SQL Reference

AWS Clean Rooms

AWS Clean Rooms: SQL Reference

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 and naming rules used in this SQL Reference.

Topics

The following sections provide information about the literals, data types, SQL commands, types of SQL functions, and SQL conditions you can use in AWS Clean Rooms.

- AWS Clean Rooms Spark SQL

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.
[ ]	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

Character	Description
	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.

Topics

- Configured table association names and columns
- Reserved words

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 can't start or end with a hyphen.

- (*Custom analysis rule only*) They can use the dollar sign ($) but can't 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 can't contain consecutive hyphen (-) characters.
- They can't begin with any of the following prefixes:

  `padb_`, `pg_`, `stcs_`, `stl_`, `stll_`, `stv_`, `svcs_`, `svl_`, `svv_`, `sys_`, `systable_`
- They can't contain backslash characters (\) , quotation marks ('), or spaces that aren't 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 can't be used in AWS Clean Rooms (even with quotes):
  - oid
  - tableoid
  - xmin
  - cmin
  - xmax
  - cmax
  - ctid

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_U SER_IDDEFAULT	INTO	OVERLAPS	VERBOSE
DEFERRABLE	IS	PARALLELPARTITION	WALLETWHEN
DEFLATE	ISNULL	PERCENT	WHERE
DEFRAG	JOIN	PERMISSIONS	WITH
DELTA	LANGUAGE	PIVOTPLACING	WITHOUT

Data type support by SQL engine

AWS Clean Rooms supports multiple SQL engines and dialects. Understanding the data type systems across these implementations is crucial for successful data collaboration and analysis. The following tables show the equivalent data types across AWS Clean Rooms SQL, Snowflake SQL, and Spark SQL.

Numeric data types

Numeric types represent various kinds of numbers, from precise integers to approximate floating-point values. The choice of numeric type affects both storage requirements and computational precision. Integer types vary by byte size, while decimal and floating-point types offer different precision and scale options.

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
8-byte Integer	BIGINT	Not supported	BIGINT, LONG	Signed integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
4-byte Integer	INT	Not supported	INT, INTEGER	Signed integers from -2,147,483,648 to 2,147,483,647
2-byte Integer	SMALLINT	Not supported	SMALLINT, SHORT	Signed integers from -32,768 to 32,767
1-byte Integer	Not supported	Not supported	TINYINT, BYTE	Signed integers from -128 to 127

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Double Precision Float	DOUBLE, DOUBLE PRECISION	FLOAT, FLOAT4, FLOAT8, DOUBLE, DOUBLE PRECISION, REAL	DOUBLE	8-byte double-precision floating point numbers
Single Precision Float	REAL, FLOAT	Not supported	FLOAT	4-byte single-precision floating point numbers
Decimal (fixed precision)	DECIMAL	DECIMAL, NUMERIC, NUMBER **Note** Snowflake automatically aliases smaller-width exact numeric types (INT, BIGINT, SMALLINT, etc.) to NUMBER.	DECIMAL, NUMERIC,	Arbitrary-precision signed decimal numbers

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Decimal (with precision)	DECIMAL(p)	DECIMAL(p), NUMBER(p)	DECIMAL(p)	Fixed-pre cision decimal numbers
Decimal (with scale)	DECIMAL(p,s)	DECIMAL(p,s), NUMBER(p,s)	DECIMAL(p,s)	Fixed-pre cision decimal numbers with scale

Boolean data types

Boolean types represent simple true/false logical values. These types are consistent across SQL engines and are commonly used for flags, conditions, and logical operations.

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Boolean	BOOLEAN	BOOLEAN	BOOLEAN	Represent s true/false values

Date and time data types

Date and time types handle temporal data, with varying levels of precision and time zone awareness. These types support different formats for storing dates, times, and timestamps, with options for including or excluding time zone information.

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Date	DATE	DATE	DATE	Date values (year, month,

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
				day) without time zone
Time	TIME	Not supported	Not supported	Time of day in UTC, without time zone
Time with TZ	TIMETZ	Not supported	Not supported	Time of day in UTC, with time zone
Timestamp	TIMESTAMP	TIMESTAMP , TIMESTAMP _NTZ	TIMESTAMP _NTZ	Timestamp without time zone **Note** NTZ indicates "No Time Zone"
Timestamp with TZ	TIMESTAMPTZ	TIMESTAMP _LTZ	TIMESTAMP , TIMESTAMP _LTZ	Timestamp with local time zone **Note** LTZ indicates "Local Time Zone"

Character data types

Character types store textual data, offering both fixed-length and variable-length options. These types handle text strings and binary data, with optional length specifications to control storage allocation.

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Fixed-length Character	CHAR	CHAR, CHARACTER	CHAR, CHARACTER	Fixed-length character string
Fixed-length Character with Length	CHAR(n)	CHAR(n), CHARACTER(n)	CHAR(n), CHARACTER(n)	Fixed-length character string with specified length
Variable-length Character	VARCHAR	VARCHAR, STRING, TEXT	VARCHAR, STRING	Variable-length character string
Variable-length Character with Length	VARCHAR(n)	VARCHAR(n), STRING(n), TEXT(n)	VARCHAR(n)	Variable-length character string with length limit
Binary	VARBYTE	BINARY, VARBINARY	BINARY	Binary byte sequence
Binary with Length	VARBYTE(n)	Not supported	Not supported	Binary byte sequence with length limit

Structured data types

Structured types allow for complex data organization by combining multiple values into single fields. These include arrays for ordered collections, maps for key-value pairs, and structs for creating custom data structures with named fields.

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Array	ARRAY<type>	ARRAY(type)	ARRAY<type>	Ordered sequence of elements of the same type **Note** Array types must contain elements of the same type
Map	MAP<key,value>	MAP(key,value)	MAP<key,value>	Collection of key-value pairs **Note** Map types must contain elements of the

Data type	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
				same type
Struct	STRUCT< field1: type1, field2: type2>	OBJECT(field1 type1, field2 type2)	STRUCT< field1: type1, field2: type2 >	Structure with named fields of specified types **Note** Structured type syntax may vary slightly between implementations
Super	SUPER	Not supported	Not supported	Flexible type supporting all data types including complex types

AWS Clean Rooms Spark SQL

AWS Clean Rooms Spark SQL enforces rules regarding the use of data types, expressions, and literals.

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

The following topics provide information about the literals, data types, commands, functions, and conditions supported in AWS Clean Rooms Spark SQL.

Topics

Literals

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

AWS Clean Rooms Spark SQL supports several types of literals, including:

- Numeric literals for integer, decimal, and floating-point numbers.
- Character literals, also referred to as strings, character strings, or character constants, used to specify a character string value.
- Date, time, and timestamp literals, used with datetime data types. For more information, see Date, time, and timestamp literals.
- Interval literals. For more information, see Interval literals.
- Boolean literals. For more information, see Boolean literals.
- Null literals, used to specify a null value.
- Only TAB, CARRIAGE RETURN (CR), and LINE FEED (LF) Unicode control characters from the Unicode general category (Cc) are supported.

AWS Clean Rooms Spark SQL doesn't support direct references to string literals in the SELECT clause, but they can be used within functions such as CAST.

+ (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;
```

Data types

Each value that AWS Clean Rooms Spark SQL 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 Spark SQL.

Data type name	Data type	Aliases	Description
ARRAY	the section called "Nested type"	Not applicable	Array nested data type
BIGINT	the section called "Numeric types"	Not applicable	Signed eight-byte integer
BINARY	the section called "Binary type"	Not applicable	Byte sequence values
BOOLEAN	the section called "Boolean type"	BOOL	Logical Boolean (true/false)
BYTE	the section called "Numeric types"	Not applicable	1-byte signed integer numbers, from -128 to 127
CHAR	the section called "Character types"	CHARACTER	Fixed-length character string
DATE	the section called "Datetime types"	Not applicable	Calendar date (year, month, day)

Data type name	Data type	Aliases	Description
DECIMAL	the section called "Numeric types"	NUMERIC	Exact numeric of selectable precision
FLOAT	the section called "Numeric types"	FLOAT8, DOUBLE PRECISION	Double precision floating-point number
INTEGER	the section called "Numeric types"	INT	Signed four-byte integer
INTERVAL	the section called "Datetime types"	Not applicable	Time duration in day to time order or year to month order
LONG	the section called "Numeric types"	Not applicable	8-byte signed integer numbers
MAP	the section called "Nested type"	Not applicable	Map nested data type
REAL	the section called "Numeric types"	FLOAT4	Single precision floating-point number
SHORT	the section called "Numeric types"	Not applicable	2-byte signed integer numbers.
SMALLINT	the section called "Numeric types"	Not applicable	Signed two-byte integer
STRUCT	the section called "Nested type"	Not applicable	Struct nested data type
TIMESTAMP_LTZ	the section called "Datetime types"	Not applicable	Time of day with local time zone

Data type name	Data type	Aliases	Description
TIMESTAMP_NTZ	the section called "Datetime types"	Not applicable	Time of day without time zone
TINYINT	the section called "Numeric types"	Not applicable	1-byte signed integer numbers, from -128 to 127
VARCHAR	the section called "Character types"	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

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

Topics

- Integer types
- DECIMAL or NUMERIC type
- Floating-point types
- Computations with numeric values

Integer types

Use the following 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
SHORT	2 bytes	-32768 to +32767
INTEGER or INT	4 bytes	-2147483648 to +2147483647
BIGINT	8 bytes	-92233720 36854775808 to 922337203 6854775807
LONG	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 AWS Clean Rooms Spark SQL functions.

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.

Operand 1	Operand 2	Return type
SMALLINT or SHORT	SMALLINT or SHORT	SMALLINT or SHORT
SMALLINT or SHORT	INTEGER	INTEGER
SMALLINT or SHORT	BIGINT	BIGINT
SMALLINT or SHORT	DECIMAL	DECIMAL
SMALLINT or SHORT	FLOAT4	FLOAT8
SMALLINT or SHORT	FLOAT8	FLOAT8

Operand 1	Operand 2	Return type
INTEGER	INTEGER	INTEGER
INTEGER	BIGINT or LONG	BIGINT or LONG
INTEGER	DECIMAL	DECIMAL
INTEGER	FLOAT4	FLOAT8
INTEGER	FLOAT8	FLOAT8
BIGINT or LONG	BIGINT or LONG	BIGINT or LONG
BIGINT or LONG	DECIMAL	DECIMAL
BIGINT or LONG	FLOAT4	FLOAT8
BIGINT or LONG	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL
DECIMAL	FLOAT4	FLOAT8
DECIMAL	FLOAT8	FLOAT8
FLOAT4	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. `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 don't 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 or SHORT is cast as DECIMAL(5,0)
- INTEGER is cast as DECIMAL(10,0)
- BIGINT or LONG 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).

Topics

- CHAR or CHARACTER
- VARCHAR or CHARACTER VARYING
- Significance of trailing blanks

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.

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.

Name	Storage	Range (width of column)
CHAR or CHARACTER	Length of string, including trailing blanks (if any)	4096 bytes

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)
```

VARCHAR data types are defined in terms of bytes, not characters. 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)
VARCHAR or CHARACTER VARYING	4 bytes + total bytes for characters, where each character can be 1 to 4 bytes.	65535 bytes (64K -1)

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, TIMESTAMP_LTZ, and TIMESTAMP_NTZ.

Topics

- DATE
- TIMESTAMP_LTZ
- TIMESTAMP_NTZ
- Examples with datetime types
- Date, time, and timestamp literals
- Interval literals
- Interval data types and literals

DATE

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

Name	Storage	Range	Resolution
DATE	4 bytes	4713 BC to 294276 AD	1 day

TIMESTAMP_LTZ

Use the TIMESTAMP_LTZ data type to store complete timestamp values that include the date, the time of day, and the local time zone.

TIMESTAMP represents values comprising values of fields `year`, `month`, `day`, `hour`, `minute`, and `second`, with the session local timezone. The `timestamp` value represents an absolute point in time.

TIMESTAMP in Spark is a user-specified alias associated with one of the TIMESTAMP_LTZ and TIMESTAMP_NTZ variations. You can set the default timestamp type as TIMESTAMP_LTZ (default value) or TIMESTAMP_NTZ via the configuration `spark.sql.timestampType`.

TIMESTAMP_NTZ

Use the TIMESTAMP_NTZ data type to store complete timestamp values that include the date, the time of day, without the local time zone.

TIMESTAMP represents values comprising values of fields `year`, `month`, `day`, `hour`, `minute`, and `second`. All operations are performed without taking any time zone into account.

TIMESTAMP in Spark is a user-specified alias associated with one of the TIMESTAMP_LTZ and TIMESTAMP_NTZ variations. You can set the default timestamp type as TIMESTAMP_LTZ (default value) or TIMESTAMP_NTZ via the configuration `spark.sql.timestampType`.

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
```

Date, time, and timestamp literals

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

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

Input date	Full date
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
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

Input times	Description (of time part)
16:05	4:05 PM exactly

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.
`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 DATE_ADD function.

```
select date_add('today', 1);

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

select date_add('now', 1);

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 Spark SQL.

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 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
```

Interval data types and literals

You can use an interval data type to store durations of time in units such as, `seconds`, `minutes`, `hours`, `days`, `months`, and `years`. Interval data types and literals can be used in datetime calculations, such as, adding intervals to dates and timestamps, summing intervals, and subtracting an interval from a date or timestamp. Interval literals can be used as input values to interval data type columns in a table.

Syntax of interval data type

To specify an interval data type to store a duration of time in years and months:

```
INTERVAL year_to_month_qualifier
```

To specify an interval data type to store a duration in days, hours, minutes, and seconds:

```
INTERVAL day_to_second_qualifier [ (fractional_precision) ]
```

Syntax of interval literal

To specify an interval literal to define a duration of time in years and months:

```
INTERVAL quoted-string year_to_month_qualifier
```

To specify an interval literal to define a duration in days, hours, minutes, and seconds:

```
INTERVAL quoted-string day_to_second_qualifier [ (fractional_precision) ]
```

Arguments

quoted-string

Specifies a positive or negative numeric value specifying a quantity and the datetime unit as an input string. If the *quoted-string* contains only a numeric, then AWS Clean Rooms determines the units from the *year_to_month_qualifier* or *day_to_second_qualifier*. For example, `'23' MONTH` represents `1 year 11 months`, `'-2' DAY` represents `-2 days 0 hours 0 minutes 0.0 seconds`, `'1-2' MONTH` represents `1 year 2 months`, and `'13 day 1 hour 1 minute 1.123 seconds' SECOND` represents `13 days 1 hour 1 minute 1.123 seconds`. For more information about output formats of an interval, see Interval styles.

year_to_month_qualifier

Specifies the range of the interval. If you use a qualifier and create an interval with time units smaller than the qualifier, AWS Clean Rooms truncates and discards the smaller parts of the interval. Valid values for *year_to_month_qualifier* are:

- `YEAR`
- `MONTH`
- `YEAR TO MONTH`

day_to_second_qualifier

Specifies the range of the interval. If you use a qualifier and create an interval with time units smaller than the qualifier, AWS Clean Rooms truncates and discards the smaller parts of the interval. Valid values for *day_to_second_qualifier* are:

- `DAY`
- `HOUR`
- `MINUTE`
- `SECOND`
- `DAY TO HOUR`
- `DAY TO MINUTE`
- `DAY TO SECOND`
- `HOUR TO MINUTE`
- `HOUR TO SECOND`
- `MINUTE TO SECOND`

The output of the INTERVAL literal is truncated to the smallest INTERVAL component specified. For example, when using a MINUTE qualifier, AWS Clean Rooms discards the time units smaller than MINUTE.

```
select INTERVAL '1 day 1 hour 1 minute 1.123 seconds' MINUTE
```

The resulting value is truncated to `'1 day 01:01:00'`.

fractional_precision

Optional parameter that specifies the number of fractional digits allowed in the interval. The *fractional_precision* argument should only be specified if your interval contains SECOND. For example, `SECOND(3)` creates an interval that allows only three fractional digits, such as 1.234 seconds. The maximum number of fractional digits is six.

The session configuration `interval_forbid_composite_literals` determines whether an error is returned when an interval is specified with both YEAR TO MONTH and DAY TO SECOND parts.

Interval arithmetic

You can use interval values with other datetime values to perform arithmetic operations. The following tables describe the available operations and what data type results from each operation.

Note

Operations that can produce both `date` and `timestamp` results do so based on the smallest unit of time involved in the equation. For example, when you add an `interval` to a `date` the result is a `date` if it is a YEAR TO MONTH interval, and a timestamp if it is a DAY TO SECOND interval.

Operations where the first operand is an `interval` produce the following results for the given second operand:

Operator	Date	Timestamp	Interval	Numeric
-	N/A	N/A	Interval	N/A

Operator	Date	Timestamp	Interval	Numeric
+	Date	Date/Timestamp	Interval	N/A
*	N/A	N/A	N/A	Interval
/	N/A	N/A	N/A	Interval

Operations where the first operand is a `date` produce the following results for the given second operand:

Operator	Date	Timestamp	Interval	Numeric
-	Numeric	Interval	Date/Timestamp	Date
+	N/A	N/A	N/A	N/A

Operations where the first operand is a `timestamp` produce the following results for the given second operand:

Operator	Date	Timestamp	Interval	Numeric
-	Numeric	Interval	Timestamp	Timestamp
+	N/A	N/A	N/A	N/A

Interval styles

- `postgres` – follows PostgreSQL style. This is the default.
- `postgres_verbose` – follows PostgreSQL verbose style.
- `sql_standard` – follows the SQL standard interval literals style.

The following command sets the interval style to `sql_standard`.

```
SET IntervalStyle to 'sql_standard';
```

postgres output format

The following is the output format for `postgres` interval style. Each numeric value can be negative.

```
'<numeric> <unit> [, <numeric> <unit> ...]'
```

```
select INTERVAL '1-2' YEAR TO MONTH::text

varchar
---------------
1 year 2 mons
```

```
select INTERVAL '1 2:3:4.5678' DAY TO SECOND::text

varchar
------------------
1 day 02:03:04.5678
```

postgres_verbose output format

postgres_verbose syntax is similar to postgres, but postgres_verbose outputs also contain the unit of time.

```
'[@] <numeric> <unit> [, <numeric> <unit> ...] [direction]'
```

```
select INTERVAL '1-2' YEAR TO MONTH::text

varchar
-----------------
@ 1 year 2 mons
```

```
select INTERVAL '1 2:3:4.5678' DAY TO SECOND::text

varchar
---------------------------
@ 1 day 2 hours 3 mins 4.56 secs
```

sql_standard output format

Interval year to month values are formatted as the following. Specifying a negative sign before the interval indicates the interval is a negative value and applies to the entire interval.

```
'[-]yy-mm'
```

Interval day to second values are formatted as the following.

```
'[-]dd hh:mm:ss.ffffff'
```

```
SELECT INTERVAL '1-2' YEAR TO MONTH::text

varchar
-------
1-2
```

```
select INTERVAL '1 2:3:4.5678' DAY TO SECOND::text

varchar
---------------
1 2:03:04.5678
```

Examples of interval data type

The following examples demonstrate how to use INTERVAL data types with tables.

```
create table sample_intervals (y2m interval month, h2m interval hour to minute);
insert into sample_intervals values (interval '20' month, interval '2 days
 1:1:1.123456' day to second);
select y2m::text, h2m::text from sample_intervals;


      y2m      |       h2m
---------------+-----------------
 1 year 8 mons | 2 days 01:01:00
```

```
update sample_intervals set y2m = interval '2' year where y2m = interval '1-8' year to
 month;
select * from sample_intervals;

   y2m   |       h2m
---------+-----------------
```

```
 2 years | 2 days 01:01:00
```

```
delete from sample_intervals where h2m = interval '2 1:1:0' day to second;
select * from sample_intervals;

 y2m | h2m
-----+-----
```

Examples of interval literals

The following examples are run with interval style set to `postgres`.

The following example demonstrates how to create an INTERVAL literal of 1 year.

```
select INTERVAL '1' YEAR

intervaly2m
---------------
1 years 0 mons
```

If you specify a *quoted-string* that exceeds the qualifier, the remaining units of time are truncated from the interval. In the following example, an interval of 13 months becomes 1 year and 1 month, but the remaining 1 month is left out because of the YEAR qualifier.

```
select INTERVAL '13 months' YEAR

intervaly2m
---------------
1 years 0 mons
```

If you use a qualifier lower than your interval string, leftover units are included.

```
select INTERVAL '13 months' MONTH

intervaly2m
---------------
1 years 1 mons
```

Specifying a precision in your interval truncates the number of fractional digits to the specified precision.

```
select INTERVAL '1.234567' SECOND (3)

intervald2s
--------------------------------
0 days 0 hours 0 mins 1.235 secs
```

If you don't specify a precision, AWS Clean Rooms uses the maximum precision of 6.

```
select INTERVAL '1.23456789' SECOND

intervald2s
-----------------------------------
0 days 0 hours 0 mins 1.234567 secs
```

The following example demonstrates how to create a ranged interval.

```
select INTERVAL '2:2' MINUTE TO SECOND

intervald2s
------------------------------
0 days 0 hours 2 mins 2.0 secs
```

Qualifiers dictate the units that you're specifying. For example, even though the following example uses the same *quoted-string* of '2:2' as the previous example, AWS Clean Rooms recognizes that it uses different units of time because of the qualifier.

```
select INTERVAL '2:2' HOUR TO MINUTE

intervald2s
------------------------------
0 days 2 hours 2 mins 0.0 secs
```

Abbreviations and plurals of each unit are also supported. For example, `5s`, `5 second`, and `5 seconds` are equivalent intervals. Supported units are years, months, hours, minutes, and seconds.

```
select INTERVAL '5s' SECOND

intervald2s
------------------------------
0 days 0 hours 0 mins 5.0 secs
```

```
select INTERVAL '5 HOURS' HOUR

intervald2s
------------------------------
0 days 5 hours 0 mins 0.0 secs
```

```
select INTERVAL '5 h' HOUR

intervald2s
------------------------------
0 days 5 hours 0 mins 0.0 secs
```

Examples of interval literals without qualifier syntax

> **Note**
>
> The following examples demonstrate using an interval literal without a `YEAR TO MONTH` or `DAY TO SECOND` qualifier. For information about using the recommended interval literal with a qualifier, see Interval data types and literals.

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

An interval literal 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`).

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

The following 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 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 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 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 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 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 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 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

State	Valid literal values	Storage
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    |
```

Boolean literals

The following rules are for working with Boolean literals that are supported by AWS Clean Rooms Spark SQL.

Use a Boolean literal to specify a Boolean value, such as TRUE or FALSE.

Syntax

```
TRUE | FALSE
```

Example

The following example shows a column with a specified value of TRUE .

```
SELECT TRUE AS col;
+----+
| col|
+----+
|true|
+----+
```

Binary type

Use the BINARY data type to store and manage fixed-length, uninterpreted binary data, providing efficient storage and comparison capabilities for specific use cases.

The BINARY data type stores a fixed number of bytes, regardless of the actual length of the data being stored. The maximum length is typically 255 bytes.

BINARY is used to store raw, uninterpreted binary data, such as images, documents, or other types of files. The data is stored exactly as it is provided, without any character encoding or interpretation. Binary data stored in BINARY columns is compared and sorted byte-by-byte, based on the actual binary values, rather than any character encoding or collation rules.

The following example query shows the binary representation of the string `"abc"`. Each character in the string is represented by its ASCII code in hexadecimal format: "a" is 0x61, "b" is 0x62, and "c" is 0x63. When combined, these hexadecimal values form the binary representation `"616263"`.

```
SELECT 'abc'::binary;
binary
---------
 616263
```

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.

Topics

ARRAY type

Use the ARRAY type to represent values comprising a sequence of elements with the type of `elementType`.

```
array(elementType, containsNull)
```

Use containsNull to indicate if elements in an ARRAY type can have null values.

MAP type

Use the MAP type to represent values comprising a set of key-value pairs.

```
map(keyType, valueType, valueContainsNull)
```

keyType: the data type of keys

valueType: the data type of values

Keys aren't allowed to have null values. Use valueContainsNull to indicate if values of a MAP type value can have null values.

STRUCT type

Use the STRUCT type to represent values with the structure described by a sequence of StructFields (fields).

```
struct(name, dataType, nullable)
```

StructField(name, dataType, nullable): Represents a field in a StructType.

dataType: the data type a field

name: the name of a field

Use nullable to indicate if values of these fields can have null values.

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.

Type compatibility and conversion

The following topics describe how type conversion rules and data type compatibility work in AWS Clean Rooms Spark SQL.

Topics

- Compatibility
- General compatibility and conversion rules
- Implicit conversion types

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.

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 or SHORT
	VARCHAR
CHAR	VARCHAR
DATE	CHAR
	VARCHAR

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

From type	To type
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	SMALLINT or SHORT
	VARCHAR
REAL (FLOAT4)	BIGINT or LONG
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	SMALLINT or SHORT
	VARCHAR
SMALLINT	BIGINT or LONG
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER (INT)
	REAL (FLOAT4)
	VARCHAR
TIME	VARCHAR
	TIMETZ

Note

Implicit conversions between DATE, TIME, TIMESTAMP_LTZ, TIMESTAMP_NTZ, 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.

AWS Clean Rooms Spark SQL commands

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

Topics

- CACHE TABLE
- Hints
- SELECT

CACHE TABLE

The CACHE TABLE command caches an existing table's data or creates and caches a new table containing query results.

Note

The cached data persists for the entire query.

The syntax, arguments, and some examples come from the Apache Spark SQL Reference.

Syntax

The CACHE TABLE command supports three syntax patterns:

With AS (without parentheses): Creates and caches a new table based on the query results.

```
CACHE TABLE cache_table_identifier AS query;
```

With AS and parentheses: Functions similarly to the first syntax but uses parentheses to explicitly group the query.

```
CACHE TABLE cache_table_identifier AS ( query );
```

Without AS: Caches an existing table, using the SELECT statement to filter which rows to cache.

```
CACHE TABLE cache_table_identifier query;
```

Where:

- All statements should end with a semicolon (;)
- `query` is typically a SELECT statement
- Parentheses around the query are optional with AS
- The AS keyword is optional

Parameters

cache_table_identifier

The name for the cached table. Can include an optional database name qualifier.

AS

A keyword used when creating and caching a new table from query results.

query

A SELECT statement or other query that defines the data to be cached.

Examples

In the following examples, the cached table persists for the entire query. After caching, subsequent queries that reference *`cache_table_identifier`* will read from the cached version rather than recomputing or reading from *`sourceTable`*. This can improve query performance for frequently accessed data.

Create and cache a filtered table from query results

The first example demonstrates how to create and cache a new table from query results. This command uses the AS keyword without parentheses around the SELECT statement. It creates a new table named 'cache_table_identifier' containing only the rows from 'sourceTable' where the status is 'active'. It runs the query, stores the results in the new table, and caches the new table's contents. The original 'sourceTable' remains unchanged, and subsequent queries must reference 'cache_table_identifier' to use the cached data.

```
CACHE TABLE cache_table_identifier AS
    SELECT * FROM sourceTable
    WHERE status = 'active';
```

Cache query results with parenthesized SELECT statements

The second example demonstrates how to cache the results of a query as a new table with a specified name (cache_table_identifier), using parentheses around the SELECT statement. This command creates a new table named 'cache_table_identifier' containing only the rows from 'sourceTable' where the status is 'active'. It runs the query, stores the results in the new table, and caches the new table's contents. The original 'sourceTable' remains unchanged. Subsequent queries must reference 'cache_table_identifier' to use the cached data.

```
CACHE TABLE cache_table_identifier AS (
    SELECT * FROM sourceTable
    WHERE status = 'active'
);
```

Cache an existing table with filter conditions

The third example demonstrates how to cache an existing table using a different syntax. This syntax, which omits the 'AS' keyword and parentheses, typically caches the specified rows from an existing table named 'cache_table_identifier' rather than creating a new table. The SELECT statement acts as a filter to determine which rows to cache.

> **Note**
>
> The exact behavior of this syntax varies across database systems. Always verify the correct syntax for your specific AWS service.

```
CACHE TABLE cache_table_identifier
SELECT * FROM sourceTable
WHERE status = 'active';
```

Hints

Hints for SQL analyses provide optimization directives that guide query execution strategies in AWS Clean Rooms, enabling you to improve query performance and reduce compute costs. Hints suggest how the Spark analytics engine should generate its execution plan.

Syntax

```
SELECT /*+ hint_name(parameters), hint_name(parameters) */ column_list
FROM table_name;
```

Hints are embedded in SQL queries using comment-style syntax and must be placed directly after the SELECT keyword.

Supported hint types

AWS Clean Rooms supports two categories of hints: Join hints and Partitioning hints.

Topics

- Join hints
- Partitioning hints

Join hints

Join hints suggest join strategies for query execution. The syntax, arguments, and some examples come from the Apache Spark SQL Reference for more information

BROADCAST

Suggests that AWS Clean Rooms use broadcast join. The join side with the hint will be broadcast regardless of autoBroadcastJoinThreshold. If both sides of the join have the broadcast hints, the one with the smaller size (based on stats) will be broadcast.

Aliases: BROADCASTJOIN, MAPJOIN

Parameters: Table identifiers (optional)

Examples:

```
-- Broadcast a specific table
SELECT /*+ BROADCAST(students) */ e.name, s.course
FROM employees e JOIN students s ON e.id = s.id;

-- Broadcast multiple tables
SELECT /*+ BROADCASTJOIN(s, d) */ *
FROM employees e
JOIN students s ON e.id = s.id
JOIN departments d ON e.dept_id = d.id;
```

MERGE

Suggests that AWS Clean Rooms use shuffle sort merge join.

Aliases: SHUFFLE_MERGE, MERGEJOIN

Parameters: Table identifiers (optional)

Examples:

```
-- Use merge join for a specific table
SELECT /*+ MERGE(employees) */ *
FROM employees e JOIN students s ON e.id = s.id;

-- Use merge join for multiple tables
SELECT /*+ MERGEJOIN(e, s, d) */ *
FROM employees e
JOIN students s ON e.id = s.id
JOIN departments d ON e.dept_id = d.id;
```

SHUFFLE_HASH

Suggests that AWS Clean Rooms use shuffle hash join. If both sides have the shuffle hash hints, the query optimizer chooses the smaller side (based on stats) as the build side.

Parameters: Table identifiers (optional)

Examples:

```
-- Use shuffle hash join
SELECT /*+ SHUFFLE_HASH(students) */ *
```

```
FROM employees e JOIN students s ON e.id = s.id;
```

SHUFFLE_REPLICATE_NL

Suggests that AWS Clean Rooms use shuffle-and-replicate nested loop join.

Parameters: Table identifiers (optional)

Examples:

```
-- Use shuffle-replicate nested loop join
SELECT /*+ SHUFFLE_REPLICATE_NL(students) */ *
FROM employees e JOIN students s ON e.id = s.id;
```

Troubleshooting Hints in Spark SQL

The following table shows common scenarios where hints are not applied in SparkSQL. For additional information, see the section called "Considerations and limitations".

Use Case	Example Query
Table reference not found	`SELECT /*+ BROADCAST(fake_table) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
Table not participating in join operation	`SELECT /*+ BROADCAST(s) */ * FROM students s WHERE s.age > 25;`
Table reference in nested subquery	`SELECT /*+ BROADCAST(s) */ * FROM employees e INNER JOIN (SELECT * FROM students s WHERE s.age > 20) sub ON e.eid = sub.sid;`
Column name instead of table reference	`SELECT /*+ BROADCAST(e.eid) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

<table>
<tr><th>Use Case</th><th>Example Query</th></tr>
<tr><td>Hint without required parameters</td><td><pre>SELECT /*+ BROADCAST */ *
FROM employees e
INNER JOIN students s ON e.eid = s.sid;</pre></td></tr>
<tr><td>Base table name instead of table alias</td><td><pre>SELECT /*+ BROADCAST(employees) */ *
FROM employees e
INNER JOIN students s ON e.eid = s.sid;</pre></td></tr>
</table>

Partitioning hints

Partitioning hints control data distribution across executor nodes. When multiple partitioning hints are specified, multiple nodes are inserted into the logical plan, but the leftmost hint is picked by the optimizer.

COALESCE

Reduces the number of partitions to the specified number of partitions.

Parameters: Numeric value (required) - must be a positive integer between 1 and 2147483647

Examples:

```
-- Reduce to 5 partitions
SELECT /*+ COALESCE(5) */ employee_id, salary
FROM employees;
```

REPARTITION

Repartitions data to the specified number of partitions using the specified partitioning expressions. Uses round-robin distribution.

Parameters:

- Numeric value (optional) - number of partitions; Must be a positive integer between 1 and 2147483647
- Column identifiers (optional) - columns to partition by; These columns must exist in the input schema.

- If both are specified, numeric value must come first

Examples:

```
-- Repartition to 10 partitions
SELECT /*+ REPARTITION(10) */ *
FROM employees;

-- Repartition by column
SELECT /*+ REPARTITION(department) */ *
FROM employees;

-- Repartition to 8 partitions by department
SELECT /*+ REPARTITION(8, department) */ *
FROM employees;

-- Repartition by multiple columns
SELECT /*+ REPARTITION(8, department, location) */ *
FROM employees;
```

REPARTITION_BY_RANGE

Repartitions data to the specified number of partitions using range partitioning on the specified columns.

Parameters:

- Numeric value (optional) - number of partitions; Must be a positive integer between 1 and 2147483647
- Column identifiers (optional) - columns to partition by; These columns must exist in the input schema.
- If both are specified, numeric value must come first

Examples:

```
SELECT /*+ REPARTITION_BY_RANGE(10) */ *
FROM employees;

-- Repartition by range on age column
SELECT /*+ REPARTITION_BY_RANGE(age) */ *
```

```
FROM employees;

-- Repartition to 5 partitions by range on age
SELECT /*+ REPARTITION_BY_RANGE(5, age) */ *
FROM employees;

-- Repartition by range on multiple columns
SELECT /*+ REPARTITION_BY_RANGE(5, age, salary) */ *
FROM employees;
```

REBALANCE

Rebalances the query result output partitions so that every partition is of a reasonable size (not too small and not too big). This is a best-effort operation: if there are skews, AWS Clean Rooms will split the skewed partitions to make them not too big. This hint is useful when you need to write the result of a query to a table to avoid too small or too big files.

Parameters:

- Numeric value (optional) - number of partitions; Must be a positive integer between 1 and 2147483647
- Column identifiers (optional) - columns must appear in the SELECT output list
- If both are specified, numeric value must come first

Examples:

```
-- Rebalance to 10 partitions
SELECT /*+ REBALANCE(10) */ employee_id, name
FROM employees;

-- Rebalance by specific columns in output
SELECT /*+ REBALANCE(employee_id, name) */ employee_id, name
FROM employees;

-- Rebalance to 8 partitions by specific columns
SELECT /*+ REBALANCE(8, employee_id, name) */ employee_id, name, department
FROM employees;
```

Combining multiple hints

You can specify multiple hints in a single query by separating them with commas:

```
-- Combine join and partitioning hints
SELECT /*+ BROADCAST(d), REPARTITION(8) */ e.name, d.dept_name
FROM employees e JOIN departments d ON e.dept_id = d.id;

-- Multiple join hints
SELECT /*+ BROADCAST(s), MERGE(d) */ *
FROM employees e
JOIN students s ON e.id = s.id
JOIN departments d ON e.dept_id = d.id;

-- Hints within separate hint blocks within the same query
SELECT /*+ REPARTITION(100) */ /*+ COALESCE(500) */ /*+ REPARTITION_BY_RANGE(3, c) */ *
 FROM t;
```

Considerations and limitations

- Hints are optimization suggestions, not commands. The query optimizer may ignore hints based on resource constraints or execution conditions.
- Hints are embedded directly in SQL query strings for both CreateAnalysisTemplate and StartProtectedQuery APIs.
- Hints must be placed directly after the SELECT keyword.
- Named parameters are not supported with hints and will throw an exception.
- Column names in REPARTITION amd REPARTITION_BY_RANGE hints must exist in the input schema.
- Column names in REBALANCE hints must appear in the SELECT output list.
- Numeric parameters must be positive integers between 1 and 2147483647. Scientific notations like *1e1* are not supported
- Hints are not supported in Differential Privacy SQL queries.
- Hints for SQL queries are not supported in PySpark jobs. To provide directives for execution plans in a PySpark job, use the data frame API. See Apache Spark DataFrame API Docs for more information.

SELECT

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

The following SELECT SQL commands, clauses, and set operators are supported in AWS Clean Rooms Spark SQL:

Topics

The syntax, arguments, and some examples come from the Apache Spark SQL Reference.

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

Parameters

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 can't use an alias in other expressions defined after it in the same target list.

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, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, or EXCEPT ALL

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
- Parameters
- Usage notes

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 clause

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. Left and right outer joins retain values from one of the joined tables when no match is found in the other table.

The combination of the JOIN type and the join condition determines which rows are included in the final result set. The SELECT and WHERE clauses then control which columns are returned and how the rows are filtered. Understanding the different JOIN types and how to use them effectively is a crucial skill in SQL, because it allows you to combine data from multiple tables in a flexible and powerful way.

Syntax

```
SELECT column1, column2, ..., columnn
FROM table1
join_type table2
ON table1.column = table2.column;
```

Parameters

SELECT column1, column2, ..., columnN

The columns you want to include in the result set. You can select columns from either or both of the tables involved in the JOIN.

FROM table1

The first (left) table in the JOIN operation.

[JOIN | INNER JOIN | LEFT [OUTER] JOIN | RIGHT [OUTER] JOIN | FULL [OUTER] JOIN] table2:

The type of JOIN to be performed. JOIN or INNER JOIN returns only the rows with matching values in both tables.

LEFT [OUTER] JOIN returns all rows from the left table, with matching rows from the right table.

RIGHT [OUTER] JOIN returns all rows from the right table, with matching rows from the left table.

FULL [OUTER] JOIN returns all rows from both tables, regardless of whether there is a match or not.

CROSS JOIN creates a Cartesian product of the rows from the two tables.

ON table1.column = table2.column

The join condition, which specifies how the rows in the two tables are matched. The join condition can be based on one or more columns.

WHERE condition:

An optional clause that can be used to filter the result set further, based on a specified condition.

Example

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
```

Join types

INNER

This is the default join type. Returns the rows that have matching values in both table references.

The INNER JOIN is the most common type of join used in SQL. It's a powerful way to combine data from multiple tables based on a common column or set of columns.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

The following query will return all the rows where there is a matching customer_id value between the customers and orders tables. The result set will contain the customer_id, name, order_id, and order_date columns.

```
SELECT customers.customer_id, customers.name, orders.order_id, orders.order_date
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id;
```

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 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 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
```

LEFT [OUTER]

Returns all values from the left table reference and the matched values from the right table reference, or appends NULL if there is no match. It's also referred to as a *left outer join*.

It returns all the rows from the left (first) table, and the matching rows from the right (second) table. If there is no match in the right table, the result set will contain NULL values for the columns from the right table. The OUTER keyword can be omitted, and the join can be written as simply LEFT JOIN. The opposite of a LEFT OUTER JOIN is a RIGHT OUTER JOIN, which returns all the rows from the right table and the matching rows from the left table.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
LEFT [OUTER] JOIN table2
```

```
ON table1.column = table2.column;
```

The following query will return all the rows from the customers table, along with the matching rows from the orders table. If a customer has no orders, the result set will still include that customer's information, with NULL values for the order_id and order_date columns.

```
SELECT customers.customer_id, customers.name, orders.order_id, orders.order_date
FROM customers
LEFT OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

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 didn't 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
```

RIGHT [OUTER]

Returns all values from the right table reference and the matched values from the left table reference, or appends NULL if there is no match. It's also referred to as a *right outer join*.

It returns all the rows from the right (second) table, and the matching rows from the left (first) table. If there is no match in the left table, the result set will contain NULL values for the columns from the left table. The OUTER keyword can be omitted, and the join can be written as simply RIGHT JOIN. The opposite of a RIGHT OUTER JOIN is a LEFT OUTER JOIN, which returns all the rows from the left table and the matching rows from the right table.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

The following query will return all the rows from the customers table, along with the matching rows from the orders table. If a customer has no orders, the result set will still include that customer's information, with NULL values for the order_id and order_date columns.

```
SELECT orders.order_id, orders.order_date, customers.customer_id, customers.name
FROM orders
RIGHT OUTER JOIN customers
ON orders.customer_id = customers.customer_id;
```

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
```

FULL [OUTER]

Returns all values from both relations, appending NULL values on the side that doesn't have a match. It's also referred to as a *full outer join*.

It returns all the rows from both the left and right tables, regardless of whether there is a match or not. If there is no match, the result set will contain NULL values for the columns from the table that doesn't have a matching row. The OUTER keyword can be omitted, and the join can be written as simply FULL JOIN. The FULL OUTER JOIN is less commonly used than the LEFT OUTER JOIN or RIGHT OUTER JOIN, but it can be useful in certain scenarios where you need to see all the data from both tables, even if there are no matches.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

The following query will return all the rows from both the customers and orders tables. If a customer has no orders, the result set will still include that customer's information, with NULL values for the order_id and order_date columns. If an order has no associated customer, the result set will include that order, with NULL values for the customer_id and name columns.

```
SELECT customers.customer_id, customers.name, orders.order_id, orders.order_date
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

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 didn't 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
```

[LEFT] SEMI

Returns values from the left side of the table reference that has a match with the right. It's also referred to as a *left semi join*.

It returns only the rows from the left (first) table that have a matching row in the right (second) table. It does not return any columns from the right table - only the columns from the left table. The LEFT SEMI JOIN is useful when you want to find the rows in one table that have a match in another table, without needing to return any data from the second table. The LEFT SEMI JOIN is a more efficient alternative to using a subquery with an IN or EXISTS clause.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
LEFT SEMI JOIN table2
ON table1.column = table2.column;
```

The following query will return only the customer_id and name columns from the customers table, for the customers who have at least one order in the orders table. The result set won't include any columns from the orders table.

```
SELECT customers.customer_id, customers.name
FROM customers
LEFT SEMI JOIN orders
```

```
ON customers.customer_id = orders.customer_id;
```

CROSS JOIN

Returns the Cartesian product of two relations. This means that the result set will contain all possible combinations of rows from the two tables, without any condition or filter applied.

The CROSS JOIN is useful when you need to generate all possible combinations of data from two tables, such as in the case of creating a report that displays all possible combinations of customer and product information. The CROSS JOIN is different from other join types (INNER JOIN, LEFT JOIN, etc.) because it doesn't have a join condition in the ON clause. The join condition isn't required for a CROSS JOIN.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
CROSS JOIN table2;
```

The following query will return a result set that contains all possible combinations of customer_id, customer_name, product_id, and product_name from the customers and products tables. If the customers table has 10 rows and the products table has 20 rows, the result set of the CROSS JOIN will contain 10 x 20 = 200 rows.

```
SELECT customers.customer_id, customers.name, products.product_id,
 products.product_name
FROM customers
CROSS JOIN products;
```

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
```

ANTI JOIN

Returns the values from the left table reference that have no match with the right table reference. It's also referred to as a *left anti join*.

The ANTI JOIN is a useful operation when you want to find the rows in one table that don't have a match in another table.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
LEFT ANTI JOIN table2
ON table1.column = table2.column;
```

The following query will return all the customers who haven't placed any orders.

```
SELECT customers.customer_id, customers.name
FROM customers
```

```
LEFT ANTI JOIN orders
ON customers.customer_id = orders.customer_id
WHERE orders.order_id IS NULL;
```

NATURAL

Specifies that the rows from the two relations will implicitly be matched on equality for all columns with matching names.

It automatically matches columns with the same name and data type between the two tables. It doesn't require you to explicitly specify the join condition in the ON clause. It combines all the matching columns between the two tables into the result set.

The NATURAL JOIN is a convenient shorthand when the tables you're joining have columns with the same names and data types. However, it's generally recommended to use the more explicit INNER JOIN ... ON syntax to make the join conditions more explicit and easier to understand.

Syntax:

```
SELECT column1, column2, ..., columnn
FROM table1
NATURAL JOIN table2;
```

The following example is a natural join between two tables, `employees` and `departments`, with the following columns:

- `employees` table: `employee_id`, `first_name`, `last_name`, `department_id`
- `departments` table: `department_id`, `department_name`

The following query will return a result set that includes the first name, last name, and department name for all matching rows between the two tables, based on the `department_id` column.

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees e
NATURAL JOIN departments d;
```

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
```

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)
```

VALUES clause

The VALUES clause is used to provide a set of row values directly in the query, without the need to reference a table.

The VALUES clause can be used in the following scenarios:

- You can use the VALUES clause in an INSERT INTO statement to specify the values for the new rows being inserted into a table.
- You can use the VALUES clause on its own to create a temporary result set, or inline table, without the need to reference a table.
- You can combine the VALUES clause with other SQL clauses, such as WHERE, ORDER BY, or LIMIT, to filter, sort, or limit the rows in the result set.

This clause is particularly useful when you need to insert, query, or manipulate a small set of data directly in your SQL statement, without the need to create or reference a permanent table. It allows you to define the column names and the corresponding values for each row, giving you the flexibility to create temporary result sets or insert data on the fly, without the overhead of managing a separate table.

Syntax

```
VALUES ( expression [ , ... ] ) [ table_alias ]
```

Parameters

expression

An expression that specifies a combination of one or more values, operators and SQL functions that results in a value.

table_alias

An alias that specifies a temporary name with an optional column name list.

Example

The following example creates an inline table, temporary table-like result set with two columns, `col1` and `col2`. The single row in the result set contains the values `"one"` and `1`, respectively. The

SELECT * FROM part of the query simply retrieves all the columns and rows from this temporary result set. The column names (col1 and col2) are automatically generated by the database system, because the VALUES clause doesn't explicitly specify the column names.

```
SELECT * FROM VALUES ("one", 1);
+----+----+
|col1|col2|
+----+----+
| one|   1|
+----+----+
```

If you want to define custom column names, you can do so by using an AS clause after the VALUES clause, like this:

```
SELECT * FROM (VALUES ("one", 1)) AS my_table (name, id);
+------+----+
| name | id |
+------+----+
| one  |  1 |
+------+----+
```

This would create a temporary result set with the column names name and id, instead of the default col1 and col2.

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 AWS Clean Rooms Spark SQL functions.

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 *set operators* are used to compare and merge the results of two separate query expressions.

AWS Clean Rooms Spark SQL supports the following set operators listed in the following table.

Set operator
INTERSECT
INTERSECT ALL
EXCEPT
EXCEPT ALL
UNION
UNION ALL

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 can't 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

Syntax

```
subquery1
{ { UNION [ ALL | DISTINCT ] |
              INTERSECT [ ALL | DISTINCT ] |
              EXCEPT [ ALL | DISTINCT ] } subquery2 } [...] }
```

Parameters

subquery1, subquery2

A query expression that corresponds, in the form of its select list, to a second query expression that follows the UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, or EXCEPT ALL 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 [ALL | DISTINCT]

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

INTERSECT [ALL | DISTINCT]

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

EXCEPT [ALL | DISTINCT]

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.

EXCEPT ALL doesn't remove duplicates from the result rows.

MINUS and EXCEPT are exact synonyms.

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
- Parameters
- Usage notes
- Examples with ORDER BY

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 Example 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);
```

AWS Clean Rooms Spark SQL functions

AWS Clean Rooms Spark SQL supports the following SQL functions:

Topics

Aggregate functions

Aggregate functions in AWS Clean Rooms Spark SQL are used to perform calculations or operations on a group of rows and return a single value. They are essential for data analysis and summarization tasks.

AWS Clean Rooms Spark SQL supports the following aggregate functions:

Topics

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 (expression[, isIgnoreNull] )
```

Arguments

expression

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

isIgnoreNull

A boolean that determines if the function should return only non-null values.

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
```

APPROX COUNT_DISTINCT function

APPROX COUNT_DISTINCT provides an efficient way to estimate the number of unique values in a column or dataset.

Syntax

```
approx_count_distinct(expr[, relativeSD])
```

Arguments

expr

The expression or column for which you want to estimate the number of unique values.

It can be a single column, a complex expression, or a combination of columns.

relativeSD

An optional parameter that specifies the desired relative standard deviation of the estimate.

It is a value between 0 and 1, representing the maximum acceptable relative error of the estimate. A smaller relativeSD value will result in a more accurate but slower estimation.

If this parameter isn't provided, a default value (usually around 0.05 or 5%) is used.

Returns

Returns the estimated cardinality by HyperLogLog++. relativeSD defines the maximum relative standard deviation allowed.

Example

The following query estimates the number of unique values in the `col1` column, with a relative standard deviation of 1% (0.01).

```
SELECT approx_count_distinct(col1, 0.01)
```

The following query estimates that there are 3 unique values in the `col1` column (the values 1, 2, and 3).

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

APPROX PERCENTILE function

APPROX PERCENTILE is used to estimate the percentile value of a given expression or column without having to sort the entire dataset. This function is useful in scenarios where you need to quickly understand the distribution of a large dataset or track percentile-based metrics, without the computational overhead of performing an exact percentile calculation. However, it's important to understand the trade-offs between speed and accuracy, and to choose the appropriate error tolerance based on the specific requirements of your use case.

Syntax

```
APPROX_PERCENTILE(expr, percentile [, accuracy])
```

Arguments

expr

The expression or column for which you want to estimate the percentile value.

It can be a single column, a complex expression, or a combination of columns.

percentile

The percentile value you want to estimate, expressed as a value between 0 and 1.

For example, 0.5 would correspond to the 50th percentile (median).

accuracy

An optional parameter that specifies the desired accuracy of the percentile estimate. It is a value between 0 and 1, representing the maximum acceptable relative error of the estimate. A smaller `accuracy` value will result in a more precise but slower estimation. If this parameter isn't provided, a default value (usually around 0.05 or 5%) is used.

Returns

Returns the approximate percentile of the numeric or ANSI interval column col which is the smallest value in the ordered col values (sorted from least to greatest) such that no more than percentage of col values is less than the value or equal to that value.

The value of percentage must be between 0.0 and 1.0. The accuracy parameter (default: 10000) is a positive numeric literal which controls approximation accuracy at the cost of memory.

Higher value of accuracy yields better accuracy, `1.0/accuracy` is the relative error of the approximation.

When percentage is an array, each value of the percentage array must be between 0.0 and 1.0. In this case, returns the approximate percentile array of column col at the given percentage array.

Examples

The following query estimates the 95th percentile of the `response_time` column, with a maximum relative error of 1% (0.01).

```
SELECT APPROX_PERCENTILE(response_time, 0.95, 0.01) AS p95_response_time
FROM my_table;
```

The following query estimates the 50th, 40th, and 10th percentile values of the `col` column in the `tab` table.

```
SELECT approx_percentile(col, array(0.5, 0.4, 0.1), 100) FROM VALUES (0), (1), (2),
 (10) AS tab(col)
```

The following query estimates the 50th percentile (median) of the values in the col column.

```
SELECT approx_percentile(col, 0.5, 100) FROM VALUES (0), (6), (7), (9), (10) AS
 tab(col)
```

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
- FLOAT

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
```

CARDINALITY function

The CARDINALITY function returns the size of an ARRAY or MAP expression (*expr*).

This function is useful to find the size or length of an array.

Syntax

```
cardinality(expr)
```

Arguments

expr

An ARRAY or MAP expression.

Returns

Returns the size of an array or a map (INTEGER).

The function returns `NULL` for null input if `sizeOfNull` is set to `false` or `enabled` is set to `true`.

Otherwise, the function returns `-1` for null input. With the default settings, the function returns `-1` for null input.

Example

The following query calculates the cardinality, or the number of elements, in the given array. The array ('b', 'd', 'c', 'a') has 4 elements, so the output of this query would be 4.

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

COLLECT_LIST function

The COLLECT_LIST function collects and returns a list of non-unique elements.

This type of function is useful when you want to collect multiple values from a set of rows into a single array or list data structure.

> **Note**
>
> The function is non-deterministic because the order of the collected results depends on the order of the rows, which may be non-deterministic after a shuffle operation is performed.

Syntax

```
collect_list(expr)
```

Arguments

expr

An expression of any type.

Returns

Returns an ARRAY of the argument type. The order of elements in the array is non-deterministic.

NULL values are excluded.

If DISTINCT is specified, the function collects only unique values and is a synonym for `collect_set` aggregate function.

Example

The following query collects all the values from the col column into a list. The VALUES clause is used to create an inline table with three rows, where each row has a single column col with the values 1, 2, and 1 respectively. The `collect_list()` function is then used to aggregate all the values from the col column into a single array. The output of this SQL statement would be the array `[1,2,1]`, which contains all the values from the col column in the order they appeared in the input data.

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

COLLECT_SET function

The COLLECT_SET function collects and returns a set of unique elements.

This function is useful when you want to collect all the distinct values from a set of rows into a single data structure, without including any duplicates.

> **Note**
>
> The function is non-deterministic because the order of the collected results depends on the order of the rows, which may be non-deterministic after a shuffle operation is performed.

Syntax

```
collect_set(expr)
```

Arguments

expr

An expression of any type except MAP.

Returns

Returns an ARRAY of the argument type. The order of elements in the array is non-deterministic.

NULL values are excluded.

Example

The following query collects all the unique values from the col column into a set. The VALUES clause is used to create an inline table with three rows, where each row has a single column col with the values 1, 2, and 1 respectively. The `collect_set()` function is then used to aggregate all the unique values from the col column into a single set. The output of this SQL statement would be the set `[1,2]`, which contains the unique values from the col column. The duplicate value of 1 is only included once in the result.

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

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 (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.

Syntax

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

```
COUNT ( [ DISTINCT | ALL ] 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.

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
...
```

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 any numerical data type.

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

Syntax

```
MEDIAN ( median_expression )
```

Arguments

median_expression

The target column or expression that the function operates on.

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 any numerical data type.

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 function

The PERCENTILE function is used to calculates the exact percentile value by first sorting the values in the `col` column and then finding the value at the specified `percentage`.

The PERCENTILE function is useful when you need to calculate the exact percentile value and the computational cost is acceptable for your use case. It provides more accurate results than the APPROX_PERCENTILE function, but may be slower, especially for large datasets.

In contrast, the APPROX_PERCENTILE function is a more efficient alternative that can provide an estimate of the percentile value with a specified error tolerance, making it more suitable for scenarios where speed is a higher priority than absolute precision.

Syntax

```
percentile(col, percentage [, frequency])
```

Arguments

col

The expression or column for which you want to calculate the percentile value.

percentage

The percentile value you want to calculate, expressed as a value between 0 and 1.

For example, 0.5 would correspond to the 50th percentile (median).

frequency

An optional parameter that specifies the frequency or weight of each value in the `col` column. If provided, the function will calculate the percentile based on the frequency of each value.

Returns

Returns the exact percentile value of numeric or ANSI interval column col at the given percentage.

The value of percentage must be between 0.0 and 1.0.

The value of frequency should be positive integral

Example

The following query finds the value that is greater than or equal to 30% of the values in the `col` column. Since the values are 0 and 10, the 30th percentile is 3.0, because it is the value that is greater than or equal to 30% of the data.

```
SELECT percentile(col, 0.3) FROM VALUES (0), (10) AS tab(col);
 3.0
```

SKEWNESS function

The SKEWNESS function returns the skewness value calculated from values of a group.

Skewness is a statistical measure that describes the asymmetry or lack of symmetry in a dataset. It provides information about the shape of the data distribution.

This function can be useful in understanding the statistical properties of a dataset and informing further analysis or decision-making.

Syntax

```
skewness(expr)
```

Arguments

expr

An expression that evaluates to a numeric.

Returns

Returns DOUBLE.

If DISTINCT is specified, the function operates only on a unique set of *expr* values.

Examples

The following query calculates the skewness of the values in the `col` column. In this example, the `VALUES` clause is used to create an inline table with four rows, where each row has a single column `col` with the values -10, -20, 100, and 1000. The `skewness()` function is then used to calculate the skewness of the values in the `col` column. The result, 1.1135657469022011, represents the degree and direction of skewness in the data. A positive skewness value indicates that the data is skewed to the right, with the bulk of the values concentrated on the left side of the distribution. A negative skewness value indicates that the data is skewed to the left, with the bulk of the values concentrated on the right side of the distribution.

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

The following query calculates the skewness of the values in the col column. Similar to the previous example, the `VALUES` clause is used to create an inline table with four rows, where each row has a single column `col` with the values -1000, -100, 10, and 20. The `skewness()` function is then used to calculate the skewness of the values in the `col` column. The result, -1.1135657469022011, represents the degree and direction of skewness in the data. In this case, the negative skewness value indicates that the data is skewed to the left, with the bulk of the values concentrated on the right side of the distribution.

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

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 numeric 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 (DISTINCT column )
```

Arguments

column

The target column that the function operates on. The column is any numeric data types.

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

Creates an array with the given elements.

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 an ARRAY with the elements in the expression.

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_CONTAINS function

The ARRAY_CONTAINS function can be used to perform basic membership checks on array data structures. The ARRAY_CONTAINS function is useful when you need to check if a specific value is present within an array.

Syntax

```
array_contains(array, value)
```

Arguments

array

An ARRAY to be searched.

value

An expression with a type sharing a least common type with the array elements.

Return type

The ARRAY_CONTAINS function returns a BOOLEAN.

If value is NULL, the result is NULL.

If any element in array is NULL, the result is NULL if value is not matched to any other element.

Examples

The following example checks if the array `[1, 2, 3]` contains the value 4. Since the array `[1, 2, 3]` doesn't contain the value 4, the array_contains function returns `false`.

```
SELECT array_contains(array(1, 2, 3), 4)
false
```

The following example checks if the array `[1, 2, 3]` contains the value 2. Since the array `[1, 2, 3]` does contain the value 2, the array_contains function returns `true`.

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

ARRAY_DISTINCT function

The ARRAY_DISTINCT function can be used to remove duplicate values from an array. The ARRAY_DISTINCT function is useful when you need to remove duplicates from an array and work with only the unique elements. This can be helpful in scenarios where you want to perform operations or analyses on a dataset without the interference of repeated values.

Syntax

```
array_distinct(array)
```

Arguments

array

An ARRAY expression.

Return type

The ARRAY_DISTINCT function returns an ARRAY that contains only the unique elements from the input array.

Examples

In this example, the input array `[1, 2, 3, null, 3]` contains a duplicate value of 3. The `array_distinct` function removes this duplicate value 3 and returns a new array with the unique elements: `[1, 2, 3, null]`.

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

In this example, the input array `[1, 2, 2, 3, 3, 3]` contains duplicate values of 2 and 3. The `array_distinct` function removes these duplicates and returns a new array with the unique elements: `[1, 2, 3]`.

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

ARRAY_EXCEPT function

The ARRAY_EXCEPT function takes two arrays as arguments and returns a new array that contains only the elements that are present in the first array but not the second array.

The ARRAY_EXCEPT is useful when you need to find the elements that are unique to one array compared to another. This can be helpful in scenarios where you need to perform set-like operations on arrays, such as finding the difference between two sets of data.

Syntax

```
array_except(array1, array2)
```

Arguments

array1

An ARRAY of any type with comparable elements.

array2

An ARRAY of elements sharing a least common type with the elements of *array1*.

Return type

The ARRAY_EXCEPT function returns an ARRAY of matching type to *array1* with no duplicates.

Examples

In this example, the first array `[1, 2, 3]` contains the elements 1, 2, and 3. The second array `[2, 3, 4]` contains the elements 2, 3, and 4. The `array_except` function removes the elements 2 and 3 from the first array, since they're also present in the second array. The resulting output is the array `[1]`.

```
SELECT array_except(array(1, 2, 3), array(2, 3, 4))
  [1]
```

In this example, the first array `[1, 2, 3]` contains the elements 1, 2, and 3. The second array `[1, 3, 5]` contains the elements 1, 3, and 5. The `array_except` function removes the elements 1 and 3 from the first array, since they're also present in the second array. The resulting output is the array `[2]`.

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

ARRAY_INTERSECT function

The ARRAY_INTERSECT function takes two arrays as arguments and returns a new array that contains the elements that are present in both input arrays. This function is useful when you need to find the common elements between two arrays. This can be helpful in scenarios where you need to perform set-like operations on arrays, such as finding the intersection between two sets of data.

Syntax

```
array_intersect(array1, array2)
```

Arguments

array1

An ARRAY of any type with comparable elements.

array2

An ARRAY of elements sharing a least common type with the elements of array1.

Return type

The ARRAY_INTERSECT function returns an ARRAY of matching type to array1 with no duplicates and elements contained in both array1 and array2.

Examples

In this example, the first array `[1, 2, 3]` contains the elements 1, 2, and 3. The second array `[1, 3, 5]` contains the elements 1, 3, and 5. The ARRAY_INTERSECT function identifies the common elements between the two arrays, which are 1 and 3. The resulting output array is `[1, 3]`.

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

ARRAY_JOIN function

The ARRAY_JOIN function takes two arguments: The first argument is the input array that will be joined. The second argument is the separator string that will be used to concatenate the array elements. This function is useful when you need to convert an array of strings (or any other data type) into a single concatenated string. This can be helpful in scenarios where you want to present an array of values as a single formatted string, such as for display purposes or for use in further processing.

Syntax

```
array_join(array, delimiter[, nullReplacement])
```

Arguments

array

Any ARRAY type, but its elements are interpreted as strings.

delimiter

A STRING used to separate the concatenated array elements.

nullReplacement

A STRING used to express a NULL value in the result.

Return type

The ARRAY_JOIN function returns a STRING where the elements of array are separated by delimiter and null elements are substituted for `nullReplacement`. If `nullReplacement` is omitted, `null` elements are filtered out. If any argument is `NULL`, the result is `NULL`.

Examples

In this example, the ARRAY_JOIN function takes the array `['hello', 'world']` and joins the elements using the separator `' '` (a space character). The resulting output is the string `'hello world'`.

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

In this example, the ARRAY_JOIN function takes the array `['hello', null, 'world']` and joins the elements using the separator `' '` (a space character). The `null` value is replaced with the provided replacement string `','` (a comma). The resulting output is the string `'hello , world'`.

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

ARRAY_REMOVE function

The ARRAY_REMOVE function takes two arguments: The first argument is the input array from which the elements will be removed. The second argument is the value that will be removed from the array. This function is useful when you need to remove specific elements from an array. This can be helpful in scenarios where you need to perform data cleaning or preprocessing on an array of values.

Syntax

```
array_remove(array, element)
```

Arguments

array

An ARRAY.

element

An expression of a type sharing a least common type with the elements of array.

Return type

The ARRAY_REMOVE function returns the result type matched the type of the array. If the element to be removed is `NULL`, the result is `NULL`.

Examples

In this example, the ARRAY_REMOVE function takes the array `[1, 2, 3, null, 3]` and removes all occurrences of the value 3. The resulting output is the array `[1, 2, null]`.

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

ARRAY_UNION function

The ARRAY_UNION function takes two arrays as arguments and returns a new array that contains the unique elements from both input arrays. This function is useful when you need to combine two arrays and eliminate any duplicate elements. This can be helpful in scenarios where you need to perform set-like operations on arrays, such as finding the union between two sets of data.

Syntax

```
array_union(array1, array2)
```

Arguments

array1

An ARRAY.

array2

An ARRAY of the same type as *array1*.

Return type

The ARRAY_UNION function returns an ARRAY of the same type as array.

Example

In this example, the first array `[1, 2, 3]` contains the elements 1, 2, and 3. The second array `[1, 3, 5]` contains the elements 1, 3, and 5. The ARRAY_UNION function combines the unique elements from both arrays, resulting in the output array `[1, 2, 3, 5]`. T

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

EXPLODE function

The EXPLODE function is used to transform a single row with an array or map column into multiple rows, where each row corresponds to a single element from the array or map.

Syntax

```
explode(expr)
```

Arguments

expr

An array expression or a map expression.

Return type

The EXPLODE function returns a set of rows, where each row represents a single element from the input array or map.

The data type of the output rows depends on the data type of the elements in the input array or map.

Examples

The following example takes the single-row array [10, 20] and transforms it into two separate rows, each containing one of the array elements (10 and 20).

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

In the first example, the input array was directly passed as an argument to `explode()`. In this example, the input array is specified using the => syntax, where the column name (`collection`) is explicitly provided.

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

Both approaches are valid and achieve the same result, but the second syntax can be more useful when you need to explode a column from a larger dataset, rather than just a simple array literal.

FLATTEN function

The FLATTEN function is used to "flatten" a nested array structure into a single flat array.

Syntax

```
flatten(arrayOfArrays)
```

Arguments

arrayOfArrays

An array of arrays.

Return type

The FLATTEN function returns an array.

Example

In this example, the input is a nested array with two inner arrays, and the output is a single flat array containing all the elements from the inner arrays. The FLATTEN function takes the nested array `[[1, 2], [3, 4]]` and combines all the elements into a single array `[1, 2, 3, 4]`.

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

Conditional expressions

In SQL, conditional expressions are used to make decisions based on certain conditions. They allow you to control the flow of your SQL statements and return different values or perform different actions based on the evaluation of one or more conditions.

AWS Clean Rooms supports the following conditional expressions:

Topics

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 expression

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)
```

IF expression

The IF conditional function returns one of two values based on a condition.

This function is a common control flow statement used in SQL to make decisions and return different values based on the evaluation of a condition. It's useful for implementing simple if-else logic within a query.

Syntax

```
if(expr1, expr2, expr3)
```

Arguments

expr1

The condition or expression that is evaluated. If it is `true`, the function will return the value of *expr2*. If *expr1* is `false`, the function will return the value of *expr3*.

expr2

The expression that is evaluated and returned if *expr1* is `true`.

expr3

The expression that is evaluated and returned if *expr1* is `false`.

Returns

If `expr1` evaluates to `true`, then returns `expr2`; otherwise returns `expr3`.

Example

The following example uses the `if()` function to return one of two values based on a condition. The condition being evaluated is `1 < 2`, which is `true`, so the first value `'a'` is returned.

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

IS_NULL expression

The IS_NULL conditional expression is used to check if a value is null.

This expression is a synonym for IS NULL.

Syntax

```
is_null(expr)
```

Arguments

expr

An expression of any type.

Returns

The IS_NULL conditional expression returns a Boolean. If `expr1` is NULL, returns `true`, otherwise returns `false`.

Examples

The following example checks if the value 1 is null, and returns the boolean result `true` because 1 is a valid, non-null value.

```
SELECT is not null(1);
 true
```

The following example selects the `id` column from the `squirrels` table, but only for the rows where the age column is `null`.

```
SELECT id FROM squirrels WHERE is_null(age)
```

IS_NOT_NULL expression

The IS_NOT_NULL conditional expression is used to check if a value is not null.

This expression is a synonym for IS NOT NULL.

Syntax

```
is_not_null(expr)
```

Arguments

expr

An expression of any type.

Returns

The IS_NOT_NULL conditional expression returns a Boolean. If `expr1` is not NULL, returns `true`, otherwise returns `false`.

Examples

The following example checks if the value 1 is not null, and returns the boolean result `true` because 1 is a valid, non-null value.

```
SELECT is not null(1);
 true
```

The following example selects the `id` column from the `squirrels` table, but only for the rows where the age column is not `null`.

```
SELECT id FROM squirrels WHERE is_not_null(age)
```

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

Compares two arguments and returns null if the arguments are equal. If they aren't equal, the first argument is returned.

Syntax

The NULLIF expression compares two arguments and returns null if the arguments are equal. If they aren't 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.

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)
```

Constructor functions

A SQL constructor function is a function that is used to create new data structures, such as arrays or maps.

They take some input values and return a new data structure object. Constructor functions are typically named after the data type they create, such as ARRAY or MAP.

Constructor functions are different from scalar functions or aggregate functions, which operate on existing data and return a single value. Constructor functions are used to create new data structures that can then be used in further data processing or analysis.

AWS Clean Rooms supports the following constructor functions:

Topics

MAP constructor function

The MAP constructor function creates a map with the given key/value pairs.

Constructor functions like MAP are useful when you need to create new data structures programmatically within your SQL queries. They allow you to build complex data structures that can be used in further data processing or analysis.

Syntax

```
map(key0, value0, key1, value1, ...)
```

Arguments

key0

An expression of any comparable type. All *key0* must share a least common type.

value0

An expression of any type. All *valueN* must share a least common type.

Returns

The MAP function returns a MAP with keys typed as the least common type of *key0* and values typed as the least common type of *value0*.

Examples

The following example creates a new map with two key-value pairs: The key `1.0` is associated with the value `'2'`. The key `3.0` is associated with the value `'4'`. The resulting map is then returned as the output of the SQL statement.

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

NAMED_STRUCT constructor function

The NAMED_STRUCT constructor function creates a struct with the given field names and values.

Constructor functions like NAMED_STRUCT are useful when you need to create new data structures programmatically within your SQL queries. They allow you to build complex data structures, such as structs or records, that can be used in further data processing or analysis.

Syntax

```
named_struct(name1, val1, name2, val2, ...)
```

Arguments

name1

A STRING literal naming field 1.

val1

An expression of any type specifying the value for field 1.

Returns

The NAMED_STRUCT function returns a struct with field 1 matching the type of *val1*.

Examples

The following example creates a new struct with three named fields: The field `"a"` is assigned the value 1. The field `"b"` is assigned the value `2.` The field `"c"` is assigned the value 3. The resulting struct is then returned as the output of the SQL statement.

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

STRUCT constructor function

The STRUCT constructor function creates a struct with the given field values.

Constructor functions like STRUCT are useful when you need to create new data structures programmatically within your SQL queries. They allow you to build complex data structures, such as structs or records, that can be used in further data processing or analysis.

Syntax

```
struct(col1, col2, col3, ...)
```

Arguments

col1

A column name or any valid expression.

Returns

The STRUCT function returns a struct with *field1* matching the type of *expr1*.

If the arguments are named references, the names are used to name the field. Otherwise, the fields are named *colN*, where N is the position of the field in the struct.

Examples

The following example creates a new struct with three fields: The first field is assigned the value 1. The second field is assigned the value 2. The third field is assigned the value 3. By default, the fields in the resulting struct are named `col1`, `col2`, and `col3`, based on their position in the argument list. The resulting struct is then returned as the output of the SQL statement.

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

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 Spark SQL supports several data type formatting functions.

Topics

- Numeric format strings

BASE64 function

The BASE64 function converts an expression to a base 64 string using RFC2045 Base64 transfer encoding for MIME.

Syntax

```
base64(expr)
```

Arguments

expr

A BINARY expression or a STRING which the function will interpret as BINARY.

Return type

STRING

Example

To convert the given string input into its Base64 encoded representation. use the following example. The result is the Base64 encoded representation of the input string 'Spark SQL', which is 'U3BhcmsgU1FM'.

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

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.

Certain data types require an explicit conversion to other data types using the CAST function. Other data types can be converted implicitly, as part of another command, without using CAST. 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 )
```

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 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 | 72800000000000000000000.00
       2 |  7600000000000000000000.00
       3 | 35000000000000000000000.00
       4 | 17500000000000000000000.00
       5 | 15400000000000000000000.00
       6 | 39400000000000000000000.00
       7 | 78800000000000000000000.00
       8 | 19700000000000000000000.00
       9 | 59100000000000000000000.00

(9 rows)
```

DECODE function

The DECODE function is the counterpart to the ENCODE function, which is used to convert a string to a binary format using a specific character encoding. The DECODE function takes the binary data and converts it back to a readable string format using the specified character encoding.

This function is useful when you need to work with binary data stored in a database and need to present it in a human-readable format, or when you need to convert data between different character encodings.

Syntax

```
decode(expr, charset)
```

Arguments

expr

A BINARY expression encoded in charset.

charset

A STRING expression.

Supported character set encodings (case-insensitive): `'US-ASCII'`, `'ISO-8859-1'`, `'UTF-8'`, `'UTF-16BE'`, `'UTF-16LE'`, and `'UTF-16'`.

Return type

The DECODE function returns a STRING.

Example

The following example has a table called `messages` with a column called `message_text` that stores message data in a binary format using the UTF-8 character encoding. The DECODE function converts the binary data back to a readable string format. The output of this query is the readable text of the message stored in the messages table, with the ID `123`, converted from the binary format to a string using the `'utf-8'` encoding.

```
SELECT decode(message_text, 'utf-8') AS message
FROM messages
WHERE message_id = 123;
```

ENCODE function

The ENCODE function is used to convert a string to its binary representation using a specified character encoding.

This function is useful when you need to work with binary data or when you need to convert between different character encodings. For example, you might use the ENCODE function when storing data in a database that requires binary storage, or when you need to transfer data between systems that use different character encodings.

Syntax

```
encode(str, charset)
```

Arguments

str

A STRING expression to be encoded.

charset

A STRING expression specifying the encoding.

Supported character set encodings (case-insensitive): `'US-ASCII'`, `'ISO-8859-1'`, `'UTF-8'`, `'UTF-16BE'`, `'UTF-16LE'`, and `'UTF-16'`.

Return type

The ENCODE function returns a BINARY.

Example

The following example converts the string `'abc'` to its binary representation using the `'utf-8'` encoding, which in this case results in the original string being returned. This is because the `'utf-8'` encoding is a variable-width character encoding that can represent the entire ASCII character set (which includes the letters `'a'`, `'b'`, and `'c'`) using a single byte per character. Therefore, the binary representation of `'abc'` using `'utf-8'` is the same as the original string.

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

HEX function

The HEX function converts a numeric value (either an integer or a floating-point number) to its corresponding hexadecimal string representation.

Hexadecimal is a numeral system that uses 16 distinct symbols (0-9 and A-F) to represent numeric values. It is commonly used in computer science and programming to represent binary data in a more compact and human-readable format.

Syntax

```
hex(expr)
```

Arguments

expr

A BIGINT, BINARY, or STRING expression.

Return type

HEX returns a STRING. The function returns the hexadecimal representation of the argument.

Example

The following example takes the integer value 17 as input and applies the HEX() function to it. The output is 11, which is the hexadecimal representation of the input value 17.

```
SELECT hex(17);
 11
```

The following example converts the string `'Spark_SQL'` to its hexadecimal representation. The output is `537061726B2053514C`, which is the hexadecimal representation of the input string `'Spark_SQL'`.

```
SELECT hex('Spark_SQL');
 537061726B2053514C
```

In this example, the string 'Spark_SQL' is converted as follows:

- 'S' -> 53
- 'p' -> 70
- 'a' -> 61
- 'r' -> 72 '
- k' -> 6B
- '_' -> 20
- 'S' -> 53

- 'Q' -> 51
- 'L' -> 4C

The concatenation of these hexadecimal values results in the final output `"537061726B2053514C"`.

STR_TO_MAP function

The STR_TO_MAP function is a string-to-map conversion function. It converts a string representation of a map (or dictionary) into an actual map data structure.

This function is useful when you need to work with map data structures in SQL, but the data is initially stored as a string. By converting the string representation to an actual map, you can then perform operations and manipulations on the map data.

Syntax

```
str_to_map(text[, pairDelim[, keyValueDelim]])
```

Arguments

text

A STRING expression that represents the map.

pairDelim

An optional STRING literal that specifies how to separate entries. It defaults to a comma (`','`).

keyValueDelim

An optional STRING literal that specifies how to separate each key-value pair. It defaults to a colon (`':'`).

Return type

The STR_TO_MAP function returns a MAP of STRING for both keys and values. Both *pairDelim* and *keyValueDelim* are treated as regular expressions.

Example

The following example takes the input string and the two delimiter arguments, and converts the string representation into an actual map data structure. In this specific example, the input string `'a:1,b:2,c:3'` represents a map with the following key-value pairs: `'a'` is the key, and `'1'` is the value. `'b'` is the key, and `'2'` is the value. `'c'` is the key, and `'3'` is the value. The `','` delimiter is used to separate the key-value pairs, and the `':'` delimiter is used to separate the key and value within each pair. The output of this query is: `{"a":"1","b":"2","c":"3"}`. This is the resulting map data structure, where the keys are `'a'`, `'b'`, and `'c'`, and the corresponding values are `'1'`, `'2'`, and `'3'`.

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

The following example demonstrates that the STR_TO_MAP function expects the input string to be in a specific format, with the key-value pairs delimited correctly. If the input string doesn't match the expected format, the function will still attempt to create a map, but the resulting values may not be as expected.

```
SELECT str_to_map('a');
 {"a":null}
```

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 (date_str)
```

```
TO_DATE (date_str, format)
```

Arguments

date_str

A date string or a data type that can be cast into a date string.

format

A string literal that matches Spark's datetime patterns. For valid datetime patterns, see Datetime Patterns for Formatting and Parsing.

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 MMM yyyy');

to_date
------------
2001-10-02
(1 row)
```

The following SQL statement converts the string `20010631` to a date.

```
select to_date('20010631', 'yyyyMMdd');
```

The following SQL statement converts the string `20010631` to a date:

```
to_date('20010631', 'YYYYMMDD', TRUE);
```

The result is a null value because there are only 30 days in June.

```

to_date
------------
NULL
```

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
```

UNBASE64 function

The UNBASE64 function converts an argument from a base 64 string to a binary.

Base64 encoding is commonly used to represent binary data (such as images, files, or encrypted information) in a textual format that is safe for transmission over various communication channels (such as email, URL parameters, or database storage).

The UNBASE64 function allows you to reverse this process and recover the original binary data. This type of functionality can be useful in scenarios where you need to work with data that has been encoded in Base64 format, such as when integrating with external systems or APIs that use Base64 as a data transfer mechanism.

Syntax

```
unbase64(expr)
```

Arguments

expr

A STRING expression in a base64 format.

Return type

`BINARY`

Example

In the following example, the Base64-encoded string `'U3BhcmsgU1FM'` is converted back to the original string `'Spark SQL'`.

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

UNHEX function

The UNHEX function converts a hexadecimal string back to its original string representation.

This function can be useful in scenarios where you need to work with data that has been stored or transmitted in a hexadecimal format, and you need to restore the original string representation for further processing or display.

The UNHEX function is the counterpart to the HEX function.

Syntax

```
unhex(expr)
```

Arguments

expr

A STRING expression of hexadecimal characters.

Return type

UNHEX returns a BINARY.

If the length of *expr* is odd, the first character is discarded and the result is padded with a null byte. If *expr* contains non hex characters the result is NULL.

Example

The following example converts a hexadecimal string back to its original string representation by using the UNHEX() and DECODE() functions together. The first part of the query, uses the UNHEX() function to convert the hexadecimal string '537061726B2053514C' to its binary representation. The second part of the query, uses the DECODE() function to convert the binary data obtained from the UNHEX() function back to a string, using the 'UTF-8' character encoding. The output of the query, is he original string 'Spark_SQL' that was converted to hexadecimal and then back to a string.

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

Datetime format strings

You can use datetime patterns in the following common scenarios:

- When working with CSV and JSON data sources to parse and format datetime content

- When converting between string types and date or timestamp types using functions such as:
  - unix_timestamp
  - date_format
  - to_unix_timestamp
  - from_unixtime
  - to_date
  - to_timestamp
  - from_utc_timestamp
  - to_utc_timestamp

Use the pattern letters in the following table for date and timestamp parsing and formatting.

Datepart or timepart	Meaning	Examples
a	AM or PM of the day, presented as am-pm	PM
D	Day of the year, presented as a 3-digit number	189
d	Day of the month, presented as a 2-digit number	28
E	Day of the week, presented as a text	Tue Tuesday
F	Aligned day of the week in the month, presented as a 1-digit number	3
G	Era indicator, presented as text	AD Anno Domini
h	Clock-hour of AM or PM, presented as a 2-digit number	12

Datepart or timepart	Meaning	Examples
H	Hour of day, presented as a 2-digit number from 0–23	0
k	Clock-hour of day, presented as a 2-digit number from 1–24	1
K	Hour of AM or PM, presented as a 2-digit number from 0–11	0
m	Minute of hour, presented as a 2-digit number	30
M/L	Month of the year, presented as a month	7 07 Jul July
O	Localized zone offset from UTC	GMT+8 GMT+8:00 UTC-08:00
Q/q	Quarter of the year, presented as a number (1 to 4) or text	3 03 Q3 3rd quarter
s	Second of minute, presented as a 2-digit number	55

Datepart or timepart	Meaning	Examples
S	Fraction of a second, presented as a fraction	978
V	Time zone identifier, presented as a zone-id	America/Los_Angeles Z 08:30
x	Zone offset from UTC (offset-X)	+0000 -08 -0830 -08:30 -083015 -08:30:15
X	Zone offset from UTC; where Z is for zero	Z -08 -0830 -08:30 -083015 -08:30:15
y	Year, presented as a year	2020 20
z	Time zone name, presented as text	Pacific Standard Time PST

Datepart or timepart	Meaning	Examples
Z	Zone offset from UTC (offset-Z)	+0000 -0800 -08:00
'	Escape for text, presented as a delimiter	N/A
''	Single quote, presented as a literal	'
[	Optional section start	N/A
]	Optional section end	N/A

The number of pattern letters determines the format type:

Text Format

- Use 1-3 letters for the abbreviated form (for example, "Mon" for Monday)
- Use exactly 4 letters for the full form (for example, "Monday")
- Don't use 5 or more letters - this will cause an error

Number Format (n)

- The value n represents the maximum number of letters allowed
- For single letter patterns:
  - Output uses minimum digits without padding
- For multiple letter patterns:
  - Output is padded with zeros to match the letter count width
- When parsing, input must contain the exact number of digits

Number/Text Format

- For 3 or more letters, follow the Text Format rules
- For fewer letters, follow the Number Format rules

Fraction Format

- Use 1-9 'S' characters (for example, SSSSSS)
- For parsing:
  - Accept fractions between 1 and the number of S characters
- For formatting:
  - Pad with zeros to match the number of S characters
- Supports up to 6 digits for microsecond precision
- Can parse nanoseconds but truncates extra digits

Year Format

- The letter count sets the minimum field width for padding
- For two letters:
  - Prints the last two digits
  - Parses years between 2000-2099
- For less than four letters (except two):
  - Shows the sign only for negative years
- Don't use 7 or more letters - this will cause an error

Month Format

- Use 'M' for standard form or 'L' for standalone form
- Single 'M' or 'L':
  - Shows month numbers 1-12 without padding
- 'MM' or 'LL':
  - Shows month numbers 01-12 with padding
- 'MMM':
  - Shows abbreviated month name in standard form

  - Must be part of a full date pattern
- 'LLL':
  - Shows abbreviated month name in standalone form
  - Use for month-only formatting
- 'MMMM':
  - Shows full month name in standard form
  - Use for dates and timestamps
- 'LLLL':
  - Shows full month name in standalone form
  - Use for month-only formatting

Time Zone Formats

- am-pm: Use 1 letter only
- Zone ID (V): Use 2 letters only
- Zone names (z):
  - 1-3 letters: Shows short name
  - 4 letters: Shows full name
  - Don't use 5 or more letters

Offset Formats

- X and x:
  - 1 letter: Shows hour (+01) or hour-minute (+0130)
  - 2 letters: Shows hour-minute without colon (+0130)
  - 3 letters: Shows hour-minute with colon (+01:30)
  - 4 letters: Shows hour-minute-second without colon (+013015)
  - 5 letters: Shows hour-minute-second with colon (+01:30:15)
  - X uses 'Z' for zero offset
  - x uses '+00', '+0000', or '+00:00' for zero offset
- O:
  - 1 letter: Shows short form (GMT+8)

  - 4 letters: Shows full form (GMT+08:00)
- Z:
  - 1-3 letters: Shows hour-minute without colon (+0130)
  - 4 letters: Shows full localized form
  - 5 letters: Shows hour-minute-second with colon

Optional Sections

- Use square brackets [ ] to mark optional content
- You can nest optional sections
- All valid data appears in output
- Input can omit entire optional sections

> **Note**
>
> The symbols 'E', 'F', 'q', and 'Q' work only for datetime formatting (like date_format). Don't use them for datetime parsing (like to_timestamp).

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.

Format	Description
, (comma)	Thousands separator.
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.

Date and time functions

Date and time functions allow you to perform a wide range of operations on date and time data, such as extracting parts of a date, performing date calculations, formatting dates and times, and working with the current date and time. These functions are essential for tasks such as data analysis, reporting, and data manipulation involving temporal data.

AWS Clean Rooms supports the following date and time functions:

Topics

ADD_MONTHS function

ADD_MONTHS adds the specified number of months to a date or timestamp value or expression. The DATE_ADD 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
```

CURRENT_TIMESTAMP function

CURRENT_TIMESTAMP returns the current date and time, including the date, time, and (optionally) the milliseconds or microseconds.

This function is useful when you need to get the current date and time, for example, to record the timestamp of an event, to perform time-based calculations, or to populate date/time columns.

Syntax

```
current_timestamp()
```

Return type

The CURRENT_TIMESTAMP function returns a DATE.

Example

The following example returns current date and time at the moment the query is executed, which is April 25, 2020, at 15:49:11.914 (3:49:11.914 PM).

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

The following example retrieves the current date and time for each row in the squirrels table.

```
SELECT current_timestamp() FROM squirrels
```

DATE_ADD function

Returns the date that is num_days after start_date.

Syntax

```
date_add(start_date, num_days)
```

Arguments

start_date

The starting date value.

num_days

The number of days to add (integer). A positive number adds days, a negative number subtracts days.

Return type

DATE

Examples

The following example adds one day to a date:

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

Result:
```

```
2016-07-31
```

The following example adds multiple days.

```
SELECT date_add('2016-07-30', 5);

Result:
2016-08-04
```

Usage notes

This documentation is for Spark SQL's DATE_ADD function, which provides a simpler interface for adding days to dates compared to some other SQL variants. For adding other intervals like months or years, different functions may be required.

DATE_DIFF function

DATE_DIFF returns the difference between the date parts of two date or time expressions.

Syntax

```
date_diff(endDate, startDate)
```

Arguments

endDate

A DATE expression.

startDate

A DATE expression.

Return type

BIGINT

Examples with a DATE column

The following example finds the difference, in number of weeks, between two literal date values.

```
select date_diff(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 date_diff(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 date_diff(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 date_diff(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 DATE_DIFF function is DATE_DIFF.

```
select date_diff(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,
date_diff(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(date_diff(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 date_diff(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 date_diff(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 date_diff(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 date_diff(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

```
datepart(field, source)
```

Arguments

field

Which part of the source should be extracted, and supported string values are the same as the fields of the equivalent function EXTRACT.

source

A DATE or INTERVAL column from where field should be extracted.

Return type

If *field* is 'SECOND', a DECIMAL(8, 6). In all other cases, an INTEGER.

Example

The following example extracts the day of the year (DOY) from a date value. The output shows that the day of the year for the date "2019-08-12" is 224. This means that August 12, 2019 is the 224th day of the year 2019.

```
SELECT datepart('doy', DATE'2019-08-12');
 224
```

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(format, datetime)
```

Arguments

format

The format representing the unit to be truncated to. Valid formats are as follows:

- "YEAR", "YYYY", "YY" - truncate to the first date of the year that the ts falls in, the time part will be zero out
- "QUARTER" - truncate to the first date of the quarter that the ts falls in, the time part will be zero out
- "MONTH", "MM", "MON" - truncate to the first date of the month that the ts falls in, the time part will be zero out
- "WEEK" - truncate to the Monday of the week that the ts falls in, the time part will be zero out

- "DAY", "DD" - zero out the time part
- "HOUR" - zero out the minute and second with fraction part
- "MINUTE"- zero out the second with fraction part
- "SECOND" - zero out the second fraction part
- "MILLISECOND" - zero out the microseconds
- "MICROSECOND" - everything remains

ts

A datetime value

Return type

Returns timestamp *ts* truncated to the unit specified by the format model

Examples

The following example truncates a date value to the beginning of the year. The output shows that the date "2015-03-05" has been truncated to "2015-01-01", which is the beginning of the year 2015.

```
SELECT date_trunc('YEAR', '2015-03-05');

 date_trunc
-----------
2015-01-01
```

DAY function

The DAY function returns the day of month of the date/timestamp.

Date extraction functions are useful when you need to work with specific components of a date or timestamp, such as when performing date-based calculations, filtering data, or formatting date values.

Syntax

```
day(date)
```

Arguments

date

A DATE or TIMESTAMP expression.

Returns

The DAY function returns an INTEGER.

Examples

The following example extracts the day of the month (`30`) from the input date `'2009-07-30'`.

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

The following example extracts the day of the month from the `birthday` column of the `squirrels` table and returns the results as the output of the SELECT statement. The output of this query will be a list of day values, one for each row in the `squirrels` table, representing the day of the month for each squirrel's birthday.

```
SELECT day(birthday) FROM squirrels
```

DAYOFMONTH function

The DAYOFMONTH function returns the day of month of the date/timestamp (a value between 1 and 31, depending on the month and year).

The DAYOFMONTH function is similar to the DAY function, but they have slightly different names and slightly different behavior. The DAY function is more commonly used, but the DAYOFMONTH function can be used as an alternative. This type of query can be useful when you need to perform date-based analysis or filtering on a table that contains date or timestamp data, such as extracting specific components of a date for further processing or reporting.

Syntax

```
dayofmonth(date)
```

Arguments

date

A DATE or TIMESTAMP expression.

Returns

The DAYOFMONTH function returns an INTEGER.

Example

The following example extracts the day of the month (`30`) from the input date `'2009-07-30'`.

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

The following example applies the DAYOFMONTH function to the `birthday` column of the `squirrels` table. For each row in the `squirrels` table, the day of the month from the `birthday` column will be extracted and returned as the output of the SELECT statement. The output of this query will be a list of day values, one for each row in the `squirrels` table, representing the day of the month for each squirrel's birthday.

```
SELECT dayofmonth(birthday) FROM squirrels
```

DAYOFWEEK function

The DAYOFWEEK function takes a date or timestamp as input and returns the day of the week as a number (1 for Sunday, 2 for Monday, ..., 7 for Saturday).

This date extraction function is useful when you need to work with specific components of a date or timestamp, such as when performing date-based calculations, filtering data, or formatting date values.

Syntax

```
dayofweek(date)
```

Arguments

date

A DATE or TIMESTAMP expression.

Returns

The DAYOFWEEK function returns an INTEGER where

1 = Sunday

2 = Monday

3 = Tuesday

4 = Wednesday

5 = Thursday

6 = Friday

7 = Saturday

Examples

The following example extracts the day of the week from this date, which is 5 (representing Thursday).

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

The following example extracts the day of the week from the `birthday` column of the `squirrels` table and returns the results as the output of the SELECT statement. The output of this query will be a list of day of the week values, one for each row in the `squirrels` table, representing the day of the week for each squirrel's birthday.

```
SELECT dayofweek(birthday) FROM squirrels
```

DAYOFYEAR function

The DAYOFYEAR function is a date extraction function that takes a date or timestamp as input and returns the day of the year (a value between 1 and 366, depending on the year and whether it's a leap year).

This function is useful when you need to work with specific components of a date or timestamp, such as when performing date-based calculations, filtering data, or formatting date values.

Syntax

```
dayofyear(date)
```

Arguments

date

A DATE or TIMESTAMP expression.

Returns

The DAYOFYEAR function returns an INTEGER (between 1 and 366, depending on the year and whether it's a leap year).

Examples

The following example extracts the day of the year (`100`) from the input date `'2016-04-09'`.

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

The following example extracts the day of the year from the `birthday` column of the `squirrels` table and returns the results as the output of the SELECT statement.

```
SELECT dayofyear(birthday) FROM squirrels
```

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 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
```

FROM_UTC_TIMESTAMP function

The FROM_UTC_TIMESTAMP function converts the input date from UTC (Coordinated Universal Time) to the specified time zone.

This function is useful when you need to convert date and time values from UTC to a specific time zone. This can be important when working with data that originates from different parts of the world and needs to be presented in the appropriate local time.

Syntax

```
from_utc_timestamp(timestamp, timezone
```

Arguments

timestamp

A TIMESTAMP expression with a UTC timestamp.

timezone

A STRING expression that is a valid timezone to which the input date or timestamp should be converted.

Returns

The FROM_UTC_TIMESTAMP function returns a TIMESTAMP.

Example

The following example converts the input date from UTC to the specified time zone (`'Asia/Seoul'`), which in this case is 9 hours ahead of UTC. The resulting output is the date and time in the Seoul time zone, which is `2016-08-31 09:00:00`.

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

HOUR function

The HOUR function is a time extraction function that takes a time or timestamp as input and returns the hour component (a value between 0 and 23).

This time extraction function is useful when you need to work with specific components of a time or timestamp, such as when performing time-based calculations, filtering data, or formatting time values.

Syntax

```
hour(timestamp)
```

Arguments

timestamp

A TIMESTAMP expression.

Returns

The HOUR function returns an INTEGER.

Example

The following example extracts the hour component (12) from the input timestamp `'2009-07-30 12:58:59'`.

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

MINUTE function

The MINUTE function is a time extraction function that takes a time or timestamp as input and returns the minute component (a value between 0 and 60).

Syntax

```
minute(timestamp)
```

Arguments

timestamp

A TIMESTAMP expression or a STRING of a valid timestamp format.

Returns

The MINUTE function returns an INTEGER.

Example

The following example extracts the minute component (58) from the input timestamp `'2009-07-30 12:58:59'`.

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

MONTH function

The MONTH function is a time extraction function that takes a time or timestamp as input and returns the month component (a value between 0 and 12).

Syntax

```
month(date)
```

Arguments

date

A TIMESTAMP expression or a STRING of a valid timestamp format.

Returns

The MONTH function returns an INTEGER.

Example

The following example extracts the month component (7) from the input timestamp `'2016-07-30'`.

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

SECOND function

The SECOND function is a time extraction function that takes a time or timestamp as input and returns the second component (a value between 0 and 60).

Syntax

```
second(timestamp)
```

Arguments

timestamp

A TIMESTAMP expression.

Returns

The SECOND function returns an INTEGER.

Example

The following example extracts the second component (59) from the input timestamp `'2009-07-30 12:58:59'`.

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

TIMESTAMP function

The TIMESTAMP function takes a value (typically a number) and converts it to a timestamp data type.

This function is useful when you need to convert a numeric value representing a time or date to a timestamp data type. This can be helpful when you are working with data that is stored in a numeric format, such as Unix timestamps or epoch time.

Syntax

```
timestamp(expr)
```

Arguments

expr

Any expression that can be cast to TIMESTAMP.

Returns

The TIMESTAMP function returns a TIMESTAMP.

Example

The following example converts a numeric Unix timestamp (`1632416400`) to its corresponding timestamp data type: September 22, 2021 at 12:00:00 PM UTC.

```
SELECT timestamp(1632416400);
 2021-09-22 12:00:00 UTC
```

TO_TIMESTAMP function

TO_TIMESTAMP converts a TIMESTAMP string to TIMESTAMPTZ.

Syntax

```
to_timestamp (timestamp)
```

```
to_timestamp (timestamp, format)
```

Arguments

timestamp

A timestamp string or a data type that can be cast into a timestamp string.

format

A string literal that matches Spark's datetime patterns. For valid datetime patterns, see [Datetime Patterns for Formatting and Parsing].

Return type

TIMESTAMP

Examples

The following example demonstrates using the TO_TIMESTAMP function to convert a TIMESTAMP string to a TIMESTAMP.

```
select current_timestamp() as timestamp, to_timestamp( current_timestamp(), '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 TIMESTAMP. The result is a TIMESTAMP 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
```

YEAR function

The YEAR function is a date extraction function that takes a date or timestamp as input and returns the year component (a four-digit number).

Syntax

```
year(date)
```

Arguments

date

A DATE or TIMESTAMP expression.

Returns

The YEAR function returns an INTEGER.

Example

The following example extracts the year component (`2016`) from the input date `'2016-07-30'`.

```
SELECT year('2016-07-30');
 2016
```

The following example extracts the year component from the `birthday` column of the `squirrels` table and returns the results as the output of the SELECT statement. The output of this query will be a list of year values, one for each row in the `squirrels` table, representing the year of each squirrel's birthday.

```
SELECT year(birthday) FROM squirrels
```

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:

- DATE_ADD
- DATE_DIFF
- 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
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

Date part or time part	Abbreviations
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)
```

Encryption and decryption functions

Encryption and decryption functions help SQL developers protect sensitive data from unauthorized access or misuse by converting it between a readable, plaintext form and an unreadable, ciphertext form.

AWS Clean Rooms Spark SQL supports the following encryption and decryption functions:

Topics

- AES_ENCRYPT function
- AES_DECRYPT function

AES_ENCRYPT function

The AES_ENCRYPT function is used for encrypting data using the Advanced Encryption Standard (AES) algorithm.

Syntax

```
aes_encrypt(expr, key[, mode[, padding[, iv[, aad]]]])
```

Arguments

expr

The binary value to encrypt.

key

The passphrase to use to encrypt the data.

Key lengths of 16, 24 and 32 bits are supported.

mode

Specifies which block cipher mode should be used to encrypt messages.

Valid modes: ECB (Electronic CodeBook), GCM (Galois/Counter Mode), CBC (Cipher-Block Chaining).

padding

Specifies how to pad messages whose length isn't a multiple of the block size.

Valid values: PKCS, NONE, DEFAULT.

The DEFAULT padding means PKCS (Public Key Cryptography Standards) for ECB, NONE for GCM and PKCS for CBC.

Supported combinations of (*mode*, *padding*) are ('ECB', 'PKCS'), ('GCM', 'NONE') and ('CBC', 'PKCS').

iv

Optional initialization vector (IV). Only supported for CBC and GCM modes.

Valid values: 12-bytes long for GCM and 16 bytes for CBC.

aad

Optional additional authenticated data (AAD). Only supported for GCM mode. This can be any free-form input and must be provided for both encryption and decryption.

Return type

The AES_ENCRYPT function returns an encrypted value of *expr* using AES in given mode with the specified padding.

Examples

The following example demonstrates how to use the Spark SQL AES_ENCRYPT function to securely encrypt a string of data (in this case, the word "Spark") using a specified encryption key. The resulting ciphertext is then Base64-encoded to make it easier to store or transmit.

```
SELECT base64(aes_encrypt('Spark', 'abcdefghijklmnop'));
  4A5jOAh9FNGwoMeuJukfllrLdHEZxA2DyuSQAWz77dfn
```

The following example demonstrates how to use the Spark SQL AES_ENCRYPT function to securely encrypt a string of data (in this case, the word "Spark") using a specified encryption key. The resulting ciphertext is then represented in hexadecimal format, which can be useful for tasks such as data storage, transmission, or debugging.

```
SELECT hex(aes_encrypt('Spark', '0000111122223333'));
 83F16B2AA704794132802D248E6BFD4E380078182D1544813898AC97E709B28A94
```

The following example demonstrates how to use the Spark SQL AES_ENCRYPT function to securely encrypt a string of data (in this case, "Spark SQL") using a specified encryption key, encryption mode, and padding mode. The resulting ciphertext is then Base64-encoded to make it easier to store or transmit.

```
SELECT base64(aes_encrypt('Spark SQL', '1234567890abcdef', 'ECB', 'PKCS'));
 3lmwu+Mw0H3fi5NDvcu9lg==
```

AES_DECRYPT function

The AES_DECRYPT function is used for decrypting data using the Advanced Encryption Standard (AES) algorithm.

Syntax

```
aes_decrypt(expr, key[, mode[, padding[, aad]]])
```

Arguments

expr

The binary value to decrypt.

key

The passphrase to use to decrypt the data.

The passphrase must match the key originally used to produce the encrypted value and be 16, 24, or 32 bytes long.

mode

Specifies which block cipher mode should be used to decrypt messages.

Valid modes: ECB, GCM, CBC.

padding

Specifies how to pad messages whose length isn't a multiple of the block size.

Valid values: PKCS, NONE, DEFAULT.

The DEFAULT padding means PKCS for ECB, NONE for GCM and PKCS for CBC.

aad

Optional additional authenticated data (AAD). Only supported for GCM mode. This can be any free-form input and must be provided for both encryption and decryption.

Return type

Returns a decrypted value of *expr* using AES in mode with padding.

Examples

The following example demonstrates how to use the Spark SQL AES_ENCRYPT function to securely encrypt a string of data (in this case, the word "Spark") using a specified encryption key. The resulting ciphertext is then Base64-encoded to make it easier to store or transmit.

```
SELECT base64(aes_encrypt('Spark', 'abcdefghijklmnop'));
  4A5jOAh9FNGwoMeuJukfllrLdHEZxA2DyuSQAWz77dfn
```

The following example demonstrates how to use the Spark SQL AES_DECRYPT function to decrypt data that has been previously encrypted and Base64-encoded. The decryption process requires the correct encryption key and parameters (encryption mode and padding mode) to successfully recover the original plaintext data.

```
SELECT aes_decrypt(unbase64('3lmwu+Mw0H3fi5NDvcu9lg=='), '1234567890abcdef', 'ECB',
 'PKCS');
 Spark SQL
```

Hash functions

A hash function is a mathematical function that converts a numerical input value into another value.

AWS Clean Rooms Spark SQL supports the following hash functions:

Topics

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);
```

xxHASH64 function

The xxhash64 function returns a 64-bit hash value of the arguments.

The xxhash64() function is a non-cryptographic hash function designed to be fast and efficient. It's often used in data processing and storage applications, where a unique identifier for a piece of data is needed, but the exact contents of the data don't need to be kept secret.

In the context of a SQL query, the xxhash64() function could be used for various purposes, such as:

- Generating a unique identifier for a row in a table
- Partitioning data based on a hash value
- Implementing custom indexing or data distribution strategies

The specific use case would depend on the requirements of the application and the data being processed.

Syntax

```
xxhash64(expr1, expr2, ...)
```

Arguments

expr1

An expression of any type.

expr2

An expression of any type.

Returns

Returns a 64-bit hash value of the arguments (BIGINT). Hash seed is 42.

Example

The following example generates a 64-bit hash value (5602566077635097486) based on the provided input. The first argument is a string value, in this case, the word "Spark". The second argument is an array containing the single integer value 123. The third argument is an integer value representing the seed for the hash function.

```
SELECT xxhash64('Spark', array(123), 2);
 5602566077635097486
```

Hyperloglog functions

The HyperLogLog (HLL) functions in SQL provide a way to efficiently estimate the number of unique elements (cardinality) in a large dataset, even when the actual set of unique elements isn't stored.

The main benefits of using HLL functions are:

- **Memory efficiency**: HLL sketches require much less memory than storing the full set of unique elements, making them suitable for large datasets.
- **Distributed computing**: HLL sketches can be combined across multiple data sources or processing nodes, allowing for efficient distributed unique count estimation.
- **Approximate results**: HLL provides an approximate unique count estimation, with a tunable trade-off between accuracy and memory usage (via the precision parameter).

These functions are particularly useful in scenarios where you need to estimate the number of unique items, such as in analytics, data warehousing, and real-time stream processing applications.

AWS Clean Rooms supports the following HLL functions.

Topics

HLL_SKETCH_AGG function

The HLL_SKETCH_AGG aggregate function creates an HLL sketch from the values in the specified column. It returns an HLLSKETCH data type that encapsulates the input expression values.

The HLL_SKETCH_AGG aggregate function works with any data type and ignores NULL values.

When there are no rows in a table or all rows are NULL, the resulting sketch has no index-value pairs such as `{"version":1,"logm":15,"sparse":{"indices":[],"values":[]}}`.

Syntax

```
HLL_SKETCH_AGG (aggregate_expression[, lgConfigK ] )
```

Argument

aggregate_expression

Any expression of type INT, BIGINT, STRING, or BINARY against which unique counting will occur. Any `NULL` values are ignored.

lgConfigK

An optional INT constant between 4 and 21 inclusive with default 12. The log-base-2 of K, where K is the number of buckets or slots for the sketch.

Return type

The HLL_SKETCH_AGG function returns a non-NULL BINARY buffer containing the HyperLogLog sketch computed because of consuming and aggregating all input values in the aggregation group.

Examples

The following examples use the HyperLogLog (HLL) algorithm to estimate the distinct count of values in the `col` column. The `hll_sketch_agg(col, 12)` function aggregates the values in the col column, creating an HLL sketch using a precision of 12. The `hll_sketch_estimate()` function is then used to estimate the distinct count of values based on the generated HLL sketch. The final result of the query is 3, which represents the estimated distinct count of values in the `col` column. In this case, the distinct values are 1, 2, and 3.

```
SELECT hll_sketch_estimate(hll_sketch_agg(col, 12))
    FROM VALUES (1), (1), (2), (2), (3) tab(col);
  3
```

The following example also uses the HLL algorithm to estimate the distinct count of values in the `col` column, but it doesn't specify a precision value for the HLL sketch. In this case, it uses the default precision of 14. The `hll_sketch_agg(col)` function takes the values in the `col` column and creates an HyperLogLog (HLL) sketch, which is a compact data structure that can be used to estimate the distinct count of elements. The `hll_sketch_estimate(hll_sketch_agg(col))` function takes the HLL sketch created in the previous step and calculates an estimate of the distinct count of values in the `col` column. The final result of the query is 3, which represents the estimated distinct count of values in the `col` column. In this case, the distinct values are 1, 2, and 3.

```
SELECT hll_sketch_estimate(hll_sketch_agg(col))
FROM VALUES (1), (1), (2), (2), (3) tab(col);
3
```

HLL_SKETCH_ESTIMATE function

The HLL_SKETCH_ESTIMATE function takes an HLL sketch and estimates the number of unique elements represented by the sketch. It uses the HyperLogLog (HLL) algorithm to count a probabilistic approximation of the number of unique values in a given column, consuming a binary representation known as a sketch buffer previously generated by the HLL_SKETCH_AGG function and returning the result as a big integer.

The HLL sketching algorithm provides an efficient way to estimate the number of unique elements, even for large datasets, without having to store the full set of unique values.

The `hll_union` and `hll_union_agg` functions can also combine sketches together by consuming and merging these buffers as inputs.

Syntax

```
HLL_SKETCH_ESTIMATE (hllsketch_expression)
```

Argument

hllsketch_expression

A `BINARY` expression holding a sketch generated by HLL_SKETCH_AGG

Return type

The HLL_SKETCH_ESTIMATE function returns a BIGINT value that is the approximate distinct count represented by the input sketch.

Examples

The following examples use the HyperLogLog (HLL) sketching algorithm to estimate the cardinality (unique count) of values in the `col` column. The `hll_sketch_agg(col, 12)` function takes the `col` column and creates an HLL sketch using a precision of 12 bits. The HLL sketch is an approximate data structure that can efficiently estimate the number of unique elements in a set. The `hll_sketch_estimate()` function takes the HLL sketch created by `hll_sketch_agg` and estimates the cardinality (unique count) of the values represented by the sketch. The `FROM VALUES (1), (1), (2), (2), (3) tab(col);` generates a test dataset with 5 rows, where the `col` column contains the values 1, 1, 2, 2, and 3. The result of this query is the estimated unique count of the values in the `col` column, which is 3.

```
SELECT hll_sketch_estimate(hll_sketch_agg(col, 12))
    FROM VALUES (1), (1), (2), (2), (3) tab(col);
  3
```

The difference between the following example and the previous one is that the precision parameter (12 bits) isn't specified in the `hll_sketch_agg` function call. In this case, the default precision of 14 bits is used, which may provide a more accurate estimate for the unique count compared to the previous example that used 12 bits of precision.

```
SELECT hll_sketch_estimate(hll_sketch_agg(col))
FROM VALUES (1), (1), (2), (2), (3) tab(col);
3
```

HLL_UNION function

The HLL_UNION function combines two HLL sketches into a single, unified sketch. It uses the HyperLogLog (HLL) algorithm to combine two sketches into a single sketch. Queries can use the resulting buffers to compute approximate unique counts as long integers with the `hll_sketch_estimate` function.

Syntax

```
HLL_UNION (( expr1, expr2 [, allowDifferentLgConfigK ] ))
```

Argument

exprN

A `BINARY` expression holding a sketch generated by HLL_SKETCH_AGG.

allowDifferentLgConfigK

A optional BOOLEAN expression controlling whether to allow merging two sketches with different lgConfigK values. The default value is `false`.

Return type

The HLL_UNION function returns a BINARY buffer containing the HyperLogLog sketch computed as a result of combining the input expressions. When the `allowDifferentLgConfigK` parameter is `true`, the result sketch uses the smaller of the two provided `lgConfigK` values.

Examples

The following examples use the HyperLogLog (HLL) sketching algorithm to estimate the unique count of values across two columns, `col1` and `col2`, in a dataset.

The `hll_sketch_agg(col1)` function creates an HLL sketch for the unique values in the `col1` column.

The `hll_sketch_agg(col2)` function creates an HLL sketch for the unique values in the col2 column.

The `hll_union(...)` function combines the two HLL sketches created in steps 1 and 2 into a single, unified HLL sketch.

The `hll_sketch_estimate(...)` function takes the combined HLL sketch and estimates the unique count of values across both `col1` and `col2`.

The `FROM VALUES` clause generates a test dataset with 5 rows, where `col1` contains the values 1, 1, 2, 2, and 3, and `col2` contains the values 4, 4, 5, 5, and 6.

The result of this query is the estimated unique count of values across both `col1` and `col2`, which is 6. The HLL sketching algorithm provides an efficient way to estimate the number of unique elements, even for large datasets, without having to store the full set of unique values. In this example, the `hll_union` function is used to combine the HLL sketches from the two columns, which allows the unique count to be estimated across the entire dataset, rather than just for each column individually.

```
SELECT hll_sketch_estimate(
  hll_union(
    hll_sketch_agg(col1),
```

```
    hll_sketch_agg(col2)))
  FROM VALUES
    (1, 4),
    (1, 4),
    (2, 5),
    (2, 5),
    (3, 6) AS tab(col1, col2);
  6
```

The difference between the following example and the previous one is that the precision parameter (12 bits) isn't specified in the `hll_sketch_agg` function call. In this case, the default precision of 14 bits is used, which may provide a more accurate estimate for the unique count compared to the previous example that used 12 bits of precision.

```
SELECT hll_sketch_estimate(
  hll_union(
    hll_sketch_agg(col1, 14),
    hll_sketch_agg(col2, 14)))
  FROM VALUES
    (1, 4),
    (1, 4),
    (2, 5),
    (2, 5),
    (3, 6) AS tab(col1, col2);
```

HLL_UNION_AGG function

The HLL_UNION_AGG function combines multiple HLL sketches into a single, unified sketch. It uses the HyperLogLog (HLL) algorithm to combine a group of sketches into a single one. Queries can use the resulting buffers to compute approximate unique counts with the `hll_sketch_estimate` function.

Syntax

```
HLL_UNION_AGG ( expr [, allowDifferentLgConfigK ] )
```

Argument

expr

A `BINARY` expression holding a sketch generated by HLL_SKETCH_AGG.

allowDifferentLgConfigK

A optional BOOLEAN expression controlling whether to allow merging two sketches with different lgConfigK values. The default value is `false`.

Return type

The HLL_UNION_AGG function returns a BINARY buffer containing the HyperLogLog sketch computed as a result of combining the input expressions of the same group. When the `allowDifferentLgConfigK` parameter is `true`, the result sketch uses the smaller of the two provided `lgConfigK` values.

Examples

The following examples use the HyperLogLog (HLL) sketching algorithm to estimate the unique count of values across multiple HLL sketches.

The first example estimates the unique count of values in a dataset.

```
SELECT hll_sketch_estimate(hll_union_agg(sketch, true))
    FROM (SELECT hll_sketch_agg(col) as sketch
            FROM VALUES (1) AS tab(col)
          UNION ALL
          SELECT hll_sketch_agg(col, 20) as sketch
            FROM VALUES (1) AS tab(col));
  1
```

The inner query creates two HLL sketches:

- The first SELECT statement creates a sketch from a single value of 1.
- The second SELECT statement creates a sketch from another single value of 1, but with a precision of 20.

The outer query uses the HLL_UNION_AGG function to combine the two sketches into a single sketch. Then it applies the HLL_SKETCH_ESTIMATE function to this combined sketch to estimate the unique count of values.

The result of this query is the estimated unique count of the values in the `col` column, which is 1. This means that the two input values of 1 are considered to be unique, even though they have the same value.

The second example includes a different precision parameter for the HLL_UNION_AGG function. In this case, both HLL sketches are created with a precision of 14 bits, which allows them to be successfully combined using `hll_union_agg` with the `true` parameter.

```
SELECT hll_sketch_estimate(hll_union_agg(sketch, true))
    FROM (SELECT hll_sketch_agg(col, 14) as sketch
            FROM VALUES (1) AS tab(col)
          UNION ALL
          SELECT hll_sketch_agg(col, 14) as sketch
            FROM VALUES (1) AS tab(col));
  1
```

The final result of the query is the estimated unique count, which in this case is also 1. This means that the two input values of 1 are considered to be unique, even though they have the same value.

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.

Topics

- GET_JSON_OBJECT function
- TO_JSON function

GET_JSON_OBJECT function

The GET_JSON_OBJECT function extracts a json object from `path`.

Syntax

```
get_json_object(json_txt, path)
```

Arguments

json_txt

A STRING expression containing well formed JSON.

path

A STRING literal with a well formed JSON path expression.

Returns

Returns a STRING.

A NULL is returned if the object can't be found.

Example

The following example extracts a value from a JSON object.. The first argument is a JSON string that represents a simple object with a single key-value pair. The second argument is a JSON path expression. The $ symbol represents the root of the JSON object, and the `.a` part specifies that we want to extract the value associated with the "a" key. The output of the function is 'b', which is the value associated with the "a" key in the input JSON object.

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

TO_JSON function

The TO_JSON function converts an input expression into a JSON string representation. The function handles the conversion of different data types (such as numbers, strings, and booleans) into their corresponding JSON representations.

The TO_JSON function is useful when you need to convert structured data (such as database rows or JSON objects) into a more portable, self-describing format like JSON. This can be particularly helpful when you need to interact with other systems or services that expect JSON-formatted data.

Syntax

```
to_json(expr[, options])
```

Arguments

expr

The input expression that you want to convert to a JSON string. It can be a value, a column, or any other valid SQL expression.

options

An optional set of configuration options that can be used to customize the JSON conversion process. These options may include things like the handling of null values, the representation of numeric values, and the treatment of special characters..

Returns

Returns a JSON string with a given struct value

Examples

The following example converts a named struct (a type of structured data) into a JSON string. The first argument (`named_struct('a', 1, 'b', 2)`) is the input expression that is passed to the `to_json()` function. It creates a named struct with two fields: "a" with a value of 1, and "b" with a value of 2. The to_json() function takes the named struct as its argument and converts it into a JSON string representation. The output is `{"a":1,"b":2}`, which is a valid JSON string that represented the named struct.

```
SELECT to_json(named_struct('a', 1, 'b', 2));
 {"a":1,"b":2}
```

The following example converts a named struct that contains a timestamp value into a JSON string, with a customized timestamp format. The first argument (`named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd'))`) creates a named struct with a single field 'time' that contains the timestamp value. The second argument (`map('timestampFormat', 'dd/MM/yyyy')`) creates a map (key-value dictionary) with a single key-value pair, where the key is 'timestampFormat' and the value is 'dd/MM/yyyy'. This map is used to specify the desired format for the timestamp value when converting it to JSON. The to_json() function converts the named struct into a JSON string. The second argument, the map, is used to customize the timestamp format to 'dd/MM/yyyy'. The output is `{"time":"26/08/2015"}`, which is a JSON string with a single field 'time' that contains the timestamp value in the desired 'dd/MM/yyyy' format.

```
SELECT to_json(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')),
 map('timestampFormat', 'dd/MM/yyyy'));
 {"time":"26/08/2015"}
```

Math functions

This section describes the mathematical operators and functions supported in AWS Clean Rooms Spark SQL.

Topics

- ROUND function
- SIGN function
- SIN function
- 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

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 |
+---------+
```

DIV function

The DIV operator returns the integral part of the division of dividend by divisor.

Syntax

```
dividend div divisor
```

Arguments

dividend

An expression that evaluates to a numeric or interval.

divisor

A matching interval type if `dividend` is an interval, a numeric otherwise.

Return type

`BIGINT`

Examples

The following example selects two columns from the squirrels table: the `id` column, which contains the unique identifier for each squirrel, and a `calculated` column, `age div 2`, which represents the integer division of the age column by 2. The `age div 2` calculation performs integer division on the `age` column, effectively rounding down the age to the nearest even integer. For example, if the `age` column contains values like 3, 5, 7, and 10, the `age div 2` column would contain the values 1, 2, 3, and 5, respectively.

```
SELECT id, age div 2 FROM squirrels
```

This query can be useful in scenarios where you need to group or analyze data based on age ranges, and you want to simplify the age values by rounding them down to the nearest even integer. The resulting output would provide the id and the age divided by 2 for each squirrel in the squirrels table.

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.

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 logarithm of `expr` with `base`.

Syntax

```
LOG(base, expr)
```

Argument

expr

The expression must have an integer, decimal, or floating-point data type.

base

The base for the logarithm calculation. Must be a positive number (not equal to 1) of double precision data type.

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(10, 100);
--------
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

```
{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 |
+-------------------+
```

RAND function

The RAND function generates a random floating-point number between 0 and 1. The RAND function generates a new random number each time it's called.

Syntax

```
RAND()
```

Return type

RANDOM returns a DOUBLE.

Example

The following example generates a column of random floating-point numbers between 0 and 1 for each row in the `squirrels` table. The resulting output would be a single column containing a list of random decimal values, with one value for each row in the squirrels table.

```
SELECT rand() FROM squirrels
```

This type of query is useful when you need to generate random numbers, for example, to simulate random events or to introduce randomness into your data analysis. In the context of the `squirrels` table, it might be used to assign random values to each squirrel, which could then be used for further processing or analysis.

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)
```

Scalar functions

This section describes the scalar functions supported in AWS Clean Rooms Spark SQL. A scalar function is a function that takes one or more values as input and returns a single value as output. Scalar functions operate on individual rows or elements and produce a single result for each input.

Scalar functions, such as SIZE, are different from other types of SQL functions, such as aggregate functions (count, sum, avg) and table-generating functions (explode, flatten). These other function types operate on multiple rows or generate multiple rows, whereas scalar functions work on individual rows or elements.

Topics

SIZE function

The SIZE function takes an existing array, map, or string as an argument and returns a single value representing the size or length of that data structure. It doesn't create a new data structure. It's used for querying and analyzing the properties of existing data structures, rather than for creating new ones.

This function is a useful for determining the number of elements in an array or the length of a string. It can be particularly helpful when working with arrays and other data structures in SQL, because it allows you to get information about the size or cardinality of the data.

Syntax

```
size(expr)
```

Arguments

expr

An ARRAY, MAP, or STRING expression.

Return type

The SIZE function returns an INTEGER.

Example

In this example, the SIZE function is applied to the array `['b', 'd', 'c', 'a']`, and it returns the value 4, which is the number of elements in the array.

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

In this example, the SIZE function is applied to the map `{'a': 1, 'b': 2}`, and it returns the value 2, which is the number of key-value pairs in the map.

```
SELECT size(map('a', 1, 'b', 2));
 2
```

In this example, the SIZE function is applied to the string `'hello world'`, and it returns the value 11, which is the number of characters in the string.

```
SELECT size('hello world');
11
```

String functions

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.

Topics

- SUBSTRING function
- TRANSLATE function
- TRIM function
- UPPER function
- UUID function

|| (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
```

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)
```

FORMAT_STRING function

The FORMAT_STRING function creates a formatted string by substituting placeholders in a template string with the provided arguments. It returns a formatted string from printf-style format strings.

The FORMAT_STRING function works by replacing the placeholders in the template string with the corresponding values passed as arguments. This type of string formatting can be useful when you need to dynamically construct strings that include a mix of static text and dynamic data, such as when generating output messages, reports, or other types of informative text. The FORMAT_STRING function provides a concise and readable way to create these types of formatted strings, making it easier to maintain and update the code that generates the output.

Syntax

```
format_string(strfmt, obj, ...)
```

Arguments

strfmt

A STRING expression.

obj

A STRING or numeric expression.

Return type

FORMAT_STRING returns a STRING.

Example

The following example contains a template string that contains two placeholders: `%d` for a decimal (integer) value, and `%s` for a string value. The `%d` placeholder is replaced with the decimal (integer) value (`100`), and the %s placeholder is replaced with the string value (`"days"`). The output is a template string with the placeholders replaced by the provided arguments: `"Hello World 100 days"`.

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

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)
```

LENGTH 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 doesn't 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.

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.

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)
```

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 doesn't 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.

SPLIT function

The SPLIT function allows you to extract substrings from a larger string and work with them as an array. The SPLIT function is useful when you need to break down a string into individual components based on a specific delimiter or pattern.

Syntax

```
split(str, regex, limit)
```

Arguments

str

A string expression to split.

regex

A string representing a regular expression. The *regex* string should be a Java regular expression.

limit

An integer expression which controls the number of times the *regex* is applied.

- limit > 0: The resulting array's length will not be more than limit, and the resulting array's last entry will contain all input beyond the last matched *regex*.
- limit <= 0: *regex* will be applied as many times as possible, and the resulting array can be of any size.

Return type

The SPLIT function returns an ARRAY<STRING>.

If `limit > 0`: The resulting array's length will not be more than limit, and the resulting array's last entry will contain all input beyond the last matched regex.

If `limit <= 0`: regex will be applied as many times as possible, and the resulting array can be of any size.

Example

In this example, the SPLIT function splits the input string `'oneAtwoBthreeC'` wherever it encounters the characters `'A'`, `'B'`, or `'C'` (as specified by the regular expression pattern `'[ABC]'`). The resulting output is an array of four elements: `"one"`, `"two"`, `"three"`, and an empty string `""`.

```
SELECT split('oneAtwoBthreeC', '[ABC]');
 ["one","two","three",""]
```

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
```

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(charactestring FROM start_position [ FOR numbecharacters ] )
```

```
SUBSTRING(charactestring, start_position, numbecharacters )
```

```
SUBSTRING(binary_expression, start_byte, numbebytes )
```

```
SUBSTRING(binary_expression, start_byte )
```

Arguments

charactestring

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.

numbecharacters

The number of characters to extract (the length of the substring). The *numbecharacters* 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.

numbebytes

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* + *numbecharacters* 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` + `numbecharacters` -1. For example:

```
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)
```

If `start_position` + `numbecharacters` -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
```

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 function 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)
```

UUID function

The UUID function generates a Universally Unique Identifier (UUID).

UUIDs are globally unique identifiers that are commonly used to provide unique identifiers for various purposes, such as:

- Identifying database records or other data entities.
- Generating unique names or keys for files, directories, or other resources.
- Tracking and correlating data across distributed systems.
- Providing unique identifiers for network packets, software components, or other digital assets.

The UUID function generates a UUID value that is unique with a very high probability, even across distributed systems and over long periods of time. UUIDs are typically generated using a combination of the current timestamp, the computer's network address, and other random or pseudo-random data, ensuring that each generated UUID is highly unlikely to conflict with any other UUID.

In the context of a SQL query, the UUID function can be used to generate unique identifiers for new records being inserted into a database, or to provide unique keys for data partitioning, indexing, or other purposes where a unique identifier is required.

> **Note**
>
> The UUID function is non-deterministic.

Syntax

```
uuid()
```

Arguments

The UUID function takes no argument.

Return type

UUID returns a universally unique identifier (UUID) string. The value is returned as a canonical UUID 36-character string.

Example

The following example generates a Universally Unique Identifier (UUID). The output is a 36-character string representing a Universally Unique Identifier.

```
SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

Privacy-related functions

AWS Clean Rooms provides functions to help you comply with privacy-related compliance for the following specifications.

- **Global Privacy Platform (GPP) –** A specification from the Interactive Advertising Bureau (IAB) that establishes a global, standardized framework for online privacy and use of data. For more information about the technical specifications of the GPP, see the [Global Privacy Platform documentation on GitHub](#).
- **Transparency and Consent Framework (TCF) –** A key component of the GPP, launched in 2020, which provides a standardized technical framework to help companies comply with privacy regulations such as the EU General Data Protection Regulation (GDPR). The TCF enables

customers to grant or withhold consent to data collection and processing. For more information about the technical specifications of TCF, see the TCF documentation on GitHub.

Topics

- consent_gpp_v1_decode function
- consent_tcf_v2_decode function

consent_gpp_v1_decode function

The `consent_gpp_v1_decode` function is used to decode Global Privacy Platform (GPP) v1 consent data. It takes the encoded consent string as input and returns the decoded consent data, which includes information about the user's privacy preferences and consent choices. This function is useful when working with data that includes GPP v1 consent information, as it allows you to access and analyze the consent data in a structured format.

Syntax

```
consent_gpp_v1_decode(gpp_string)
```

Arguments

gpp_string

The encoded GPP v1 consent string.

Returns

The returned dictionary includes the following key-value pairs:

- `version`: The version of the GPP specification used (currently 1).
- `cmpId`: The ID of the Consent Management Platform (CMP) that encoded the consent string.
- `cmpVersion`: The version of the CMP that encoded the consent string.
- `consentScreen`: The ID of the screen in the CMP UI where the user provided consent.
- `consentLanguage`: The language code of the consent information.
- `vendorListVersion`: The version of the vendor list used.

- `publisherCountryCode`: The country code of the publisher.
- `purposeConsent`: A list of integers representing the purposes for which the user has consented to.
- `purposeLegitimateInterest`: A list of purpose IDs for which the user's legitimate interest has been transparently communicated.
- `specialFeatureOptIns`: A list of integers representing the special features that the user has opted into.
- `vendorConsent`: A list of vendor IDs that the user has consented to.
- `vendorLegitimateInterest`: A list of vendor IDs for which the user's legitimate interest has been transparently communicated.

Example

The following example takes a single argument, which is the encoded consent string. It returns a dictionary containing the decoded consent data, including information about the user's privacy preferences, consent choices, and other metadata.

```
SELECT * FROM consent_gpp_v1_decode('ABCDEFGHIJK');
```

The basic structure of the returned consent data includes information about the consent string version, the CMP (Consent Management Platform) details, the user's consent and legitimate interest choices for different purposes and vendors, and other metadata.

```
{
    "version": 1,
    "cmpId": 12,
    "cmpVersion": 34,
    "consentScreen": 5,
    "consentLanguage": "en",
    "vendorListVersion": 89,
    "publisherCountryCode": "US",
    "purposeConsent": [1],
    "purposeLegitimateInterests": [1],
    "specialFeatureOptins": [1],
    "vendorConsent": [1],
    "vendorLegitimateInterests": [1]}
}
```

consent_tcf_v2_decode function

The `consent_tcf_v2_decode` function is used to decode Transparency and Consent Framework (TCF) v2 consent data. It takes the encoded consent string as input and returns the decoded consent data, which includes information about the user's privacy preferences and consent choices. This function is useful when working with data that includes TCF v2 consent information, as it allows you to access and analyze the consent data in a structured format.

Syntax

```
consent_tcf_v2_decode(tcf_string)
```

Arguments

tcf_string

The encoded TCF v2 consent string.

Returns

The `consent_tcf_v2_decode` function returns a dictionary containing the decoded consent data from a Transparency and Consent Framework (TCF) v2 consent string.

The returned dictionary includes the following key-value pairs:

Core segment

- `version`: The version of the TCF specification used (currently 2).
- `created`: The date and time when the consent string was created.
- `lastUpdated`: The date and time when the consent string was last updated.
- `cmpId`: The ID of the Consent Management Platform (CMP) that encoded the consent string.
- `cmpVersion`: The version of the CMP that encoded the consent string.
- `consentScreen`: The ID of the screen in the CMP UI where the user provided consent.
- `consentLanguage`: The language code of the consent information.
- `vendorListVersion`: The version of the vendor list used.
- `tcfPolicyVersion`: The version of the TCF policy that the consent string is based on.
- `isServiceSpecific`: A Boolean value indicating whether the consent is specific to a particular service or applies to all services.

- `useNonStandardStacks`: A Boolean value indicating whether non-standard stacks are used.
- `specialFeatureOptIns`: A list of integers representing the special features that the user has opted into.
- `purposeConsent`: A list of integers representing the purposes for which the user has consented to.
- `purposesLITransparency`: A list of integers representing the purposes for which the user has given legitimate interest transparency.
- `purposeOneTreatment`: A Boolean value indicating whether the user has requested the "purpose one treatment" (that is, all purposes are treated equally).
- `publisherCountryCode`: The country code of the publisher.
- `vendorConsent`: A list of vendor IDs that the user has consented to.
- `vendorLegitimateInterest`: A list of vendor IDs for which the user's legitimate interest has been transparently communicated.
- `pubRestrictionEntry`: A list of publisher restrictions. This field contains the Purpose ID, Restriction Type, and List of Vendor IDs under that Purpose restriction.

Disclosed vendor segment

- `disclosedVendors`: A list of integers representing the vendors that have been disclosed to the user.

Publisher purposes segment

- `pubPurposesConsent`: A list of integers representing the publisher-specific purposes for which the user has given consent.
- `pubPurposesLITransparency`: A list of integers representing the publisher-specific purposes for which the user has given legitimate interest transparency.
- `customPurposesConsent`: A list of integers representing the custom purposes for which the user has given consent.
- `customPurposesLITransparency`: A list of integers representing the custom purposes for which the user has given legitimate interest transparency.

This detailed consent data can be used to understand and respect the user's privacy preferences when working with personal data.

Example

The following example takes a single argument, which is the encoded consent string. It returns a dictionary containing the decoded consent data, including information about the user's privacy preferences, consent choices, and other metadata.

```
from aws_clean_rooms.functions import consent_tcf_v2_decode

consent_string = "CO1234567890abcdef"
consent_data = consent_tcf_v2_decode(consent_string)

print(consent_data)
```

The basic structure of the returned consent data includes information about the consent string version, the CMP (Consent Management Platform) details, the user's consent and legitimate interest choices for different purposes and vendors, and other metadata.

```

    /** core segment **/
    version: 2,
    created: "2023-10-01T12:00:00Z",
    lastUpdated: "2023-10-01T12:00:00Z",
    cmpId: 1234,
    cmpVersion: 5,
    consentScreen: 1,
    consentLanguage: "en",
    vendorListVersion: 2,
    tcfPolicyVersion: 2,
    isServiceSpecific: false,
    useNonStandardStacks: false,
    specialFeatureOptIns: [1, 2, 3],
    purposeConsent: [1, 2, 3],
    purposesLITransparency: [1, 2, 3],
    purposeOneTreatment: true,
    publisherCountryCode: "US",
    vendorConsent: [1, 2, 3],
    vendorLegitimateInterest: [1, 2, 3],
    pubRestrictionEntry: [
        { purpose: 1, restrictionType: 2, restrictionDescription: "Example
 restriction" },
    ],
```

```
    /** disclosed vendor segment **/
    disclosedVendors: [1, 2, 3],

    /** publisher purposes  segment **/
    pubPurposesConsent: [1, 2, 3],
    pubPurposesLITransparency: [1, 2, 3],
    customPurposesConsent: [1, 2, 3],
    customPurposesLITransparency: [1, 2, 3],
};
```

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 Spark SQL supports two types of window functions: aggregate and ranking.

Following are the supported aggregate functions:

- CUME_DIST window function
- DENSE_RANK window function
- FIRST window function
- FIRST_VALUE window function
- LAG window function
- LAST window function
- LAST_VALUE window function
- LEAD window function

Following are the supported ranking functions:

- DENSE_RANK 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	

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 window function

Given an ordered set of rows, FIRST 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 window function.

Syntax

```
FIRST( 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, 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 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 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(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
-----------+------------+--------------------------------
+------------------------------
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
...
```

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 window function

Given an ordered set of rows, The LAST 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 window function.

Syntax

```
LAST( 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 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(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
-----------+------------+--------------------------------
+------------------------------
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
...
```

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)
```

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.

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)
```

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.

AWS Clean Rooms Spark SQL conditions

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.

Syntax

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
| IN_condition
```

> **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 Spark SQL.

Topics

- Comparison operators
- Logical conditions
- Pattern-matching conditions
- BETWEEN range condition
- Null condition
- EXISTS condition

- IN condition

Comparison operators

Comparison conditions state logical relationships between two values. All comparison conditions are binary operators with a Boolean return type.

AWS Clean Rooms Spark SQL supports the comparison operators described in the following table.

Operator	Syntax	Description
!	`!expression`	The logical `NOT` operator. Used to negate a boolean expression, meaning it returns the opposite of the expressio n's value. The ! operator can also be combined with other logical operators, such as AND and OR, to create more complex boolean expressions.
<	`a < b`	The less than comparison operator. Used to compare two values and determine if the value on the left is less than the value on the right.
>	`a > b`	The greater than compariso n operator. Used to compare two values and determine if the value on the left is greater than the value on the right.
<=	`a <= b`	The less than or equal to comparison operator. Used

Operator	Syntax	Description
		to compare two values and returns `true` if the value on the left is less than or equal to the value on the right, and `false` otherwise.
>=	`a >= b`	The greater than or equal to comparison operator. Used to compare two values and determine if the value on the left is greater than or equal to the value on the right.
=	`a = b`	The equality comparison operator, which compares two values and returns `true` if they're equal, and `false` otherwise.
<> or !=	`a <> b` or `a != b`	The not equal to comparison operator, which compares two values and returns `true` if they're not equal, and `false` otherwise.

Operator	Syntax	Description
==	`a == b`	The standard equality comparison operator, which compares two values and returns `true` if they're equal, and `false` otherwise. **Note** The == operator is case-sensitive when comparing string values. If you need to perform a case-insensitive comparison, you can use functions like UPPER() or LOWER() to convert the values to the same case before the comparison.

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 the id values for all the squirrels that are not currently foraging.

```
SELECT id FROM squirrels
WHERE !is_foraging
```

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 Spark SQL 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.

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.

Syntax

```
expression [ NOT ] LIKE | pattern [ ESCAPE 'escape_char' ]
```

Arguments

expression

A valid UTF-8 character expression, such as a column name.

LIKE

LIKE performs a case-sensitive pattern match. 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 predicates don't implicitly ignore trailing spaces. To ignore trailing spaces, use RTRIM or explicitly cast a CHAR column to VARCHAR.

The ~~ operator is equivalent to LIKE. Also the `!~~` operator is equivalent to NOT LIKE.

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' 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". :

```
select distinct city from users where city like '__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)
```

RLIKE

The RLIKE operator allows you to check if a string matches a specified regular expression pattern.

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

Syntax

```
rlike(str, regexp)
```

Arguments

str

A string expression

regexp

A string expression. The regex string should be a Java regular expression.

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

Examples

The following example sets the value of the `spark.sql.parser.escapedStringLiterals` configuration parameter to `true`. This parameter is specific to the Spark SQL engine. The `spark.sql.parser.escapedStringLiterals` parameter in Spark SQL controls how the SQL parser handles escaped string literals. When set to `true`, the parser will interpret backslash characters (\) within string literals as escape characters, allowing you to include special characters like newlines, tabs, and quotation marks within your string values.

```
SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
```

For example, with `spark.sql.parser.escapedStringLiterals=true`, you could use the following string literal in your SQL query:

```
SELECT 'Hello, world!\n'
```

The newline character `\n` would be interpreted as a literal newline character in the output.

The following example performs a regular expression pattern match. The first argument is passed to the RLIKE operator. It's a string that represents a file path, where the actual username is replaced with the pattern '****'. The second argument is the regular expression pattern used for the matching. The output (`true`) indicates that the first string (`'%SystemDrive%\Users\****'`) matches the regular expression pattern (`'%SystemDrive%\\Users.*'`).

```
SELECT rlike('%SystemDrive%\Users\John', '%SystemDrive%\Users.*');
true
```

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)
```

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
Spark SQL supports Hints	AWS Clean Rooms Spark SQL supports query hints to optimize query performance and reduce compute costs.	January 20, 2026
Spark SQL supports CACHE TABLE	AWS Clean Rooms Spark SQL supports the CACHE TABLE command, which allows customers to cache existing tables or create and cache new tables from query results for improved query performance.	October 22, 2025
Spark SQL supports FIRST and LAST Window functions	AWS Clean Rooms Spark SQL supports the following Window functions: FIRST and LAST.	June 12, 2025
Spark SQL function documentation updates	Documentation-only update to accurately reflect supported Spark SQL functions. Removed documentation for 25 unsupported functions , including <=> operator,	May 20, 2025

	SIMILAR TO, LISTAGG, and ARRAY_INSERT. Corrected function names from DATEADD to DATE_ADD, DATEDIFF to DATE_DIFF , ISNULL to IS_NULL, and ISNOTNULL to IS_NOT_NU LL. Fixed a typo in DATE_PART examples.	
AWS Clean Rooms Spark SQL	Customers can now run queries using some SQL conditions, functions, commands, and conventions supported with the Spark SQL analytics engine.	October 29, 2024
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_T IMEZONE, 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