The following example returns a decimal result:

```
SELECT MOD(10.5, 4);

 mod
------
 2.5
```

You can cast parameter values:

```
SELECT MOD(CAST(16.4 as integer), 5);

 mod
------
 1
```

Check if the first parameter is even by dividing it by 2:

```
SELECT mod(5,2) = 0 as is_even;

 is_even
--------
 false
```

You can use the % as a modulo operator:

```
SELECT 11 % 4 as remainder;

 remainder
-----------
 3
```

The following example returns information for odd-numbered categories in the CATEGORY table:

```
select catid, catname
from category
where mod(catid,2)=1
order by 1,2;

 catid |  catname
-------+-----------
     1 | MLB
     3 | NFL
     5 | MLS
     7 | Plays
     9 | Pop
    11 | Classical

(6 rows)
```

PI function

The PI function returns the value of pi to 14 decimal places.

Syntax

```
PI()
```

Return type

DOUBLE PRECISION

Examples

To return the value of pi, use the following example.

```
SELECT PI();

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

POWER function

The POWER function is an exponential function that raises a numeric expression to the power of a second numeric expression. For example, 2 to the third power is calculated as POWER(2,3), with a result of 8.

Syntax

```
{POW | POWER}(expression1, expression2)
```

Arguments

expression1

Numeric expression to be raised. Must be an INTEGER, DECIMAL, or FLOAT data type.

expression2

Power to raise *expression1*. Must be an INTEGER, DECIMAL, or FLOAT data type.

Return type

DOUBLE PRECISION

Example

```
SELECT (SELECT SUM(qtysold) FROM sales, date
WHERE sales.dateid=date.dateid
AND year=2008) * POW((1+7::FLOAT/100),10) qty2010;

+------------------+
|      qty2010     |
+------------------+
| 679353.7540885945 |
+------------------+
```

RADIANS function

The RADIANS function converts an angle in degrees to its equivalent in radians.

Syntax

```
RADIANS(number)
```

Argument

number

The input parameter is a DOUBLE PRECISION number.

Return type

DOUBLE PRECISION

Example

To return the radian equivalent of 180 degrees, use the following example.

```
SELECT RADIANS(180);

+------------------+
|     radians      |
+------------------+
| 3.141592653589793 |
+------------------+
```

RANDOM function

The RANDOM function generates a random value between 0.0 (inclusive) and 1.0 (exclusive).

Syntax

```
RANDOM()
```

Return type

RANDOM returns a DOUBLE PRECISION number.

Examples

1. Compute a random value between 0 and 99. If the random number is 0 to 1, this query produces a random number from 0 to 100:

   ```
   select cast (random() * 100 as int);

   INTEGER
   ------
   24
   (1 row)
   ```

2. Retrieve a uniform random sample of 10 items:

   ```
   select *
   from sales
   order by random()
   limit 10;
   ```

   Now retrieve a random sample of 10 items, but choose the items in proportion to their prices. For example, an item that is twice the price of another would be twice as likely to appear in the query results:

   ```
   select *
   from sales
   order by log(1 - random()) / pricepaid
   limit 10;
   ```

3. This example uses the SET command to set a SEED value so that RANDOM generates a predictable sequence of numbers.

   First, return three RANDOM integers without setting the SEED value first:

```
select cast (random() * 100 as int);
INTEGER
------
6
(1 row)

select cast (random() * 100 as int);
INTEGER
------
68
(1 row)

select cast (random() * 100 as int);
INTEGER
------
56
(1 row)
```

   Now, set the SEED value to `.25`, and return three more RANDOM numbers:

```
set seed to .25;
select cast (random() * 100 as int);
INTEGER
------
21
(1 row)

select cast (random() * 100 as int);
INTEGER
------
79
(1 row)

select cast (random() * 100 as int);
INTEGER
------
12
```

```
(1 row)
```

Finally, reset the SEED value to .25, and verify that RANDOM returns the same results as the previous three calls:

```
set seed to .25;
select cast (random() * 100 as int);
INTEGER
------
21
(1 row)

select cast (random() * 100 as int);
INTEGER
------
79
(1 row)

select cast (random() * 100 as int);
INTEGER
------
12
(1 row)
```

ROUND function

The ROUND function rounds numbers to the nearest integer or decimal.

The ROUND function can optionally include a second argument as an integer to indicate the number of decimal places for rounding, in either direction. When you don't provide the second argument, the function rounds to the nearest whole number. When the second argument *>n* is specified, the function rounds to the nearest number with *n* decimal places of precision.

Syntax

```
ROUND (number [ , integer ] )
```

Argument

number

A number or expression that evaluates to a number. It can be the DECIMAL or FLOAT8 type. AWS Clean Rooms can convert other data types per the implicit conversion rules.

integer (optional)

An integer that indicates the number of decimal places for rounding in either directions.

Return type

ROUND returns the same numeric data type as the input argument(s).

Examples

Round the commission paid for a given transaction to the nearest whole number.

```
select commission, round(commission)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    28
(1 row)
```

Round the commission paid for a given transaction to the first decimal place.

```
select commission, round(commission, 1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |  28.1
(1 row)
```

For the same query, extend the precision in the opposite direction.

```
select commission, round(commission, -1)
from sales where salesid=10000;
```

```
commission | round
-----------+-------
     28.05 |    30
(1 row)
```

SIGN function

The SIGN function returns the sign (positive or negative) of a number. The result of the SIGN function is `1`, `-1`, or `0` indicating the sign of the argument.

Syntax

```
SIGN (number)
```

Argument

number

Number or expression that evaluates to a number. It can be the DECIMALor FLOAT8 type. AWS Clean Rooms can convert other data types per the implicit conversion rules.

Return type

SIGN returns the same numeric data type as the input argument(s). If the input is DECIMAL, the output is DECIMAL(1,0).

Examples

To determine the sign of the commission paid for a given transaction from the SALES table, use the following example.

```
SELECT commission, SIGN(commission)
FROM sales WHERE salesid=10000;

+------------+------+
| commission | sign |
+------------+------+
|      28.05 |    1 |
+------------+------+
```

SIN function

SIN is a trigonometric function that returns the sine of a number. The return value is between -1 and 1.

Syntax

```
SIN(number)
```

Argument

number

A DOUBLE PRECISION number in radians.

Return type

DOUBLE PRECISION

Example

To return the sine of -PI, use the following example.

```
SELECT SIN(-PI());

+-------------------------+
|           sin           |
+-------------------------+
| -0.00000000000000012246 |
+-------------------------+
```

SQRT function

The SQRT function returns the square root of a numeric value. The square root is a number multiplied by itself to get the given value.

Syntax

```
SQRT (expression)
```

Argument

expression

The expression must have an integer, decimal, or floating-point data type. The expression can include functions. The system might perform implicit type conversions.

Return type

SQRT returns a DOUBLE PRECISION number.

Examples

The following example returns the square root of a number.

```
select sqrt(16);

sqrt
---------------
4
```

The following example performs an implicit type conversion.

```
select sqrt('16');

sqrt
---------------
4
```

The following example nests functions to perform a more complex task.

```
select sqrt(round(16.4));

sqrt
---------------
4
```

The following example results in the length of the radius when given the area of a circle. It calculates the radius in inches, for instance, when given the area in square inches. The area in the sample is 20.

```
select sqrt(20/pi());
```

This returns the value 5.046265044040321.

The following example returns the square root for COMMISSION values from the SALES table. The COMMISSION column is a DECIMAL column. This example shows how you can use the function in a query with more complex conditional logic.

```
select sqrt(commission)
from sales where salesid < 10 order by salesid;

sqrt
------------------
10.4498803820905
3.37638860322683
7.24568837309472
5.1234753829798
...
```

The following query returns the rounded square root for the same set of COMMISSION values.

```
select salesid, commission, round(sqrt(commission))
from sales where salesid < 10 order by salesid;

salesid | commission | round
--------+------------+-------
      1 |     109.20 |    10
      2 |      11.40 |     3
      3 |      52.50 |     7
      4 |      26.25 |     5
...
```

For more information about sample data in AWS Clean Rooms, see Sample database.

TRUNC function

The TRUNC function truncates numbers to the previous integer or decimal.

The TRUNC function can optionally include a second argument as an integer to indicate the number of decimal places for rounding, in either direction. When you don't provide the second argument, the function rounds to the nearest whole number. When the second argument *>n*is

specified, the function rounds to the nearest number with *>n* decimal places of precision. This function also truncates a timestamp and returns a date.

Syntax

```
TRUNC (number [ , integer ] |
timestamp )
```

Arguments

number

A number or expression that evaluates to a number. It can be the DECIMAL or FLOAT8 type. AWS Clean Rooms can convert other data types per the implicit conversion rules.

integer (optional)

An integer that indicates the number of decimal places of precision, in either direction. If no integer is provided, the number is truncated as a whole number; if an integer is specified, the number is truncated to the specified decimal place.

timestamp

The function can also return the date from a timestamp. (To return a timestamp value with `00:00:00` as the time, cast the function result to a timestamp.)

Return type

TRUNC returns the same data type as the first input argument. For timestamps, TRUNC returns a date.

Examples

Truncate the commission paid for a given sales transaction.

```
select commission, trunc(commission)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   111
```

```
(1 row)
```

Truncate the same commission value to the first decimal place.

```
select commission, trunc(commission,1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 | 111.1

(1 row)
```

Truncate the commission with a negative value for the second argument; `111.15` is rounded down to `110`.

```
select commission, trunc(commission,-1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   110
(1 row)
```

Return the date portion from the result of the SYSDATE function (which returns a timestamp):

```
select sysdate;

timestamp
----------------------------
2011-07-21 10:32:38.248109
(1 row)

select trunc(sysdate);

trunc
------------
2011-07-21
(1 row)
```

Apply the TRUNC function to a TIMESTAMP column. The return type is a date.

```
select trunc(starttime) from event
order by eventid limit 1;

trunc
------------
2008-01-25
(1 row)
```

String functions

Topics

- RTRIM function
- SOUNDEX function
- SPLIT_PART function
- STRPOS function
- SUBSTR function
- SUBSTRING function
- TEXTLEN function
- TRANSLATE function
- TRIM function
- UPPER function

String functions process and manipulate character strings or expressions that evaluate to character strings. When the *string* argument in these functions is a literal value, it must be enclosed in single quotation marks. Supported data types include CHAR and VARCHAR.

The following section provides the function names, syntax, and descriptions for supported functions. All offsets into strings are one-based.

|| (Concatenation) operator

Concatenates two expressions on either side of the || symbol and returns the concatenated expression.

The concatentation operator is similar to CONCAT function.

> **Note**
>
> For both the CONCAT function and the concatenation operator, if one or both expressions is null, the result of the concatenation is null.

Syntax

```
expression1 || expression2
```

Arguments

expression1, *expression2*

Both arguments can be fixed-length or variable-length character strings or expressions.

Return type

The || operator returns a string. The type of string is the same as the input arguments.

Example

The following example concatenates the FIRSTNAME and LASTNAME fields from the USERS table:

```
select firstname || ' ' || lastname
from users
order by 1
limit 10;

concat
-----------------
Aaron Banks
Aaron Booth
Aaron Browning
Aaron Burnett
Aaron Casey
Aaron Cash
Aaron Castro
Aaron Dickerson
Aaron Dixon
Aaron Dotson
(10 rows)
```

To concatenate columns that might contain nulls, use the NVL and COALESCE functions expression. The following example uses NVL to return a 0 whenever NULL is encountered.

```
select venuename || ' seats ' || nvl(venueseats, 0)
from venue where venuestate = 'NV' or venuestate = 'NC'
order by 1
limit 10;

seating
```

```
-----------------------------------
Ballys Hotel seats 0
Bank of America Stadium seats 73298
Bellagio Hotel seats 0
Caesars Palace seats 0
Harrahs Hotel seats 0
Hilton Hotel seats 0
Luxor Hotel seats 0
Mandalay Bay Hotel seats 0
Mirage Hotel seats 0
New York New York seats 0
```

BTRIM function

The BTRIM function trims a string by removing leading and trailing blanks or by removing leading and trailing characters that match an optional specified string.

Syntax

```
BTRIM(string [, trim_chars ] )
```

Arguments

string

The input VARCHAR string to be trimmed.

trim_chars

The VARCHAR string containing the characters to be matched.

Return type

The BTRIM function returns a VARCHAR string.

Examples

The following example trims leading and trailing blanks from the string `'  abc  '`:

```
select '     abc    ' as untrim, btrim('     abc    ') as trim;

untrim    | trim
```

```
----------+------
   abc    | abc
```

The following example removes the leading and trailing `'xyz'` strings from the string `'xyzaxyzbxyzcxyz'`. The leading and trailing occurrences of `'xyz'` are removed, but occurrences that are internal within the string are not removed.

```
select 'xyzaxyzbxyzcxyz' as untrim,
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | axyzbxyzc
```

The following example removes the leading and trailing parts from the string `'setuphistorycassettes'` that match any of the characters in the *trim_chars* list `'tes'`. Any `t`, `e`, or `s` that occur before another character that is not in the *trim_chars* list at the beginning or ending of the input string are removed.

```
SELECT btrim('setuphistorycassettes', 'tes');

     btrim
----------------
 uphistoryca
```

CHAR_LENGTH function

Synonym of the LEN function.

See LEN function.

CHARACTER_LENGTH function

Synonym of the LEN function.

See LEN function.

CHARINDEX function

Returns the location of the specified substring within a string.

See POSITION function and STRPOS function for similar functions.

Syntax

```
CHARINDEX( substring, string )
```

Arguments

substring

The substring to search for within the *string*.

string

The string or column to be searched.

Return type

The CHARINDEX function returns an integer corresponding to the position of the substring (one-based, not zero-based). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

Usage notes

CHARINDEX returns 0 if the substring is not found within the `string`:

```
select charindex('dog', 'fish');

charindex
----------
0
(1 row)
```

Examples

The following example shows the position of the string `fish` within the word `dogfish`:

```
select charindex('fish', 'dogfish');
 charindex
----------
         4
(1 row)
```

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```
select distinct charindex('.', commission), count (charindex('.', commission))
from sales where charindex('.', commission) > 4 group by charindex('.', commission)
order by 1,2;

charindex | count
----------+-------
 5        |    629
(1 row)
```

CONCAT function

The CONCAT function concatenates two expressions and returns the resulting expression. To concatenate more than two expressions, use nested CONCAT functions. The concatenation operator (||) between two expressions produces the same results as the CONCAT function.

> **Note**
>
> For both the CONCAT function and the concatenation operator, if one or both expressions is null, the result of the concatenation is null.

Syntax

```
CONCAT ( expression1, expression2 )
```

Arguments

expression1, *expression2*

Both arguments can be a fixed-length character string, a variable-length character string, a binary expression, or an expression that evaluates to one of these inputs.

Return type

CONCAT returns an expression. The data type of the expression is the same type as the input arguments.

If the input expressions are of different types, AWS Clean Rooms tries to implicitly type casts one of the expressions. If values can't be cast, an error is returned.

Examples

The following example concatenates two character literals:

```
select concat('December 25, ', '2008');

concat
-------------------
December 25, 2008
(1 row)
```

The following query, using the || operator instead of CONCAT, produces the same result:

```
select 'December 25, '||'2008';

concat
-------------------
December 25, 2008
(1 row)
```

The following example uses two CONCAT functions to concatenate three character strings:

```
select concat('Thursday, ', concat('December 25, ', '2008'));

concat
----------------------------
Thursday, December 25, 2008
(1 row)
```

To concatenate columns that might contain nulls, use the NVL and COALESCE functions. The following example uses NVL to return a 0 whenever NULL is encountered.

```
select concat(venuename, concat(' seats ', nvl(venueseats, 0))) as seating
from venue where venuestate = 'NV' or venuestate = 'NC'
order by 1
limit 5;

seating
-----------------------------------
```

```
Ballys Hotel seats 0
Bank of America Stadium seats 73298
Bellagio Hotel seats 0
Caesars Palace seats 0
Harrahs Hotel seats 0
(5 rows)
```

The following query concatenates CITY and STATE values from the VENUE table:

```
select concat(venuecity, venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
------------------
DenverCO
Kansas CityMO
East RutherfordNJ
LandoverMD
(4 rows)
```

The following query uses nested CONCAT functions. The query concatenates CITY and STATE values from the VENUE table but delimits the resulting string with a comma and a space:

```
select concat(concat(venuecity,', '),venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
--------------------
Denver, CO
Kansas City, MO
East Rutherford, NJ
Landover, MD
(4 rows)
```

LEFT and RIGHT functions

These functions return the specified number of leftmost or rightmost characters from a character string.

The number is based on the number of characters, not bytes, so that multibyte characters are counted as single characters.

Syntax

```
LEFT ( string,  integer )

RIGHT ( string,  integer )
```

Arguments

string

Any character string or any expression that evaluates to a character string.

integer

A positive integer.

Return type

LEFT and RIGHT return a VARCHAR string.

Example

The following example returns the leftmost 5 and rightmost 5 characters from event names that have IDs between 1000 and 1005:

```
select eventid, eventname,
left(eventname,5) as left_5,
right(eventname,5) as right_5
from event
where eventid between 1000 and 1005
order by 1;

eventid |   eventname    | left_5 | right_5
--------+----------------+--------+---------
   1000 | Gypsy          | Gypsy  | Gypsy
   1001 | Chicago        | Chica  | icago
   1002 | The King and I | The K  | and I
   1003 | Pal Joey       | Pal J  |  Joey
   1004 | Grease         | Greas  | rease
```

```
   1005 | Chicago        | Chica  | icago
(6 rows)
```

LEN function

Returns the length of the specified string as the number of characters.

Syntax

LEN is a synonym of LENGTH function, CHAR_LENGTH function, CHARACTER_LENGTH function, and TEXTLEN function.

```
LEN(expression)
```

Argument

expression

The input parameter is a CHAR or VARCHAR or an alias of one of the valid input types.

Return type

The LEN function returns an integer indicating the number of characters in the input string.

If the input string is a character string, the LEN function returns the actual number of characters in multi-byte strings, not the number of bytes. For example, a VARCHAR(12) column is required to store three four-byte Chinese characters. The LEN function will return 3 for that same string.

Usage notes

Length calculations do not count trailing spaces for fixed-length character strings but do count them for variable-length strings.

Example

The following example returns the number of bytes and the number of characters in the string `français`.

```
select octet_length('français'),
len('français');
```

```
octet_length  | len
--------------+-----
           9  |   8
```

The following example returns the number of characters in the strings `cat` with no trailing spaces and `cat` with three trailing spaces:

```
select len('cat'), len('cat   ');
 len | len
-----+-----
   3 |   6
```

The following example returns the ten longest VENUENAME entries in the VENUE table:

```
select venuename, len(venuename)
from venue
order by 2 desc, 1
limit 10;

venuename                                | len
-----------------------------------------+-----
Saratoga Springs Performing Arts Center |  39
Lincoln Center for the Performing Arts  |  38
Nassau Veterans Memorial Coliseum       |  33
Jacksonville Municipal Stadium          |  30
Rangers BallPark in Arlington           |  29
University of Phoenix Stadium           |  29
Circle in the Square Theatre            |  28
Hubert H. Humphrey Metrodome            |  28
Oriole Park at Camden Yards             |  27
Dick's Sporting Goods Park              |  26
```

LENGTH function

Synonym of the LEN function.

See LEN function.

LOWER function

Converts a string to lowercase. LOWER supports UTF-8 multibyte characters, up to a maximum of four bytes per character.

Syntax

```
LOWER(string)
```

Argument

string

The input parameter is a VARCHAR string (or any other data type, such as CHAR, that can be implicitly converted to VARCHAR).

Return type

The LOWER function returns a character string that is the same data type as the input string.

Examples

The following example converts the CATNAME field to lowercase:

```
select catname, lower(catname) from category order by 1,2;

 catname  |   lower
----------+-----------
Classical | classical
Jazz      | jazz
MLB       | mlb
MLS       | mls
Musicals  | musicals
NBA       | nba
NFL       | nfl
NHL       | nhl
Opera     | opera
Plays     | plays
Pop       | pop
(11 rows)
```

LPAD and RPAD functions

These functions prepend or append characters to a string, based on a specified length.

Syntax

```
LPAD (string1, length, [ string2 ])
```

```
RPAD (string1, length, [ string2 ])
```

Arguments

string1

A character string or an expression that evaluates to a character string, such as the name of a character column.

length

An integer that defines the length of the result of the function. The length of a string is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. If *string1* is longer than the specified length, it is truncated (on the right). If *length* is a negative number, the result of the function is an empty string.

string2

One or more characters that are prepended or appended to *string1*. This argument is optional; if it is not specified, spaces are used.

Return type

These functions return a VARCHAR data type.

Examples

Truncate a specified set of event names to 20 characters and prepend the shorter names with spaces:

```
select lpad(eventname,20) from event
where eventid between 1 and 5 order by 1;

 lpad
--------------------
              Salome
```

```
        Il Trovatore
       Boris Godunov
     Gotterdammerung
La Cenerentola (Cind
(5 rows)
```

Truncate the same set of event names to 20 characters but append the shorter names with 0123456789.

```
select rpad(eventname,20,'0123456789') from event
where eventid between 1 and 5 order by 1;

 rpad
--------------------
Boris Godunov0123456
Gotterdammerung01234
Il Trovatore01234567
La Cenerentola (Cind
Salome01234567890123
(5 rows)
```

LTRIM function

Trims characters from the beginning of a string. Removes the longest string containing only characters in the trim characters list. Trimming is complete when a trim character does not appear in the input string.

Syntax

```
LTRIM( string [, trim_chars] )
```

Arguments

string

A string column, expression, or string literal to be trimmed.

trim_chars

A string column, expression, or string literal that represents the characters to be trimmed from the beginning of *string*. If not specified, a space is used as the trim character.

Return type

The LTRIM function returns a character string that is the same data type as the input *string* (CHAR or VARCHAR).

Examples

The following example trims the year from the `listime` column. The trim characters in string literal `'2008-'` indicate the characters to be trimmed from the left. If you use the trim characters `'028-'`, you achieve the same result.

```
select listid, listtime, ltrim(listtime, '2008-')
from listing
order by 1, 2, 3
limit 10;

listid |      listtime       |     ltrim
-------+---------------------+----------------
     1 | 2008-01-24 06:43:29 | 1-24 06:43:29
     2 | 2008-03-05 12:25:29 | 3-05 12:25:29
     3 | 2008-11-01 07:35:33 | 11-01 07:35:33
     4 | 2008-05-24 01:18:37 | 5-24 01:18:37
     5 | 2008-05-17 02:29:11 | 5-17 02:29:11
     6 | 2008-08-15 02:08:13 | 15 02:08:13
     7 | 2008-11-15 09:38:15 | 11-15 09:38:15
     8 | 2008-11-09 05:07:30 | 11-09 05:07:30
     9 | 2008-09-09 08:03:36 | 9-09 08:03:36
    10 | 2008-06-17 09:44:54 | 6-17 09:44:54
```

LTRIM removes any of the characters in *trim_chars* when they appear at the beginning of *string*. The following example trims the characters 'C', 'D', and 'G' when they appear at the beginning of VENUENAME, which is a VARCHAR column.

```
select venueid, venuename, ltrim(venuename, 'CDG')
from venue
where venuename like '%Park'
order by 2
limit 7;

venueid | venuename                  | btrim
--------+----------------------------+--------------------------
    121 | ATT Park                   | ATT Park
```

```
    109 | Citizens Bank Park         | itizens Bank Park
    102 | Comerica Park              | omerica Park
      9 | Dick's Sporting Goods Park | ick's Sporting Goods Park
     97 | Fenway Park                | Fenway Park
    112 | Great American Ball Park   | reat American Ball Park
    114 | Miller Park                | Miller Park
```

The following example uses the trim character 2 which is retrieved from the `venueid` column.

```
select ltrim('2008-01-24 06:43:29', venueid)
from venue where venueid=2;

ltrim
------------------
008-01-24 06:43:29
```

The following example does not trim any characters because a 2 is found before the `'0'` trim character.

```
select ltrim('2008-01-24 06:43:29', '0');

ltrim
-------------------
2008-01-24 06:43:29
```

The following example uses the default space trim character and trims the two spaces from the beginning of the string.

```
select ltrim('  2008-01-24 06:43:29');

ltrim
-------------------
2008-01-24 06:43:29
```

POSITION function

Returns the location of the specified substring within a string.

See CHARINDEX function and STRPOS function for similar functions.

Syntax

```
POSITION(substring IN string )
```

Arguments

substring

The substring to search for within the *string*.

string

The string or column to be searched.

Return type

The POSITION function returns an integer corresponding to the position of the substring (one-based, not zero-based). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

Usage notes

POSITION returns 0 if the substring is not found within the string:

```
select position('dog' in 'fish');

position
----------
 0
(1 row)
```

Examples

The following example shows the position of the string `fish` within the word `dogfish`:

```
select position('fish' in 'dogfish');

position
----------
 4
```

```
(1 row)
```

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```
select distinct position('.' in commission), count (position('.' in commission))
from sales where position('.' in commission) > 4 group by position('.' in commission)
order by 1,2;

position | count
---------+-------
       5 |    629
(1 row)
```

REGEXP_COUNT function

Searches a string for a regular expression pattern and returns an integer that indicates the number of times the pattern occurs in the string. If no match is found, then the function returns 0.

Syntax

```
REGEXP_COUNT ( source_string, pattern [, position [, parameters ] ] )
```

Arguments

source_string

A string expression, such as a column name, to be searched.

pattern

A string literal that represents a regular expression pattern.

position

A positive integer that indicates the position within *source_string* to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single characters. The default is 1. If *position* is less than 1, the search begins at the first character of *source_string*. If *position* is greater than the number of characters in *source_string*, the result is 0.

parameters

One or more string literals that indicate how the function matches the pattern. The possible values are the following:

- c – Perform case-sensitive matching. The default is to use case-sensitive matching.
- i – Perform case-insensitive matching.
- p – Interpret the pattern with Perl Compatible Regular Expression (PCRE) dialect.

Return type

Integer

Example

The following example counts the number of times a three-letter sequence occurs.

```
SELECT regexp_count('abcdefghijklmnopqrstuvwxyz', '[a-z]{3}');

 regexp_count
 --------------
            8
```

The following example counts the number of times the top-level domain name is either `org` or edu.

```
SELECT email, regexp_count(email,'@[^.]*\\.(org|edu)')FROM users
ORDER BY userid LIMIT 4;

                      email                      | regexp_count
-------------------------------------------------+--------------
 Etiam.laoreet.libero@sodalesMaurisblandit.edu |            1
 Suspendisse.tristique@nonnisiAenean.edu       |            1
 amet.faucibus.ut@condimentumegetvolutpat.ca   |            0
 sed@lacusUtnec.ca                             |            0
```

The following example counts the occurrences of the string FOX, using case-insensitive matching.

```
SELECT regexp_count('the fox', 'FOX', 1, 'i');
```

```
 regexp_count
 --------------
            1
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example counts the number of occurrences of such words, with case-sensitive matching.

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'p');

 regexp_count
 --------------
            2
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific connotation in PCRE. This example counts the number of occurrences of such words, but differs from the previous example in that it uses case-insensitive matching.

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'ip');

 regexp_count
 --------------
            3
```

REGEXP_INSTR function

Searches a string for a regular expression pattern and returns an integer that indicates the beginning position or ending position of the matched substring. If no match is found, then the function returns 0. REGEXP_INSTR is similar to the POSITION function, but lets you search a string for a regular expression pattern.

Syntax

```
REGEXP_INSTR ( source_string, pattern [, position [, occurrence] [, option
 [, parameters ] ] ] ] )
```

Arguments

source_string

A string expression, such as a column name, to be searched.

pattern

A string literal that represents a regular expression pattern.

position

A positive integer that indicates the position within *source_string* to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single characters. The default is 1. If *position* is less than 1, the search begins at the first character of *source_string*. If *position* is greater than the number of characters in *source_string*, the result is 0.

occurrence

A positive integer that indicates which occurrence of the pattern to use. REGEXP_INSTR skips the first *occurrence* -1 matches. The default is 1. If *occurrence* is less than 1 or greater than the number of characters in *source_string*, the search is ignored and the result is 0.

option

A value that indicates whether to return the position of the first character of the match (0) or the position of the first character following the end of the match (1). A nonzero value is the same as 1. The default value is 0.

parameters

One or more string literals that indicate how the function matches the pattern. The possible values are the following:

- c – Perform case-sensitive matching. The default is to use case-sensitive matching.
- i – Perform case-insensitive matching.
- e – Extract a substring using a subexpression.

  If *pattern* includes a subexpression, REGEXP_INSTR matches a substring using the first subexpression in *pattern*. REGEXP_INSTR considers only the first subexpression; additional subexpressions are ignored. If the pattern doesn't have a subexpression, REGEXP_INSTR ignores the 'e' parameter.
- p – Interpret the pattern with Perl Compatible Regular Expression (PCRE) dialect.

Return type

Integer

Example

The following example searches for the @ character that begins a domain name and returns the starting position of the first match.

```
SELECT email, regexp_instr(email, '@[^.]*')
FROM users
ORDER BY userid LIMIT 4;

                     email                      | regexp_instr
------------------------------------------------+--------------
 Etiam.laoreet.libero@example.com |           21
 Suspendisse.tristique@nonnisiAenean.edu       |           22
 amet.faucibus.ut@condimentumegetvolutpat.ca   |           17
 sed@lacusUtnec.ca                             |            4
```

The following example searches for variants of the word `Center` and returns the starting position of the first match.

```
SELECT venuename, regexp_instr(venuename,'[cC]ent(er|re)$')
FROM venue
WHERE regexp_instr(venuename,'[cC]ent(er|re)$') > 0
ORDER BY venueid LIMIT 4;

       venuename       | regexp_instr
-----------------------+--------------
 The Home Depot Center |           16
 Izod Center           |            6
 Wachovia Center       |           10
 Air Canada Centre     |           12
```

The following example finds the starting position of the first occurrence of the string `FOX`, using case-insensitive matching logic.

```
SELECT regexp_instr('the fox', 'FOX', 1, 1, 0, 'i');

 regexp_instr
 --------------
```

```
            5
```

The following example uses a pattern written in PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example finds the starting position of the second such word.

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'p');

 regexp_instr
 --------------
           21
```

The following example uses a pattern written in PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example finds the starting position of the second such word, but differs from the previous example in that it uses case-insensitive matching.

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'ip');

 regexp_instr
 --------------
           15
```

REGEXP_REPLACE function

Searches a string for a regular expression pattern and replaces every occurrence of the pattern with the specified string. REGEXP_REPLACE is similar to the REPLACE function, but lets you search a string for a regular expression pattern.

REGEXP_REPLACE is similar to the TRANSLATE function and the REPLACE function, except that TRANSLATE makes multiple single-character substitutions and REPLACE substitutes one entire string with another string, while REGEXP_REPLACE lets you search a string for a regular expression pattern.

Syntax

```
REGEXP_REPLACE ( source_string, pattern [, replace_string [ , position [, parameters
 ] ] ] )
```

Arguments

source_string

A string expression, such as a column name, to be searched.

pattern

A string literal that represents a regular expression pattern.

replace_string

A string expression, such as a column name, that will replace each occurrence of pattern. The default is an empty string ("").

position

A positive integer that indicates the position within *source_string* to begin searching. The position is based on the number of characters, not bytes, so that multibyte characters are counted as single characters. The default is 1. If *position* is less than 1, the search begins at the first character of *source_string*. If *position* is greater than the number of characters in *source_string*, the result is *source_string*.

parameters

One or more string literals that indicate how the function matches the pattern. The possible values are the following:

- c – Perform case-sensitive matching. The default is to use case-sensitive matching.
- i – Perform case-insensitive matching.
- p – Interpret the pattern with Perl Compatible Regular Expression (PCRE) dialect.

Return type

VARCHAR

If either *pattern* or *replace_string* is NULL, the return is NULL.

Example

The following example deletes the @ and domain name from email addresses.

```
SELECT email, regexp_replace(email, '@.*\\.(org|gov|com|edu|ca)$')
FROM users
ORDER BY userid LIMIT 4;

              email                              | regexp_replace
-------------------------------------------------+----------------
 Etiam.laoreet.libero@sodalesMaurisblandit.edu | Etiam.laoreet.libero
 Suspendisse.tristique@nonnisiAenean.edu       | Suspendisse.tristique
 amet.faucibus.ut@condimentumegetvolutpat.ca   | amet.faucibus.ut
 sed@lacusUtnec.ca                             | sed
```

The following example replaces the domain names of email addresses with this value: `internal.company.com`.

```
SELECT email, regexp_replace(email, '@.*\\.[[:alpha:]]{2,3}',
'@internal.company.com') FROM users
ORDER BY userid LIMIT 4;

                     email                      |               regexp_replace
------------------------------------------------
+-----------------------------------------
 Etiam.laoreet.libero@sodalesMaurisblandit.edu |
 Etiam.laoreet.libero@internal.company.com
 Suspendisse.tristique@nonnisiAenean.edu       |
 Suspendisse.tristique@internal.company.com
 amet.faucibus.ut@condimentumegetvolutpat.ca   | amet.faucibus.ut@internal.company.com
 sed@lacusUtnec.ca                             | sed@internal.company.com
```

The following example replaces all occurrences of the string FOX within the value `quick brown fox`, using case-insensitive matching.

```
SELECT regexp_replace('the fox', 'FOX', 'quick brown fox', 1, 'i');

   regexp_replace
--------------------
 the quick brown fox
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example replaces each occurrence of such a word with the value `[hidden]`.

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'p');

       regexp_replace
------------------------------
 [hidden] plain A1234 [hidden]
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example replaces each occurrence of such a word with the value [hidden], but differs from the previous example in that it uses case-insensitive matching.

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'ip');

         regexp_replace
---------------------------------
 [hidden] plain [hidden] [hidden]
```

REGEXP_SUBSTR function

Returns characters from a string by searching it for a regular expression pattern. REGEXP_SUBSTR is similar to the SUBSTRING function function, but lets you search a string for a regular expression pattern. If the function can't match the regular expression to any characters in the string, it returns an empty string.

Syntax

```
REGEXP_SUBSTR ( source_string, pattern [, position [, occurrence [, parameters ] ] ] )
```

Arguments

source_string

A string expression to be searched.

pattern

A string literal that represents a regular expression pattern.

position

A positive integer that indicates the position within *source_string* to begin searching. The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. The default is 1. If *position* is less than 1, the search begins at the first character of *source_string*. If *position* is greater than the number of characters in *source_string*, the result is an empty string ("").

occurrence

A positive integer that indicates which occurrence of the pattern to use. REGEXP_SUBSTR skips the first *occurrence* -1 matches. The default is 1. If *occurrence* is less than 1 or greater than the number of characters in *source_string*, the search is ignored and the result is NULL.

parameters

One or more string literals that indicate how the function matches the pattern. The possible values are the following:

- c – Perform case-sensitive matching. The default is to use case-sensitive matching.
- i – Perform case-insensitive matching.
- e – Extract a substring using a subexpression.

  If *pattern* includes a subexpression, REGEXP_SUBSTR matches a substring using the first subexpression in *pattern*. A subexpression is an expression within the pattern that is bracketed with parentheses. For example, for the pattern `'This is a (\\w+)'` matches the first expression with the string `'This is a '` followed by a word. Instead of returning *pattern*, REGEXP_SUBSTR with the `e` parameter returns only the string inside the subexpression.

  REGEXP_SUBSTR considers only the first subexpression; additional subexpressions are ignored. If the pattern doesn't have a subexpression, REGEXP_SUBSTR ignores the 'e' parameter.
- p – Interpret the pattern with Perl Compatible Regular Expression (PCRE) dialect.

Return type

VARCHAR

Example

The following example returns the portion of an email address between the @ character and the domain extension.

```
SELECT email, regexp_substr(email,'@[^.]*')
FROM users
ORDER BY userid LIMIT 4;

                     email                      |      regexp_substr
------------------------------------------------+--------------------------
 Etiam.laoreet.libero@sodalesMaurisblandit.edu | @sodalesMaurisblandit
 Suspendisse.tristique@nonnisiAenean.edu       | @nonnisiAenean
 amet.faucibus.ut@condimentumegetvolutpat.ca   | @condimentumegetvolutpat
 sed@lacusUtnec.ca                             | @lacusUtnec
```

The following example returns the portion of the input corresponding to the first occurrence of the string FOX, using case-insensitive matching.

```
SELECT regexp_substr('the fox', 'FOX', 1, 1, 'i');

 regexp_substr
---------------
 fox
```

The following example returns the first portion of the input that begins with lowercase letters. This is functionally identical to the same SELECT statement without the c parameter.

```
SELECT regexp_substr('THE SECRET CODE IS THE LOWERCASE PART OF 1931abc0EZ.', '[a-z]+',
 1, 1, 'c');

 regexp_substr
---------------
 abc
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example returns the portion of the input corresponding to the second such word.

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'p');

 regexp_substr
---------------
 a1234
```

The following example uses a pattern written in the PCRE dialect to locate words containing at least one number and one lowercase letter. It uses the ?= operator, which has a specific look-ahead connotation in PCRE. This example returns the portion of the input corresponding to the second such word, but differs from the previous example in that it uses case-insensitive matching.

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'ip');

 regexp_substr
---------------
 A1234
```

The following example uses a subexpression to find the second string matching the pattern `'this is a (\\w+)'` using case-insensitive matching. It returns the subexpression inside the parentheses.

```
select regexp_substr(
               'This is a cat, this is a dog. This is a mouse.',
               'this is a (\\w+)', 1, 2, 'ie');

 regexp_substr
---------------
 dog
```

REPEAT function

Repeats a string the specified number of times. If the input parameter is numeric, REPEAT treats it as a string.

Synonym for REPLICATE function.

Syntax

```
REPEAT(string, integer)
```

Arguments

string

The first input parameter is the string to be repeated.

integer

The second parameter is an integer indicating the number of times to repeat the string.

Return type

The REPEAT function returns a string.

Examples

The following example repeats the value of the CATID column in the CATEGORY table three times:

```
select catid, repeat(catid,3)
from category
order by 1,2;

 catid | repeat
-------+--------
     1 | 111
     2 | 222
     3 | 333
     4 | 444
     5 | 555
     6 | 666
     7 | 777
     8 | 888
     9 | 999
    10 | 101010
    11 | 111111
(11 rows)
```

REPLACE function

Replaces all occurrences of a set of characters within an existing string with other specified characters.

REPLACE is similar to the TRANSLATE function and the REGEXP_REPLACE function, except that TRANSLATE makes multiple single-character substitutions and REGEXP_REPLACE lets you search a string for a regular expression pattern, while REPLACE substitutes one entire string with another string.

Syntax

```
REPLACE(string1, old_chars, new_chars)
```

Arguments

string

CHAR or VARCHAR string to be searched search

old_chars

CHAR or VARCHAR string to replace.

new_chars

New CHAR or VARCHAR string replacing the *old_string*.

Return type

VARCHAR

If either *old_chars* or *new_chars* is NULL, the return is NULL.

Examples

The following example converts the string Shows to `Theatre` in the CATGROUP field:

```
select catid, catgroup,
replace(catgroup, 'Shows', 'Theatre')
```

```
from category
order by 1,2,3;

 catid | catgroup | replace
-------+----------+----------
     1 | Sports   | Sports
     2 | Sports   | Sports
     3 | Sports   | Sports
     4 | Sports   | Sports
     5 | Sports   | Sports
     6 | Shows    | Theatre
     7 | Shows    | Theatre
     8 | Shows    | Theatre
     9 | Concerts | Concerts
    10 | Concerts | Concerts
    11 | Concerts | Concerts
(11 rows)
```

REPLICATE function

Synonym for the REPEAT function.

See REPEAT function.

REVERSE function

The REVERSE function operates on a string and returns the characters in reverse order. For example, `reverse('abcde')` returns edcba. This function works on numeric and date data types as well as character data types; however, in most cases it has practical value for character strings.

Syntax

```
REVERSE ( expression )
```

Argument

expression

An expression with a character, date, timestamp, or numeric data type that represents the target of the character reversal. All expressions are implicitly converted to variable-length character strings. Trailing blanks in fixed-width character strings are ignored.

Return type

REVERSE returns a VARCHAR.

Examples

Select five distinct city names and their corresponding reversed names from the USERS table:

```
select distinct city as cityname, reverse(cityname)
from users order by city limit 5;

cityname | reverse
---------+----------
Aberdeen | needrebA
Abilene  | enelibA
Ada      | adA
Agat     | tagA
Agawam   | mawagA
(5 rows)
```

Select five sales IDs and their corresponding reversed IDs cast as character strings:

```
select salesid, reverse(salesid)::varchar
from sales order by salesid desc limit 5;

salesid | reverse
--------+---------
 172456 | 654271
 172455 | 554271
 172454 | 454271
 172453 | 354271
 172452 | 254271
(5 rows)
```

RTRIM function

The RTRIM function trims a specified set of characters from the end of a string. Removes the longest string containing only characters in the trim characters list. Trimming is complete when a trim character does not appear in the input string.

Syntax

```
RTRIM( string, trim_chars )
```

Arguments

string

A string column, expression, or string literal to be trimmed.

trim_chars

A string column, expression, or string literal that represents the characters to be trimmed from the end of *string*. If not specified, a space is used as the trim character.

Return type

A string that is the same data type as the *string* argument.

Example

The following example trims leading and trailing blanks from the string `' abc '`:

```
select '     abc    ' as untrim, rtrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    |    abc
```

The following example removes the trailing `'xyz'` strings from the string `'xyzaxyzbxyzcxyz'`. The trailing occurrences of `'xyz'` are removed, but occurrences that are internal within the string are not removed.

```
select 'xyzaxyzbxyzcxyz' as untrim,
rtrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | xyzaxyzbxyzc
```

The following example removes the trailing parts from the string `'setuphistorycassettes'` that match any of the characters in the *trim_chars* list `'tes'`. Any `t`, `e`, or `s` that occur before another character that is not in the *trim_chars* list at the ending of the input string are removed.

```
SELECT rtrim('setuphistorycassettes', 'tes');

     rtrim
-----------------
 setuphistoryca
```

The following example trims the characters 'Park' from the end of VENUENAME where present:

```
select venueid, venuename, rtrim(venuename, 'Park')
from venue
order by 1, 2, 3
limit 10;

venueid |         venuename          |          rtrim
--------+----------------------------+-------------------------
      1 | Toyota Park                | Toyota
      2 | Columbus Crew Stadium      | Columbus Crew Stadium
      3 | RFK Stadium                | RFK Stadium
      4 | CommunityAmerica Ballpark  | CommunityAmerica Ballp
      5 | Gillette Stadium           | Gillette Stadium
      6 | New York Giants Stadium    | New York Giants Stadium
      7 | BMO Field                  | BMO Field
      8 | The Home Depot Center      | The Home Depot Cente
      9 | Dick's Sporting Goods Park | Dick's Sporting Goods
     10 | Pizza Hut Park             | Pizza Hut
```

Note that RTRIM removes any of the characters `P`, `a`, `r`, or `k` when they appear at the end of a VENUENAME.

SOUNDEX function

The SOUNDEX function returns the American Soundex value consisting of the first letter followed by a 3–digit encoding of the sounds that represent the English pronunciation of the string that you specify.

Syntax

```
SOUNDEX(string)
```

Arguments

string

You specify a CHAR or VARCHAR string that you want to convert to an American Soundex code value.

Return type

The SOUNDEX function returns a VARCHAR(4) string consisting of a capital letter followed by a three–digit encoding of the sounds that represent the English pronunciation.

Usage notes

The SOUNDEX function converts only English alphabetical lowercase and uppercase ASCII characters, including a–z and A–Z. SOUNDEX ignores other characters. SOUNDEX returns a single Soundex value for a string of multiple words separated by spaces.

```
select soundex('AWS Amazon');
```

```
 soundex
---------
 A252
```

SOUNDEX returns an empty string if the input string doesn't contain any English letters.

```
select soundex('+-*/%');
```

```
  soundex
---------

```

Example

The following example returns the Soundex A525 for the word Amazon.

```
select soundex('Amazon');
```

```
 soundex
---------
 A525
```

SPLIT_PART function

Splits a string on the specified delimiter and returns the part at the specified position.

Syntax

```
SPLIT_PART(string, delimiter, position)
```

Arguments

string

A string column, expression, or string literal to be split. The string can be CHAR or VARCHAR.

delimiter

The delimiter string indicating sections of the input *string*.

If *delimiter* is a literal, enclose it in single quotation marks.

position

Position of the portion of *string* to return (counting from 1). Must be an integer greater than 0. If *position* is larger than the number of string portions, SPLIT_PART returns an empty string. If *delimiter* is not found in *string*, then the returned value contains the contents of the specified part, which might be the entire *string* or an empty value.

Return type

A CHAR or VARCHAR string, the same as the *string* parameter.

Examples

The following example splits a string literal into parts using the $ delimiter and returns the second part.

```
select split_part('abc$def$ghi','$',2)

split_part
----------
def
```

The following example splits a string literal into parts using the $ delimiter. It returns an empty string because part 4 is not found.

```
select split_part('abc$def$ghi','$',4)

split_part
----------


```

The following example splits a string literal into parts using the # delimiter. It returns the entire string, which is the first part, because the delimiter is not found.

```
select split_part('abc$def$ghi','#',1)

split_part
------------
abc$def$ghi
```

The following example splits the timestamp field LISTTIME into year, month, and day components.

```
select listtime, split_part(listtime,'-',1) as year,
split_part(listtime,'-',2) as month,
split_part(split_part(listtime,'-',3),' ',1) as day
from listing limit 5;

      listtime       | year | month | day
---------------------+------+-------+------
 2008-03-05 12:25:29 | 2008 | 03    | 05
 2008-09-09 08:03:36 | 2008 | 09    | 09
 2008-09-26 05:43:12 | 2008 | 09    | 26
 2008-10-04 02:00:30 | 2008 | 10    | 04
 2008-01-06 08:33:11 | 2008 | 01    | 06
```

The following example selects the LISTTIME timestamp field and splits it on the '-' character to get the month (the second part of the LISTTIME string), then counts the number of entries for each month:

```
select split_part(listtime,'-',2) as month, count(*)
from listing
group by split_part(listtime,'-',2)
order by 1, 2;

 month | count
-------+-------
    01 | 18543
    02 | 16620
    03 | 17594
    04 | 16822
    05 | 17618
    06 | 17158
    07 | 17626
    08 | 17881
    09 | 17378
    10 | 17756
    11 | 12912
    12 | 4589
```

STRPOS function

Returns the position of a substring within a specified string.

See CHARINDEX function and POSITION function for similar functions.

Syntax

```
STRPOS(string, substring )
```

Arguments

string

The first input parameter is the string to be searched.

substring

The second parameter is the substring to search for within the *string*.

Return type

The STRPOS function returns an integer corresponding to the position of the substring (one-based, not zero-based). The position is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters.

Usage notes

STRPOS returns 0 if the *substring* is not found within the *string*:

```
select strpos('dogfish', 'fist');
strpos
--------
0
(1 row)
```

Examples

The following example shows the position of the string `fish` within the word `dogfish`:

```
select strpos('dogfish', 'fish');
strpos
--------
4
(1 row)
```

The following example returns the number of sales transactions with a COMMISSION over 999.00 from the SALES table:

```
select distinct strpos(commission, '.'),
count (strpos(commission, '.'))
from sales
where strpos(commission, '.') > 4
group by strpos(commission, '.')
order by 1, 2;

strpos | count
-------+-------
 5     |    629
(1 row)
```

SUBSTR function

Synonym of the SUBSTRING function.

See SUBSTRING function.

SUBSTRING function

Returns the subset of a string based on the specified start position.

If the input is a character string, the start position and number of characters extracted are based on characters, not bytes, so that multi-byte characters are counted as single characters. If the input is a binary expression, the start position and extracted substring are based on bytes. You can't specify a negative length, but you can specify a negative starting position.

Syntax

```
SUBSTRING(character_string FROM start_position [ FOR number_characters ] )
```

```
SUBSTRING(character_string, start_position, number_characters )
```

```
SUBSTRING(binary_expression, start_byte, number_bytes )
```

```
SUBSTRING(binary_expression, start_byte )
```

Arguments

character_string

The string to be searched. Non-character data types are treated like a string.

start_position

The position within the string to begin the extraction, starting at 1. The *start_position* is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. This number can be negative.

number_characters

The number of characters to extract (the length of the substring). The *number_characters* is based on the number of characters, not bytes, so that multi-byte characters are counted as single characters. This number cannot be negative.

start_byte

The position within the binary expression to begin the extraction, starting at 1. This number can be negative.

number_bytes

The number of bytes to extract, that is, the length of the substring. This number can't be negative.

Return type

VARCHAR

Usage notes for character strings

The following example returns a four-character string beginning with the sixth character.

```
select substring('caterpillar',6,4);
substring
-----------
pill
(1 row)
```

If the *start_position* + *number_characters* exceeds the length of the *string*, SUBSTRING returns a substring starting from the *start_position* until the end of the string. For example:

```
select substring('caterpillar',6,8);
substring
-----------
pillar
(1 row)
```

If the `start_position` is negative or 0, the SUBSTRING function returns a substring beginning at the first character of string with a length of `start_position` + `number_characters` -1. For example:

```
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)
```

If `start_position` + `number_characters` -1 is less than or equal to zero, SUBSTRING returns an empty string. For example:

```
select substring('caterpillar',-5,4);
substring
-----------

(1 row)
```

Examples

The following example returns the month from the LISTTIME string in the LISTING table:

```
select listid, listtime,
substring(listtime, 6, 2) as month
from listing
order by 1, 2, 3
limit 10;

 listid |      listtime       | month
--------+---------------------+-------
      1 | 2008-01-24 06:43:29 | 01
      2 | 2008-03-05 12:25:29 | 03
      3 | 2008-11-01 07:35:33 | 11
      4 | 2008-05-24 01:18:37 | 05
      5 | 2008-05-17 02:29:11 | 05
      6 | 2008-08-15 02:08:13 | 08
      7 | 2008-11-15 09:38:15 | 11
      8 | 2008-11-09 05:07:30 | 11
      9 | 2008-09-09 08:03:36 | 09
     10 | 2008-06-17 09:44:54 | 06
(10 rows)
```

The following example is the same as above, but uses the FROM...FOR option:

```
select listid, listtime,
```

```
substring(listtime from 6 for 2) as month
from listing
order by 1, 2, 3
limit 10;

 listid |      listtime       | month
--------+---------------------+-------
      1 | 2008-01-24 06:43:29 | 01
      2 | 2008-03-05 12:25:29 | 03
      3 | 2008-11-01 07:35:33 | 11
      4 | 2008-05-24 01:18:37 | 05
      5 | 2008-05-17 02:29:11 | 05
      6 | 2008-08-15 02:08:13 | 08
      7 | 2008-11-15 09:38:15 | 11
      8 | 2008-11-09 05:07:30 | 11
      9 | 2008-09-09 08:03:36 | 09
     10 | 2008-06-17 09:44:54 | 06
(10 rows)
```

You can't use SUBSTRING to predictably extract the prefix of a string that might contain multi-byte characters because you need to specify the length of a multi-byte string based on the number of bytes, not the number of characters. To extract the beginning segment of a string based on the length in bytes, you can CAST the string as VARCHAR(*byte_length*) to truncate the string, where *byte_length* is the required length. The following example extracts the first 5 bytes from the string `'Fourscore and seven'`.

```
select cast('Fourscore and seven' as varchar(5));

varchar
-------
Fours
```

The following example returns the first name `Ana` which appears after the last space in the input string `Silva, Ana`.

```
select reverse(substring(reverse('Silva, Ana'), 1, position(' ' IN reverse('Silva,
 Ana'))))

 reverse
-----------
 Ana
```

TEXTLEN function

Synonym of LEN function.

See LEN function.

TRANSLATE function

For a given expression, replaces all occurrences of specified characters with specified substitutes. Existing characters are mapped to replacement characters by their positions in the *characters_to_replace* and *characters_to_substitute* arguments. If more characters are specified in the *characters_to_replace* argument than in the *characters_to_substitute* argument, the extra characters from the *characters_to_replace* argument are omitted in the return value.

TRANSLATE is similar to the REPLACE function and the REGEXP_REPLACE function, except that REPLACE substitutes one entire string with another string and REGEXP_REPLACE lets you search a string for a regular expression pattern, while TRANSLATE makes multiple single-character substitutions.

If any argument is null, the return is NULL.

Syntax

```
TRANSLATE ( expression, characters_to_replace, characters_to_substitute )
```

Arguments

expression

The expression to be translated.

characters_to_replace

A string containing the characters to be replaced.

characters_to_substitute

A string containing the characters to substitute.

Return type

VARCHAR

Examples

The following example replaces several characters in a string:

```
select translate('mint tea', 'inea', 'osin');

translate
-----------
most tin
```

The following example replaces the at sign (@) with a period for all values in a column:

```
select email, translate(email, '@', '.') as obfuscated_email
from users limit 10;

email                                          obfuscated_email
-------------------------------------------------------------------------------------------
Etiam.laoreet.libero@sodalesMaurisblandit.edu
 Etiam.laoreet.libero.sodalesMaurisblandit.edu
amet.faucibus.ut@condimentumegetvolutpat.ca
 amet.faucibus.ut.condimentumegetvolutpat.ca
turpis@accumsanlaoreet.org                 turpis.accumsanlaoreet.org
ullamcorper.nisl@Cras.edu                 ullamcorper.nisl.Cras.edu
arcu.Curabitur@senectusetnetus.com              arcu.Curabitur.senectusetnetus.com
ac@velit.ca                                  ac.velit.ca
Aliquam.vulputate.ullamcorper@amalesuada.org
 Aliquam.vulputate.ullamcorper.amalesuada.org
vel.est@velitegestas.edu                        vel.est.velitegestas.edu
dolor.nonummy@ipsumdolorsit.ca                  dolor.nonummy.ipsumdolorsit.ca
et@Nunclaoreet.ca                               et.Nunclaoreet.ca
```

The following example replaces spaces with underscores and strips out periods for all values in a column:

```
select city, translate(city, ' .', '_') from users
where city like 'Sain%' or city like 'St%'
group by city
order by city;

city            translate
--------------+------------------
Saint Albans     Saint_Albans
```

```
Saint Cloud      Saint_Cloud
Saint Joseph     Saint_Joseph
Saint Louis      Saint_Louis
Saint Paul       Saint_Paul
St. George       St_George
St. Marys        St_Marys
St. Petersburg   St_Petersburg
Stafford         Stafford
Stamford         Stamford
Stanton          Stanton
Starkville       Starkville
Statesboro       Statesboro
Staunton         Staunton
Steubenville     Steubenville
Stevens Point    Stevens_Point
Stillwater       Stillwater
Stockton         Stockton
Sturgis          Sturgis
```

TRIM function

Trims a string by removing leading and trailing blanks or by removing leading and trailing characters that match an optional specified string.

Syntax

```
TRIM( [ BOTH ] [ trim_chars FROM ] string
```

Arguments

trim_chars

(Optional) The characters to be trimmed from the string. If this parameter is omitted, blanks are trimmed.

string

The string to be trimmed.

Return type

The TRIM function returns a VARCHAR or CHAR string. If you use the TRIM function with a SQL command, AWS Clean Rooms implicitly converts the results to VARCHAR. If you use the TRIM function in the SELECT list for a SQL function, AWS Clean Rooms does not implicitly convert the results, and you might need to perform an explicit conversion to avoid a data type mismatch error. See the CAST function and CONVERT function functions for information about explicit conversions.

Example

The following example trims leading and trailing blanks from the string `' abc '`:

```
select '     abc    ' as untrim, trim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

The following example removes the double quotation marks that surround the string "dog":

```
select trim('"' FROM '"dog"');

btrim
-------
dog
```

TRIM removes any of the characters in *trim_chars* when they appear at the beginning of *string*. The following example trims the characters 'C', 'D', and 'G' when they appear at the beginning of VENUENAME, which is a VARCHAR column.

```
select venueid, venuename, trim(venuename, 'CDG')
from venue
where venuename like '%Park'
order by 2
limit 7;

venueid | venuename                  | btrim
--------+----------------------------+--------------------------
    121 | ATT Park                   | ATT Park
    109 | Citizens Bank Park         | itizens Bank Park
```

```
    102 | Comerica Park              | omerica Park
      9 | Dick's Sporting Goods Park | ick's Sporting Goods Park
     97 | Fenway Park                | Fenway Park
    112 | Great American Ball Park   | reat American Ball Park
    114 | Miller Park                | Miller Park
```

UPPER function

Converts a string to uppercase. UPPER supports UTF-8 multibyte characters, up to a maximum of four bytes per character.

Syntax

```
UPPER(string)
```

Arguments

string

The input parameter is a VARCHAR string (or any other data type, such as CHAR, that can be implicitly converted to VARCHAR).

Return type

The UPPER function returns a character string that is the same data type as the input string.

Examples

The following example converts the CATNAME field to uppercase:

```
select catname, upper(catname) from category order by 1,2;

 catname  |   upper
----------+-----------
Classical | CLASSICAL
Jazz      | JAZZ
MLB       | MLB
MLS       | MLS
Musicals  | MUSICALS
```

```
NBA       | NBA
NFL       | NFL
NHL       | NHL
Opera     | OPERA
Plays     | PLAYS
Pop       | POP
(11 rows)
```

SUPER type information functions

This section describes the information functions for SQL to derive the dynamic information from inputs of the SUPER data type supported in AWS Clean Rooms.

Topics

DECIMAL_PRECISION function

Checks the precision of the maximum total number of decimal digits to be stored. This number includes both the left and right digits of the decimal point. The range of the precision is from 1 to 38, with a default of 38.

Syntax

```
DECIMAL_PRECISION(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

INTEGER

Example

To apply the DECIMAL_PRECISION function to the table t, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (3.14159);

SELECT DECIMAL_PRECISION(s) FROM t;

+-------------------+
| decimal_precision |
+-------------------+
|                 6 |
+-------------------+
```

DECIMAL_SCALE function

Checks the number of decimal digits to be stored to the right of the decimal point. The range of the scale is from 0 to the precision point, with a default of 0.

Syntax

```
DECIMAL_SCALE(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

INTEGER

Example

To apply the DECIMAL_SCALE function to the table t, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (3.14159);

SELECT DECIMAL_SCALE(s) FROM t;

+---------------+
| decimal_scale |
+---------------+
|             5 |
+---------------+
```

IS_ARRAY function

Checks whether a variable is an array. The function returns `true` if the variable is an array. The function also includes empty arrays. Otherwise, the function returns `false` for all other values, including null.

Syntax

```
IS_ARRAY(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `[1,2]` is an array using the IS_ARRAY function, use the following example.

```
SELECT IS_ARRAY(JSON_PARSE('[1,2]'));

+----------+
| is_array |
+----------+
| true     |
+----------+
```

IS_BIGINT function

Checks whether a value is a `BIGINT`. The IS_BIGINT function returns `true` for numbers of scale 0 in the 64-bit range. Otherwise, the function returns `false` for all other values, including null and floating point numbers.

The IS_BIGINT function is a superset of IS_INTEGER.

Syntax

```
IS_BIGINT(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if 5 is a `BIGINT` using the IS_BIGINT function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (5);

SELECT s, IS_BIGINT(s) FROM t;

+---+-----------+
| s | is_bigint |
+---+-----------+
| 5 | true      |
+---+-----------+
```

IS_CHAR function

Checks whether a value is a CHAR. The IS_CHAR function returns `true` for strings that have only ASCII characters, because the CHAR type can store only characters that are in the ASCII format. The function returns `false` for any other values.

Syntax

```
IS_CHAR(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `t` is a CHAR using the IS_CHAR function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES ('t');
```

```
SELECT s, IS_CHAR(s) FROM t;

+-----+---------+
|  s  | is_char |
+-----+---------+
| "t" | true    |
+-----+---------+
```

IS_DECIMAL function

Checks whether a value is a DECIMAL. The IS_DECIMAL function returns `true` for numbers that are not floating points. The function returns `false` for any other values, including null.

The IS_DECIMAL function is a superset of IS_BIGINT.

Syntax

```
IS_DECIMAL(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `1.22` is a DECIMAL using the IS_DECIMAL function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (1.22);

SELECT s, IS_DECIMAL(s) FROM t;
```

```
+------+------------+
|  s   | is_decimal |
+------+------------+
| 1.22 | true       |
+------+------------+
```

IS_FLOAT function

Checks whether a value is a floating point number. The IS_FLOAT function returns `true` for floating point numbers (FLOAT4 and FLOAT8). The function returns `false` for any other values.

The set of IS_DECIMAL the set of IS_FLOAT are disjoint.

Syntax

```
IS_FLOAT(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `2.22::`FLOAT is a FLOAT using the IS_FLOAT function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES(2.22::FLOAT);

SELECT s, IS_FLOAT(s) FROM t;

+---------+----------+
|    s    | is_float |
+---------+----------+
```

```
| 2.22e+0 | true     |
+---------+----------+
```

IS_INTEGER function

Returns `true` for numbers of scale 0 in the 32-bit range, and `false` for anything else (including null and floating point numbers).

The IS_INTEGER function is a superset of the IS_SMALLINT function.

Syntax

```
IS_INTEGER(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if 5 is an `INTEGER` using the IS_INTEGER function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (5);

SELECT s, IS_INTEGER(s) FROM t;

+---+------------+
| s | is_integer |
+---+------------+
| 5 | true       |
+---+------------+
```

IS_OBJECT function

Checks whether a variable is an object. The IS_OBJECT function returns `true` for objects, including empty objects. The function returns `false` for any other values, including null.

Syntax

```
IS_OBJECT(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `{"name": "Joe"}` is an object using the IS_OBJECT function, use the following example.

```
CREATE TABLE t(s super);

INSERT INTO t VALUES (JSON_PARSE('{"name": "Joe"}'));

SELECT s, IS_OBJECT(s) FROM t;

+----------------+-----------+
|       s        | is_object |
+----------------+-----------+
| {"name":"Joe"} | true      |
+----------------+-----------+
```

IS_SCALAR function

Checks whether a variable is a scalar. The IS_SCALAR function returns `true` for any value that is not an array or an object. The function returns `false` for any other values, including null.

The set of IS_ARRAY, IS_OBJECT, and IS_SCALAR cover all values except nulls.

Syntax

```
IS_SCALAR(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if `{"name": "Joe"}` is a scalar using the IS_SCALAR function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (JSON_PARSE('{"name": "Joe"}'));

SELECT s, IS_SCALAR(s.name) FROM t;

+----------------+-----------+
|       s        | is_scalar |
+----------------+-----------+
| {"name":"Joe"} | true      |
+----------------+-----------+
```

IS_SMALLINT function

Checks whether a variable is a SMALLINT. The IS_SMALLINT function returns `true` for numbers of scale 0 in the 16-bit range. The function returns `false` for any other values, including null and floating point numbers.

Syntax

```
IS_SMALLINT(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return

BOOLEAN

Example

To check if 5 is a SMALLINT using the IS_SMALLINT function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES (5);

SELECT s, IS_SMALLINT(s) FROM t;

+---+-------------+
| s | is_smallint |
+---+-------------+
| 5 | true        |
+---+-------------+
```

IS_VARCHAR function

Checks whether a variable is a VARCHAR. The IS_VARCHAR function returns `true` for all strings. The function returns `false` for any other values.

The IS_VARCHAR function is a superset of the IS_CHAR function.

Syntax

```
IS_VARCHAR(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

BOOLEAN

Example

To check if abc is a VARCHAR using the IS_VARCHAR function, use the following example.

```
CREATE TABLE t(s SUPER);

INSERT INTO t VALUES ('abc');

SELECT s, IS_VARCHAR(s) FROM t;

+-------+------------+
|   s   | is_varchar |
+-------+------------+
| "abc" | true       |
+-------+------------+
```

JSON_TYPEOF function

The JSON_TYPEOF scalar function returns a VARCHAR with values boolean, number, string, object, array, or null, depending on the dynamic type of the SUPER value.

Syntax

```
JSON_TYPEOF(super_expression)
```

Arguments

super_expression

A SUPER expression or column.

Return type

VARCHAR

Example

To check the type of JSON for the array `[1,2]` using the JSON_TYPEOF function, use the following example.

```
SELECT JSON_TYPEOF(ARRAY(1,2));

+-------------+
| json_typeof |
+-------------+
| array       |
+-------------+
```

VARBYTE functions

AWS Clean Rooms supports the following VARBYTE functions.

Topics

- FROM_HEX function
- FROM_VARBYTE function
- TO_HEX function
- TO_VARBYTE function

FROM_HEX function

FROM_HEX converts a hexadecimal to a binary value.

Syntax

```
FROM_HEX(hex_string)
```

Arguments

hex_string

Hexadecimal string of data type VARCHAR or TEXT to be converted. The format must be a literal value.

Return type

VARBYTE

Example

To convert the hexadecimal representation of `'6162'` to a binary value, use the following example. The result is automatically shown as the hexadecimal representation of the binary value.

```
SELECT FROM_HEX('6162');

+----------+
| from_hex |
+----------+
|     6162 |
+----------+
```

FROM_VARBYTE function

FROM_VARBYTE converts a binary value to a character string in the specified format.

Syntax

```
FROM_VARBYTE(binary_value, format)
```

Arguments

binary_value

A binary value of data type VARBYTE.

format

The format of the returned character string. Case insensitive valid values are `hex`, `binary`, `utf-8`, and `utf8`.

Return type

VARCHAR

Example

To convert the binary value `'ab'` to hexadecimal, use the following example.

```
SELECT FROM_VARBYTE('ab', 'hex');

+--------------+
| from_varbyte |
+--------------+
|         6162 |
+--------------+
```

TO_HEX function

TO_HEX converts a number or binary value to a hexadecimal representation.

Syntax

```
TO_HEX(value)
```

Arguments

value

Either a number or binary value (VARBYTE) to be converted.

Return type

VARCHAR

Example

To convert a number to its hexadecimal representation, use the following example.

```
SELECT TO_HEX(2147676847);
```

```
+----------+
|  to_hex  |
+----------+
| 8002f2af |
+----------+To create a table, insert the VARBYTE representation of 'abc' to a
 hexadecimal number, and select the column with the value, use the following example.
```

TO_VARBYTE function

TO_VARBYTE converts a string in a specified format to a binary value.

Syntax

```
TO_VARBYTE(string, format)
```

Arguments

string

A CHAR or VARCHAR string.

format

The format of the input string. Case insensitive valid values are `hex`, `binary`, `utf-8`, and `utf8`.

Return type

VARBYTE

Example

To convert the hex 6162 to a binary value, use the following example. The result is automatically shown as the hexadecimal representation of the binary value.

```
SELECT TO_VARBYTE('6162', 'hex');

+------------+
| to_varbyte |
+------------+
```

```
|       6162 |
+------------+
```

Window functions

By using window functions, you can create analytic business queries more efficiently. Window functions operate on a partition or "window" of a result set, and return a value for every row in that window. In contrast, non-windowed functions perform their calculations with respect to every row in the result set. Unlike group functions that aggregate result rows, window functions retain all rows in the table expression.

The values returned are calculated by using values from the sets of rows in that window. For each row in the table, the window defines a set of rows that is used to compute additional attributes. A window is defined using a window specification (the OVER clause), and is based on three main concepts:

- *Window partitioning,* which forms groups of rows (PARTITION clause)
- *Window ordering*, which defines an order or sequence of rows within each partition (ORDER BY clause)
- *Window frames*, which are defined relative to each row to further restrict the set of rows (ROWS specification)

Window functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the window functions are processed. Therefore, window functions can appear only in the select list or ORDER BY clause. You can use multiple window functions within a single query with different frame clauses. You can also use window functions in other scalar expressions, such as CASE.

Window function syntax summary

Window functions follow a standard syntax, which is as follows.

```
function (expression) OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list [ frame_clause ] ] )
```

Here, *function* is one of the functions described in this section.

The *expr_list* is as follows.

```
expression | column_name [, expr_list ]
```

The *order_list* is as follows.

```
expression | column_name [ ASC | DESC ]
[ NULLS FIRST | NULLS LAST ]
[, order_list ]
```

The *frame_clause* is as follows.

```
ROWS
{ UNBOUNDED PRECEDING | unsigned_value PRECEDING | CURRENT ROW } |

{ BETWEEN
{ UNBOUNDED PRECEDING | unsigned_value { PRECEDING | FOLLOWING } | CURRENT ROW}
AND
{ UNBOUNDED FOLLOWING | unsigned_value { PRECEDING | FOLLOWING } | CURRENT ROW }}
```

Arguments

function

The window function. For details, see the individual function descriptions.

OVER

The clause that defines the window specification. The OVER clause is mandatory for window functions, and differentiates window functions from other SQL functions.

PARTITION BY *expr_list*

(Optional) The PARTITION BY clause subdivides the result set into partitions, much like the GROUP BY clause. If a partition clause is present, the function is calculated for the rows in each partition. If no partition clause is specified, a single partition contains the entire table, and the function is computed for that complete table.

The ranking functions DENSE_RANK, NTILE, RANK, and ROW_NUMBER require a global comparison of all the rows in the result set. When a PARTITION BY clause is used, the query optimizer can run each aggregation in parallel by spreading the workload across multiple

slices according to the partitions. If the PARTITION BY clause is not present, the aggregation step must be run serially on a single slice, which can have a significant negative impact on performance, especially for large clusters.

AWS Clean Rooms doesn't support string literals in PARTITION BY clauses.

ORDER BY *order_list*

(Optional) The window function is applied to the rows within each partition sorted according to the order specification in ORDER BY. This ORDER BY clause is distinct from and completely unrelated to ORDER BY clauses in the *frame_clause*. The ORDER BY clause can be used without the PARTITION BY clause.

For ranking functions, the ORDER BY clause identifies the measures for the ranking values. For aggregation functions, the partitioned rows must be ordered before the aggregate function is computed for each frame. For more about window function types, see Window functions.

Column identifiers or expressions that evaluate to column identifiers are required in the order list. Neither constants nor constant expressions can be used as substitutes for column names.

NULLS values are treated as their own group, sorted and ranked according to the NULLS FIRST or NULLS LAST option. By default, NULL values are sorted and ranked last in ASC ordering, and sorted and ranked first in DESC ordering.

AWS Clean Rooms doesn't support string literals in ORDER BY clauses.

If the ORDER BY clause is omitted, the order of the rows is nondeterministic.

> **Note**
>
> In any parallel system such as AWS Clean Rooms, when an ORDER BY clause doesn't produce a unique and total ordering of the data, the order of the rows is nondeterministic. That is, if the ORDER BY expression produces duplicate values (a partial ordering), the return order of those rows might vary from one run of AWS Clean Rooms to the next. In turn, window functions might return unexpected or inconsistent results. For more information, see Unique ordering of data for window functions.

column_name

Name of a column to be partitioned by or ordered by.

ASC | DESC

Option that defines the sort order for the expression, as follows:

- ASC: ascending (for example, low to high for numeric values and 'A' to 'Z' for character strings). If no option is specified, data is sorted in ascending order by default.
- DESC: descending (high to low for numeric values; 'Z' to 'A' for strings).

NULLS FIRST | NULLS LAST

Option that specifies whether NULLS should be ordered first, before non-null values, or last, after non-null values. By default, NULLS are sorted and ranked last in ASC ordering, and sorted and ranked first in DESC ordering.

frame_clause

For aggregate functions, the frame clause further refines the set of rows in a function's window when using ORDER BY. It enables you to include or exclude sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers.

The frame clause doesn't apply to ranking functions. Also, the frame clause isn't required when no ORDER BY clause is used in the OVER clause for an aggregate function. If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required.

When no ORDER BY clause is specified, the implied frame is unbounded, equivalent to ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

ROWS

This clause defines the window frame by specifying a physical offset from the current row.

This clause specifies the rows in the current window or partition that the value in the current row is to be combined with. It uses arguments that specify row position, which can be before or after the current row. The reference point for all window frames is the current row. Each row becomes the current row in turn as the window frame slides forward in the partition.

The frame can be a simple set of rows up to and including the current row.

```
{UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW}
```

Or it can be a set of rows between two boundaries.

```
BETWEEN
{ UNBOUNDED PRECEDING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
AND
{ UNBOUNDED FOLLOWING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
```

UNBOUNDED PRECEDING indicates that the window starts at the first row of the partition; *offset* PRECEDING indicates that the window starts a number of rows equivalent to the value of offset before the current row. UNBOUNDED PRECEDING is the default.

CURRENT ROW indicates the window begins or ends at the current row.

UNBOUNDED FOLLOWING indicates that the window ends at the last row of the partition; *offset* FOLLOWING indicates that the window ends a number of rows equivalent to the value of offset after the current row.

offset identifies a physical number of rows before or after the current row. In this case, *offset* must be a constant that evaluates to a positive numeric value. For example, 5 FOLLOWING ends the frame five rows after the current row.

Where BETWEEN is not specified, the frame is implicitly bounded by the current row. For example, `ROWS 5 PRECEDING` is equal to `ROWS BETWEEN 5 PRECEDING AND CURRENT ROW`. Also, `ROWS UNBOUNDED FOLLOWING` is equal to `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`.

Note

You can't specify a frame in which the starting boundary is greater than the ending boundary. For example, you can't specify any of the following frames.

```
between 5 following and 5 preceding
between current row and 2 preceding
between 3 following and current row
```

Unique ordering of data for window functions

If an ORDER BY clause for a window function doesn't produce a unique and total ordering of the data, the order of the rows is nondeterministic. If the ORDER BY expression produces duplicate

values (a partial ordering), the return order of those rows can vary in multiple runs. In this case, window functions can also return unexpected or inconsistent results.

For example, the following query returns different results over multiple runs. These different results occur because `order by dateid` doesn't produce a unique ordering of the data for the SUM window function.

```
select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid |   sumpaid
--------+-----------+-------------
1827 |   1730.00 |     1730.00
1827 |    708.00 |     2438.00
1827 |    234.00 |     2672.00
...

select dateid, pricepaid,
sum(pricepaid) over(order by dateid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid |   sumpaid
--------+-----------+-------------
1827 |    234.00 |      234.00
1827 |    472.00 |      706.00
1827 |    347.00 |     1053.00
...
```

In this case, adding a second ORDER BY column to the window function can solve the problem.

```
select dateid, pricepaid,
sum(pricepaid) over(order by dateid, pricepaid rows unbounded preceding) as sumpaid
from sales
group by dateid, pricepaid;

dateid | pricepaid | sumpaid
--------+-----------+---------
1827 |    234.00 |  234.00
1827 |    337.00 |  571.00
1827 |    347.00 |  918.00
```

```
...
```

Supported functions

AWS Clean Rooms supports two types of window functions: aggregate and ranking.

Following are the supported aggregate functions:

- AVG window function
- COUNT window function
- CUME_DIST window function
- DENSE_RANK window function
- FIRST_VALUE window function
- LAG window function
- LAST_VALUE window function
- LEAD window function
- LISTAGG window function
- MAX window function
- MEDIAN window function
- MIN window function
- NTH_VALUE window function
- PERCENTILE_CONT window function
- PERCENTILE_DISC window function
- RATIO_TO_REPORT window function
- STDDEV_SAMP and STDDEV_POP window functions (STDDEV_SAMP and STDDEV are synonyms)
- SUM window function
- VAR_SAMP and VAR_POP window functions (VAR_SAMP and VARIANCE are synonyms)

Following are the supported ranking functions:

- DENSE_RANK window function
- NTILE window function

- PERCENT_RANK window function
- RANK window function
- ROW_NUMBER window function

Sample table for window function examples

You can find specific window function examples with each function description. Some of the examples use a table named WINSALES, which contains 11 rows, as shown in the following table.

SALESID	DATEID	SELLERID	BUYERID	QTY	QTY_SHIPPED
30001	8/2/2003	3	B	10	10
10001	12/24/2003	1	C	10	10
10005	12/24/2003	1	A	30	
40001	1/9/2004	4	A	40	
10006	1/18/2004	1	C	10	
20001	2/12/2004	2	B	20	20
40005	2/12/2004	4	A	10	10
20002	2/16/2004	2	C	20	20
30003	4/18/2004	3	B	15	
30004	4/18/2004	3	B	20	
30007	9/7/2004	3	C	30	

AVG window function

The AVG window function returns the average (arithmetic mean) of the input expression values. The AVG function works with numeric values and ignores NULL values.

Syntax

```
AVG ( [ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list
                        frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression for counting. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the AVG function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

The argument types supported by the AVG function are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

The return types supported by the AVG function are:

- BIGINT for SMALLINT or INTEGER arguments
- NUMERIC for BIGINT arguments
- DOUBLE PRECISION for floating point arguments

Examples

The following example computes a rolling average of quantities sold by date; order the results by date ID and sales ID:

```
select salesid, dateid, sellerid, qty,
avg(qty) over
(order by dateid, salesid rows unbounded preceding) as avg
from winsales
order by 2,1;

salesid |   dateid   | sellerid | qty | avg
---------+------------+----------+-----+-----
30001 | 2003-08-02 |        3 |  10 |  10
10001 | 2003-12-24 |        1 |  10 |  10
10005 | 2003-12-24 |        1 |  30 |  16
40001 | 2004-01-09 |        4 |  40 |  22
10006 | 2004-01-18 |        1 |  10 |  20
20001 | 2004-02-12 |        2 |  20 |  20
40005 | 2004-02-12 |        4 |  10 |  18
20002 | 2004-02-16 |        2 |  20 |  18
30003 | 2004-04-18 |        3 |  15 |  18
30004 | 2004-04-18 |        3 |  20 |  18
30007 | 2004-09-07 |        3 |  30 |  19
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

COUNT window function

The COUNT window function counts the rows defined by the expression.

The COUNT function has two variations. COUNT(*) counts all the rows in the target table whether they include nulls or not. COUNT(expression) computes the number of rows with non-NULL values in a specific column or expression.

Syntax

```
COUNT ( * | [ ALL ] expression) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list
                        frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression for counting. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the COUNT function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

The COUNT function supports all argument data types.

The return type supported by the COUNT function is BIGINT.

Examples

The following example shows the sales ID, quantity, and count of all rows from the beginning of the data window:

```
select salesid, qty,
count(*) over (order by salesid rows unbounded preceding) as count
from winsales
order by salesid;

salesid | qty | count
---------+-----+-----
10001 |  10 |   1
10005 |  30 |   2
10006 |  10 |   3
20001 |  20 |   4
20002 |  20 |   5
30001 |  10 |   6
30003 |  15 |   7
30004 |  20 |   8
30007 |  30 |   9
40001 |  40 |   10
40005 |  10 |   11
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example shows how the sales ID, quantity, and count of non-null rows from the beginning of the data window. (In the WINSALES table, the QTY_SHIPPED column contains some NULLs.)

```
select salesid, qty, qty_shipped,
count(qty_shipped)
over (order by salesid rows unbounded preceding) as count
from winsales
order by salesid;

salesid | qty | qty_shipped | count
---------+-----+-------------+-------
10001 |  10 |          10 |   1
10005 |  30 |             |   1
10006 |  10 |             |   1
20001 |  20 |          20 |   2
20002 |  20 |          20 |   3
30001 |  10 |          10 |   4
30003 |  15 |             |   4
30004 |  20 |             |   4
30007 |  30 |             |   4
40001 |  40 |             |   4
40005 |  10 |          10 |   5
(11 rows)
```

CUME_DIST window function

Calculates the cumulative distribution of a value within a window or partition. Assuming ascending ordering, the cumulative distribution is determined using this formula:

```
count of rows with values <= x / count of rows in the window or partition
```

where *x* equals the value in the current row of the column specified in the ORDER BY clause. The following dataset illustrates use of this formula:

```
Row# Value   Calculation    CUME_DIST
1        2500    (1)/(5)    0.2
2        2600    (2)/(5)    0.4
3        2800    (3)/(5)    0.6
4        2900    (4)/(5)    0.8
5        3100    (5)/(5)    1.0
```

The return value range is >0 to 1, inclusive.

Syntax

```
CUME_DIST ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

Arguments

OVER

A clause that specifies the window partitioning. The OVER clause cannot contain a window frame specification.

PARTITION BY *partition_expression*

Optional. An expression that sets the range of records for each group in the OVER clause.

ORDER BY *order_list*

The expression on which to calculate cumulative distribution. The expression must have either a numeric data type or be implicitly convertible to one. If ORDER BY is omitted, the return value is 1 for all rows.

If ORDER BY doesn't produce a unique ordering, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

FLOAT8

Examples

The following example calculates the cumulative distribution of the quantity for each seller:

```
select sellerid, qty, cume_dist()
over (partition by sellerid order by qty)
from winsales;
```

```
sellerid   qty    cume_dist
-------------------------------------------------
1          10.00    0.33
1          10.64    0.67
1          30.37    1
3          10.04    0.25
3          15.15    0.5
3          20.75    0.75
3          30.55    1
2          20.09    0.5
2          20.12    1
4          10.12    0.5
4          40.23    1
```

For a description of the WINSALES table, see Sample table for window function examples.

DENSE_RANK window function

The DENSE_RANK window function determines the rank of a value in a group of values, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the rankings are reset for each group of rows. Rows with equal values for the ranking criteria receive the same rank. The DENSE_RANK function differs from RANK in one respect: If two or more rows tie, there is no gap in the sequence of ranked values. For example, if two rows are ranked 1, the next rank is 2.

You can have ranking functions with different PARTITION BY and ORDER BY clauses in the same query.

Syntax

```
DENSE_RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

Arguments

( )

The function takes no arguments, but the empty parentheses are required.

OVER

The window clauses for the DENSE_RANK function.

PARTITION BY *expr_list*

Optional. One or more expressions that define the window.

ORDER BY *order_list*

Optional. The expression on which the ranking values are based. If no PARTITION BY is specified, ORDER BY uses the entire table. If ORDER BY is omitted, the return value is 1 for all rows.

If ORDER BY doesn't produce a unique ordering, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

INTEGER

Examples

The following example orders the table by the quantity sold (in descending order), and assign both a dense rank and a regular rank to each row. The results are sorted after the window function results are applied.

```
select salesid, qty,
dense_rank() over(order by qty desc) as d_rnk,
rank() over(order by qty desc) as rnk
from winsales
order by 2,1;

salesid | qty | d_rnk | rnk
---------+-----+-------+-----
10001 |  10 |     5 |   8
10006 |  10 |     5 |   8
30001 |  10 |     5 |   8
40005 |  10 |     5 |   8
30003 |  15 |     4 |   7
20001 |  20 |     3 |   4
20002 |  20 |     3 |   4
```

```
30004 |  20 |     3 |   4
10005 |  30 |     2 |   2
30007 |  30 |     2 |   2
40001 |  40 |     1 |   1
(11 rows)
```

Note the difference in rankings assigned to the same set of rows when the DENSE_RANK and RANK functions are used side by side in the same query. For a description of the WINSALES table, see Sample table for window function examples.

The following example partitions the table by SELLERID and orders each partition by the quantity (in descending order) and assign a dense rank to each row. The results are sorted after the window function results are applied.

```
select salesid, sellerid, qty,
dense_rank() over(partition by sellerid order by qty desc) as d_rnk
from winsales
order by 2,3,1;

salesid | sellerid | qty | d_rnk
---------+----------+-----+-------
10001 |        1 |  10 |     2
10006 |        1 |  10 |     2
10005 |        1 |  30 |     1
20001 |        2 |  20 |     1
20002 |        2 |  20 |     1
30001 |        3 |  10 |     4
30003 |        3 |  15 |     3
30004 |        3 |  20 |     2
30007 |        3 |  30 |     1
40005 |        4 |  10 |     2
40001 |        4 |  40 |     1
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

FIRST_VALUE window function

Given an ordered set of rows, FIRST_VALUE returns the value of the specified expression with respect to the first row in the window frame.

For information about selecting the last row in the frame, see LAST_VALUE window function .

Syntax

```
FIRST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

IGNORE NULLS

When this option is used with FIRST_VALUE, the function returns the first value in the frame that is not NULL (or NULL if all values are NULL).

RESPECT NULLS

Indicates that AWS Clean Rooms should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER

Introduces the window clauses for the function.

PARTITION BY *expr_list*

Defines the window for the function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY clause is specified, ORDER BY sorts the entire table. If you specify an ORDER BY clause, you must also specify a *frame_clause*.

The results of the FIRST_VALUE function depends on the ordering of the data. The results are nondeterministic in the following cases:

- When no ORDER BY clause is specified and a partition contains two different values for an expression
- When the expression evaluates to different values that correspond to the same value in the ORDER BY list.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows in the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Return type

These functions support expressions that use primitive AWS Clean Rooms data types. The return type is the same as the data type of the *expression*.

Examples

The following example returns the seating capacity for each venue in the VENUE table, with the results ordered by capacity (high to low). The FIRST_VALUE function is used to select the name of the venue that corresponds to the first row in the frame: in this case, the row with the highest number of seats. The results are partitioned by state, so when the VENUESTATE value changes, a new first value is selected. The window frame is unbounded so the same first value is selected for each row in each partition.

For California, `Qualcomm Stadium` has the highest number of seats (`70561`), so this name is the first value for all of the rows in the `CA` partition.

```
select venuestate, venueseats, venuename,
first_value(venuename)
over(partition by venuestate
order by venueseats desc
rows between unbounded preceding and unbounded following)
from (select * from venue where venueseats >0)
order by venuestate;

venuestate | venueseats |           venuename            |         first_value
-----------+------------+--------------------------------
+-----------------------------
CA         |      70561 | Qualcomm Stadium               | Qualcomm Stadium
CA         |      69843 | Monster Park                   | Qualcomm Stadium
CA         |      63026 | McAfee Coliseum                | Qualcomm Stadium
CA         |      56000 | Dodger Stadium                 | Qualcomm Stadium
CA         |      45050 | Angel Stadium of Anaheim       | Qualcomm Stadium
```

```
CA         |      42445 | PETCO Park                     | Qualcomm Stadium
CA         |      41503 | AT&T Park                      | Qualcomm Stadium
CA         |      22000 | Shoreline Amphitheatre         | Qualcomm Stadium
CO         |      76125 | INVESCO Field                  | INVESCO Field
CO         |      50445 | Coors Field                    | INVESCO Field
DC         |      41888 | Nationals Park                 | Nationals Park
FL         |      74916 | Dolphin Stadium                | Dolphin Stadium
FL         |      73800 | Jacksonville Municipal Stadium | Dolphin Stadium
FL         |      65647 | Raymond James Stadium          | Dolphin Stadium
FL         |      36048 | Tropicana Field                | Dolphin Stadium
...
```

LAG window function

The LAG window function returns the values for a row at a given offset above (before) the current row in the partition.

Syntax

```
LAG (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

Arguments

value_expr

The target column or expression that the function operates on.

offset

An optional parameter that specifies the number of rows before the current row to return values for. The offset can be a constant integer or an expression that evaluates to an integer. If you do not specify an offset, AWS Clean Rooms uses 1 as the default value. An offset of `0` indicates the current row.

IGNORE NULLS

An optional specification that indicates that AWS Clean Rooms should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.

> **Note**
>
> You can use an NVL or COALESCE expression to replace the null values with another value.

RESPECT NULLS

Indicates that AWS Clean Rooms should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER

Specifies the window partitioning and ordering. The OVER clause cannot contain a window frame specification.

PARTITION BY *window_partition*

An optional argument that sets the range of records for each group in the OVER clause.

ORDER BY *window_ordering*

Sorts the rows within each partition.

The LAG window function supports expressions that use any of the AWS Clean Rooms data types. The return type is the same as the type of the *value_expr*.

Examples

The following example shows the quantity of tickets sold to the buyer with a buyer ID of 3 and the time that buyer 3 bought the tickets. To compare each sale with the previous sale for buyer 3, the query returns the previous quantity sold for each sale. Since there is no purchase before 1/16/2008, the first previous quantity sold value is null:

```
select buyerid, saletime, qtysold,
lag(qtysold,1) over (order by buyerid, saletime) as prev_qtysold
from sales where buyerid = 3 order by buyerid, saletime;

buyerid |      saletime       | qtysold | prev_qtysold
---------+---------------------+---------+--------------
3 | 2008-01-16 01:06:09 |       1 |
3 | 2008-01-28 02:10:01 |       1 |            1
```

```
3 | 2008-03-12 10:39:53 |       1 |            1
3 | 2008-03-13 02:56:07 |       1 |            1
3 | 2008-03-29 08:21:39 |       2 |            1
3 | 2008-04-27 02:39:01 |       1 |            2
3 | 2008-08-16 07:04:37 |       2 |            1
3 | 2008-08-22 11:45:26 |       2 |            2
3 | 2008-09-12 09:11:25 |       1 |            2
3 | 2008-10-01 06:22:37 |       1 |            1
3 | 2008-10-20 01:55:51 |       2 |            1
3 | 2008-10-28 01:30:40 |       1 |            2
(12 rows)
```

LAST_VALUE window function

Given an ordered set of rows, The LAST_VALUE function returns the value of the expression with respect to the last row in the frame.

For information about selecting the first row in the frame, see FIRST_VALUE window function .

Syntax

```
LAST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

IGNORE NULLS

The function returns the last value in the frame that is not NULL (or NULL if all values are NULL).

RESPECT NULLS

Indicates that AWS Clean Rooms should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER

Introduces the window clauses for the function.

PARTITION BY *expr_list*

Defines the window for the function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY clause is specified, ORDER BY sorts the entire table. If you specify an ORDER BY clause, you must also specify a *frame_clause*.

The results depend on the ordering of the data. The results are nondeterministic in the following cases:

- When no ORDER BY clause is specified and a partition contains two different values for an expression
- When the expression evaluates to different values that correspond to the same value in the ORDER BY list.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows in the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Return type

These functions support expressions that use primitive AWS Clean Rooms data types. The return type is the same as the data type of the *expression*.

Examples

The following example returns the seating capacity for each venue in the VENUE table, with the results ordered by capacity (high to low). The LAST_VALUE function is used to select the name of the venue that corresponds to the last row in the frame: in this case, the row with the least number of seats. The results are partitioned by state, so when the VENUESTATE value changes, a new last value is selected. The window frame is unbounded so the same last value is selected for each row in each partition.

For California, Shoreline Amphitheatre is returned for every row in the partition because it has the lowest number of seats (22000).

```
select venuestate, venueseats, venuename,
last_value(venuename)
over(partition by venuestate
order by venueseats desc
rows between unbounded preceding and unbounded following)
from (select * from venue where venueseats >0)
order by venuestate;

venuestate | venueseats |           venuename            |          last_value
-----------+------------+--------------------------------
+-----------------------------
CA         |      70561 | Qualcomm Stadium               | Shoreline Amphitheatre
CA         |      69843 | Monster Park                   | Shoreline Amphitheatre
CA         |      63026 | McAfee Coliseum                | Shoreline Amphitheatre
CA         |      56000 | Dodger Stadium                 | Shoreline Amphitheatre
CA         |      45050 | Angel Stadium of Anaheim       | Shoreline Amphitheatre
CA         |      42445 | PETCO Park                     | Shoreline Amphitheatre
CA         |      41503 | AT&T Park                      | Shoreline Amphitheatre
CA         |      22000 | Shoreline Amphitheatre         | Shoreline Amphitheatre
CO         |      76125 | INVESCO Field                  | Coors Field
CO         |      50445 | Coors Field                    | Coors Field
DC         |      41888 | Nationals Park                 | Nationals Park
FL         |      74916 | Dolphin Stadium                | Tropicana Field
FL         |      73800 | Jacksonville Municipal Stadium | Tropicana Field
FL         |      65647 | Raymond James Stadium          | Tropicana Field
FL         |      36048 | Tropicana Field                | Tropicana Field
...
```

LEAD window function

The LEAD window function returns the values for a row at a given offset below (after) the current row in the partition.

Syntax

```
LEAD (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

Arguments

value_expr

The target column or expression that the function operates on.

offset

An optional parameter that specifies the number of rows below the current row to return values for. The offset can be a constant integer or an expression that evaluates to an integer. If you do not specify an offset, AWS Clean Rooms uses 1 as the default value. An offset of `0` indicates the current row.

IGNORE NULLS

An optional specification that indicates that AWS Clean Rooms should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.

> **Note**
>
> You can use an NVL or COALESCE expression to replace the null values with another value.

RESPECT NULLS

Indicates that AWS Clean Rooms should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER

Specifies the window partitioning and ordering. The OVER clause cannot contain a window frame specification.

PARTITION BY *window_partition*

An optional argument that sets the range of records for each group in the OVER clause.

ORDER BY *window_ordering*

Sorts the rows within each partition.

The LEAD window function supports expressions that use any of the AWS Clean Rooms data types. The return type is the same as the type of the *value_expr*.

Examples

The following example provides the commission for events in the SALES table for which tickets were sold on January 1, 2008 and January 2, 2008 and the commission paid for ticket sales for the subsequent sale.

```
select eventid, commission, saletime,
lead(commission, 1) over (order by saletime) as next_comm
from sales where saletime between '2008-01-01 00:00:00' and '2008-01-02 12:59:59'
order by saletime;

eventid | commission |      saletime       | next_comm
---------+------------+---------------------+-----------
6213 |      52.05 | 2008-01-01 01:00:19 |    106.20
7003 |     106.20 | 2008-01-01 02:30:52 |    103.20
8762 |     103.20 | 2008-01-01 03:50:02 |     70.80
1150 |      70.80 | 2008-01-01 06:06:57 |     50.55
1749 |      50.55 | 2008-01-01 07:05:02 |    125.40
8649 |     125.40 | 2008-01-01 07:26:20 |     35.10
2903 |      35.10 | 2008-01-01 09:41:06 |    259.50
6605 |     259.50 | 2008-01-01 12:50:55 |    628.80
6870 |     628.80 | 2008-01-01 12:59:34 |     74.10
6977 |      74.10 | 2008-01-02 01:11:16 |     13.50
4650 |      13.50 | 2008-01-02 01:40:59 |     26.55
4515 |      26.55 | 2008-01-02 01:52:35 |     22.80
5465 |      22.80 | 2008-01-02 02:28:01 |     45.60
5465 |      45.60 | 2008-01-02 02:28:02 |     53.10
7003 |      53.10 | 2008-01-02 02:31:12 |     70.35
4124 |      70.35 | 2008-01-02 03:12:50 |     36.15
1673 |      36.15 | 2008-01-02 03:15:00 |   1300.80
...
(39 rows)
```

LISTAGG window function

For each group in a query, the LISTAGG window function orders the rows for that group according to the ORDER BY expression, then concatenates the values into a single string.

LISTAGG is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
LISTAGG( [DISTINCT] expression [, 'delimiter' ] )
[ WITHIN GROUP (ORDER BY order_list) ]
OVER ( [PARTITION BY partition_expression] )
```

Arguments

DISTINCT

(Optional) A clause that eliminates duplicate values from the specified expression before concatenating. Trailing spaces are ignored, so the strings `'a'` and `'a '` are treated as duplicates. LISTAGG uses the first value encountered. For more information, see Significance of trailing blanks.

aggregate_expression

Any valid expression (such as a column name) that provides the values to aggregate. NULL values and empty strings are ignored.

delimiter

(Optional) The string constant to separate the concatenated values. The default is NULL.

AWS Clean Rooms supports any amount of leading or trailing whitespace around an optional comma or colon as well as an empty string or any number of spaces.

Examples of valid values are:

`", "`

`": "`

`" "`

WITHIN GROUP (ORDER BY *order_list*)

(Optional) A clause that specifies the sort order of the aggregated values. Deterministic only if ORDER BY provides unique ordering. The default is to aggregate all rows and return a single value.

OVER

A clause that specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.

PARTITION BY *partition_expression*

(Optional) Sets the range of records for each group in the OVER clause.

Returns

VARCHAR(MAX). If the result set is larger than the maximum VARCHAR size (64K – 1, or 65535), then LISTAGG returns the following error:

```
Invalid operation: Result size exceeds LISTAGG limit
```

Examples

The following examples uses the WINSALES table. For a description of the WINSALES table, see Sample table for window function examples.

The following example returns a list of seller IDs, ordered by seller ID.

```
select listagg(sellerid)
within group (order by sellerid)
over() from winsales;

  listagg
------------
 11122333344
...
...
 11122333344
 11122333344
   (11 rows)
```

The following example returns a list of seller IDs for buyer B, ordered by date.

```
select listagg(sellerid)
within group (order by dateid)
over () as seller
from winsales
where buyerid = 'b' ;

  seller
---------
```

```
    3233
    3233
    3233
    3233

(4 rows)
```

The following example returns a comma-separated list of sales dates for buyer B.

```
select listagg(dateid,',')
within group (order by sellerid desc,salesid asc)
over () as dates
from winsales
where buyerid  = 'b';

            dates
------------------------------------------
2003-08-02,2004-04-18,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-04-18,2004-02-12

(4 rows)
```

The following example uses DISTINCT to return a list of unique sales dates for buyer B.

```
select listagg(distinct dateid,',')
within group (order by sellerid desc,salesid asc)
over () as dates
from winsales
where buyerid  = 'b';

           dates
--------------------------------
2003-08-02,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-02-12
2003-08-02,2004-04-18,2004-02-12

(4 rows)
```

The following example returns a comma-separated list of sales IDs for each buyer ID.

```
select buyerid,
listagg(salesid,',')
within group (order by salesid)
over (partition by buyerid) as sales_id
from winsales
order by buyerid;

   buyerid | sales_id
-----------+------------------------
        a  |10005,40001,40005
        a  |10005,40001,40005
        a  |10005,40001,40005
        b  |20001,30001,30004,30003
        b  |20001,30001,30004,30003
        b  |20001,30001,30004,30003
        b  |20001,30001,30004,30003
        c  |10001,20002,30007,10006
        c  |10001,20002,30007,10006
        c  |10001,20002,30007,10006
        c  |10001,20002,30007,10006
(11 rows)
```

MAX window function

The MAX window function returns the maximum of the input expression values. The MAX function works with numeric values and ignores NULL values.

Syntax

```
MAX ( [ ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

A clause that specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the MAX function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

Accepts any data type as input. Returns the same data type as *expression*.

Examples

The following example shows the sales ID, quantity, and maximum quantity from the beginning of the data window:

```
select salesid, qty,
max(qty) over (order by salesid rows unbounded preceding) as max
from winsales
order by salesid;

salesid | qty | max
---------+-----+-----
10001 |  10 |  10
```

```
10005 |  30 |  30
10006 |  10 |  30
20001 |  20 |  30
20002 |  20 |  30
30001 |  10 |  30
30003 |  15 |  30
30004 |  20 |  30
30007 |  30 |  30
40001 |  40 |  40
40005 |  10 |  40
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example shows the salesid, quantity, and maximum quantity in a restricted frame:

```
select salesid, qty,
max(qty) over (order by salesid rows between 2 preceding and 1 preceding) as max
from winsales
order by salesid;

salesid | qty | max
---------+-----+-----
10001 |  10 |
10005 |  30 |  10
10006 |  10 |  30
20001 |  20 |  30
20002 |  20 |  20
30001 |  10 |  20
30003 |  15 |  20
30004 |  20 |  15
30007 |  30 |  20
40001 |  40 |  30
40005 |  10 |  40
(11 rows)
```

MEDIAN window function

Calculates the median value for the range of values in a window or partition. NULL values in the range are ignored.

MEDIAN is an inverse distribution function that assumes a continuous distribution model.

MEDIAN is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
MEDIAN ( median_expression )
OVER ( [ PARTITION BY partition_expression ] )
```

Arguments

median_expression

An expression, such as a column name, that provides the values for which to determine the median. The expression must have either a numeric or datetime data type or be implicitly convertible to one.

OVER

A clause that specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.

PARTITION BY *partition_expression*

Optional. An expression that sets the range of records for each group in the OVER clause.

Data types

The return type is determined by the data type of *median_expression*. The following table shows the return type for each *median_expression* data type.

Input Type	Return Type
NUMERIC, DECIMAL	DECIMAL
FLOAT, DOUBLE	DOUBLE
DATE	DATE

Usage notes

If the *median_expression* argument is a DECIMAL data type defined with the maximum precision of 38 digits, it is possible that MEDIAN will return either an inaccurate result or an error. If the return value of the MEDIAN function exceeds 38 digits, the result is truncated to fit, which causes a loss of precision. If, during interpolation, an intermediate result exceeds the maximum precision, a numeric overflow occurs and the function returns an error. To avoid these conditions, we recommend either using a data type with lower precision or casting the *median_expression* argument to a lower precision.

For example, a SUM function with a DECIMAL argument returns a default precision of 38 digits. The scale of the result is the same as the scale of the argument. So, for example, a SUM of a DECIMAL(5,2) column returns a DECIMAL(38,2) data type.

The following example uses a SUM function in the *median_expression* argument of a MEDIAN function. The data type of the PRICEPAID column is DECIMAL (8,2), so the SUM function returns DECIMAL(38,2).

```
select salesid, sum(pricepaid), median(sum(pricepaid))
over() from sales where salesid < 10 group by salesid;
```

To avoid a potential loss of precision or an overflow error, cast the result to a DECIMAL data type with lower precision, as the following example shows.

```
select salesid, sum(pricepaid), median(sum(pricepaid)::decimal(30,2))
over() from sales where salesid < 10 group by salesid;
```

Examples

The following example calculates the median sales quantity for each seller:

```
select sellerid, qty, median(qty)
over (partition by sellerid)
from winsales
order by sellerid;


sellerid qty median
---------------------------
```

```
1  10 10.0
1  10 10.0
1  30 10.0
2  20 20.0
2  20 20.0
3  10 17.5
3  15 17.5
3  20 17.5
3  30 17.5
4  10 25.0
4  40 25.0
```

For a description of the WINSALES table, see Sample table for window function examples.

MIN window function

The MIN window function returns the minimum of the input expression values. The MIN function works with numeric values and ignores NULL values.

Syntax

```
MIN ( [ ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the MIN function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

Accepts any data type as input. Returns the same data type as *expression*.

Examples

The following example shows the sales ID, quantity, and minimum quantity from the beginning of the data window:

```
select salesid, qty,
min(qty) over
(order by salesid rows unbounded preceding)
from winsales
order by salesid;

salesid | qty | min
---------+-----+-----
10001 |  10 |  10
10005 |  30 |  10
10006 |  10 |  10
20001 |  20 |  10
20002 |  20 |  10
30001 |  10 |  10
30003 |  15 |  10
30004 |  20 |  10
30007 |  30 |  10
40001 |  40 |  10
```

```
40005 |  10 |  10
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example shows the sales ID, quantity, and minimum quantity in a restricted frame:

```
select salesid, qty,
min(qty) over
(order by salesid rows between 2 preceding and 1 preceding) as min
from winsales
order by salesid;

salesid | qty | min
---------+-----+-----
10001 |  10 |
10005 |  30 |  10
10006 |  10 |  10
20001 |  20 |  10
20002 |  20 |  10
30001 |  10 |  20
30003 |  15 |  10
30004 |  20 |  10
30007 |  30 |  15
40001 |  40 |  20
40005 |  10 |  30
(11 rows)
```

NTH_VALUE window function

The NTH_VALUE window function returns the expression value of the specified row of the window frame relative to the first row of the window.

Syntax

```
NTH_VALUE (expr, offset)
[ IGNORE NULLS | RESPECT NULLS ]
OVER
( [ PARTITION BY window_partition ]
[ ORDER BY window_ordering
                        frame_clause ] )
```

Arguments

expr

The target column or expression that the function operates on.

offset

Determines the row number relative to the first row in the window for which to return the expression. The *offset* can be a constant or an expression and must be a positive integer that is greater than 0.

IGNORE NULLS

An optional specification that indicates that AWS Clean Rooms should skip null values in the determination of which row to use. Null values are included if IGNORE NULLS is not listed.

RESPECT NULLS

Indicates that AWS Clean Rooms should include null values in the determination of which row to use. RESPECT NULLS is supported by default if you do not specify IGNORE NULLS.

OVER

Specifies the window partitioning, ordering, and window frame.

PARTITION BY *window_partition*

Sets the range of records for each group in the OVER clause.

ORDER BY *window_ordering*

Sorts the rows within each partition. If ORDER BY is omitted, the default frame consists of all rows in the partition.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows in the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

The NTH_VALUE window function supports expressions that use any of the AWS Clean Rooms data types. The return type is the same as the type of the *expr*.

Examples

The following example shows the number of seats in the third largest venue in California, Florida, and New York compared to the number of seats in the other venues in those states:

```
select venuestate, venuename, venueseats,
nth_value(venueseats, 3)
ignore nulls
over(partition by venuestate order by venueseats desc
rows between unbounded preceding and unbounded following)
as third_most_seats
from (select * from venue where venueseats > 0 and
venuestate in('CA', 'FL', 'NY'))
order by venuestate;

venuestate |           venuename            | venueseats | third_most_seats
-----------+--------------------------------+------------+------------------
CA         | Qualcomm Stadium               |      70561 |            63026
CA         | Monster Park                   |      69843 |            63026
CA         | McAfee Coliseum                |      63026 |            63026
CA         | Dodger Stadium                 |      56000 |            63026
CA         | Angel Stadium of Anaheim       |      45050 |            63026
CA         | PETCO Park                     |      42445 |            63026
CA         | AT&T Park                      |      41503 |            63026
CA         | Shoreline Amphitheatre         |      22000 |            63026
FL         | Dolphin Stadium                |      74916 |            65647
FL         | Jacksonville Municipal Stadium |      73800 |            65647
FL         | Raymond James Stadium          |      65647 |            65647
FL         | Tropicana Field                |      36048 |            65647
NY         | Ralph Wilson Stadium           |      73967 |            20000
NY         | Yankee Stadium                 |      52325 |            20000
NY         | Madison Square Garden          |      20000 |            20000
(15 rows)
```

NTILE window function

The NTILE window function divides ordered rows in the partition into the specified number of ranked groups of as equal size as possible and returns the group that a given row falls into.

Syntax

```
NTILE (expr)
```

```
OVER (
[ PARTITION BY expression_list ]
[ ORDER BY order_list ]
)
```

Arguments

expr

The number of ranking groups and must result in a positive integer value (greater than 0) for each partition. The *expr* argument must not be nullable.

OVER

A clause that specifies the window partitioning and ordering. The OVER clause cannot contain a window frame specification.

PARTITION BY *window_partition*

Optional. The range of records for each group in the OVER clause.

ORDER BY *window_ordering*

Optional. An expression that sorts the rows within each partition. If the ORDER BY clause is omitted, the ranking behavior is the same.

If ORDER BY does not produce a unique ordering, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

BIGINT

Examples

The following example ranks into four ranking groups the price paid for Hamlet tickets on August 26, 2008. The result set is 17 rows, divided almost evenly among the rankings 1 through 4:

```
select eventname, caldate, pricepaid, ntile(4)
over(order by pricepaid desc) from sales, event, date
where sales.eventid=event.eventid and event.dateid=date.dateid and eventname='Hamlet'
```

```
and caldate='2008-08-26'
order by 4;

eventname |  caldate   | pricepaid | ntile
----------+------------+-----------+-------
Hamlet    | 2008-08-26 |   1883.00 |     1
Hamlet    | 2008-08-26 |   1065.00 |     1
Hamlet    | 2008-08-26 |    589.00 |     1
Hamlet    | 2008-08-26 |    530.00 |     1
Hamlet    | 2008-08-26 |    472.00 |     1
Hamlet    | 2008-08-26 |    460.00 |     2
Hamlet    | 2008-08-26 |    355.00 |     2
Hamlet    | 2008-08-26 |    334.00 |     2
Hamlet    | 2008-08-26 |    296.00 |     2
Hamlet    | 2008-08-26 |    230.00 |     3
Hamlet    | 2008-08-26 |    216.00 |     3
Hamlet    | 2008-08-26 |    212.00 |     3
Hamlet    | 2008-08-26 |    106.00 |     3
Hamlet    | 2008-08-26 |    100.00 |     4
Hamlet    | 2008-08-26 |     94.00 |     4
Hamlet    | 2008-08-26 |     53.00 |     4
Hamlet    | 2008-08-26 |     25.00 |     4
(17 rows)
```

PERCENT_RANK window function

Calculates the percent rank of a given row. The percent rank is determined using this formula:

`(x - 1) / (the number of rows in the window or partition - 1)`

where *x* is the rank of the current row. The following dataset illustrates use of this formula:

```
Row# Value Rank Calculation PERCENT_RANK
1 15 1 (1-1)/(7-1) 0.0000
2 20 2 (2-1)/(7-1) 0.1666
3 20 2 (2-1)/(7-1) 0.1666
4 20 2 (2-1)/(7-1) 0.1666
5 30 5 (5-1)/(7-1) 0.6666
6 30 5 (5-1)/(7-1) 0.6666
7 40 7 (7-1)/(7-1) 1.0000
```

The return value range is 0 to 1, inclusive. The first row in any set has a PERCENT_RANK of 0.

Syntax

```
PERCENT_RANK ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

Arguments

( )

The function takes no arguments, but the empty parentheses are required.

OVER

A clause that specifies the window partitioning. The OVER clause cannot contain a window frame specification.

PARTITION BY *partition_expression*

Optional. An expression that sets the range of records for each group in the OVER clause.

ORDER BY *order_list*

Optional. The expression on which to calculate percent rank. The expression must have either a numeric data type or be implicitly convertible to one. If ORDER BY is omitted, the return value is 0 for all rows.

If ORDER BY does not produce a unique ordering, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

FLOAT8

Examples

The following example calculates the percent rank of the sales quantities for each seller:

```
select sellerid, qty, percent_rank()
over (partition by sellerid order by qty)
from winsales;
```

```
sellerid qty  percent_rank
---------------------------------------
1  10.00  0.0
1  10.64  0.5
1  30.37  1.0
3  10.04  0.0
3  15.15  0.33
3  20.75  0.67
3  30.55  1.0
2  20.09  0.0
2  20.12  1.0
4  10.12  0.0
4  40.23  1.0
```

For a description of the WINSALES table, see Sample table for window function examples.

PERCENTILE_CONT window function

PERCENTILE_CONT is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into the given percentile value with respect to the sort specification.

PERCENTILE_CONT computes a linear interpolation between values after ordering them. Using the percentile value `(P)` and the number of not null rows `(N)` in the aggregation group, the function computes the row number after ordering the rows according to the sort specification. This row number `(RN)` is computed according to the formula `RN = (1+ (P*(N-1))`. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN = CEILING(RN)` and `FRN = FLOOR(RN)`.

The final result will be as follows.

If `(CRN = FRN = RN)` then the result is `(value of expression from row at RN)`

Otherwise the result is as follows:

`(CRN - RN) * (value of expression for row at FRN) + (RN - FRN) * (value of expression for row at CRN)`.

You can specify only the PARTITION clause in the OVER clause. If PARTITION is specified, for each row, PERCENTILE_CONT returns the value that would fall into the specified percentile among a set of values within a given partition.

PERCENTILE_CONT is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
PERCENTILE_CONT ( percentile )
WITHIN GROUP (ORDER BY expr)
OVER (  [ PARTITION BY expr_list ]  )
```

Arguments

percentile

Numeric constant between 0 and 1. Nulls are ignored in the calculation.

WITHIN GROUP (ORDER BY *expr*)

Specifies numeric or date/time values to sort and compute the percentile over.

OVER

Specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.

PARTITION BY *expr*

Optional argument that sets the range of records for each group in the OVER clause.

Returns

The return type is determined by the data type of the ORDER BY expression in the WITHIN GROUP clause. The following table shows the return type for each ORDER BY expression data type.

Input Type	Return Type
SMALLINTINTEGERBIGINTNUMERIC, DECIMAL	DECIMAL
FLOAT, DOUBLE	DOUBLE
DATE	DATE

Input Type	Return Type
TIMESTAMP	TIMESTAMP

Usage notes

If the ORDER BY expression is a DECIMAL data type defined with the maximum precision of 38 digits, it is possible that PERCENTILE_CONT will return either an inaccurate result or an error. If the return value of the PERCENTILE_CONT function exceeds 38 digits, the result is truncated to fit, which causes a loss of precision. If, during interpolation, an intermediate result exceeds the maximum precision, a numeric overflow occurs and the function returns an error. To avoid these conditions, we recommend either using a data type with lower precision or casting the ORDER BY expression to a lower precision.

For example, a SUM function with a DECIMAL argument returns a default precision of 38 digits. The scale of the result is the same as the scale of the argument. So, for example, a SUM of a DECIMAL(5,2) column returns a DECIMAL(38,2) data type.

The following example uses a SUM function in the ORDER BY clause of a PERCENTILE_CONT function. The data type of the PRICEPAID column is DECIMAL (8,2), so the SUM function returns DECIMAL(38,2).

```
select salesid, sum(pricepaid), percentile_cont(0.6)
within group (order by sum(pricepaid) desc) over()
from sales where salesid < 10 group by salesid;
```

To avoid a potential loss of precision or an overflow error, cast the result to a DECIMAL data type with lower precision, as the following example shows.

```
select salesid, sum(pricepaid), percentile_cont(0.6)
within group (order by sum(pricepaid)::decimal(30,2) desc) over()
from sales where salesid < 10 group by salesid;
```

Examples

The following examples uses the WINSALES table. For a description of the WINSALES table, see Sample table for window function examples.

```
select sellerid, qty, percentile_cont(0.5)
```

```
within group (order by qty)
over() as median from winsales;

 sellerid | qty | median
----------+-----+--------
        1 |  10 |   20.0
        1 |  10 |   20.0
        3 |  10 |   20.0
        4 |  10 |   20.0
        3 |  15 |   20.0
        2 |  20 |   20.0
        3 |  20 |   20.0
        2 |  20 |   20.0
        3 |  30 |   20.0
        1 |  30 |   20.0
        4 |  40 |   20.0
(11 rows)
```

```
select sellerid, qty, percentile_cont(0.5)
within group (order by qty)
over(partition by sellerid) as median from winsales;

 sellerid | qty | median
----------+-----+--------
        2 |  20 |   20.0
        2 |  20 |   20.0
        4 |  10 |   25.0
        4 |  40 |   25.0
        1 |  10 |   10.0
        1 |  10 |   10.0
        1 |  30 |   10.0
        3 |  10 |   17.5
        3 |  15 |   17.5
        3 |  20 |   17.5
        3 |  30 |   17.5
(11 rows)
```

The following example calculates the PERCENTILE_CONT and PERCENTILE_DISC of the ticket sales for sellers in Washington state.

```
SELECT sellerid, state, sum(qtysold*pricepaid) sales,
percentile_cont(0.6) within group (order by sum(qtysold*pricepaid::decimal(14,2) )
 desc) over(),
```

```
percentile_disc(0.6) within group (order by sum(qtysold*pricepaid::decimal(14,2) )
 desc) over()
from sales s, users u
where s.sellerid = u.userid and state = 'WA' and sellerid < 1000
group by sellerid, state;

 sellerid | state |  sales  | percentile_cont | percentile_disc
----------+-------+---------+-----------------+-----------------
      127 | WA    | 6076.00 |         2044.20 |         1531.00
      787 | WA    | 6035.00 |         2044.20 |         1531.00
      381 | WA    | 5881.00 |         2044.20 |         1531.00
      777 | WA    | 2814.00 |         2044.20 |         1531.00
       33 | WA    | 1531.00 |         2044.20 |         1531.00
      800 | WA    | 1476.00 |         2044.20 |         1531.00
        1 | WA    | 1177.00 |         2044.20 |         1531.00
(7 rows)
```

PERCENTILE_DISC window function

PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the given set.

For a given percentile value P, PERCENTILE_DISC sorts the values of the expression in the ORDER BY clause and returns the value with the smallest cumulative distribution value (with respect to the same sort specification) that is greater than or equal to P.

You can specify only the PARTITION clause in the OVER clause.

PERCENTILE_DISC is a compute-node only function. The function returns an error if the query doesn't reference a user-defined table or AWS Clean Rooms system table.

Syntax

```
PERCENTILE_DISC ( percentile )
WITHIN GROUP (ORDER BY expr)
OVER (  [ PARTITION BY expr_list ]  )
```

Arguments

percentile

Numeric constant between 0 and 1. Nulls are ignored in the calculation.

WITHIN GROUP (ORDER BY *expr*)

Specifies numeric or date/time values to sort and compute the percentile over.

OVER

Specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.

PARTITION BY *expr*

Optional argument that sets the range of records for each group in the OVER clause.

Returns

The same data type as the ORDER BY expression in the WITHIN GROUP clause.

Examples

The following examples uses the WINSALES table. For a description of the WINSALES table, see Sample table for window function examples.

```
select sellerid, qty, percentile_disc(0.5)
within group (order by qty)
over() as median from winsales;

sellerid | qty | median
----------+-----+--------
        1 |  10 |     20
        3 |  10 |     20
        1 |  10 |     20
        4 |  10 |     20
        3 |  15 |     20
        2 |  20 |     20
        2 |  20 |     20
        3 |  20 |     20
        1 |  30 |     20
        3 |  30 |     20
        4 |  40 |     20
(11 rows)
```

```
select sellerid, qty, percentile_disc(0.5)
```

```
within group (order by qty)
over(partition by sellerid) as median from winsales;

sellerid | qty | median
----------+-----+--------
        2 |  20 |     20
        2 |  20 |     20
        4 |  10 |     10
        4 |  40 |     10
        1 |  10 |     10
        1 |  10 |     10
        1 |  30 |     10
        3 |  10 |     15
        3 |  15 |     15
        3 |  20 |     15
        3 |  30 |     15
(11 rows)
```

RANK window function

The RANK window function determines the rank of a value in a group of values, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the rankings are reset for each group of rows. Rows with equal values for the ranking criteria receive the same rank. AWS Clean Rooms adds the number of tied rows to the tied rank to calculate the next rank and thus the ranks might not be consecutive numbers. For example, if two rows are ranked 1, the next rank is 3.

RANK differs from the DENSE_RANK window function in one respect: For DENSE_RANK, if two or more rows tie, there is no gap in the sequence of ranked values. For example, if two rows are ranked 1, the next rank is 2.

You can have ranking functions with different PARTITION BY and ORDER BY clauses in the same query.

Syntax

```
RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

Arguments

( )

The function takes no arguments, but the empty parentheses are required.

OVER

The window clauses for the RANK function.

PARTITION BY *expr_list*

Optional. One or more expressions that define the window.

ORDER BY *order_list*

Optional. Defines the columns on which the ranking values are based. If no PARTITION BY is specified, ORDER BY uses the entire table. If ORDER BY is omitted, the return value is 1 for all rows.

If ORDER BY does not produce a unique ordering, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

INTEGER

Examples

The following example orders the table by the quantity sold (default ascending), and assign a rank to each row. A rank value of 1 is the highest ranked value. The results are sorted after the window function results are applied:

```
select salesid, qty,
rank() over (order by qty) as rnk
from winsales
order by 2,1;

salesid | qty | rnk
--------+-----+-----
10001 |  10 |  1
10006 |  10 |  1
```

```
30001 |  10 |  1
40005 |  10 |  1
30003 |  15 |  5
20001 |  20 |  6
20002 |  20 |  6
30004 |  20 |  6
10005 |  30 |  9
30007 |  30 |  9
40001 |  40 |  11
(11 rows)
```

Note that the outer ORDER BY clause in this example includes columns 2 and 1 to make sure that AWS Clean Rooms returns consistently sorted results each time this query is run. For example, rows with sales IDs 10001 and 10006 have identical QTY and RNK values. Ordering the final result set by column 1 ensures that row 10001 always falls before 10006. For a description of the WINSALES table, see Sample table for window function examples.

In the following example, the ordering is reversed for the window function (`order by qty desc`). Now the highest rank value applies to the largest QTY value.

```
select salesid, qty,
rank() over (order by qty desc) as rank
from winsales
order by 2,1;

 salesid | qty | rank
---------+-----+-----
   10001 |  10 |   8
   10006 |  10 |   8
   30001 |  10 |   8
   40005 |  10 |   8
   30003 |  15 |   7
   20001 |  20 |   4
   20002 |  20 |   4
   30004 |  20 |   4
   10005 |  30 |   2
   30007 |  30 |   2
   40001 |  40 |   1
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example partitions the table by SELLERID and order each partition by the quantity (in descending order) and assign a rank to each row. The results are sorted after the window function results are applied.

```
select salesid, sellerid, qty, rank() over
(partition by sellerid
order by qty desc) as rank
from winsales
order by 2,3,1;

salesid | sellerid | qty | rank
--------+----------+-----+-----
  10001 |        1 |  10 |  2
  10006 |        1 |  10 |  2
  10005 |        1 |  30 |  1
  20001 |        2 |  20 |  1
  20002 |        2 |  20 |  1
  30001 |        3 |  10 |  4
  30003 |        3 |  15 |  3
  30004 |        3 |  20 |  2
  30007 |        3 |  30 |  1
  40005 |        4 |  10 |  2
  40001 |        4 |  40 |  1
(11 rows)
```

RATIO_TO_REPORT window function

Calculates the ratio of a value to the sum of the values in a window or partition. The ratio to report value is determined using the formula:

`value of` *ratio_expression* `argument for the current row / sum of` *ratio_expression* `argument for the window or partition`

The following dataset illustrates use of this formula:

```
Row# Value Calculation RATIO_TO_REPORT
1 2500 (2500)/(13900) 0.1798
2 2600 (2600)/(13900) 0.1870
3 2800 (2800)/(13900) 0.2014
4 2900 (2900)/(13900) 0.2086
5 3100 (3100)/(13900) 0.2230
```

The return value range is 0 to 1, inclusive. If *ratio_expression* is NULL, then the return value is NULL.

Syntax

```
RATIO_TO_REPORT ( ratio_expression )
OVER ( [ PARTITION BY partition_expression ] )
```

Arguments

ratio_expression

An expression, such as a column name, that provides the value for which to determine the ratio. The expression must have either a numeric data type or be implicitly convertible to one.

You cannot use any other analytic function in *ratio_expression*.

OVER

A clause that specifies the window partitioning. The OVER clause cannot contain a window ordering or window frame specification.

PARTITION BY *partition_expression*

Optional. An expression that sets the range of records for each group in the OVER clause.

Return type

FLOAT8

Examples

The following example calculates the ratios of the sales quantities for each seller:

```
select sellerid, qty, ratio_to_report(qty)
over (partition by sellerid)
from winsales;

sellerid qty  ratio_to_report
----------------------------------------
2  20.12312341      0.5
2  20.08630000      0.5
4  10.12414400      0.2
4  40.23000000      0.8
```

```
1  30.37262000      0.6
1  10.64000000      0.21
1  10.00000000      0.2
3  10.03500000      0.13
3  15.14660000      0.2
3  30.54790000      0.4
3  20.74630000      0.27
```

For a description of the WINSALES table, see Sample table for window function examples.

ROW_NUMBER window function

Determines the ordinal number of the current row within a group of rows, counting from 1, based on the ORDER BY expression in the OVER clause. If the optional PARTITION BY clause is present, the ordinal numbers are reset for each group of rows. Rows with equal values for the ORDER BY expressions receive the different row numbers nondeterministically.

Syntax

```
ROW_NUMBER () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

Arguments

( )

The function takes no arguments, but the empty parentheses are required.

OVER

The window clauses for the ROW_NUMBER function.

PARTITION BY *expr_list*

Optional. One or more expressions that define the ROW_NUMBER function.

ORDER BY *order_list*

Optional. The expression that defines the columns on which the row numbers are based. If no PARTITION BY is specified, ORDER BY uses the entire table.

If ORDER BY does not produce a unique ordering or is omitted, the order of the rows is nondeterministic. For more information, see Unique ordering of data for window functions.

Return type

BIGINT

Examples

The following example partitions the table by SELLERID and orders each partition by QTY (in ascending order), then assigns a row number to each row. The results are sorted after the window function results are applied.

```
select salesid, sellerid, qty,
row_number() over
(partition by sellerid
 order by qty asc) as row
from winsales
order by 2,4;

 salesid | sellerid | qty | row
---------+----------+-----+-----
   10006 |        1 |  10 |   1
   10001 |        1 |  10 |   2
   10005 |        1 |  30 |   3
   20001 |        2 |  20 |   1
   20002 |        2 |  20 |   2
   30001 |        3 |  10 |   1
   30003 |        3 |  15 |   2
   30004 |        3 |  20 |   3
   30007 |        3 |  30 |   4
   40005 |        4 |  10 |   1
   40001 |        4 |  40 |   2
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

STDDEV_SAMP and STDDEV_POP window functions

The STDDEV_SAMP and STDDEV_POP window functions return the sample and population standard deviation of a set of numeric values (integer, decimal, or floating-point). See also STDDEV_SAMP and STDDEV_POP functions.

STDDEV_SAMP and STDDEV are synonyms for the same function.

Syntax

```
STDDEV_SAMP | STDDEV | STDDEV_POP
( [ ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list
                        frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

The argument types supported by the STDDEV functions are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

Regardless of the data type of the expression, the return type of a STDDEV function is a double precision number.

Examples

The following example shows how to use STDDEV_POP and VAR_POP functions as window functions. The query computes the population variance and population standard deviation for PRICEPAID values in the SALES table.

```
select salesid, dateid, pricepaid,
round(stddev_pop(pricepaid) over
(order by dateid, salesid rows unbounded preceding)) as stddevpop,
round(var_pop(pricepaid) over
(order by dateid, salesid rows unbounded preceding)) as varpop
from sales
order by 2,1;

salesid | dateid | pricepaid | stddevpop | varpop
--------+--------+-----------+-----------+---------
  33095 |   1827 |    234.00 |         0 |       0
  65082 |   1827 |    472.00 |       119 |   14161
  88268 |   1827 |    836.00 |       248 |   61283
  97197 |   1827 |    708.00 |       230 |   53019
 110328 |   1827 |    347.00 |       223 |   49845
 110917 |   1827 |    337.00 |       215 |   46159
 150314 |   1827 |    688.00 |       211 |   44414
 157751 |   1827 |   1730.00 |       447 |  199679
 165890 |   1827 |   4192.00 |      1185 | 1403323
...
```

The sample standard deviation and variance functions can be used in the same way.

SUM window function

The SUM window function returns the sum of the input column or expression values. The SUM function works with numeric values and ignores NULL values.

Syntax

```
SUM ( [ ALL ] expression ) OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list
                        frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the SUM function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of

rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

The argument types supported by the SUM function are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

The return types supported by the SUM function are:

- BIGINT for SMALLINT or INTEGER arguments
- NUMERIC for BIGINT arguments
- DOUBLE PRECISION for floating-point arguments

Examples

The following example creates a cumulative (rolling) sum of sales quantities ordered by date and sales ID:

```
select salesid, dateid, sellerid, qty,
sum(qty) over (order by dateid, salesid rows unbounded preceding) as sum
from winsales
order by 2,1;

salesid |   dateid   | sellerid | qty | sum
---------+------------+----------+-----+-----
30001 | 2003-08-02 |        3 |  10 |  10
10001 | 2003-12-24 |        1 |  10 |  20
10005 | 2003-12-24 |        1 |  30 |  50
40001 | 2004-01-09 |        4 |  40 |  90
10006 | 2004-01-18 |        1 |  10 | 100
20001 | 2004-02-12 |        2 |  20 | 120
40005 | 2004-02-12 |        4 |  10 | 130
20002 | 2004-02-16 |        2 |  20 | 150
30003 | 2004-04-18 |        3 |  15 | 165
30004 | 2004-04-18 |        3 |  20 | 185
30007 | 2004-09-07 |        3 |  30 | 215
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example creates a cumulative (rolling) sum of sales quantities by date, partition the results by seller ID, and order the results by date and sales ID within the partition:

```
select salesid, dateid, sellerid, qty,
sum(qty) over (partition by sellerid
order by dateid, salesid rows unbounded preceding) as sum
from winsales
order by 2,1;

salesid |   dateid   | sellerid | qty | sum
---------+------------+----------+-----+-----
30001 | 2003-08-02 |        3 |  10 |  10
10001 | 2003-12-24 |        1 |  10 |  10
10005 | 2003-12-24 |        1 |  30 |  40
40001 | 2004-01-09 |        4 |  40 |  40
10006 | 2004-01-18 |        1 |  10 |  50
20001 | 2004-02-12 |        2 |  20 |  20
40005 | 2004-02-12 |        4 |  10 |  50
20002 | 2004-02-16 |        2 |  20 |  40
30003 | 2004-04-18 |        3 |  15 |  25
30004 | 2004-04-18 |        3 |  20 |  45
30007 | 2004-09-07 |        3 |  30 |  75
(11 rows)
```

The following example numbers all of the rows sequentially in the result set, ordered by the SELLERID and SALESID columns:

```
select salesid, sellerid, qty,
sum(1) over (order by sellerid, salesid rows unbounded preceding) as rownum
from winsales
order by 2,1;

salesid | sellerid |  qty | rownum
--------+----------+------+--------
10001 |        1 |   10 |     1
10005 |        1 |   30 |     2
10006 |        1 |   10 |     3
20001 |        2 |   20 |     4
20002 |        2 |   20 |     5
30001 |        3 |   10 |     6
30003 |        3 |   15 |     7
30004 |        3 |   20 |     8
30007 |        3 |   30 |     9
```

```
40001 |        4 |   40 |    10
40005 |        4 |   10 |    11
(11 rows)
```

For a description of the WINSALES table, see Sample table for window function examples.

The following example numbers all rows sequentially in the result set, partition the results by SELLERID, and order the results by SELLERID and SALESID within the partition:

```
select salesid, sellerid, qty,
sum(1) over (partition by sellerid
order by sellerid, salesid rows unbounded preceding) as rownum
from winsales
order by 2,1;

salesid | sellerid | qty | rownum
---------+----------+-----+--------
10001 |        1 |  10 |      1
10005 |        1 |  30 |      2
10006 |        1 |  10 |      3
20001 |        2 |  20 |      1
20002 |        2 |  20 |      2
30001 |        3 |  10 |      1
30003 |        3 |  15 |      2
30004 |        3 |  20 |      3
30007 |        3 |  30 |      4
40001 |        4 |  40 |      1
40005 |        4 |  10 |      2
(11 rows)
```

VAR_SAMP and VAR_POP window functions

The VAR_SAMP and VAR_POP window functions return the sample and population variance of a set of numeric values (integer, decimal, or floating-point). See also VAR_SAMP and VAR_POP functions.

VAR_SAMP and VARIANCE are synonyms for the same function.

Syntax

```
VAR_SAMP | VARIANCE | VAR_POP
( [ ALL ] expression ) OVER
(
```

```
[ PARTITION BY expr_list ]
[ ORDER BY order_list
                        frame_clause ]
)
```

Arguments

expression

The target column or expression that the function operates on.

ALL

With the argument ALL, the function retains all duplicate values from the expression. ALL is the default. DISTINCT is not supported.

OVER

Specifies the window clauses for the aggregation functions. The OVER clause distinguishes window aggregation functions from normal set aggregation functions.

PARTITION BY *expr_list*

Defines the window for the function in terms of one or more expressions.

ORDER BY *order_list*

Sorts the rows within each partition. If no PARTITION BY is specified, ORDER BY uses the entire table.

frame_clause

If an ORDER BY clause is used for an aggregate function, an explicit frame clause is required. The frame clause refines the set of rows in a function's window, including or excluding sets of rows within the ordered result. The frame clause consists of the ROWS keyword and associated specifiers. See Window function syntax summary.

Data types

The argument types supported by the VARIANCE functions are SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, REAL, and DOUBLE PRECISION.

Regardless of the data type of the expression, the return type of a VARIANCE function is a double precision number.

SQL conditions in AWS Clean Rooms

Conditions are statements of one or more expressions and logical operators that evaluate to true, false, or unknown. Conditions are also sometimes referred to as predicates.

> **Note**
>
> All string comparisons and LIKE pattern matches are case-sensitive. For example, 'A' and 'a' do not match. However, you can do a case-insensitive pattern match by using the ILIKE predicate.

The following SQL conditions are supported in AWS Clean Rooms.

Topics

Comparison conditions

Comparison conditions state logical relationships between two values. All comparison conditions are binary operators with a Boolean return type. AWS Clean Rooms supports the comparison operators described in the following table.

Operator	Syntax	Description
<	a < b	Value a is less than value b.
>	a > b	Value a is greater than value b.

Operator	Syntax	Description
<=	`a <= b`	Value a is less than or equal to value b.
>=	`a >= b`	Value a is greater than or equal to value b.
=	`a = b`	Value a is equal to value b.
<> or !=	`a <> b or a != b`	Value a is not equal to value b.
a = TRUE	`a IS TRUE`	Value a is Boolean TRUE.

Usage notes

= ANY | SOME

The ANY and SOME keywords are synonymous with the *IN* condition. The ANY and SOME keywords return true if the comparison is true for at least one value returned by a subquery that returns one or more values. AWS Clean Rooms supports only the = (equals) condition for ANY and SOME. Inequality conditions are not supported.

> **Note**
>
> The ALL predicate is not supported.

<> ALL

The ALL keyword is synonymous with NOT IN (see IN condition condition) and returns true if the expression is not included in the results of the subquery. AWS Clean Rooms supports only the <> or != (not equals) condition for ALL. Other comparison conditions are not supported.

IS TRUE/FALSE/UNKNOWN

Non-zero values equate to TRUE, 0 equates to FALSE, and null equates to UNKNOWN. See the Boolean type data type.

Examples

Here are some simple examples of comparison conditions:

```
a = 5
a < b
min(x) >= 5
qtysold = any (select qtysold from sales where dateid = 1882
```

The following query returns venues with more than 10,000 seats from the VENUE table:

```
select venueid, venuename, venueseats from venue
where venueseats > 10000
order by venueseats desc;

venueid |           venuename            | venueseats
---------+--------------------------------+------------
83 | FedExField                     |      91704
 6 | New York Giants Stadium        |      80242
79 | Arrowhead Stadium              |      79451
78 | INVESCO Field                  |      76125
69 | Dolphin Stadium                |      74916
67 | Ralph Wilson Stadium           |      73967
76 | Jacksonville Municipal Stadium |      73800
89 | Bank of America Stadium        |      73298
72 | Cleveland Browns Stadium       |      73200
86 | Lambeau Field                  |      72922
...
(57 rows)
```

This example selects the users (USERID) from the USERS table who like rock music:

```
select userid from users where likerock = 't' order by 1 limit 5;

userid
--------
3
5
6
13
16
(5 rows)
```

This example selects the users (USERID) from the USERS table where it is unknown whether they like rock music:

```
select firstname, lastname, likerock
from users
where likerock is unknown
order by userid limit 10;

firstname | lastname | likerock
----------+----------+----------
Rafael    | Taylor   |
Vladimir  | Humphrey |
Barry     | Roy      |
Tamekah   | Juarez   |
Mufutau   | Watkins  |
Naida     | Calderon |
Anika     | Huff     |
Bruce     | Beck     |
Mallory   | Farrell  |
Scarlett  | Mayer    |
(10 rows
```

Examples with a TIME column

The following example table TIME_TEST has a column TIME_VAL (type TIME) with three values inserted.

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

The following example extracts the hours from each timetz_val.

```
select time_val from time_test where time_val < '3:00';
   time_val
---------------
 00:00:00.5550
 00:58:00
```

The following example compares two time literals.

```
select time '18:25:33.123456' = time '18:25:33.123456';
 ?column?
----------
 t
```

Examples with a TIMETZ column

The following example table TIMETZ_TEST has a column TIMETZ_VAL (type TIMETZ) with three values inserted.

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

The following example selects only the TIMETZ values less than `3:00:00` UTC. The comparison is made after converting the value to UTC.

```
select timetz_val from timetz_test where timetz_val < '3:00:00 UTC';

   timetz_val
---------------
 00:00:00.5550+00
```

The following example compares two TIMETZ literals. The time zone is ignored for the comparison.

```
select time '18:25:33.123456 PST' < time '19:25:33.123456 EST';

 ?column?
----------
 t
```

Logical conditions

Logical conditions combine the result of two conditions to produce a single result. All logical conditions are binary operators with a Boolean return type.

Syntax

```
expression
{ AND | OR }
expression
NOT expression
```

Logical conditions use a three-valued Boolean logic where the null value represents an unknown relationship. The following table describes the results for logical conditions, where E1 and E2 represent expressions:

E1	E2	E1 AND E2	E1 OR E2	NOT E2
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
FALSE	TRUE	FALSE	TRUE	
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	
UNKNOWN	TRUE	UNKNOWN	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	

The NOT operator is evaluated before AND, and the AND operator is evaluated before the OR operator. Any parentheses used may override this default order of evaluation.

Examples

The following example returns USERID and USERNAME from the USERS table where the user likes both Las Vegas and sports:

```
select userid, username from users
```

```
where likevegas = 1 and likesports = 1
order by userid;

userid | username
--------+----------
1 | JSG99FHE
67 | TWU10MZT
87 | DUF19VXU
92 | HYP36WEQ
109 | FPL38HZK
120 | DMJ24GUZ
123 | QZR22XGQ
130 | ZQC82ALK
133 | LBN45WCH
144 | UCX04JKN
165 | TEY68OEB
169 | AYQ83HGO
184 | TVX65AZX
...
(2128 rows)
```

The next example returns the USERID and USERNAME from the USERS table where the user likes Las Vegas, or sports, or both. This query returns all of the output from the previous example plus the users who like only Las Vegas or sports.

```
select userid, username from users
where likevegas = 1 or likesports = 1
order by userid;

userid | username
--------+----------
1 | JSG99FHE
2 | PGL08LJI
3 | IFT66TXU
5 | AEB55QTM
6 | NDQ15VBM
9 | MSD36KVR
10 | WKW41AIW
13 | QTF33MCG
15 | OWU78MTR
16 | ZMG93CDD
22 | RHT62AGI
27 | KOY02CVE
```

```
29 | HUH27PKK
...
(18968 rows)
```

The following query uses parentheses around the OR condition to find venues in New York or California where Macbeth was performed:

```
select distinct venuename, venuecity
from venue join event on venue.venueid=event.venueid
where (venuestate = 'NY' or venuestate = 'CA') and eventname='Macbeth'
order by 2,1;

venuename                |   venuecity
-----------------------------------------+---------------
Geffen Playhouse                         | Los Angeles
Greek Theatre                            | Los Angeles
Royce Hall                               | Los Angeles
American Airlines Theatre                | New York City
August Wilson Theatre                    | New York City
Belasco Theatre                          | New York City
Bernard B. Jacobs Theatre                | New York City
...
```

Removing the parentheses in this example changes the logic and results of the query.

The following example uses the NOT operator:

```
select * from category
where not catid=1
order by 1;

catid | catgroup |  catname  |                 catdesc
-------+----------+-----------+--------------------------------------------
2 | Sports   | NHL       | National Hockey League
3 | Sports   | NFL       | National Football League
4 | Sports   | NBA       | National Basketball Association
5 | Sports   | MLS       | Major League Soccer
...
```

The following example uses a NOT condition followed by an AND condition:

```
select * from category
```

```
where (not catid=1) and catgroup='Sports'
order by catid;

catid | catgroup | catname |             catdesc
-------+----------+---------+---------------------------------
2 | Sports   | NHL     | National Hockey League
3 | Sports   | NFL     | National Football League
4 | Sports   | NBA     | National Basketball Association
5 | Sports   | MLS     | Major League Soccer
(4 rows)
```

Pattern-matching conditions

A pattern-matching operator searches a string for a pattern specified in the conditional expression and returns true or false depending on whether it finds a match. AWS Clean Rooms uses the following methods for pattern matching:

- LIKE expressions

  The LIKE operator compares a string expression, such as a column name, with a pattern that uses the wildcard characters % (percent) and _ (underscore). LIKE pattern matching always covers the entire string. LIKE performs a case-sensitive match and ILIKE performs a case-insensitive match.

- SIMILAR TO regular expressions

  The SIMILAR TO operator matches a string expression with a SQL standard regular expression pattern, which can include a set of pattern-matching metacharacters that includes the two supported by the LIKE operator. SIMILAR TO matches the entire string and performs a case-sensitive match.

Topics

LIKE

The LIKE operator compares a string expression, such as a column name, with a pattern that uses the wildcard characters % (percent) and _ (underscore). LIKE pattern matching always covers the

entire string. To match a sequence anywhere within a string, the pattern must start and end with a percent sign.

LIKE is case-sensitive; ILIKE is case-insensitive.

Syntax

```
expression [ NOT ] LIKE | ILIKE pattern [ ESCAPE 'escape_char' ]
```

Arguments

expression

A valid UTF-8 character expression, such as a column name.

LIKE | ILIKE

LIKE performs a case-sensitive pattern match. ILIKE performs a case-insensitive pattern match for single-byte UTF-8 (ASCII) characters. To perform a case-insensitive pattern match for multibyte characters, use the LOWER function on *expression* and *pattern* with a LIKE condition.

In contrast to comparison predicates, such as = and <>, LIKE and ILIKE predicates do not implicitly ignore trailing spaces. To ignore trailing spaces, use RTRIM or explicitly cast a CHAR column to VARCHAR.

The ~~ operator is equivalent to LIKE, and ~~* is equivalent to ILIKE. Also the !~~ and !~~* operators are equivalent to NOT LIKE and NOT ILIKE.

pattern

A valid UTF-8 character expression with the pattern to be matched.

escape_char

A character expression that will escape metacharacters characters in the pattern. The default is two backslashes ('\\').

If *pattern* does not contain metacharacters, then the pattern only represents the string itself; in that case LIKE acts the same as the equals operator.

Either of the character expressions can be CHAR or VARCHAR data types. If they differ, AWS Clean Rooms converts *pattern* to the data type of *expression*.

LIKE supports the following pattern-matching metacharacters:

Operator	Description
%	Matches any sequence of zero or more characters.
_	Matches any single character.

Examples

The following table shows examples of pattern matching using LIKE:

Expression	Returns
'abc' LIKE 'abc'	True
'abc' LIKE 'a%'	True
'abc' LIKE '_B_'	False
'abc' ILIKE '_B_'	True
'abc' LIKE 'c%'	False

The following example finds all cities whose names start with "E":

```
select distinct city from users
where city like 'E%' order by city;
city
---------------
East Hartford
East Lansing
East Rutherford
East St. Louis
Easthampton
Easton
Eatontown
Eau Claire
...
```

The following example finds users whose last name contains "ten" :

```
select distinct lastname from users
where lastname like '%ten%' order by lastname;
lastname
-------------
Christensen
Wooten
...
```

The following example finds cities whose third and fourth characters are "ea". The command uses ILIKE to demonstrate case insensitivity:

```
select distinct city from users where city ilike '__EA%' order by city;
city
-------------
Brea
Clearwater
Great Falls
Ocean City
Olean
Wheaton
(6 rows)
```

The following example uses the default escape string (\\) to search for strings that include "start_" (the text `start` followed by an underscore _):

```
select tablename, "column" from my_table_def


where "column" like '%start\\_%'
limit 5;

     tablename     |    column
-------------------+---------------
 my_s3client       | start_time
 my_tr_conflict    | xact_start_ts
 my_undone         | undo_start_ts
 my_unload_log     | start_time
 my_vacuum_detail | start_row
(5 rows)
```

The following example specifies '^' as the escape character, then uses the escape character to search for strings that include "start_" (the text `start` followed by an underscore _):

```
select tablename, "column" from my_table_def

where "column" like '%start^_%' escape '^'
limit 5;

     tablename     |    column
-------------------+---------------
 my_s3client       | start_time
 my_tr_conflict    | xact_start_ts
 my_undone         | undo_start_ts
 my_unload_log     | start_time
 my_vacuum_detail | start_row
(5 rows)
```

The following example uses the ~~* operator to do a case-insensitive (ILIKE) search for cities that start with "Ag".

```
select distinct city from users where city ~~* 'Ag%' order by city;

city
------------
Agat
Agawam
Agoura Hills
Aguadilla
```

SIMILAR TO

The SIMILAR TO operator matches a string expression, such as a column name, with a SQL standard regular expression pattern. A SQL regular expression pattern can include a set of pattern-matching metacharacters, including the two supported by the LIKE operator.

The SIMILAR TO operator returns true only if its pattern matches the entire string, unlike POSIX regular expression behavior, where the pattern can match any portion of the string.

SIMILAR TO performs a case-sensitive match.

Note

Regular expression matching using SIMILAR TO is computationally expensive. We recommend using LIKE whenever possible, especially when processing a very large number of rows. For example, the following queries are functionally identical, but the query that uses LIKE runs several times faster than the query that uses a regular expression:

```
select count(*) from event where eventname SIMILAR TO '%(Ring|Die)%';
select count(*) from event where eventname LIKE '%Ring%' OR eventname LIKE '%Die
%';
```

Syntax

```
expression [ NOT ] SIMILAR TO pattern [ ESCAPE 'escape_char' ]
```

Arguments

expression

A valid UTF-8 character expression, such as a column name.

SIMILAR TO

SIMILAR TO performs a case-sensitive pattern match for the entire string in *expression*.

pattern

A valid UTF-8 character expression representing a SQL standard regular expression pattern.

escape_char

A character expression that will escape metacharacters in the pattern. The default is two backslashes ('\\').

If *pattern* does not contain metacharacters, then the pattern only represents the string itself.

Either of the character expressions can be CHAR or VARCHAR data types. If they differ, AWS Clean Rooms converts *pattern* to the data type of *expression*.

SIMILAR TO supports the following pattern-matching metacharacters:

Operator	Description
%	Matches any sequence of zero or more characters.
_	Matches any single character.
|	Denotes alternation (either of two alternatives).
*	Repeat the previous item zero or more times.
+	Repeat the previous item one or more times.
?	Repeat the previous item zero or one time.
{m}	Repeat the previous item exactly *m* times.
{m,}	Repeat the previous item *m* or more times.
{m,n}	Repeat the previous item at least *m* and not more than *n* times.
()	Parentheses group items into a single logical item.
[...]	A bracket expression specifies a character class, just as in POSIX regular expressions.

Examples

The following table shows examples of pattern matching using SIMILAR TO:

Expression	Returns
`'abc' SIMILAR TO 'abc'`	True
`'abc' SIMILAR TO '_b_'`	True
`'abc' SIMILAR TO '_A_'`	False
`'abc' SIMILAR TO '%(b|d)%'`	True
`'abc' SIMILAR TO '(b|c)%'`	False

Expression	Returns
`'AbcAbcdefgefg12efgefg12' SIMILAR TO '((Ab)?c)+d((efg)+(12))+'`	True
`'aaaaaab11111xy' SIMILAR TO 'a{6}_[0-9]{5}(x|y){2}'`	True
`'$0.87' SIMILAR TO '$[0-9]+(.[0-9][0-9])?'`	True

The following example finds cities whose names contain "E" or "H":

```
SELECT DISTINCT city FROM users
WHERE city SIMILAR TO '%E%|%H%' ORDER BY city LIMIT 5;

      city
-----------------
 Agoura Hills
 Auburn Hills
 Benton Harbor
 Beverly Hills
 Chicago Heights
```

The following example uses the default escape string ('\\') to search for strings that include "_":

```
SELECT tablename, "column" FROM my_table_def
WHERE "column" SIMILAR TO '%start\\_%'

ORDER BY tablename, "column" LIMIT 5;

        tablename         |       column
--------------------------+---------------------
 my_abort_idle            | idle_start_time
 my_abort_idle            | txn_start_time
 my_analyze_compression | start_time
 my_auto_worker_levels    | start_level
 my_auto_worker_levels    | start_wlm_occupancy
```

The following example specifies '^' as the escape string, then uses the escape string to search for strings that include "_":

```
SELECT tablename, "column" FROM my_table_def

WHERE "column" SIMILAR TO '%start^_%' ESCAPE '^'
ORDER BY tablename, "column" LIMIT 5;

        tablename         |       column
--------------------------+---------------------
 stcs_abort_idle          | idle_start_time
 stcs_abort_idle          | txn_start_time
 stcs_analyze_compression | start_time
 stcs_auto_worker_levels  | start_level
 stcs_auto_worker_levels  | start_wlm_occupancy
```

BETWEEN range condition

A BETWEEN condition tests expressions for inclusion in a range of values, using the keywords BETWEEN and AND.

Syntax

```
expression [ NOT ] BETWEEN expression AND expression
```

Expressions can be numeric, character, or datetime data types, but they must be compatible. The range is inclusive.

Examples

The first example counts how many transactions registered sales of either 2, 3, or 4 tickets:

```
select count(*) from sales
where qtysold between 2 and 4;

count
--------
104021
(1 row)
```

The range condition includes the begin and end values.

```
select min(dateid), max(dateid) from sales
where dateid between 1900 and 1910;

min  | max
-----+-----
1900 | 1910
```

The first expression in a range condition must be the lesser value and the second expression the greater value. The following example will always return zero rows due to the values of the expressions:

```
select count(*) from sales
where qtysold between 4 and 2;

count
-------
0
(1 row)
```

However, applying the NOT modifier will invert the logic and produce a count of all rows:

```
select count(*) from sales
where qtysold not between 4 and 2;

count
--------
172456
(1 row)
```

The following query returns a list of venues with 20000 to 50000 seats:

```
select venueid, venuename, venueseats from venue
where venueseats between 20000 and 50000
order by venueseats desc;

venueid |       venuename               | venueseats
---------+-------------------------------+------------
116 | Busch Stadium                 |      49660
106 | Rangers BallPark in Arlington |      49115
```

```
96 | Oriole Park at Camden Yards   |      48876
...
(22 rows)
```

The following example demonstrates using BETWEEN for date values:

```
select salesid, qtysold, pricepaid, commission, saletime
from sales
where eventid between 1000 and 2000
   and saletime between '2008-01-01' and '2008-01-03'
order by saletime asc;

salesid | qtysold | pricepaid | commission |   saletime
--------+---------+-----------+------------+---------------
  65082 |       4 |       472 |       70.8 | 1/1/2008 06:06
 110917 |       1 |       337 |      50.55 | 1/1/2008 07:05
 112103 |       1 |       241 |      36.15 | 1/2/2008 03:15
 137882 |       3 |      1473 |     220.95 | 1/2/2008 05:18
  40331 |       2 |        58 |        8.7 | 1/2/2008 05:57
 110918 |       3 |      1011 |     151.65 | 1/2/2008 07:17
  96274 |       1 |       104 |       15.6 | 1/2/2008 07:18
 150499 |       3 |       135 |      20.25 | 1/2/2008 07:20
  68413 |       2 |       158 |       23.7 | 1/2/2008 08:12
```

Note that although BETWEEN's range is inclusive, dates default to having a time value of 00:00:00. The only valid January 3 row for the sample query would be a row with a saletime of `1/3/2008 00:00:00`.

Null condition

The NULL condition tests for nulls, when a value is missing or unknown.

Syntax

```
expression IS [ NOT ] NULL
```

Arguments

expression

Any expression such as a column.

IS NULL

Is true when the expression's value is null and false when it has a value.

IS NOT NULL

Is false when the expression's value is null and true when it has a value.

Example

This example indicates how many times the SALES table contains null in the QTYSOLD field:

```
select count(*) from sales
where qtysold is null;
count
-------
0
(1 row)
```

EXISTS condition

EXISTS conditions test for the existence of rows in a subquery, and return true if a subquery returns at least one row. If NOT is specified, the condition returns true if a subquery returns no rows.

Syntax

```
[ NOT ] EXISTS (table_subquery)
```

Arguments

EXISTS

Is true when the *table_subquery* returns at least one row.

NOT EXISTS

Is true when the *table_subquery* returns no rows.

table_subquery

A subquery that evaluates to a table with one or more columns and one or more rows.

Example

This example returns all date identifiers, one time each, for each date that had a sale of any kind:

```
select dateid from date
where exists (
select 1 from sales
where date.dateid = sales.dateid
)
order by dateid;

dateid
--------
1827
1828
1829
...
```

IN condition

An IN condition tests a value for membership in a set of values or in a subquery.

Syntax

```
expression [ NOT ] IN (expr_list | table_subquery)
```

Arguments

expression

A numeric, character, or datetime expression that is evaluated against the *expr_list* or *table_subquery* and must be compatible with the data type of that list or subquery.

expr_list

One or more comma-delimited expressions, or one or more sets of comma-delimited expressions bounded by parentheses.

table_subquery

A subquery that evaluates to a table with one or more rows, but is limited to only one column in its select list.

IN | NOT IN

IN returns true if the expression is a member of the expression list or query. NOT IN returns true if the expression is not a member. IN and NOT IN return NULL and no rows are returned in the following cases: If *expression* yields null; or if there are no matching *expr_list* or *table_subquery* values and at least one of these comparison rows yields null.

Examples

The following conditions are true only for those values listed:

```
qtysold in (2, 4, 5)
date.day in ('Mon', 'Tues')
date.month not in ('Oct', 'Nov', 'Dec')
```

Optimization for Large IN Lists

To optimize query performance, an IN list that includes more than 10 values is internally evaluated as a scalar array. IN lists with fewer than 10 values are evaluated as a series of OR predicates. This optimization is supported for SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, BOOLEAN, CHAR, VARCHAR, DATE, TIMESTAMP, and TIMESTAMPTZ data types.

Look at the EXPLAIN output for the query to see the effect of this optimization. For example:

```
explain select * from sales
QUERY PLAN
------------------------------------------------------------------------
XN Seq Scan on sales  (cost=0.00..6035.96 rows=86228 width=53)
Filter: (salesid = ANY ('{1,2,3,4,5,6,7,8,9,10,11}'::integer[]))
(2 rows)
```

Syntax

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
```

```
| IN_condition
```

Querying nested data

AWS Clean Rooms offers SQL-compatible access to relational and nested data.

AWS Clean Rooms uses dotted notation and array subscript for path navigation when accessing nested data. It also enables the FROM clause items to iterate over arrays and use for unnest operations. The following topics provide descriptions of the different query patterns that combine the use of the array/struct/map data type with path and array navigation, unnesting, and joins.

Topics

Navigation

AWS Clean Rooms enables navigation into arrays and structures using the `[...]` bracket and dot notation respectively. Furthermore, you can mix navigation into structures using the dot notation and arrays using the bracket notation.

Example

For example, the following example query assumes that the `c_orders` array data column is an array with a structure and an attribute is named `o_orderkey`.

```
SELECT cust.c_orders[0].o_orderkey FROM customer_orders_lineitem AS cust;
```

You can use the dot and bracket notations in all types of queries, such as filtering, join, and aggregation. You can use these notations in a query in which there are normally column references.

Example

The following example uses a SELECT statement that filters results.

```
SELECT count(*) FROM customer_orders_lineitem WHERE c_orders[0].o_orderkey IS NOT NULL;
```

Example

The following example uses the bracket and dot navigation in both GROUP BY and ORDER BY clauses.

```
SELECT c_orders[0].o_orderdate,
       c_orders[0].o_orderstatus,
       count(*)
FROM customer_orders_lineitem
WHERE c_orders[0].o_orderkey IS NOT NULL
GROUP BY c_orders[0].o_orderstatus,
         c_orders[0].o_orderdate
ORDER BY c_orders[0].o_orderdate;
```

Unnesting queries

To unnest queries, AWS Clean Rooms enables iteration over arrays. It does this by navigating the array using the FROM clause of a query.

Example

Using the previous example, the following example iterates over the attribute values for `c_orders`.

```
SELECT o FROM customer_orders_lineitem c, c.c_orders o;
```

The unnesting syntax is an extension of the FROM clause. In standard SQL, the FROM clause `x (AS) y` means that `y` iterates over each tuple in relation `x`. In this case, `x` refers to a relation and `y` refers to an alias for relation `x`. Similarly, the syntax of unnesting using the FROM clause item `x (AS) y` means that `y` iterates over each value in array expression `x`. In this case, `x` is an array expression and `y` is an alias for `x`.

The left operand can also use the dot and bracket notation for regular navigation.

Example

In the previous example:

- `customer_orders_lineitem c` is the iteration over the `customer_order_lineitem` base table
- `c.c_orders o` is the iteration over the `c.c_orders array`

To iterate over the o_lineitems attribute, which is an array within an array, you add multiple clauses.

```
SELECT o, l FROM customer_orders_lineitem c, c.c_orders o, o.o_lineitems l;
```

AWS Clean Rooms also supports an array index when iterating over the array using the AT keyword. The clause x AS y AT z iterates over array x and generates the field z, which is the array index.

Example

The following example shows how an array index works.

```
SELECT c_name,
       orders.o_orderkey AS orderkey,
       index AS orderkey_index
FROM customer_orders_lineitem c, c.c_orders AS orders AT index
ORDER BY orderkey_index;
c_name             | orderkey | orderkey_index
-------------------+----------+----------------
Customer#000008251 | 3020007  |        0
Customer#000009452 | 4043971  |        0  (2 rows)
```

Example

The following example iterates over a scalar array.

```
CREATE TABLE bar AS SELECT json_parse('{"scalar_array": [1, 2.3, 45000000]}') AS data;

SELECT index, element FROM bar AS b, b.data.scalar_array AS element AT index;

 index | element
-------+----------
     0 | 1
1 | 2.3
2 | 45000000
(3 rows)
```

Example

The following example iterates over an array of multiple levels. The example uses multiple unnest clauses to iterate into the innermost arrays. The f.multi_level_array AS array

iterates over `multi_level_array`. The array AS element is the iteration over the arrays within `multi_level_array`.

```
CREATE TABLE foo AS SELECT json_parse('[[1.1, 1.2], [2.1, 2.2], [3.1, 3.2]]') AS
 multi_level_array;

SELECT array, element FROM foo AS f, f.multi_level_array AS array, array AS element;

   array   | element
-----------+---------
 [1.1,1.2] | 1.1
 [1.1,1.2] | 1.2
 [2.1,2.2] | 2.1
 [2.1,2.2] | 2.2
 [3.1,3.2] | 3.1
 [3.1,3.2] | 3.2
(6 rows)
```

Lax semantics

By default, navigation operations on nested data values return null instead of returning an error out when the navigation is invalid. Object navigation is invalid if the nested data value is not an object or if the nested data value is an object but doesn't contain the attribute name used in the query.

Example

For example, the following query accesses an invalid attribute name in the nested data column `c_orders`:

```
SELECT c.c_orders.something FROM customer_orders_lineitem c;
```

Array navigation returns null if the nested data value is not an array or the array index is out of bounds.

Example

The following query returns null because `c_orders[1][1]` is out of bounds.

```
SELECT c.c_orders[1][1] FROM customer_orders_lineitem c;
```

Types of introspection

Nested data type columns support inspection functions that return the type and other type information about the value. AWS Clean Rooms supports the following boolean functions for nested data columns:

- DECIMAL_PRECISION
- DECIMAL_SCALE
- IS_ARRAY
- IS_BIGINT
- IS_CHAR
- IS_DECIMAL
- IS_FLOAT
- IS_INTEGER
- IS_OBJECT
- IS_SCALAR
- IS_SMALLINT
- IS_VARCHAR
- JSON_TYPEOF

All these functions return false if the input value is null. IS_SCALAR, IS_OBJECT, and IS_ARRAY are mutually exclusive and cover all possible values except for null. To infer the types corresponding to the data, AWS Clean Rooms uses the JSON_TYPEOF function that returns the type of (the top level of) the nested data value as shown in the following example:

```
SELECT JSON_TYPEOF(r_nations) FROM region_nations;
 json_typeof
 -------------
array
(1 row)
```

```
SELECT JSON_TYPEOF(r_nations[0].n_nationkey) FROM region_nations;
 json_typeof
 -------------
 number
```

Document history for the AWS Clean Rooms SQL Reference

The following table describes the documentation releases for the AWS Clean Rooms SQL Reference.

For notification about updates to this documentation, you can subscribe to the RSS feed. To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser you are using.

Change	Description	Date
SQL commands and SQL functions - update	Examples have been added for the JOIN clause, EXCEPT set operator, CASE conditional expression, and the following functions: ANY_VALUE, NVL and COALESCE, NULLIF, CAST, CONVERT, CONVERT_TIMEZONE, EXTRACT, MOD, SIGN, CONCAT, FIRST_VALUE, and LAST_VALUE.	February 28, 2024
SQL functions - update	AWS Clean Rooms now supports the following SQL functions: Array, SUPER, and VARBYTE. The following math functions are now supported: ACOS, ASIN, ATAN, ATAN2, COT, DEXP, PI, POW, RADIANS, and SIN. The following JSON functions are now supported: CAN_JSON_PARSE, JSON_PARSE, and JSON_SERIALIZE.	October 6, 2023

Nested data type support	AWS Clean Rooms now supports nested data types.	August 30, 2023
SQL naming rules - update	Documentation-only change to clarify reserved column names.	August 16, 2023
General availability	The AWS Clean Rooms SQL Reference is now generally available.	July 31, 2023