SQL 参考

AWS Clean Rooms

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Clean Rooms: SQL 参考

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

中的 SQL 概述 AWS Clean Rooms

欢迎使用《AWS Clean Rooms SQL 参考》。

AWS Clean Rooms围绕行业标准的结构化查询语言 (SQL) 构建，结构化查询语言是一种由用于处理数据库和数据库对象的命令和函数组成的查询语言。SQL 还会强制实施有关数据类型、表达式和文本使用的规则。

以下主题提供有关本 SQL 参考中使用的约定和命名规则的一般信息。

主题

- SQL 参考惯例
- SQL 命名规则
- SQL 引擎支持的数据类型

以下各节提供有关可以在中使用的文字、数据类型、SQL 命令、SQL 函数类型和 SQL 条件的信息。AWS Clean Rooms

- AWS Clean Rooms 火花 SQL

有关的更多信息AWS Clean Rooms，请参阅《AWS Clean Rooms用户指南》和《AWS Clean Rooms API 参考》。

SQL 参考惯例

本节介绍用于为 SQL 表达式、命令和函数编写语法的约定。

字符	描述
CAPS	采用大写字母的字样为关键字。
[ ]	方括号表示可选参数。方括号中的多个参数表示您可选择任意数量的参数。此外，不同行的括号中的参数表示分析程序需要按照参数在语法中列出的顺序获取参数。
{ }	大括号表示您需要选择大括号内的参数之一。

字符	描述
|	管道表示您可以在不同参数之间选择。
斜体	斜体字样表示占位符。必须插入适当的值以替换斜体字样。
...	省略号表示可以重复前面的元素。
'	单引号中的字样表示必须键入引号。

SQL 命名规则

以下各节说明了 AWS Clean Rooms中的 SQL 命名规则。

主题

配置表关联名称和列

可以查询的成员使用配置表关联名称作为查询中的表名。配置表关联名称和配置表列可以在查询中使用别名。

以下命名规则适用于配置表关联名称、配置表的列名和别名：

- 它们只能使用字母数字、下划线 (_) 或连字符 (-)，但不能以连字符开头或结尾。
  - （仅限自定义分析规则）他们可以使用美元符号 ($)，但不能使用遵循美元引号字符串常量的模式。

    用美元括起来的字符串常量包括：
    - 一个美元符号 ($)
    - 零个或多个字符（可选“标签”）
    - 另一个美元符号
    - 构成字符串内容的任意字符序列
    - 一个美元符号 ($)

  - 以美元引号开头的同一个标签
  - 一个美元符号

    例如：`$$invalid$$`
- 它们不能包含连续的连字符 (-)。
- 它们不能以以下任何前缀开头：

  `padb_`, `pg_`, `stcs_`, `stl_`, `stll_`, `stv_`, `svcs_`, `svl_`, `svv_`, `sys_`, `systable_`
- 它们不能包含反斜杠字符 (\)、引号 (') 或非双引号的空格。
- 如果它们以非字母字符开头，则必须位于双引号 (" ") 中。
- 如果它们包含连字符 (-)，则必须位于双引号 (" ") 内。
- 它们的长度必须在 1 到 127 个字符之间。
- 保留字必须位于双引号 (" ") 内。
- 以下列名已保留，不能用于 AWS Clean Rooms （即使带引号也是如此）：
  - oid
  - tableoid
  - xmin
  - cmin
  - xmax
  - cmax
  - ctid

保留字

以下是中的保留字列表 AWS Clean Rooms。

AES128	DELTA32KDESC	LEADING	PRIMARY
AES256ALL	DISTINCT	LEFTLIKE	RAW
ALLOWOVER WRITEANALYSE	DO	LIMIT	READRATIO

ANALYZE	DISABLE	LOCALTIME	RECOVERREFERENCES
AND	ELSE	LOCALTIMESTAMP	REJECTLOG
ANY	EMPTYASNULLENABLE	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
BLANKSASNULLBOTH	FREEZE	NEW	SYSDATESYSTEM
BYTEDICT	FROM	NOT	TABLE
BZIP2CASE	FULL	NOTNULL	TAG
CAST	GLOBALDICT256	NULL	TDES
CHECK	GLOBALDICT64KGRANT	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	TRUNCATECOLUMNSUNION
CURRENT_TIME	INNER	OR	UNIQUE
CURRENT_TIMESTAMP	INTERSECT	ORDER	UNNEST
CURRENT_USER	INTERVAL	OUTER	USING
CURRENT_USER_IDDEFAULT	INTO	OVERLAPS	VERBOSE
DEFERRABLE	IS	PARALLELPARTITION	WALLETWHEN
DEFLATE	ISNULL	PERCENT	WHERE
DEFRAG	JOIN	PERMISSIONS	WITH
DELTA	LANGUAGE	PIVOTPLACING	WITHOUT

SQL 引擎支持的数据类型

AWS Clean Rooms 支持多个 SQL 引擎和方言。了解这些实施中的数据类型系统对于成功进行数据协作和分析至关重要。下表显示了 SQL、Snowflak AWS Clean Rooms e SQL 和 Spark SQL 中的等效数据类型。

数值数据类型

数字类型表示各种数字，从精确的整数到近似的浮点值。数字类型的选择会影响存储要求和计算精度。整数类型因字节大小而异，而十进制和浮点类型则提供不同的精度和缩放选项。

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
8 字节整数	BIGINT	不支持	BIGINT，长	从-9,223,372,036,036,854,775,808到9,223,372,036,854,775,807的带符号整数。
4 字节整数	INT	不支持	INT，INTEGER	从 -2,147,483,648 到 2,147,483,647 之间的带符号整数
2 字节整数	SMALLINT	不支持	小，短	从 -32,768 到 32,767 之间的带符号整数
1 字节整数	不支持	不支持	TINYINT，BYTE	从 -128 到 127 之间的有符号整数
双精度浮动	双精度、双精度	浮点型、FLOAT4、FLOAT8、双精度、双精度、实数	DOUBLE	8 字节双精度浮点数
单精度浮球	真实，浮动	不支持	FLOAT	4 字节单精度浮点数
十进制（固定精度）	DECIMAL	十进制、数字、数字	十进制、数字、	任意精度的带符号十进制数

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
		Note Snowflake 会自动将较小宽度的精确数字类型（INT、BIGINT SMALLINT 等）别名为数字。		
十进制（精确）	十进制 (p)	十进制 (p)、数字 (p)	十进制 (p)	固定精度的十进制数字
十进制（带刻度）	DECIMAL (p,s)	十进制 (p, s)、数字 (p, s)	DECIMAL (p,s)	带刻度的固定精度十进制数字

布尔数据类型

布尔类型表示简单的 true/false 逻辑值。这些类型在 SQL 引擎中是一致的，通常用于标志、条件和逻辑操作。

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
布尔值	BOOLEAN	BOOLEAN	BOOLEAN	代表 true/false 值

日期和时间数据类型

日期和时间类型处理时态数据，其精度和时区感知程度各不相同。这些类型支持不同的格式来存储日期、时间和时间戳，并提供包含或排除时区信息的选项。

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
日期	DATE	DATE	DATE	不带时区的日期值（年、月、日）
时间	TIME	不支持	不支持	一天中的时间，以 UTC 表示，不含时区
与 TZ 共度时光	TIMETZ	不支持	不支持	一天中的时间（UTC），含时区
Timestamp	TIMESTAMP	时间戳，TIMESTAMP_NTZ	TIMESTAMP_NTZ	Timestamp without time zone **Note** NTZ 表示 “无时区”
带有 TZ 的时间戳	TIMESTAMPTZ	TIMESTAMP_LTZ	时间戳，TIMESTAMP_LTZ	带有本地时区的时间戳

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
				Note LTZ 表示 “本地时区”

字符数据类型

字符类型存储文本数据，提供固定长度和可变长度选项。这些类型处理文本字符串和二进制数据，并具有可选的长度规格来控制存储分配。

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
固定长度的字符	CHAR	CHAR，CHARACTER	CHAR，CHARACTER	固定长度字符串
带长度的固定长度字符	CHAR(n)	CHAR(n)，CHARACTER(n)	CHAR(n)，CHARACTER(n)	具有指定长度的固定长度字符串
长度可变的字符	VARCHAR	VARCHAR、字符串、文本	VARCHAR，字符串	长度可变的字符串
带长度的可变长度字符	VARCHAR (n)	VARCHAR (n)、字符串 (n)、文本 (n)	VARCHAR (n)	有长度限制的可变长度字符串
二元	VARBYTE	BINARY、VARBINARY	BINARY	二进制字节序列
带长度的二进制	VARBYTE(n)	不支持	不支持	有长度限制的二进制字节序列

结构化数据类型

结构化类型允许通过将多个值组合成单个字段来组织复杂的数据。其中包括用于有序集合的数组、键值对的映射以及用于使用命名字段创建自定义数据结构的结构。

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
数组	数组 <type>	数组（类型）	数组 <type>	相同类型元素的有序序列 Note 数组类型必须包含相同类型的元素
Map	地图<key, value>	MAP（键、值）	地图<key, value>	键值对的集合 Note 地图类型必须包含相同类型的元素
结构体	结构< field1: type1, field2: type2>	对象（字段 1 类型 1，字段 2 类型 2）	结构< field1: type1, field2: type2 >	包含指定类型的命名字段的结构 Note 不同实现的结

数据类型	AWS Clean Rooms SQL	雪花 SQL	Spark SQL	说明
				构化类型语法可能略有不同
Super	SUPER	不支持	不支持	灵活的类型，支持所有数据类型，包括复杂类型

AWS Clean Rooms 火花 SQL

AWS Clean Rooms Spark SQL 强制执行有关使用数据类型、表达式和文字的规则。

有关 AWS Clean Rooms Spark SQL 的更多信息，请参阅AWS Clean Rooms 用户指南和 AWS Clean Rooms API 参考。

以下主题提供有关 AWS Clean Rooms Spark SQL 中支持的文字、数据类型、命令、函数和条件的信息。

主题

文本

文本或常量是固定数据值，由一系列字符或数字常量组成。

AWS Clean Rooms Spark SQL 支持多种类型的文字，包括：

- 整数、小数和浮点数的数字文本。
- 字符字面值，也称为字符串、字符串或字符常量，用于指定字符串值。
- 与日期时间数据类型一起使用的日期、时间和时间戳文本。有关更多信息，请参阅 日期、时间和时间戳文本。
- 间隔文本。有关更多信息，请参阅 间隔文本。
- 布尔字面值。有关更多信息，请参阅 布尔字面值。
- 空文本，用于指定空值。
- 只有 TAB, CARRIAGE RETURN (CR)，以及 LINE FEED (LF) 支持 Unicode 通用类别 (Cc) 中的 Unicode 控制字符。

AWS Clean Rooms Spark SQL 不支持在 SELECT 子句中直接引用字符串文字，但可以在诸如 CAST 之类的函数中使用它们。

+ (串联) 运算符

连接数值文本、字符串文本和/或日期时间和时间间隔文本。它们位于 + 符号的两侧，并根据 + 符号两侧的输入返回不同的类型。

语法

```
numeric + string
```

```
date + time
```

```
date + timetz
```

参数的顺序可以反转。

参数

numeric literals

表示数字的文本或常量可以是整数或浮点。

string literals

字符串、字符字符串或字符常量

date

A DATE 列或隐式转换为 a 的表达式 DATE.

time

A TIME 列或隐式转换为 a 的表达式 TIME.

timetz

A TIMETZ 列或隐式转换为 a 的表达式 TIMETZ.

示例

以下示例表 TIME_TEST 有专栏 TIME_VAL （类型 TIME)，其中插入了三个值。

```
select date '2000-01-02' + time_val as ts from time_test;
```

数据类型

AWS Clean Rooms Spark SQL 存储或检索的每个值都有一个数据类型，其中包含一组固定的关联属性。数据类型是在创建表时声明的。数据类型约束了列或参数可包含的一组值。

下表列出了您可以在 AWS Clean Rooms Spark SQL 中使用的数据类型。

数据类型名称	数据类型	别名	说明
ARRAY	the section called “嵌套类型”	不适用	ARRAY 嵌套数据类型
BIGINT	the section called “数字类型”	不适用	有符号的八字节整数
BINARY	the section called “二进制类型”	不适用	字节序列值
BOOLEAN	the section called “布尔值类型”	BOOL	逻辑布尔值 (true/false)
BYTE	the section called “数字类型”	不适用	1 字节有符号整数，从 -128 到 127
CHAR	the section called “字符类型”	CHARACTER	固定长度字符串
DATE	the section called “日期时间类型”	不适用	日历日期（年、月、日）
DECIMAL	the section called “数字类型”	NUMERIC	可选精度的精确数字
FLOAT	the section called “数字类型”	FLOAT8，双精度	双精度浮点数

数据类型名称	数据类型	别名	说明
INTEGER	the section called “数字类型”	INT	有符号的四字节整数
INTERVAL	the section called “日期时间类型”	不适用	按日到时间顺序或按年到月顺序排列的时间持续时间
LONG	the section called “数字类型”	不适用	8 字节有符号整数
MAP	the section called “嵌套类型”	不适用	MAP 嵌套数据类型
REAL	the section called “数字类型”	FLOAT4	单精度浮点数
SHORT	the section called “数字类型”	不适用	2 字节有符号整数。
SMALLINT	the section called “数字类型”	不适用	有符号的二字节整数
STRUCT	the section called “嵌套类型”	不适用	STRUCT 嵌套数据类型
TIMESTAMP_LTZ	the section called “日期时间类型”	不适用	一天中的时间和当地时区
TIMESTAMP_NTZ	the section called “日期时间类型”	不适用	一天中没有时区的时间
TINYINT	the section called “数字类型”	不适用	1 字节有符号整数，从 -128 到 127
VARCHAR	the section called “字符类型”	CHARACTER VARYING	具有用户定义的限制的可变长度字符串，

> **Note**
>
> ARRAY、STRUCT 和 MAP 嵌套数据类型目前仅适用于自定义分析规则。有关更多信息，请参阅 嵌套类型。

多字节字符

VARCHAR 数据类型支持多达 4 个字节的 UTF-8 多字节字符。不支持 5 个字节或更长的字符。要计算包含多字节字符的 VARCHAR 列的大小，请用字符数乘以每个字符的字节数。例如，如果一个字符串包含四个中文字符，并且每个字符的长度为三个字节，则您需要一个 VARCHAR(12) 列才能存储该字符串。

VARCHAR 数据类型不支持下列无效的 UTF-8 代码点：

`0xD800 – 0xDFFF`（字节序列：`ED A0 80 – ED BF BF`）

CHAR 数据类型不支持多字节字符。

数字类型

数字数据类型包括整数、小数和浮点数。

主题

- 整数类型
- DECIMAL 或 NUMERIC 类型
- 浮点类型
- 数值计算

整数类型

使用以下数据类型来存储不同范围的整数。您无法存储每种类型所允许范围之外的值。

Name	存储	范围
SMALLINT	2 字节	-32768 到 +32767
SHORT	2 字节	-32768 到 +32767

Name	存储	范围
INTEGER 或 INT	4 字节	-2147483648 到 +2147483647
BIGINT	8 字节	-9223372036854775808 到 9223372036854775807
LONG	8 字节	-9223372036854775808 到 9223372036854775807

DECIMAL 或 NUMERIC 类型

使用 DECIMAL 或 NUMERIC 数据类型存储具有用户定义的精度 的值。DECIMAL 和 NUMERIC 关键字是可互换的。在本文档中，小数 是此数据类型的首选术语。术语数字 一般用于指整数、小数和浮点数据类型。

存储	范围
可变，对于未压缩的 DECIMAL 类型可以多达 128 位。	128 位有符号整数，精度位数可以多达 38 位。

通过指定*precision*和在表中定义 DECIMAL 列*scale*：

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

precision

整个值中有效位的总数：小数点两边的位数。例如，数字 48.2891 的精度为 6，小数位数为 4。如果未指定，默认精度为 18。最大精度为 38。

如果输入值中的小数点左侧的位数超出了列的精度减去其小数位数的差值，则此值无法复制到列中（或无法插入或更新）。此规则适用于列定义范围之外的任何值。例如，numeric(5,2) 列所允许的值范围为 -999.99 到 999.99。

scale

值的小数部分中小数点右侧的小数位数。整数的小数位数为零。在列规范中，小数位数值必须小于或等于精度值。如果未指定，默认小数位数为 0。最大小数位数为 37。

如果加载到表中的输入值的小数位数大于列的小数位数，则该值将四舍五入到指定小数位数。例如，SALES 表中的 PRICEPAID 列为 DECIMAL(8,2) 列。如果将 DECIMAL(8,4) 值插入到 PRICEPAID 列中，则该值将四舍五入到小数位数 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)
```

但是，对于显示强制转换表中选定的值而得出的结果，不会进行四舍五入。

> **Note**
>
> 您可插入到 DECIMAL(19,0) 列中的最大正值为 9223372036854775807 (2^{63} -1)。最大负值为 -9223372036854775807。例如，尝试插入值 9999999999999999999（19 个 9）将导致溢出错误。无论小数点的位置如何，AWS Clean Rooms 可表示为 DECIMAL 数的最大字符串为 9223372036854775807。例如，您可加载到 DECIMAL(19,18) 列中的最大值为 9.223372036854775807。
> 制定这些规则的原因如下：
>
> - 有效精度为 19 位或更少的 DECIMAL 值在内部存储为 8 字节整数。
> - 有效精度为 20-38 的 DECIMAL 值在内部存储为 16 字节整数。

有关使用 128 位 DECIMAL 或 NUMERIC 列的说明

请勿为 DECIMAL 列随意分配最高精度，除非您确定您的应用程序需要此精度。128 位值使用的磁盘空间是 64 位值的两倍，并且可能会降低查询执行速度。

浮点类型

使用 REAL 和 DOUBLE PRECISION 数据类型可存储具有可变精度 的数值。这些类型是不精确的 类型，意味着一些值是作为估计值存储的，因此存储和返回某个特定值可能导致细微的差异。如果您需要精确的存储和计算（如货币金额），请使用 DECIMAL 数据类型。

根据 IEEE 浮点运算标准 754，REAL 表示单精度浮点数格式。它的精度约为 6 位，范围在 1E-37 到 1E+37 之间。您也可以将此数据类型指定为 FLOAT4。

根据 IEEE 二进制浮点运算标准 754，DOUBLE PRECISION 表示双精度浮点数格式。它的精度约为 15 位，范围在 1E-307 到 1E+308 之间。您也可以将此数据类型指定为 FLOAT 或 FLOAT8。

数值计算

在中 AWS Clean Rooms，计算是指二进制数学运算：加法、减法、乘法和除法。此部分介绍这些运算的预期返回类型，以及在涉及 DECIMAL 数据类型时应用于确定精度和小数位数的特定公式。

当在查询处理期间计算数值时，您可能会遇到无法计算和查询返回数字溢出错误的情况。您还可能会遇到计算值的小数位数发生变化或出乎意料的情况。对于一些运算，您可使用显式强制转换（类型提升）或 AWS Clean Rooms 配置参数来解决这些问题。

有关类似使用 SQL 函数的计算的结果的信息，请参阅AWS Clean Rooms 火花 SQL 函数。

计算的返回类型

给定中支持的数值数据类型集 AWS Clean Rooms，下表显示了加法、减法、乘法和除法运算的预期返回类型。表左侧第一列表示计算中的第一个操作数，顶部行表示第二个操作数。

操作数 1	操作数 2	返回类型
SMALLINT 或简短	SMALLINT 或简短	SMALLINT 或简短
SMALLINT 或简短	INTEGER	INTEGER
SMALLINT 或简短	BIGINT	BIGINT
SMALLINT 或简短	DECIMAL	DECIMAL
SMALLINT 或简短	FLOAT4	FLOAT8
SMALLINT 或简短	FLOAT8	FLOAT8

操作数 1	操作数 2	返回类型
INTEGER	INTEGER	INTEGER
INTEGER	大或长	大或长
INTEGER	DECIMAL	DECIMAL
INTEGER	FLOAT4	FLOAT8
INTEGER	FLOAT8	FLOAT8
大或长	大或长	大或长
大或长	DECIMAL	DECIMAL
大或长	FLOAT4	FLOAT8
大或长	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL
DECIMAL	FLOAT4	FLOAT8
DECIMAL	FLOAT8	FLOAT8
FLOAT4	FLOAT8	FLOAT8
FLOAT8	FLOAT8	FLOAT8

计算的 DECIMAL 结果的精度和小数位数

下表汇总了在数学运算返回 DECIMAL 结果时用于计算生成的精度和小数位数的规则。在此表中，p1和s1表示计算中第一个操作数的精度和小数位数。 p2并s2表示第二个操作数的精度和小数位数。（不管这些计算如何，最大的结果精度为 38，最大的结果小数位数为 38）。

运算	结果精度和小数位数
+ 或者 -	小数位数 = `max(s1,s2)`

运算	结果精度和小数位数
	精度 = max(p1-s1,p2-s2)+1+scale
*	小数位数 = s1+s2 精度 = p1+p2+1
/	小数位数 = max(4,s1+p2-s2+1) 精度 = p1-s1+ s2+scale

例如，SALES 表中的 PRICEPAID 和 COMMISSION 列均为 DECIMAL(8,2) 列。如果您用 PRICEPAID 除以 COMMISSION（或者反过来），采用的公式如下所示：

```
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)
```

以下计算是使用集合运算符（如 UNION、INTERSECT 和 EXCEPT）或 COALESCE 和 DECODE 等函数计算对 DECIMAL 值执行的运算的最终精度和小数位数的一般规则：

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

例如，将 DEC1 具有一个十进制 (7,2) 列的 DEC2 表与具有一个十进制 (15,3) 列的表连接以创建表。DEC3 的架构 DEC3 显示该列变为数字 (15,3) 列。

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

在上例中，采用的公式如下所示：

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

Scale = max(2,3) = 3
```

```
Result = DECIMAL(15,3)
```

有关除法运算的说明

对于除法运算， divide-by-zero条件会返回错误。

在计算精度和小数位数之后，将应用小数位数最多为 100 的限制。如果计算所得的结果小数位数大于 100，则除法运算结果的范围如下所示：

- 精度 = `precision - (scale - max_scale)`
- 小数位数 = `max_scale`

如果计算所得的精度大于最大精度 (38)，则精度将减少为 38，小数位数的结果将介于以下范围：`max(38 + scale - precision), min(4, 100))`

溢出条件

将检查所有数值计算是否存在溢出情况。精度为 19 或 19 以下的 DECIMAL 数据存储为 64 位整数。精度大于 19 的 DECIMAL 数据存储为 128 位整数。所有 DECIMAL 值的最大精度为 38，最大小数位数为 37。当值超出这些限制时将出现溢出错误，中间结果集和最终结果集都存在这种情况：

- 当特定数据值不符合强制转换函数指定的请求精度或比例时，显式转换会导致运行时溢出错误。例如，您无法强制转换 SALES 表中 PRICEPAID 列（DECIMAL(8,2) 列）的所有值并返回 DECIMAL(7,3) 结果：

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

  此错误出现的原因是 PRICEPAID 列中的一些 较大的值无法强制转换。
- 乘法运算的乘积结果中的小数位数为所有操作数的小数位数之和。例如，如果两个操作数的小数位数都为 4，则结果的小数位数为 8，小数点左侧仅剩下 10 位。因此，在具有有效小数位数的两个大数相乘时，遇到溢出的情况相对容易一些。

INTEGER 和 DECIMAL 类型的数值计算

如果计算中的一个操作数具有 INTEGER 数据类型且另一个操作数为 DECIMAL，则 INTEGER 操作数将隐式强制转换为 DECIMAL：

- SMALLINT 或 SHORT 被转换为十进制 (5,0)
- INTEGER 将强制转换为 DECIMAL(10,0)
- BIGINT 或 LONG 被转换为十进制 (19,0)

例如，如果将 SALES.COMMISSION（DECIMAL(8,2) 列）和 SALES.QTYSOLD（SMALLINT 列）相乘，则此计算将强制转换为：

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

字符类型

字符数据类型包括 CHAR（字符）和 VARCHAR（字符变体）。

主题

- CHAR 或 CHARACTER
- VARCHAR 或 CHARACTER VARYING
- 尾部空格的意义

CHAR 或 CHARACTER

使用 CHAR 或 CHARACTER 列存储固定长度字符串。这些字符串将使用空格填补，因此 CHAR(10) 列始终占用 10 字节的存储。

```
char(10)
```

未指定长度的 CHAR 列将生成 CHAR(1) 列。

CHAR 和 VARCHAR 数据类型是按照字节而不是字符来定义的。CHAR 列只能包含单字节字符，因此 CHAR(10) 列可包含最大长度为 10 字节的字符串。

Name	存储	范围（列宽度）
CHAR 或 CHARACTER	字符串的长度，包括尾部空格（如有）	4096 字节

VARCHAR 或 CHARACTER VARYING

使用 VARCHAR 或 CHARACTER VARYING 列存储具有固定限制的可变长度字符串。这些字符串不会使用空格填补，因此 VARCHAR(120) 列最多包含 120 个单字节字符、60 个双字节字符、40 个三字节字符或 30 个四字节字符。

```
varchar(120)
```

VARCHAR 数据类型是根据字节而不是字符来定义的。VARCHAR 可包含多字节字符，并且每个字符最多可以有 4 个字节。例如，VARCHAR(12) 列可包含 12 个单字节字符、6 个双字节字符、4 个三字节字符或 3 个四字节字符。

名称	存储	范围（列宽度）
VARCHAR 或 CHARACTER VARYING	4 字节 + 字符的总字节，其中每个字符可为 1 至 4 个字节。	65535 字节 (64K -1)

尾部空格的意义

CHAR 和 VARCHAR 数据类型存储长度最多为 n 字节的字符串。尝试将更长的字符串存储到这些类型的列中将导致错误。但是，如果额外的字符全为空格，则字符串将截断至最大长度。如果字符串短于最大长度，CHAR 值将使用空格填补，但 VARCHAR 值将存储不带空格的字符串。

CHAR 值中的尾部空格始终无语义意义。当比较两个 CHAR 值时将忽视尾部空格，而不将其包含在 LENGTH 计算中，在将 CHAR 值转换为其他字符串类型时将删除尾部空格。

VARCHAR 和 CHAR 值中的尾部空格将在比较值时视为无语义意义。

长度计算将返回 VARCHAR 字符串的包含尾部空格在内的长度。尾部空格不会计入固定长度字符串的长度中。

日期时间类型

日期时间数据类型包括日期、时间、TIMESTAMP_LTZ 和 TIMESTAMP_NTZ。

主题

DATE

使用 DATE 数据类型存储没有时间戳的简单日历日期。

Name	存储	范围	解析
DATE	4 字节	4713 BC 到 294276 AD	1 天

TIMESTAMP_LTZ

使用 TIMESTAMP_LTZ 数据类型存储完整的时间戳值，包括日期、一天中的时间和本地时区。

TIMESTAMP 表示包含字段`year`、`month`、、`dayhourminutesecond`、和的值以及会话本地时区的值。该`timestamp`值表示绝对的时间点。

Spark 中的 TIMESTAMP 是用户指定的别名，与 TIMESTAMP_LTZ 和 TIMESTAMP_NTZ 变体之一相关联。您可以通过配置将默认时间戳类型设置为 TIMESTAMP_LTZ（默认值）或 TIMESTAMP_NTZ。`spark.sql.timestampType`

TIMESTAMP_NTZ

使用 TIMESTAMP_NTZ 数据类型存储完整的时间戳值，包括日期、一天中的时间，不包括本地时区。

TIMESTAMP 表示包含字段`year`、、`month`、`dayhourminute`、和`second`值的值。所有操作都是在不考虑任何时区的情况下执行的。

Spark 中的 TIMESTAMP 是用户指定的别名，与 TIMESTAMP_LTZ 和 TIMESTAMP_NTZ 变体之一相关联。您可以通过配置将默认时间戳类型设置为 TIMESTAMP_LTZ（默认值）或 TIMESTAMP_NTZ。`spark.sql.timestampType`

日期时间类型的示例

以下示例向您展示如何处理 AWS Clean Rooms支持的日期时间类型。

日期示例

以下示例插入具有不同格式的日期并显示输出。

```
select * from datetable order by 1;

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

如果您将时间戳值插入 DATE 列，时间部分会被忽略，只会加载日期。

时间示例

以下示例插入具有不同格式的 TIME 和 TIMETZ 值并显示输出。

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

日期、时间和时间戳文本

以下是使用 AWS Clean Rooms Spark SQL 支持的日期、时间和时间戳文字的规则。

日期

下表显示了输入日期，这些日期是您可以加载到 AWS Clean Rooms 表中的文字日期值的有效示例。默认 `MDY DateStyle` 模式被认为是有效的。此模式意味着在字符串中，月份值将位于日期值之前，如 `1999-01-08` 和 `01/02/00`。

> **Note**
>
> 当您将日期或时间戳文本加载到表中时，这些文本必须用引号括起来。

输入日期	完整日期
1999 年 1 月 8 日	1999 年 1 月 8 日
1999-01-08	1999 年 1 月 8 日
1/8/1999	1999 年 1 月 8 日
01/02/00	2000 年 1 月 2 日
2000-Jan-31	2000 年 1 月 31 日
Jan-31-2000	2000 年 1 月 31 日
31-Jan-2000	2000 年 1 月 31 日
20080215	2008 年 2 月 15 日
080215	2008 年 2 月 15 日
2008.366	2008 年 12 月 31 日（三位数的日期部分必须介于 001 和 366 之间）

Times

下表显示了输入时间，这些时间是您可以加载到 AWS Clean Rooms 表中的文字时间值的有效示例。

输入时间	描述（时间部分）
04:05:06.789	上午 4:05 过 6.789 秒
04:05:06	上午 4:05 过 6 秒
04:05	恰好上午 4:05
040506	上午 4:05 过 6 秒
04:05 AM	恰好上午 4:05；AM 为可选
04:05 PM	恰好下午 4:05；小时值必须小于 12

输入时间	描述（时间部分）
16:05	恰好下午 4:05

特殊日期时间值

下列显示可用作日期时间文本和日期函数参数的特殊值。它们需要单引号，并在查询处理期间转换为常规时间戳值。

特殊值	描述
now	计算结果为当前事务的开始时间并返回具有微秒精度的时间戳。
today	计算结果为相应的日期并返回时间部分为零的时间戳。
tomorrow	计算结果为相应的日期并返回时间部分为零的时间戳。
yesterday	计算结果为相应的日期并返回时间部分为零的时间戳。

以下示例说明了如何now使用 DATE_ADD 函数。today

```
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)
```

间隔文本

以下是使用 AWS Clean Rooms Spark SQL 支持的区间文字的规则。

使用间隔文本标识特定时间段（如 12 hours 或 6 weeks）。您可在涉及日期时间表达式的条件和计算中使用这些间隔文本。

> **Note**
>
> 不能对 AWS Clean Rooms 表中的列使用 INTERVAL 数据类型。

间隔用 INTERVAL 关键字与数量和支持的日期部分的组合表示；例如 INTERVAL '7 days' 或 INTERVAL '59 minutes'。您可以将许多数量和单位连接在一起以形成更精确的间隔；例如：INTERVAL '7 days, 3 hours, 59 minutes'。还支持每个单位的缩写和复数；例如：5 s、5 second 和 5 seconds 是等效的间隔。

如果您未指定日期部分，则间隔值表示秒。您可指定数量值作为小数（例如：0.5 days）。

示例

以下示例显示了具有不同间隔值的一系列计算。

以下示例向指定日期添加 1 秒。

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

以下示例向指定日期添加 1 分钟。

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

以下示例向指定日期添加 3 小时 35 分钟。

```
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)
```

以下示例向指定日期添加 52 周。

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

以下示例向指定日期添加 1 周、1 小时、1 分钟和 1 秒。

```
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)
```

以下示例向指定日期添加 12 小时（半天）。

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

以下示例将从 2023 年 3 月 31 日减去 4 个月，结果为 2022 年 11 月 30 日。计算时会考虑一个月中的天数。

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

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

间隔数据类型和文字

您可以使用间隔数据类型以 seconds、minutes、hours、days、months、和 years 等为单位存储时间段。间隔数据类型和文字可用于日期时间计算，例如在日期和时间戳中添加间隔、对间隔求和以及从日期或时间戳中减去间隔。间隔文字可用作表中间隔数据类型列的输入值。

间隔数据类型的语法

指定间隔数据类型以存储以年和月为单位的持续时间：

```
INTERVAL year_to_month_qualifier
```

指定间隔数据类型以存储以天、小时、分钟和秒为单位的持续时间：

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

间隔文字的语法

指定间隔文字以定义以年和月为单位的持续时间：

```
INTERVAL quoted-string year_to_month_qualifier
```

指定间隔文字以定义以天、小时、分钟和秒为单位的持续时间：

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

参数

引用字符串

指定正数值或负数值，以指定数量和日期时间单位作为输入字符串。如果带引号的字符串仅包含数字，则从 y ear_to_month_month_qualifier 或 day_to_secon d_ qualifier 中 AWS Clean Rooms 确定单位。例如，'23' MONTH 表示 1 year 11 months、'-2' DAY 表示 -2 days 0 hours 0 minutes 0.0 seconds，'1-2' MONTH 表示 1 year 2 months、'13 day 1 hour 1

minute 1.123 seconds' SECOND 表示 13 days 1 hour 1 minute 1.123 seconds。有关间隔输出格式的更多信息，请参阅间隔样式。

year_to_month_qualifier

指定间隔的范围。如果您使用限定符并创建时间单位小于限定符的时间间隔，则会 AWS Clean Rooms 截断并丢弃间隔中较小的部分。year_to_month_qualifier 的有效值为：

- YEAR
- MONTH
- YEAR TO MONTH

day_to_second_qualifier

指定间隔的范围。如果您使用限定符并创建时间单位小于限定符的时间间隔，则会 AWS Clean Rooms 截断并丢弃间隔中较小的部分。day_to_second_qualifier 的有效值为：

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

INTERVAL 文字的输出会被截断为指定的最小 INTERVAL 分量。例如，使用 MINUTE 限定符时，AWS Clean Rooms 丢弃小于 MINUTE 的时间单位。

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

结果值被截断为 '1 day 01:01:00'。

fractional_precision

可选参数，用于指定间隔中允许的小数位数。仅在间隔包含 SECOND 时，才应指定 fractional_precision 参数。例如，SECOND(3) 创建的间隔仅允许三个小数位，例如 1.234 秒。最大小数位数为六位。

会话配置 interval_forbid_composite_literals 确定在同时指定 YEAR TO MONTH 和 DAY TO SECOND 部分的间隔时是否返回错误。

间隔算术

您可以将间隔值与其他日期时间值一起使用来执行算术运算。下表介绍了可用的运算以及每种运算产生的数据类型。

> **Note**
>
> 能同时产生 date 和 timestamp 结果的运算是以等式中涉及的最小时间单位为基础的。例如，在将 interval 添加 date 时，如果是 YEAR TO MONTH 间隔，则结果为 date；如果是 DAY TO SECOND 间隔，则结果为时间戳。

第一操作数为 interval 的运算会对给定的第二操作数产生以下结果：

运算符	日期	Timestamp	Interval	数值
-	不适用	不适用	Interval	不适用
+	日期	日期/时间戳	Interval	不适用
*	不适用	不适用	不适用	Interval
/	不适用	不适用	不适用	Interval

第一操作数为 date 的运算会对给定的第二操作数产生以下结果：

运算符	日期	Timestamp	Interval	数值
-	数值	Interval	日期/时间戳	日期
+	不适用	不适用	不适用	不适用

第一操作数为 timestamp 的运算会对给定的第二操作数产生以下结果：

运算符	日期	Timestamp	Interval	数值
-	数值	Interval	Timestamp	Timestamp
+	不适用	不适用	不适用	不适用

间隔样式

- `postgres` – 遵循 PostgreSQL 风格。这是默认值。
- `postgres_verbose` – 遵循 PostgreSQL 的详细风格。
- `sql_standard` – 遵循 SQL 标准间隔文字风格。

以下命令将间隔风格设置为 `sql_standard`。

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

postgres 输出格式

以下是 `postgres` 间隔风格的输出格式。每个数值都可以是负数。

```
'<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 输出格式

postgres_verbose 语法与 postgres 类似，但是 postgres_verbose 输出还包含时间单位。

```
'[@] <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 输出格式

年至月间隔值的格式如下所示。在间隔之前指定负数表示间隔为负值，适用于整个间隔。

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

日至秒值间隔的格式如下所示。

```
'[-]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
```

间隔数据类型示例

以下示例演示如何将 INTERVAL 数据类型与表结合使用。

```
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
-----+-----
```

间隔文字示例

以下示例在间隔风格设置为 postgres 的情况下运行。

以下示例演示如何创建间隔为 1 年的 INTERVAL 文字。

```
select INTERVAL '1' YEAR

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

如果您指定的引用字符串超过限定词，则剩余的时间单位将从间隔处截断。在以下示例中，13 个月的间隔变为 1 年零 1 个月，但由于使用 YEAR 限定词，剩余的 1 个月被排除在外。

```
select INTERVAL '13 months' YEAR
```

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

如果您使用的限定词小于间隔字符串，则会包括排除的单位。

```
select INTERVAL '13 months' MONTH

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

在间隔中指定精度会将小数位数截断为指定的精度。

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

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

如果未指定精度，则 AWS Clean Rooms 使用最大精度 6。

```
select INTERVAL '1.23456789' SECOND

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

以下示例演示如何创建范围间隔。

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

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

限定词决定了您要指定的单位。例如，尽管以下示例使用与前一个示例相同的带引号的 '2:2' 字符串，AWS Clean Rooms 但由于限定符的原因，它使用了不同的时间单位。

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

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

还支持每个单位的缩写和复数。例如，5s、5 second、和 5 seconds 是等效间隔。支持的单位为年、月、小时、分钟和秒。

```
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
```

不带限定词语法的间隔文字示例

> **Note**
>
> 以下示例演示如何使用不带 YEAR TO MONTH 或 DAY TO SECOND 限定词的间隔文字。有关将推荐的间隔文字与限定词一起使用的信息，请参阅间隔数据类型和文字。

使用间隔文本标识特定时间段（如 12 hours 或 6 months）。您可在涉及日期时间表达式的条件和计算中使用这些间隔文本。

间隔文字用 INTERVAL 关键字与数量和支持的日期部分的组合来表示；例如 INTERVAL '7 days' 或 INTERVAL '59 minutes'。您可以将许多数量和单位连接在一起以形成更精确的间隔；例

如：INTERVAL '7 days, 3 hours, 59 minutes'。还支持每个单位的缩写和复数；例如：5 s、5 second 和 5 seconds 是等效的间隔。

如果您未指定日期部分，则间隔值表示秒。您可指定数量值作为小数（例如：0.5 days）。

以下示例显示了具有不同间隔值的一系列计算。

以下选项向指定日期添加 1 秒。

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

以下选项向指定日期添加 1 分钟。

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

以下选项向指定日期添加 3 小时 35 分钟。

```
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)
```

以下选项向指定日期添加 52 周。

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

```
(1 row)
```

以下选项向指定日期添加 1 周、1 小时、1 分钟和 1 秒：

```
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)
```

以下选项向指定日期添加 12 小时（半天）。

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

以下计算将从 2023 年 2 月 15 日减去 4 个月，结果为 2022 年 10 月 15 日。

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

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

以下计算将从 2023 年 3 月 31 日减去 4 个月，结果为 2022 年 11 月 30 日。计算时会考虑一个月中的天数。

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

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

布尔值类型

使用 BOOLEAN 数据类型在单字节列中存储 true 和 false 值。下表描述了布尔值的三种可能状态以及导致这些状态的文本值。不管输入字符串如何，Boolean 列将存储和输出“t”表示 true，“f”表示 false。

州	有效的文本值	存储
True	TRUE 't' 'true' 'y' 'yes' '1'	1 字节
False	FALSE 'f' 'false' 'n' 'no' '0'	1 字节
Unknown	NULL	1 字节

您可以使用 IS 比较将布尔值仅作为 WHERE 子句中的谓词进行检查。不能将 IS 比较与 SELECT 列表中的布尔值一起使用。

示例

您可使用 BOOLEAN 列将每个客户的“活跃/非活跃”状态存储在 CUSTOMER 表中。

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

在此示例中，以下查询从 USERS 表中选择喜欢运动而不喜欢电影院的用户：

```
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)
```

以下示例从 USERS 表中选择不清楚是否喜欢摇滚音乐的用户。

```
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)
```

以下示例返回错误，因为它在 SELECT 列表中使用了 IS 比较。

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

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

以下示例成功，因为它在 SELECT 列表中使用了等于比较 (=) 而不是 IS 比较。

```
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    |
```

布尔字面值

以下规则适用于处理 AWS Clean Rooms Spark SQL 支持的布尔文字。

使用布尔字面值来指定布尔值，例如TRUE或FALSE。

语法

```
TRUE | FALSE
```

示例

以下示例显示了指定值为的列TRUE。

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

二进制类型

使用 BINARY 数据类型存储和管理固定长度、未解释的二进制数据，为特定用例提供高效的存储和比较功能。

无论存储数据的实际长度如何，BINARY 数据类型都存储固定数量的字节。最大长度通常为 255 字节。

BINARY 用于存储未经解释的原始二进制数据，例如图像、文档或其他类型的文件。数据完全按照提供的方式存储，没有任何字符编码或解释。存储在 BINARY 列中的二进制数据是根据实际的二进制值进行比较和排序 byte-by-byte的，而不是根据任何字符编码或排序规则进行比较和排序。

以下示例查询显示了字符串的二进制表示形式"abc"。字符串中的每个字符都由其十六进制格式的 ASCII 码表示：“a” 为 0x61，“b” 为 0x62，“c” 为 0x63。组合后，这些十六进制值构成二进制表示形式。"616263"

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

嵌套类型

AWS Clean Rooms 支持涉及嵌套数据类型的查询，特别是 AWS Glue STRUCT、ARRAY 和 MAP 列类型。只有自定义分析规则支持嵌套数据类型。

值得注意的是，嵌套数据类型并不符合 SQL 数据库关系数据模型严格的表格结构。

嵌套数据类型包含引用数据中不同实体的标签。它们可以包含复杂的值，如数组、嵌套结构和其他与序列化格式（如 JSON）相关联的复杂结构。嵌套数据类型支持单个嵌套数据类型字段或对象最多 1 MB 的数据。

主题

- 数组类型
- 地图类型
- 结构类型
- 嵌套数据类型的示例

数组类型

使用 ARRAY 类型来表示包含类型为的元素序列的值`elementType`。

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

用于`containsNull`指示 ARRAY 类型中的元素是否可以有`null`值。

地图类型

使用 MAP 类型来表示包含一组键值对的值。

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

keyType: 密钥的数据类型

valueType: 值的数据类型

密钥不允许有null值。用于valueContainsNull指示 MAP 类型值的值是否可以有null值。

结构类型

使用 STRUCT 类型以一系列由 StructFields （字段）描述的结构来表示值。

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

StructField（名称，dataType，可为空）：表示 a 中的一个字段。 StructType

dataType: 数据类型 a 字段

name: 字段的名称

nullable用于指示这些字段的值是否可以有null值。

嵌套数据类型的示例

对于 struct<given:varchar, family:varchar> 类型，有两个属性名称：given 和 family，每个名称对应一个 varchar 值。

对于 array<varchar> 类型，数组指定为 varchar 的列表。

array<struct<shipdate:timestamp, price:double>> 类型是指具有 struct<shipdate:timestamp, price:double> 类型的元素列表。

map 数据类型的行为类似于 structs 的 array，其中数组中每个元素的属性名称用 key 表示并映射到 value。

Example

例如，map<varchar(20), varchar(20)> 类型被视为 array<struct<key:varchar(20), value:varchar(20)>>，其中 key 和 value 指的是底层数据中的映射的属性。

有关如何 AWS Clean Rooms 启用对数组和结构的导航的信息，请参见导航。

有关如何通过使用查询的 FROM 子句浏览数组来 AWS Clean Rooms 启用对数组的迭代的信息，请参见取消嵌套查询。

类型兼容性和转换

以下主题介绍类型转换规则和数据类型兼容性在 AWS Clean Rooms Spark SQL 中的工作原理。

主题

兼容性

在各种数据库操作期间，将会出现数据类型匹配以及文本值和常量与数据类型的匹配，包括以下情况：

- 表中的数据操控语言 (DML) 操作
- UNION、INTERSECT 和 EXCEPT 查询
- CASE 表达式
- 谓词（如 LIKE 和 IN）的计算
- 执行数据比较或提取的 SQL 函数的计算
- 数学运算符的比较

这些运算的结果取决于类型转换规则和数据类型兼容性。兼容性意味着并非总是需要 one-to-one匹配特定值和特定数据类型。由于一些数据类型是兼容的，因此可进行隐式转换或强制转换。有关更多信息，请参阅 隐式转换类型。如果数据类型不兼容，您有时可通过使用显式转换函数将值从一种数据类型转换为另一种数据类型。

一般兼容性和转换规则

请注意下列兼容性和转换规则：

- 一般来说，同属一种类型类别的数据类型（如不同的数字数据类型）是兼容的并且可隐式转换。

  例如，通过使用隐式转换，您可以将一个小数值插入整数列。小数进位为整数。或者，您可以从日期中提取一个数字值（如 2008）并将其插入到整数列中。

- 数字数据类型会强制执行尝试插入 out-of-range值时出现的溢出条件。例如，精度为 5 的小数值无法放入到精度定义为 4 的小数列中。整数或小数的整数部分永远不会被截断。不过，小数的小数部分可以酌情向上或向下舍入。但是，对于显示强制转换表中选定的值而得出的结果，不会进行四舍五入。
- 不同类型的字符字符串是兼容的。包含单字节数据的 VARCHAR 列字符串和 CHAR 列字符串是兼容且可隐式转换的。包含多字节数据的 VARCHAR 字符串是不可兼容的。此外，如果字符串是适当的文本值，则您可以将字符字符串转换为日期、时间、时间戳或数字值。将忽略任何前导空格或尾随空格。反过来，您也可以将日期、时间、时间戳或数字值转换为固定长度或可变长度的字符串。

  > **Note**
  >
  > 您要强制转换为数字类型的字符串必须包含数字的字符表示形式。例如，您可将字符串 `'1.0'` 或 `'5.9'` 强制转换为小数值，但无法将字符串 `'ABC'` 强制转换为任何数字类型。

- 如果将 DECIMAL 值与字符串进行比较，则会 AWS Clean Rooms 尝试将字符串转换为 DECIMAL 值。在将所有其他数值与字符串进行比较时，数值将转换为字符串。如果要强制进行相反的转换（例如，将字符串转换为正数，或者将 DECIMAL 值转换为字符串），请使用显式函数，例如 CAST 函数。
- 若要将 64 位 DECIMAL 或 NUMERIC 值转换为更高的精度，必须使用显式转换函数（如 CAST 或 CONVERT）。

隐式转换类型

有两种隐式转换类型：

- 赋值中的隐式转化，如 INSERT 或 UPDATE 命令中的设置值。
- 表达式中的隐式转化，例如在 WHERE 子句中执行比较

下表列出了在赋值或表达式中可隐式转换的数据类型。您还可使用显式转换函数执行这些转换。

源类型	目标类型
BIGINT	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)

源类型	目标类型
	双精度 (FLOAT8)
	INTEGER
	真实 (FLOAT4)
	SMALLINT 或简短
	VARCHAR
CHAR	VARCHAR
DATE	CHAR
	VARCHAR
	TIMESTAMP
	TIMESTAMPTZ
DECIMAL (NUMERIC)	大或长
	CHAR
	双精度 (FLOAT8)
	INTEGER (INT)
	真实 (FLOAT4)
	SMALLINT 或简短
	VARCHAR
双精度 (FLOAT8)	大或长
	CHAR
	DECIMAL (NUMERIC)

源类型	目标类型
	INTEGER (INT)
	真实 (FLOAT4)
	SMALLINT 或简短
	VARCHAR
INTEGER (INT)	大或长
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	双精度 (FLOAT8)
	真实 (FLOAT4)
	SMALLINT 或简短
	VARCHAR
真实 (FLOAT4)	大或长
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	SMALLINT 或简短
	VARCHAR
SMALLINT	大或长
	BOOLEAN

源类型	目标类型
	CHAR
	DECIMAL (NUMERIC)
	双精度 (FLOAT8)
	INTEGER (INT)
	真实 (FLOAT4)
	VARCHAR
TIME	VARCHAR
	TIMETZ

> **Note**
>
> 日期、时间、TIMESTAMP_LTZ、TIMESTAMP_NTZ 或字符串之间的隐式转换使用当前会话时区。
> 无法将 VARBYTE 数据类型隐式转换为任何其它数据类型。有关更多信息，请参阅 CAST 函数。

AWS Clean Rooms Spark SQL 命令

AWS Clean Rooms Spark SQL 中支持以下 SQL 命令：

主题

- 缓存表
- 提示
- SELECT

缓存表

CACHE TABLE 命令可以缓存现有表的数据或创建并缓存包含查询结果的新表。

> **Note**
>
> 缓存的数据将在整个查询中保留。

语法、参数和一些示例来自 Apache Spark SQL 参考。

语法

CACHE TABLE 命令支持三种语法模式：

使用 AS（不带括号）：根据查询结果创建并缓存新表。

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

带有 AS 和圆括号：函数与第一种语法类似，但使用圆括号对查询进行显式分组。

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

不使用 AS：缓存现有表，使用 SELECT 语句筛选要缓存的行。

```
CACHE TABLE cache_table_identifier query;
```

其中：

- 所有语句都应以分号 (;) 结尾
- query通常是一个 SELECT 语句
- 对于 AS，查询周围的括号是可选的
- AS 关键字是可选的

参数

缓存表标识符

缓存表的名称。可以包括可选的数据库名称限定符。

AS

根据查询结果创建和缓存新表时使用的关键字。

query

定义要缓存的数据的 SELECT 语句或其他查询。

示例

在以下示例中，缓存的表将在整个查询中保留。缓存后，引用的后续查询*cache_table_identifier*将从缓存的版本中读取，而不是重新计算或从中*sourceTable*读取。这可以提高经常访问的数据的查询性能。

根据查询结果创建并缓存经过筛选的表

第一个示例演示如何根据查询结果创建和缓存新表。此命令在SELECT语句周围使用不带括号的AS关键字。它会创建一个名为 “cache_table_identifier” 的新表，其中仅包含状态为 sourceTable “” 的 “” 中的行active'。它运行查询，将结果存储在新表中，并缓存新表的内容。原始 “sourceTable” 保持不变，后续查询必须引用 “cache_table_identifier” 才能使用缓存的数据。

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

使用带括号的 SELECT 语句缓存查询结果

第二个示例演示如何使用括号将查询结果缓存为具有指定名称 (cache_table_identifier) 的新表。SELECT此命令创建一个名为 “cache_table_identifier” 的新表，其中仅包含状态为 “sourceTable” 的 “” 中的行active'。它运行查询，将结果存储在新表中，并缓存新表的内容。原来的 “sourceTable” 保持不变。后续查询必须引用 cache_table_identifier “” 才能使用缓存的数据。

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

使用筛选条件缓存现有表

第三个示例演示如何使用不同的语法缓存现有表。此语法省略了 “AS” 关键字和圆括号，通常从名为 “cache_table_identifier” 的现有表中缓存指定行，而不是创建新表。该SELECT语句充当筛选器，用于确定要缓存哪些行。

> **Note**
>
> 此语法的确切行为因数据库系统而异。请务必验证您的特定 AWS 服务的语法是否正确。

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

提示

SQL 分析提示提供了用于指导查询执行策略的优化指令 AWS Clean Rooms，使您能够提高查询性能并降低计算成本。提示建议 Spark 分析引擎应如何生成其执行计划。

语法

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

提示使用注释式语法嵌入到 SQL 查询中，必须直接放在 SELECT 关键字之后。

支持的提示类型

AWS Clean Rooms 支持两类提示：联接提示和分区提示。

主题

- 加入提示
- 分区提示

加入提示

联接提示建议查询执行的联接策略。语法、参数和一些示例来自《Apache Spark SQL 参考》以获取更多信息

广播

建议 AWS Clean Rooms 使用广播加入。无论 autoBroadcastJoin阈值如何，带有提示的加入方都会被广播。如果联接的两边都有广播提示，则将广播大小较小的那个（基于统计数据）。

别名：广播加入、MAPJOIN

参数：表标识符（可选）

示例：

```
-- 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

建议 AWS Clean Rooms 使用随机排序合并联接。

别名：SHUFFLE_MERGE、MERGEJOIN

参数：表标识符（可选）

示例：

```
-- 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;
```

SHUFF_HASH

建议 AWS Clean Rooms 使用 shuffle 哈希联接。如果双方都有 shuffle 哈希提示，则查询优化器会选择较小的一方（基于统计数据）作为构建方。

参数：表标识符（可选）

示例：

```
-- Use shuffle hash join
SELECT /*+ SHUFFLE_HASH(students) */ *
FROM employees e JOIN students s ON e.id = s.id;
```

SHUFFLE_REPLICATE_NL

建议 AWS Clean Rooms 使用 shuffle-and-replicate嵌套循环联接。

参数：表标识符（可选）

示例：

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

Spark SQL 中的故障排除提示

下表显示了在 SparkSQL 中未应用提示的常见场景。有关更多信息，请参阅 the section called “注意事项和限制”。

使用场景	示例查询
未找到表格引用	`SELECT /*+ BROADCAST(fake_table) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
表未参与联接操作	`SELECT /*+ BROADCAST(s) */ * FROM students s WHERE s.age > 25;`
嵌套子查询中的表引用	`SELECT /*+ BROADCAST(s) */ * FROM employees e INNER JOIN (SELECT * FROM students s WHERE s.age > 20) sub ON e.eid = sub.sid;`

使用场景	示例查询
列名而不是表引用	`SELECT /*+ BROADCAST(e.eid) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
没有必填参数的提示	`SELECT /*+ BROADCAST */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
基表名而不是表别名	`SELECT /*+ BROADCAST(employees) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

分区提示

分区提示控制执行器节点之间的数据分发。当指定了多个分区提示时，会将多个节点插入到逻辑计划中，但最左边的提示由优化器选择。

COALESCE

将分区数减少到指定的分区数。

参数：数值（必填）-必须是介于 1 和 2147483647 之间的正整数

示例：

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

区分

使用指定的分区表达式将数据重新分区到指定数量的分区。使用循环分发。

参数：

- 数值（可选）-分区数；必须是介于 1 和 2147483647 之间的正整数

- 列标识符（可选）-要分区的列；这些列必须存在于输入架构中。
- 如果两者都指定，则必须先指定数值

示例：

```
-- 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;
```

按范围重新分区

在指定列上使用范围分区将数据重新分区到指定数量的分区。

参数：

- 数值（可选）-分区数；必须是介于 1 和 2147483647 之间的正整数
- 列标识符（可选）-要分区的列；这些列必须存在于输入架构中。
- 如果两者都指定，则必须先指定数值

示例：

```
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;
```

再平衡

重新平衡查询结果输出分区，使每个分区的大小合理（不要太小也不要太大）。这是尽力而为的操作：如果存在偏差， AWS Clean Rooms 将拆分偏斜的分区以使其不会太大。当您需要将查询结果写入表中以避免文件太小或太大时，此提示很有用。

参数：

- 数值（可选）-分区数；必须是介于 1 和 2147483647 之间的正整数
- 列标识符（可选）-列必须出现在 SELECT 输出列表中
- 如果两者都指定，则必须先指定数值

示例：

```
-- 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;
```

组合多个提示

你可以在单个查询中指定多个提示，方法是用逗号分隔它们：

```
-- 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;
```

注意事项和限制

- 提示是优化建议，而不是命令。查询优化器可能会根据资源限制或执行条件忽略提示。
- CreateAnalysisTemplate 和的提示都直接嵌入在 SQL 查询字符串中 StartProtectedQuery APIs。
- 提示必须直接放在 SELECT 关键字之后。
- 命名参数不支持带有提示，并且会引发异常。
- REPARTITION 和 REPARTITION_BY_RANGE 提示中的列名必须存在于输入架构中。
- REBALANCE 提示中的列名必须出现在 SELECT 输出列表中。
- 数字参数必须是介于 1 和 2147483647 之间的正整数。不支持像 1e1 这样的科学记法
- 差异隐私 SQL 查询中不支持提示。
- PySpark 作业中不支持 SQL 查询的提示。要为 PySpark 任务中的执行计划提供指令，请使用数据框 API。有关更多信息，请参阅 Apache Spark DataFrame API 文档。

SELECT

SELECT 命令返回表和用户定义的函数中的行。

AWS Clean Rooms Spark SQL 中支持以下 SELECT SQL 命令、子句和集合运算符：

主题

- SELECT list
- WITH 子句
- FROM 子句
- JOIN 子句

- WHERE 子句
- 价值条款
- GROUP BY 子句
- HAVING 子句
- 集合运算符
- ORDER BY 子句
- 子查询示例
- 关联的子查询

语法、参数和一些示例来自 Apache Spark SQL 参考。

SELECT list

SELECT list 指定希望查询返回的列、函数和表达式。列表表示查询的输出。

语法

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

参数

DISTINCT

一个选项，用于根据一个或多个列中的匹配值消除结果集中的重复行。

expression

由查询引用的表中存在的一个或多个列构成的表达式。表达式可包含 SQL 函数。例如：

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

AS column_alias

在最终结果集中使用的列的临时名称。AS 关键字是可选的。例如：

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

如果您没有为不是简单列名的表达式指定别名，则结果集将对该列应用默认名称。

> **Note**
>
> 在目标列表中定义别名后，它将立即被识别。不能在同一个目标列表中在其后定义的其他表达式中使用别名。

WITH 子句

WITH 子句是一个可选子句，该子句在查询中位于 SELECT 列表之前。WITH 子句定义一个或多个 common_table_expressions。每个通用表表达式 (CTE) 均定义一个临时表，它与视图定义类似。您可以在 FROM 子句中引用这些临时表。它们仅在它们所属的查询运行时使用。WITH 子句中的每个子 CTE 均指定一个表名、一个可选的列名称列表以及一个计算结果为表的查询表达式（SELECT 语句）。

WITH 子句子查询是定义可在单个查询的执行过程中使用的表的有效方式。在所有情况下，在 SELECT 语句的主体中使用子查询可获得相同的结果，不过 WITH 子句子查询可能在编写和阅读方面更加简单。如果可能，会将已引用多次的 WITH 子句子查询优化为常用子表达式；即，可以计算 WITH 子查询一次并重用其结果。（请注意，常用子表达式不只是限于 WITH 子句中定义的子表达式。）

语法

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

其中 common_table_expression 可以是非递归的。以下是非递归形式：

```
CTE_table_name AS ( query )
```

参数

common_table_expression

定义一个您可以在 FROM 子句 中引用并且仅在执行其所属的查询期间使用的临时表。

CTE_table_name

临时表的唯一名称，该临时表用于定义 WITH 子句子查询的结果。不能在单个 WITH 子句中使用重复名称。必须为每个子查询提供一个可在 FROM 子句中引用的表名。

query

任何 AWS Clean Rooms 支持的 SELECT 查询。请参阅SELECT。

使用说明

可在以下 SQL 语句中使用 WITH 子句：

- 选择、与、并集、全部合并、全部合并、相交、全部相交、全部除外或全部除外

如果包含 WITH 子句的查询的 FROM 子句未引用 WITH 子句所定义的任何表，则将忽略 WITH 子句，并且查询将正常执行。

WITH 子句子查询所定义的表只能在 WITH 子句开始的 SELECT 查询范围内引用。例如，可以在 SELECT 列表的子查询的 FROM 子句、WHERE 子句或 HAVING 子句中引用这样的表。不能在子查询中使用 WITH 子句，也不能在主查询或其他子查询的 FROM 子句中引用其表。此查询模式会为 WITH 子句表生成 `relation table_name doesn't exist` 形式的错误消息。

不能在 WITH 子句子查询中指定另一个 WITH 子句。

不能对 WITH 子句子查询定义的表进行前向引用。例如，以下查询返回一个错误，因为在表 W1 的定义中对表 W2 进行了前向引用：

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

示例

以下示例说明了包含 WITH 语句的查询的最简单示例。在名为 VENUECOPY 的 WITH 查询中，选择 VENUE 表中的所有行。主查询又选择 VENUECOPY 中的所有行。VENUECOPY 表仅在此查询的持续时间内存在。

```
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)
```

以下示例显示一个 WITH 子句，该子句生成两个分别名为 VENUE_SALES 和 TOP_VENUES 的表。第二个 WITH 查询表从第一个表中进行选择。而主查询块的 WHERE 子句又包含一个约束 TOP_VENUES 表的子查询。

```
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)
```

以下两个示例演示基于 WITH 子句子查询的表引用范围的规则。第一个查询运行，但第二个查询失败，并出现意料中的错误。在第一个查询中，主查询的 SELECT 列表中包含 WITH 子句子查询。SELECT 列表中的子查询的 FROM 子句中将引用 WITH 子句定义的表 (HOLIDAYS)：

```
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)
```

第二个查询失败，因为它尝试在主查询和 SELECT 列表子查询中引用 HOLIDAYS 表。主查询引用超出范围。

```
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 子句

查询中的 FROM 子句列出从中选择数据的表引用（表、视图和子查询）。如果列出多个表引用，则必须在 FROM 子句或 WHERE 子句中使用适当的语法来联接表。如果未指定联接条件，则系统将查询作为交叉联接（笛卡尔乘积）进行处理。

主题

- 语法
- 参数
- 使用说明

语法

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

其中，table_reference 是下列项之一：

```
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
```

参数

with_subquery_table_name

WITH 子句中的子查询定义的表。

table_name

表或视图的名称。

alias

表或视图的临时备用名称。必须为派生自子查询的表提供别名。在其他表引用中，别名是可选的。AS 关键字始终是可选的。表别名提供了用于标识查询的其他部分（例如 WHERE 子句）中的表的快捷方法。

例如：

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

如果定义了表别名，则必须使用该别名在查询中引用该表。

例如，如果查询是 SELECT "tbl"."col" FROM "tbl" AS "t"，则查询将失败，因为表名现在基本上已被覆盖。在这种情况下，有效的查询是 SELECT "t"."col" FROM "tbl" AS "t"。

column_alias

表或视图中的列的临时备用名称。

subquery

一个计算结果为表的查询表达式。表仅在查询的持续时间内存在，并且通常会向表提供一个名称或别名。但别名不是必需的。您也可以为派生自子查询的表定义列名称。如果您希望将子查询的结果联接到其他表并且希望在查询中的其他位置选择或约束这些列，则指定列的别名是非常重要的。

子查询可以包含 ORDER BY 子句，但在未指定 LIMIT 或 OFFSET 子句的情况下，该子句可能没有任何作用。

NATURAL

定义一个联接，该联接自动将两个表中同名列的所有配对用作联接列。不需要显式联接条件。例如，如果 CATEGORY 和 EVENT 表都具有名为 CATID 的列，则这两个表的自然联接为基于其 CATID 列的联接。

> **Note**
>
> 如果指定 NATURAL 联接，但表中没有要联接的同名列配对，则查询默认为交叉联接。

join_type

指定下列类型的联接之一：

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

- CROSS JOIN

交叉联接是未限定的联接；它们返回两个表的笛卡尔乘积。

内部联接和外部联接是限定的联接。它们的限定方式包括：隐式（在自然联接中）；在 FROM 语句中使用 ON 或 USING 语法；或者使用 WHERE 子句条件。

内部联接仅基于联接条件或联接列的列表返回匹配的行。外部联接返回与内部联接相同的所有行，还返回“左侧”表和/或“右侧”表中的非匹配行。左侧表是第一个列出的表，右侧表是第二个列出的表。非匹配行包含 NULL 值以填补输出列中的空白。

ON join_condition

联接规范的类型，其中将联接列声明为紧跟 ON 关键字的条件。例如：

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

USING (join_column [, ...])

联接规范的类型，其中用圆括号将列出的联接列括起来。如果指定多个联接列，则用逗号将它们分隔开。USING 关键字必须在列表之前。例如：

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

使用说明

联接列必须具有可比较的数据类型。

NATURAL 或 USING 联接仅将每对联接列中的一个联接列保留在中间结果集中。

使用 ON 语法的联接会将两个联接列都保留在其中间结果集中。

另请参阅 WITH 子句。

JOIN 子句

SQL JOIN 子句用于根据公共字段合并两个或多个表中的数据。根据指定的联接方法，结果可能会发生变化，也可能不发生变化。当在其他表中找不到匹配项时，左外部联接和右外部联接保留某个已联接表中的值。

JOIN 类型和连接条件的组合决定了哪些行包含在最终结果集中。然后，SELECT 和 WHERE 子句控制返回哪些列以及如何筛选行。了解不同的 JOIN 类型以及如何有效使用它们是 SQL 中的一项关键技能，因为它允许你以灵活而强大的方式合并来自多个表的数据。

语法

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

参数

选择第 1 列、第 2 列、...、第 N 列

要包含在结果集中的列。您可以从 JOIN 中涉及的两个表中的一个或两个表中选择列。

来自表 1

JOIN 操作中的第一个（左）表。

[JOIN | INNER JOIN | 左 [外] 连接 | 右 [外部] 联接 | 完整 [外部] 联接] table2：

要执行的 JOIN 类型。JOIN 或 INNER JOIN 仅返回两个表中值匹配的行。

LEFT [OUTER] JOIN 返回左表中的所有行，以及右表中的匹配行。

RIGHT [OUTER] JOIN 返回右表中的所有行，以及左表中的匹配行。

FULL [OUTER] JOIN 返回两个表中的所有行，无论是否存在匹配项。

CROSS JOIN 创建两个表中行的笛卡尔乘积。

ON table 1.column = table 2.

连接条件，它指定如何匹配两个表中的行。连接条件可以基于一列或多列。

WHERE 状况：

一个可选子句，可用于根据指定条件进一步筛选结果集。

示例

以下示例是使用 USING 子句在两个表之间进行的联接。在这种情况下，列 listid 和 eventid 用作联接列。结果限制为 5 行。

```
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
```

联接类型

INNER

这是默认的联接类型。返回两个表引用中具有匹配值的行。

INNER JOIN 是 SQL 中最常用的联接类型。这是一种基于公共列或一组列合并来自多个表的数据的强大方法。

语法：

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

以下查询将返回客户表和订单表之间存在匹配的 customer_id 值的所有行。结果集将包含客户编号、姓名、订单编号和订单日期列。

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

下面的查询是 LISTING 表和 SALES 表之间的内部联接（不带 JOIN 关键字），其中 LISTING 表中的 LISTID 介于 1 和 5 之间。此查询匹配 LISTING 表（左表）和 SALES 表（右表）中的 LISTID 列值。结果显示 LISTID 1、4 和 5 符合条件。

```
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
```

以下示例是与 ON 子句的内部联接。在这种情况下，不返回 NULL 行。

```
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
```

以下查询是 FROM 子句中的两个子查询的内部联接。此查询查找不同类别的活动（音乐会和演出）的已售门票数和未售门票数：这些 FROM 子句子查询是表 子查询；它们可返回多个列和行。

```
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
```

左 [外]

返回左表引用中的所有值和右表引用中的匹配值，如果没有匹配项，则附加 NULL。它也称为左外连接。

它返回左（第一个）表中的所有行，以及右表（第二个）中的匹配行。如果右表中没有匹配项，则结果集将包含右表中各列的 NULL 值。可以省略 OUTER 关键字，连接可以简单地写成 LEFT JOIN。与 LEFT OUTER JOIN 相反的是右外连接，它返回右表中的所有行和左表中的匹配行。

语法：

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

以下查询将返回客户表中的所有行，以及订单表中的匹配行。如果客户没有订单，结果集仍将包含该客户的信息，order_id 和 order_date 列的值为空。

```
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;
```

以下查询是一个左外部联接。当在其他表中找不到匹配项时，左外部联接和右外部联接保留某个已联接表中的值。左表和右表是语法中列出的第一个表和第二个表。NULL 值用于填补结果集中的“空白”。此查询匹配 LISTING 表（左表）和 SALES 表（右表）中的 LISTID 列值。结果显示， LISTIDs 2和3没有带来任何销售。

```
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
```

右 [外部]

返回右表引用中的所有值和左表引用中的匹配值，如果没有匹配项，则附加 NULL。它也称为右外连接。

它返回右（第二个）表中的所有行，以及左表（第一个）中的匹配行。如果左表中没有匹配项，则结果集将包含左表中各列的 NULL 值。可以省略 OUTER 关键字，连接可以简单地写成 RIGHT JOIN。与 RIGHT OUTER JOIN 相反的是 LEFT OUTER JOIN，它返回左表中的所有行和右表中的匹配行。

语法：

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

以下查询将返回客户表中的所有行，以及订单表中的匹配行。如果客户没有订单，结果集仍将包含该客户的信息，order_id 和 order_date 列的值为空。

```
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;
```

以下查询是一个右外部联接。此查询匹配 LISTING 表（左表）和 SALES 表（右表）中的 LISTID 列值。结果显示 LISTIDs 1、4 和 5 符合标准。

```
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
```

完整 [外部]

返回两个关系中的所有值，在不匹配的一侧附加 NULL 值。它也被称为完全外连接。

无论是否存在匹配项，它都会返回左表和右表中的所有行。如果没有匹配项，则结果集将包含表中没有匹配行的列的 NULL 值。可以省略 OUTER 关键字，连接可以简单地写成 FULL JOIN。FULL OUTER JOIN 不如左外联接或右外联接那么常用，但在某些情况下，即使没有匹配项，也需要查看两个表中的所有数据，它可能很有用。

语法：

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

以下查询将返回客户表和订单表中的所有行。如果客户没有订单，结果集仍将包含该客户的信息，order_id 和 order_date 列的值为空。如果订单没有关联客户，则结果集将包括该订单，customer_id 和名称列的值为空。

```
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;
```

以下查询是一个完全联接。当在其他表中找不到匹配项时，完全联接保留已联接表中的值。左表和右表是语法中列出的第一个表和第二个表。NULL 值用于填补结果集中的“空白”。此查询匹配 LISTING 表（左表）和 SALES 表（右表）中的 LISTID 列值。结果显示， LISTIDs 2和3没有带来任何销售。

```
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
```

以下查询是一个完全联接。此查询匹配 LISTING 表（左表）和 SALES 表（右表）中的 LISTID 列值。只有未产生任何销售额的行（LISTIDs 2 和 3）才会出现在结果中。

```
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
```

[左] 半

从表格引用的左侧返回与右侧匹配的值。它也被称为左半联接。

它只返回左表（第一个）中右表（第二个）中具有匹配行的行。它不返回右表中的任何列，只返回左表中的列。当您想在一个表中查找另一个表中具有匹配项的行，而无需返回第二个表中的任何数据时，LEFT SEMI JOIN 非常有用。与使用带有 IN 或 EXISTS 子句的子查询相比，LEFT SEMI JOIN 是一种更有效的替代方案。

语法：

```
SELECT column1, column2, ..., columnn
FROM table1
LEFT SEMI JOIN table2
```

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

对于订单表中至少有一个订单的客户，以下查询将仅返回客户表中的 customer_id 和姓名列。结果集将不包括订单表中的任何列。

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

CROSS JOIN

返回两个关系的笛卡尔乘积。这意味着结果集将包含两个表中所有可能的行组合，不应用任何条件或筛选器。

当您需要从两个表中生成所有可能的数据组合时，例如在创建显示客户和产品信息的所有可能组合的报告时，CROSS JOIN非常有用。CROSS JOIN 与其他联接类型（INNER JOIN、LEFT JOIN 等）不同，因为它在 ON 子句中没有连接条件。交叉联接不需要连接条件。

语法：

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

以下查询将返回一个结果集，其中包含客户和产品表中客户编号、客户名称、产品编号和产品名称的所有可能组合。如果客户表有 10 行，产品表有 20 行，则 CROSS JOIN 的结果集将包含 10 x 20 = 200 行。

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

以下查询是 LISTING 表和 SALES 表的交叉联接或笛卡尔联接，其中包含限制结果的谓词。此查询匹配 SALES 表和 LISTID 表中两个表中 LISTIDs 1、2、3、4 和 5 的 LISTID 列值。结果显示 20 个行符合条件。

```
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
```

反加入

返回左表引用中与右表引用不匹配的值。它也被称为左反连接。

当您想在一个表中查找另一个表中没有匹配项的行时，ANTI JOIN 是一个非常有用的操作。

语法：

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

以下查询将返回所有未下任何订单的客户。

```
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

指定两个关系中的行将隐式匹配所有名称相匹配的列。

它会自动匹配两个表之间具有相同名称和数据类型的列。它不需要您在 ON 子句中明确指定连接条件。它将两个表之间的所有匹配列合并到结果集中。

当您要联接的表中包含具有相同名称和数据类型的列时，NATURAL JOIN 是一种便捷的简写形式。但是，通常建议使用更明确的 INNER JOIN... ON 语法使连接条件更明确、更易于理解。

语法：

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

以下示例是两个表employees和departments与以下列之间的自然联接：

- employees表:employee_id,first_name,last_name, department_id
- departments表:department_id, department_name

以下查询将根据department_id列返回一个结果集，其中包括两个表之间所有匹配行的名字、姓氏和部门名称。

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

以下示例是两个表之间的自然联接。在这种情况下，列 listid、sellerid、eventid 和 dateid 在两个表中具有相同的名称和数据类型，因此用作联接列。结果限制为 5 行。

```
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 子句

WHERE 子句包含用于联接表或将谓词应用于表中的列的条件。可在 WHERE 子句或 FROM 子句中使用适当的语法对表进行内部联接。外部联接条件必须在 FROM 子句中指定。

语法

```
[ WHERE condition ]
```

condition

任何具有布尔型结果的搜索条件，例如，联接条件或表列上的谓词。以下示例是有效的联接条件：

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

以下示例是表中的列上的有效条件：

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

条件可以是简单条件或复杂条件；对于复杂条件，可以使用圆括号来分隔逻辑单元。在下面的示例中，用圆括号将联接条件括起来。

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

使用说明

您可在 WHERE 子句中使用别名来引用选择列表表达式。

不能限制 WHERE 子句中的聚合函数的结果；要实现此目的，请使用 HAVING 子句。

WHERE 子句中受限制的列必须派生自 FROM 子句中的表引用。

示例

以下查询使用不同的 WHERE 子句限制的组合，包括 SALES 表和 EVENT 表的联接条件、EVENTNAME 列上的谓词以及 STARTTIME 列上的两个谓词。

```
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)
```

价值条款

VALUES 子句用于直接在查询中提供一组行值，无需引用表。

VALUES 子句可用于以下场景：

- 您可以在 INSERT INTO 语句中使用 VALUES 子句来指定要插入到表中的新行的值。
- 您可以单独使用 VALUES 子句来创建临时结果集或内联表，而无需引用表。

- 您可以将 VALUES 子句与其他 SQL 子句（例如 WHERE、ORDER BY 或 LIMIT）结合使用，对结果集中的行进行筛选、排序或限制。

当您需要直接在 SQL 语句中插入、查询或操作一小组数据，而无需创建或引用永久表时，此子句特别有用。它允许您为每行定义列名和相应的值，从而可以灵活地创建临时结果集或即时插入数据，而无需管理单独的表的开销。

语法

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

参数

expression

一种表达式，它指定一个或多个值、运算符和生成值的 SQL 函数的组合。

表别名

一种别名，用于指定带有可选列名列表的临时名称。

示例

以下示例创建了一个内联表、包含两列的类似表的临时结果集，col1以及。col2结果集中的单行分别包含值"one"和1。查询SELECT * FROM部分仅检索此临时结果集中的所有列和行。列名（col1和col2）是由数据库系统自动生成的，因为 VALUES 子句没有明确指定列名。

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

如果要定义自定义列名，则可以通过在 VALUES 子句之后使用 AS 子句来实现，如下所示：

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

```
+------+----+
```

这将创建一个带有列名name和的临时结果集id，而不是默认的col1和col2。

GROUP BY 子句

GROUP BY 子句标识查询的分组列。必须在查询使用标准函数（例如，SUM、AVG 和 COUNT）计算聚合时声明分组列。如果 SELECT 表达式中存在聚合函数，则 SELECT 表达式中不在聚合函数中的任何列都必须位于 GROUP BY 子句中。

有关更多信息，请参阅 AWS Clean Rooms 火花 SQL 函数。

语法

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

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

参数

expr

列或表达式的列表必须匹配查询的选择列表中的非聚合表达式的列表。例如，考虑以下简单查询。

```
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)
```

在此查询中，选择列表包含两个聚合表达式。第一个聚合表达式使用 SUM 函数，第二个聚合表达式使用 COUNT 函数。必须将其余两个列（LISTID 和 EVENTID）声明为分组列。

GROUP BY 子句中的表达式也可以使用序号来引用选择列表。例如，上一个示例的缩略形式如下。

```
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

您可以使用聚合扩展 ROLLUP 在单个语句中执行多个 GROUP BY 操作。有关聚合扩展和相关函数的更多信息，请参阅聚合扩展。

聚合扩展

AWS Clean Rooms 支持聚合扩展，以便在单个语句中执行多个 GROUP BY 操作。

GROUPING SETS

在单个语句中计算一个或多个分组集。分组集是单个 GROUP BY 子句的集合，这是一组 0 列或更多列，您可以通过这些列对查询的结果集进行分组。GROUP BY GROUPING SETS 等效于对一个按不同列分组的结果集运行 UNION ALL 查询。例如，GROUP BY GROUPING SETS((a), (b)) 等效于 GROUP BY a UNION ALL GROUP BY b。

以下示例返回按产品类别和所售产品类型分组的订单表产品的成本。

```
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

假设在一个层次结构中，前面的列被视为后续列的父列。ROLLUP 按提供的列对数据进行分组，除了分组行之外，还返回额外的小计行，表示所有分组列级别的总计。例如，您可以使用 GROUP BY ROLLUP((a), (b)) 返回先按 a 分组的结果集，然后在假设 b 是 a 的一个子部分的情况下按 b 分组。ROLLUP 还会返回包含整个结果集而不包含分组列的行。

GROUP BY ROLLUP((a), (b)) 等效于 GROUP BY GROUPING SETS((a,b), (a), ())。

以下示例返回先按类别分组，然后按产品分组，且产品是类别细分项的订单表产品的成本。

```
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

按提供的列对数据进行分组，除了分组行之外，还返回额外的小计行，表示所有分组列级别的总计。CUBE 返回与 ROLLUP 相同的行，同时为 ROLLUP 未涵盖的每个分组列组合添加额外的小计

行。例如，您可以使用 GROUP BY CUBE ((a), (b)) 返回先按 a 分组的结果集，然后在假设 b 是 a 的一个子部分的情况下按 b 分组，再然后是单独按 b 分组的结果集。CUBE 还会返回包含整个结果集而不包含分组列的行。

GROUP BY CUBE((a), (b)) 等效于 GROUP BY GROUPING SETS((a, b), (a), (b), ())。

以下示例返回先按类别分组，然后按产品分组，且产品是类别细分项的订单表产品的成本。与前面的 ROLLUP 示例不同，该语句返回每个分组列组合的结果。

```
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 子句

HAVING 子句将条件应用于查询返回的中间分组结果集。

语法

```
[ HAVING condition ]
```

例如，您可以限制 SUM 函数的结果：

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

在应用所有 WHERE 子句条件并完成 GROUP BY 操作后，应用 HAVING 条件。

条件本身采用与任何 WHERE 子句条件相同的形式。

使用说明

- HAVING 子句条件中引用的任何列必须为分组列或引用了聚合函数结果的列。
- 在 HAVING 子句中，无法指定：
  - 引用选择列表项的序号。仅 GROUP BY 和 ORDER BY 子句接受序号。

示例

以下查询按名称计算所有活动的门票总销售额，然后消除总销售额小于 $800000 的活动。HAVING 条件应用于选择列表中聚合函数的结果：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)
```

以下查询计算类似的结果集。不过，在本示例中，HAVING 条件将应用于未在选择列表中指定的聚合：sum(qtysold)。将从最终结果中消除未售出 2000 张以上的门票的活动。

```
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)
```

集合运算符

集合运算符用于比较和合并两个独立查询表达式的结果。

AWS Clean Rooms Spark SQL 支持下表中列出的以下集合运算符。

设置运算符
INTERSECT
全部相交
EXCEPT
除了全部
联合
UNION ALL

例如，如果您希望知道网站的哪些用户既是买家又是卖家且其用户名存储在单独的列或表中，则可查找这两类用户的交集。如果您希望知道哪些网站用户是买家而不是卖家，则可使用 EXCEPT 运算符查找这两个用户列表的差集。如果您希望构建一个所有用户的列表（无论角色如何），则可使用 UNION 运算符。

> **Note**
>
> ORDER BY、LIMIT、SELECT TOP 和 OFFSET 子句不能用在 UNION、UNION ALL、INTERSECT 和 EXCEPT 集合运算符合并的查询表达式中。

主题

语法

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

参数

subquery1，subquery2

一种查询表达式，以其选择列表的形式对应于 UNION、UNION ALL、INTERSECT、INTERSECT ALL、EXCEPT 或 EXCEPT ALL 运算符之后的第二个查询表达式。这两个表达式必须包含数量相同并且数据类型兼容的输出列；否则，无法比较和合并两个结果集。集合运算不允许在不同类别的数据类型之间进行隐式转换。有关更多信息，请参阅 类型兼容性和转换。

您可以构建包含无限数量的查询表达式并任意组合使用 UNION、INTERSECT 和 EXCEPT 运算符来将这些表达式链接起来的查询。例如，假定表 T1、T2 和 T3 包含兼容的列集，则以下查询结构是有效的：

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

联合 [全部 | 不同]

从两个查询表达式返回行的集合运算，无论行派生自一个查询表达式还是两个查询表达式。

相交 [全部 | 不同]

返回派生自两个查询表达式的行的集合运算。将丢弃未同时由两个表达式返回的行。

除了 [全部 | 不同]

返回派生自两个查询表达式之一的行的集合运算。要符合结果的要求，行必须存在于第一个结果表而不存在于第二个结果表中。

EXCEPT ALL 不会从结果行中删除重复项。

MINUS 和 EXCEPT 完全同义。

集合运算符的计算顺序

UNION 和 EXCEPT 集合运算符是左关联的。如果未指定圆括号来影响优先顺序，则将以从左到右的顺序来计算这些集合运算符的组合。例如，在以下查询中，首先计算 T1 和 T2 的 UNION，然后对 UNION 结果执行 EXCEPT 操作：

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

在同一个查询中使用运算符组合时，INTERSECT 运算符优先于 UNION 和 EXCEPT 运算符。例如，以下查询将计算 T2 和 T3 的交集，然后计算得到的结果与 T1 的并集：

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

通过添加圆括号，可以强制实施不同的计算顺序。在以下示例中，将 T1 和 T2 的并集结果与 T3 执行交集运算，并且查询可能会生成不同的结果。

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

```
(select * from t3)
```

使用说明

- 集合运算查询结果中返回的列名是来自第一个查询表达式中的表的列名（或别名）。由于这些列名可能会造成误解（因为列中的值派生自位于集合运算符任一侧的表），您可能需要为结果集提供有意义的别名。
- 当集合运算符查询返回小数结果时，将提升对应的结果列以返回相同的精度和小数位数。例如，在以下查询中，T1.REVENUE 为 DECIMAL(10,2) 列而 T2.REVENUE 为 DECIMAL(8,4) 列，小数结果将提升为 DECIMAL(12,4)：

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

  小数位数为 4，因为这是两个列的最大小数位数。精度为 12，因为 T1.REVENUE 要求小数点左侧有 8 位数 (12 - 4 = 8)。此类提升可确保 UNION 两侧的所有值都适合结果。对于 64 位值，最大结果精度为 19，最大结果小数位数为 18。对于 128 位值，最大结果精度为 38，最大结果小数位数为 37。

  如果生成的数据类型超过 AWS Clean Rooms 精度和小数位数限制，则查询将返回错误。
- 对于集合运算，如果对于每个相应的列对，两个数据值相等 或都为 NULL，则两个行将被视为相同。例如，如果表 T1 和 T2 都包含一列和一行，并且两个表中的行都为 NULL，则对这两个表执行的 INTERSECT 运算将返回该行。

示例 UNION 查询

在以下 UNION 查询中，SALES 表中的行将与 LISTING 表中的行合并。从每个表中选择三个兼容的列；在这种情况下，对应的列具有相同的名称和数据类型。

```
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
```

以下示例说明如何将文本值添加到 UNION 查询的输出，以便您查看哪个查询表达式生成了结果集中的每一行。查询将第一个查询表达式中的行标识为“B”（针对买家），并将第二个查询表达式中的行标识为“S”（针对卖家）。

查询标识门票事务费用等于或大于 $10000 的买家和卖家。UNION 运算符的任一侧的两个查询表达式之间的唯一差异就是 SALES 表的联接列。

```
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
```

以下示例使用 UNION ALL 运算符，因为需要在结果中保留重复行（如果发现重复行）。对于特定系列的活动 IDs，该查询会为与每个事件关联的每笔销售返回 0 行或更多行，为该事件的每个列表返回 0 或 1 行。LISTING 和 EVENT 表中的每一行 IDs 都有唯一的事件，但是销售表 IDs 中相同的事件和列表组合可能会有多笔销售。

结果集中的第三个列标识行的来源。如果行来自 SALES 表，则在 SALESROW 列中将其标记为“Yes”。（SALESROW 是 SALES.LISTID 的别名。）如果行来自 LISTING 表，则在 SALESROW 列中将其标记为“No”。

在本示例中，结果集包含针对列表 500，活动 7787 的三个销售行。换而言之，将针对此列表和活动组合执行三个不同的事务。另外两个清单（501 和 502）没有产生任何销售额，因此查询为这些列表生成的唯一一行 IDs 来自清单表（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
```

如果运行不带 ALL 关键字的相同查询，则结果只保留其中一个销售交易。

```
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
```

示例 UNION ALL 查询

以下示例使用 UNION ALL 运算符，因为需要在结果中保留重复行（如果发现重复行）。对于特定系列的活动 IDs，该查询会为与每个事件关联的每笔销售返回 0 行或更多行，为该事件的每个列表返回 0 或 1 行。LISTING 和 EVENT 表中的每一行 IDs 都有唯一的事件，但是销售表 IDs 中相同的事件和列表组合可能会有多笔销售。

结果集中的第三个列标识行的来源。如果行来自 SALES 表，则在 SALESROW 列中将其标记为“Yes”。（SALESROW 是 SALES.LISTID 的别名。）如果行来自 LISTING 表，则在 SALESROW 列中将其标记为“No”。

在本示例中，结果集包含针对列表 500，活动 7787 的三个销售行。换而言之，将针对此列表和活动组合执行三个不同的事务。另外两个清单（501 和 502）没有产生任何销售额，因此查询为这些列表生成的唯一一行 IDs 来自清单表（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
```

如果运行不带 ALL 关键字的相同查询，则结果只保留其中一个销售交易。

```
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
```

示例 INTERSECT 查询

将以下示例与第一个 UNION 示例进行比较。这两个示例之间的唯一差异是所使用的集合运算符，但结果完全不同。仅其中一行相同：

```
235494 |    23875 |    8771
```

这是在包含 5 行的有限结果中，同时在两个表中找到的唯一行。

```
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
```

下面的查询查找 3 月份同时在纽约和洛杉矶举办的活动（已销售这些活动的门票）。这两个查询表达式之间的差异是 VENUECITY 列上的约束。

```
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
```

示例 EXCEPT 查询

数据库中的 CATEGORY 表包含以下 11 行：

```
 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)
```

假定 CATEGORY_STAGE 表（临时表）包含一个额外行：

```
 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)
```

返回两个表之间的差异。换而言之，返回 CATEGORY_STAGE 表中存在但 CATEGORY 表中不存在的行：

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

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

以下等效查询使用同义词 MINUS。

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

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

如果反转 SELECT 表达式的顺序，则查询不返回任何行。

ORDER BY 子句

ORDER BY 子句对查询的结果集进行排序。

> **Note**
>
> 最外面的 ORDER BY 表达式必须仅包含位于选择列表中的列。

主题

- 语法
- 参数

- 使用说明
- 使用 ORDER BY 的示例

语法

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

参数

expression

定义查询结果排序顺序的表达式。它由选择列表中的一个或多个列组成。根据二进制 UTF-8 排序方式返回结果。您也可以指定：

- 表示选择列表条目的位置（如果不存在选择列表，则为表中列的位置）的序号
- 定义选择列表条目的别名

当 ORDER BY 子句包含多个表达式时，将根据第一个表达式对结果集进行排序，然后将第二个表达式应用于具有第一个表达式中的匹配值的行，以此类推。

ASC | DESC

一个定义表达式的排序顺序的选项，如下所示：

- ASC：升序（例如，按数值的从低到高的顺序和字符串的从 A 到 Z 的顺序）。如果未指定选项，则默认情况下将按升序对数据进行排序。
- DESC：降序（按数值的从高到低的顺序和字符串的从 Z 到 A 的顺序）。

NULLS FIRST | NULLS LAST

一个选项，指定是应将 NULL 值排在最前（位于非 null 值之前）还是排在最后（位于非 null 值之后）。默认情况下，按 ASC 顺序最后对 NULL 值进行排序和排名，按 DESC 顺序首先对 NULL 值进行排序和排名。

LIMIT number | ALL

一个选项，用于控制查询返回的排序行的数目。LIMIT 数字必须为正整数；最大值为 2147483647。

LIMIT 0 不返回任何行。可以使用此语法进行测试：检查查询运行（不显示任何行）或返回表中列的列表。如果使用 LIMIT 0 返回列的列表，则 ORDER BY 子句是多余的。默认值为 LIMIT ALL。

OFFSET start

一个选项，指定在开始返回行之前跳过 start 前的行数。OFFSET 数字必须为正整数；最大值为 2147483647。在与 LIMIT 选项结合使用时，将先跳过 OFFSET 行，然后再开始计算返回的 LIMIT 行数。如果不使用 LIMIT 选项，则结果集中的行数会减少跳过的行数。仍必须扫描 OFFSET 子句跳过的行，因此使用较大的 OFFSET 值可能会非常低效。

使用说明

请注意，使用 ORDER BY 子句时预期会发生以下行为：

- NULL 值被视为“高于”所有其他值。对于默认的升序排序顺序，NULL 值将排在最后。要更改此行为，请使用 NULLS FIRST 选项。
- 当查询不包含 ORDER BY 子句时，系统将返回具有不可预测的行顺序的结果集。同一查询执行两次可能会返回具有不同顺序的结果集。
- 可在不使用 ORDER BY 子句的情况下使用 LIMIT 和 OFFSET 选项；不过，要返回一致的行集，请将这两个选项与 ORDER BY 子句结合使用。
- 在任何并行系统中 AWS Clean Rooms，例如，当 ORDER BY 不生成唯一排序时，行的顺序是不确定的。也就是说，如果 ORDER BY 表达式生成重复的值，则这些行的返回顺序可能因其他系统而异，也可能因运行一次而异。 AWS Clean Rooms
- AWS Clean Rooms 不支持 ORDER BY 子句中的字符串文字。

使用 ORDER BY 的示例

返回 CATEGORY 表中的所有 11 行，这些行按第二列 CATGROUP 进行排序。对于具有相同 CATGROUP 值的结果，按字符串长度对 CATDESC 列值进行排序。然后，按列 CATID 和 CATNAME 排序。

```
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)
```

返回 SALES 表中的选定列（按最高的 QTYSOLD 值排序）。将结果限制为前 10 行：

```
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
```

通过使用 LIMIT 0 语法返回列的列表，但不返回行：

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

子查询示例

以下示例说明子查询适合 SELECT 查询的不同方式。有关使用子查询的另一个示例，请参阅示例。

SELECT 列表子查询

以下示例在 SELECT 列表中包含一个子查询。此子查询是标量：它只返回一列和一个值，该值将在从外部查询返回的每个行的结果中重复。此查询将子查询计算出的 Q1SALES 值与外部查询定义的 2008 年其他两个季度（第 2 季度和第 3 季度）的销售值进行比较。

```
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 子句子查询

以下示例在 WHERE 子句中包含一个表子查询。此子查询生成多个行。在本示例中，行只包含一列，但表子查询可以包含多个列和行，就像任何其他表一样。

此查询查找门票销量排名前 10 位的卖家。前 10 位卖家的列表受子查询的限制，这将删除居住在设有售票点的城市的用户。可以使用不同的方式编写此查询；例如，可将子查询重新编写为主查询中的联接。

```
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 子句子查询

请参阅WITH 子句。

关联的子查询

以下示例将关联子查询 包含在 WHERE 子句中；此类型的子查询包含其列与由外部查询生成的列之间的一个或多个关联。在本示例中，关联为 `where s.listid=l.listid`。对于外部查询生成的每一行，将执行子查询以限定或取消限定行。

```
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)
```

不支持的关联子查询模式

查询计划程序使用名为“子查询去相关性”的查询重写方法来优化多个关联子查询模式以便在 MPP 环境中执行。有几种类型的关联子查询遵循 AWS Clean Rooms 无法取消关联且不支持的模式。包含以下关联引用的查询会返回错误：

- 跳过查询块的关联引用，也称为“跨级关联引用”。例如，在以下查询中，包含关联引用的块与跳过的块由 NOT EXISTS 谓词连接：

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

  在本示例中，跳过的块是针对 LISTING 表执行的子查询。关联引用将 EVENT 表和 SALES 表关联起来。

- 来自作为外部查询中 ON 子句的一部分的子查询的关联引用：

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

  ON 子句包含从子查询中的 SALES 到外部查询中的 EVENT 的关联引用。

- 对 AWS Clean Rooms 系统表的空敏感关联引用。例如：

```
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);
```

- 来自包含窗口函数的子查询内部的关联引用。

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

- GROUP BY 列中对关联查询结果的引用。例如：

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

- 来自带聚合函数和 GROUP BY 子句（通过 IN 谓词连接到外部查询）的关联引用。（此限制不适用于 MIN 和 MAX 聚合函数。）例如：

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

AWS Clean Rooms 火花 SQL 函数

AWS Clean Rooms Spark SQL 支持以下 SQL 函数：

主题

- 聚合函数
- 数组函数
- 条件表达式
- 构造函数
- 数据类型格式设置函数
- 日期和时间函数
- 加密和解密功能
- 哈希函数
- 超级日志函数
- JSON 函数
- 数学函数
- 标量函数
- 字符串函数
- 与隐私相关的功能
- 窗口函数

聚合函数

AWS Clean Rooms Spark SQL 中的聚合函数用于对一组行执行计算或操作并返回单个值。它们对于数据分析和汇总任务至关重要。

AWS Clean Rooms Spark SQL 支持以下聚合函数：

主题

ANY_VALUE 函数

ANY_VALUE 函数以非确定方式返回输入表达式值中的任何值。如果输入表达式未导致任何行被返回，则此函数可以返回 NULL。

语法

```
ANY_VALUE (expression[, isIgnoreNull] )
```

参数

expression

对其执行函数的目标列或表达式。表达式为以下数据类型之一：

isIgnoreNull

一个布尔值，用于确定函数是否应仅返回非空值。

返回值

返回与 expression 相同的数据类型。

使用说明

如果为列指定 ANY_VALUE 函数的语句也包含第二列引用，则第二列必须出现在 GROUP BY 子句中或包含在聚合函数中。

示例

以下示例返回 eventname 为 Eagles 的任何 dateid 的实例。

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

以下是结果。

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

以下示例返回 eventname 为 Eagles 或 Cold War Kids 的任何 dateid 的实例。

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

以下是结果。

```
dateid | eventname
```

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

近似计数_区分函数

AXPROXT COUNT_DISTINCT 提供了一种估计列或数据集中唯一值数量的有效方法。

语法

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

Arguments

expr

要估计其唯一值数量的表达式或列。

它可以是单列、复杂表达式或列组合。

RelativesD

一个可选参数，用于指定估计值所需的相对标准差。

它是一个介于 0 和 1 之间的值，表示估计值的最大可接受相对误差。RelativeSD 值越小，估计值越准确，但速度越慢。

如果未提供此参数，则使用默认值（通常在 0.05 或 5% 左右）。

返回值

返回 HyperLogLog ++ 的估计基数。relativeSD 定义允许的最大相对标准差。

示例

以下查询估计`col1`列中唯一值的数量，相对标准差为 1% (0.01)。

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

以下查询估计该`col1`列中有 3 个唯一值（值 1、2 和 3）。

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

近似百分位数函数

Approxt PERCENTILE 用于估计给定表达式或列的百分位数值，而不必对整个数据集进行排序。在需要快速了解大型数据集的分布或跟踪基于百分位数的指标的情况下，此函数非常有用，而无需执行精确的百分位数计算所产生的计算开销。但是，重要的是要了解速度和准确性之间的权衡，并根据用例的具体要求选择适当的容错能力。

语法

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

Arguments

expr

要估计其百分位数值的表达式或列。

它可以是单列、复杂表达式或列组合。

percentile

要估计的百分位数值，表示为 0 到 1 之间的值。

例如，0.5 将对应于第 50 个百分位数（中位数）。

准确性

一个可选参数，用于指定百分位估计值的所需精度。它是一个介于 0 和 1 之间的值，表示估计值的最大可接受相对误差。accuracy值越小，估计值越精确，但速度越慢。如果未提供此参数，则使用默认值（通常在 0.05 或 5% 左右）。

返回值

返回数字或 ANSI 间隔列 col 的近似百分位数，该列是有序 col 值中的最小值（从最小到最大排序），因此 col 值小于或等于该值的百分比不超过百分比。

百分比的值必须介于 0.0 和 1.0 之间。精度参数（默认值：10000）是一个正数字，它以牺牲内存为代价控制近似精度。

近似值的相对误差越1.0/accuracy高，精度值越高。

当百分比为数组时，百分比数组的每个值都必须介于 0.0 和 1.0 之间。在这种情况下，返回给定百分比数组下列 col 的近似百分位数组。

示例

以下查询估计response_time列的第 95 个百分位数，最大相对误差为 1% (0.01)。

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

以下查询估计了表中该col列的第 50、40 和第 10 个百分位数值。tab

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

以下查询估计 col 列中值的第 50 个百分位数（中位数）。

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

AVG 函数

AVG 函数返回输入表达式值的平均值（算术均值）。AVG 函数使用数值并忽略 NULL 值。

语法

```
AVG (column)
```

Arguments

column

对其执行函数的目标列。列为以下数据类型之一：

- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- DOUBLE
- FLOAT

数据类型

AVG 函数支持的参数类型有 SMALLINT、INTEGER、BIGINT、DECIMAL 和 DOUBLE。

AVG 函数支持的返回类型为：

- 适用于任何整数类型参数的 BIGINT
- 适用于浮点参数的 DOUBLE
- 返回与任何其他参数类型的表达式相同的数据类型

带有 DECIMAL 参数的 AVG 函数结果的默认精度为 38。结果的小数位数与参数的小数位数相同。例如，DEC(5,2) 列的 AVG 返回 DEC(38,2) 数据类型。

示例

从 SALES 表中查找每笔交易所售的平均产品数。

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

BOOL_AND 函数

BOOL_AND 函数在单个布尔/整数列或表达式上运行。此函数将类似的逻辑应用于 BIT_AND 和 BIT_OR 函数。对于此函数，返回类型为布尔值（`true` 或 `false`）。

如果集合中的所有值为 true，则 BOOL_AND 函数返回 `true` (t)。如果任何值为 false，该函数返回 `false` (f)。

语法

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

参数

expression

对其执行函数的目标列或表达式。此表达式必须具有 BOOLEAN 或整数数据类型。该函数的返回类型为 BOOLEAN。

DISTINCT | ALL

利用参数 DISTINCT，该函数可在计算结果之前消除指定表达式的所有重复值。利用参数 ALL，该函数可保留所有重复值。ALL 是默认值。

示例

您可以对布尔表达式或整数表达式使用布尔函数。

例如，以下查询从 TICKIT 数据库中的标准 USERS 表返回结果，该表包含多个布尔列。

BOOL_AND 函数对所有 5 个行返回 `false`。并非每个州的所有用户都喜欢运动。

```
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 函数

BOOL_OR 函数在单个布尔/整数列或表达式上运行。此函数将类似的逻辑应用于 BIT_AND 和 BIT_OR 函数。对于此函数，返回类型为布尔值（`true`、`false` 或 NULL）。

如果集合中的某个值为 `true`，则 BOOL_OR 函数返回 `true`（`t`）。如果集合中的某个值为 `false`，则该函数返回 `false`（`f`）。如果值未知，则可以返回 NULL。

语法

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

参数

expression

对其执行函数的目标列或表达式。此表达式必须具有 BOOLEAN 或整数数据类型。该函数的返回类型为 BOOLEAN。

DISTINCT | ALL

利用参数 DISTINCT，该函数可在计算结果之前消除指定表达式的所有重复值。利用参数 ALL，该函数可保留所有重复值。ALL 是默认值。

示例

您可以将布尔函数与布尔表达式或整数表达式结合使用。例如，以下查询从 TICKIT 数据库中的标准 USERS 表返回结果，该表包含多个布尔列。

BOOL_OR 函数对所有 5 个行返回 `true`。每个州中至少有一个用户喜欢运动。

```
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)
```

以下示例返回 NULL。

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

基数函数

基数函数返回数组或映射表达式 (e xp r) 的大小。

此函数对于查找数组的大小或长度很有用。

语法

```
cardinality(expr)
```

Arguments

expr

数组或映射表达式。

返回值

返回数组或地图的大小（整数）。

如果设置NULL为false或设置为，enabled则sizeOfNull该函数返回空输入true。

否则，该函数将返回-1空输入。使用默认设置时，该函数返回-1空输入。

示例

以下查询计算给定数组中的基数或元素数。数组 ('b', 'd', 'c', 'a') 有 4 个元素，因此此查询的输出将是4。

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

集合列表函数

COLLECT_LIST 函数收集并返回非唯一元素的列表。

当您想要将一组行中的多个值收集到单个数组或列表数据结构中时，这种类型的函数非常有用。

> **Note**
>
> 该函数是不确定的，因为收集结果的顺序取决于行的顺序，而在执行洗牌操作后，行顺序可能不确定。

语法

```
collect_list(expr)
```

Arguments

expr

任何类型的表达式。

返回值

返回参数类型的数组。数组中元素的顺序是不确定的。

不包括空值。

如果指定了 DISTINCT，则该函数仅收集唯一值，并且是collect_set 聚合函数的同义词。

示例

以下查询将 col 列中的所有值收集到一个列表中。该VALUES子句用于创建包含三行的内联表，其中每行都有单列 col，其值分别为 1、2 和 1。然后使用该collect_list() 函数将 col 列中的所有值聚合到一个数组中。此 SQL 语句的输出将是数组[1,2,1]，其中包含 col 列中的所有值，按它们在输入数据中出现的顺序排列。

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

COLLECT_SET

COLLECT_SET 函数收集并返回一组唯一的元素。

当您想要将一组行中的所有不同值收集到单个数据结构中而不包含任何重复值时，此函数非常有用。

> **Note**
>
> 该函数是不确定的，因为收集结果的顺序取决于行的顺序，而在执行洗牌操作后，行顺序可能不确定。

语法

```
collect_set(expr)
```

Arguments

expr

除 MAP 之外的任何类型的表达式。

返回值

返回参数类型的数组。数组中元素的顺序是不确定的。

不包括空值。

示例

以下查询将 col 列中的所有唯一值收集到一个集合中。该VALUES子句用于创建包含三行的内联表，其中每行都有单列 col，其值分别为 1、2 和 1。然后使用该collect_set()函数将 col 列中的所有唯一值聚合到一个集合中。此 SQL 语句的输出将是集合[1,2]，其中包含 col 列中的唯一值。重复值 1 在结果中仅包含一次。

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

COUNT 和 COUNT DISTINCT 函数

COUNT 函数对由表达式定义的行计数。COUNT DISTINCT 函数计算某个列或表达式中不同非 NULL 值的数量。它可在执行计数之前消除指定表达式中的所有重复值。

语法

```
COUNT (DISTINCT column)
```

Arguments

column

对其执行函数的目标列。

数据类型

COUNT 函数和 COUNT DISTINCT 函数支持所有参数数据类型。

COUNT DISTINCT 函数返回 BIGINT。

示例

对来自佛罗里达州的所有用户计数。

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

数一下EVENT桌子上所有独特的 IDs 场地。

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

COUNT 函数

COUNT 函数对由表达式定义的行计数。

COUNT 函数具有以下变体。

- COUNT (*) 对目标表中的所有行计数，无论它们是否包含 null 值。
- COUNT (expression) 计算某个特定列或表达式中带非 NULL 值的行的数量。
- COUNT (DISTINCT expression) 计算某个列或表达式中非重复的非 NULL 值的数量。

语法

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

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

参数

expression

对其执行函数的目标列或表达式。COUNT 函数支持所有参数数据类型。

DISTINCT | ALL

利用参数 DISTINCT，该函数可在执行计数之前消除指定表达式中的所有重复值。利用参数 ALL，该函数可保留表达式中的所有重复值以进行计数。ALL 是默认值。

返回类型

COUNT 函数返回 BIGINT。

示例

对来自佛罗里达州的所有用户计数：

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

count
```

```
-------
510
```

对 EVENT 表中的所有事件名称计数：

```
select count(eventname) from event;

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

对 EVENT 表中的所有事件名称计数：

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

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

统计活动表 IDs 中所有独特的场地：

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

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

计算每个卖家列出 4 张以上门票出售的批次的次数。按卖家 ID 对结果进行分组：

```
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 函数

MAX 函数返回一组行中的最大值。可以使用 DISTINCT 或 ALL，但不会影响结果。

语法

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

参数

expression

对其执行函数的目标列或表达式。表达式是任何数值数据类型。

DISTINCT | ALL

利用参数 DISTINCT，该函数可在计算最大值之前消除指定表达式中的所有重复值。利用参数 ALL，该函数可保留表达式中的所有重复值以计算最大值。ALL 是默认值。

数据类型

返回与 expression 相同的数据类型。

示例

从所有销售中查找支付的最高价格：

```
select max(pricepaid) from sales;

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

从所有销售中查找每张门票的已支付最高价格：

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

max_ticket_price
-----------------
2500.00000000
```

```
(1 row)
```

MEDIAN 函数

语法

```
MEDIAN ( median_expression )
```

参数

median_expression

对其执行函数的目标列或表达式。

MIN 函数

MIN 函数返回一组行中的最小值。可以使用 DISTINCT 或 ALL，但不会影响结果。

语法

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

参数

expression

对其执行函数的目标列或表达式。表达式是任何数值数据类型。

DISTINCT | ALL

利用参数 DISTINCT，该函数可在计算最小值之前消除指定表达式中的所有重复值。利用参数 ALL，该函数可保留表达式中的所有重复值以计算最小值。ALL 是默认值。

数据类型

返回与 expression 相同的数据类型。

示例

从所有销售中查找支付的最低价格：

```
select min(pricepaid) from sales;

min
-------
20.00
(1 row)
```

从所有销售中查找每张门票的已支付最低价格：

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

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

百分位数函数

PERCENTILE 函数用于计算精确的百分位数值，方法是首先对col列中的值进行排序，然后在指定值处找到值。percentage

当您需要计算精确的百分位数值并且您的用例可以接受计算成本时，PERCENTILE 函数非常有用。它提供的结果比 APPROX_PERCENTILE 函数更准确，但速度可能会更慢，特别是对于大型数据集。

相比之下，APPROX_PERCENTILE 函数是一种更有效的替代方案，它可以提供具有指定误差容限的百分位数值的估计值，因此更适合速度优先级高于绝对精度的场景。

语法

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

Arguments

col

要计算其百分位数值的表达式或列。

百分比

要计算的百分位数值，表示为 0 到 1 之间的值。

例如，0.5 将对应于第 50 个百分位数（中位数）。

频率

一个可选参数，用于指定col列中每个值的频率或权重。如果提供，则该函数将根据每个值的频率计算百分位数。

返回值

以给定百分比返回数字或 ANSI 间隔列 col 的精确百分位数值。

百分比的值必须介于 0.0 和 1.0 之间。

频率值应为正积分

示例

以下查询查找大于或等于col列中值的 30% 的值。由于值为 0 和 10，因此第 30 个百分位数为 3.0，因为该值大于或等于数据的 30%。

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

偏度函数

SKEWNESS 函数返回根据组的值计算出的偏度值。

偏度是一种统计衡量标准，用于描述数据集中的不对称性或缺乏对称性。它提供有关数据分布形状的信息。

此函数可用于理解数据集的统计属性并为进一步的分析或决策提供信息。

语法

```
skewness(expr)
```

Arguments

expr

计算结果为数值的表达式。

返回值

返回双精度。

如果指定了 DISTINCT，则该函数仅对一组唯一的 e xpr 值进行操作。

示例

以下查询计算列中值的偏度。col在此示例中，VALUES子句用于创建包含四行的内联表，其中每行都有一列col，其值分别为 -10、-20、100 和 1000。然后，该skewness()函数用于计算列中值的偏度。col结果为1.1135657469022011，表示数据偏度的程度和方向。偏度值为正表示数据向右倾斜，大部分值集中在分布的左侧。负偏度值表示数据向左倾斜，大部分值集中在分布的右侧。

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

以下查询计算 col 列中值的偏度。与前面的示例类似，该VALUES子句用于创建包含四行的内联表，其中每行都有一列col，其值分别为 -1000、-100、10 和 20。然后，该skewness()函数用于计算列中值的偏度。col结果为-1.1135657469022011，表示数据偏度的程度和方向。在这种情况下，负偏度值表示数据向左倾斜，大部分值集中在分布的右侧。

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

STDDEV_SAMP 和 STDDEV_POP 函数

STDDEV_SAMP 和 STDDEV_POP 函数返回一组数值（整数、小数或浮点）的样本标准差和总体标准差。STDDEV_SAMP 函数的结果等于同一组值的样本方差的平方根。

STDDEV_SAMP 和 STDDEV 是同一函数的同义词。

语法

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

表达式必须具有数值数据类型。无论表达式的数据类型如何，此函数的返回类型都是双精度数。

> **Note**
>
> 使用浮点算法计算标准偏差，其计算结果可能会稍微不准确。

使用说明

当计算包含一个值的表达式的样本标准差（STDDEV 或 STDDEV_SAMP）时，函数的结果为 NULL 而不是 0。

示例

以下查询返回 VENUE 表的 VENUESEATS 列中各值的平均数，后跟同一组值的样本标准差和总体标准差。VENUESEATS 是一个 INTEGER 列。结果的小数位数已减少至 2 位。

```
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)
```

以下查询返回 SALES 表中 COMMISSION 列的样本标准差。COMMISSION 是一个 DECIMAL 列。结果的小数位数已减少至 10 位。

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

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

以下查询将 COMMISSION 列的样本标准差转换为整数。

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

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

以下查询返回 COMMISSION 列的样本标准差和样本方差的平方根。这些计算的结果相同。

```
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 和 SUM DISTINCT 函数

SUM 函数返回输入列或表达式值的和。SUM 函数使用数值并忽略 NULL 值。

SUM DISTINCT 函数可在计算和之前消除指定表达式中的所有重复值。

语法

```
SUM (DISTINCT column )
```

Arguments

column

对其执行函数的目标列。该列是任何数值数据类型。

示例

从 SALES 表中查找所有已付佣金的和：

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

从 SALES 表中查找所有不同佣金的和：

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

VAR_SAMP 和 VAR_POP 函数

VAR_SAMP 和 VAR_POP 函数返回一组数值（整数、小数或浮点）的样本方差和总体方差。VAR_SAMP 函数的结果等于同一组值的样本标准差的平方。

VAR_SAMP 和 VARIANCE 是同一函数的同义词。

语法

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

表达式必须具有整数、小数或浮点数据类型。无论表达式的数据类型如何，此函数的返回类型都是双精度数。

> **Note**
>
> 这些函数的结果可能跨数据仓库集群而异，具体取决于每个案例中集群的配置。

使用说明

当计算包含一个值的表达式的样本方差（VARIANCE 或 VAR_SAMP）时，函数的结果为 NULL 而不是 0。

示例

以下查询返回 LISTING 表中 NUMTICKETS 列的已取整样本方差和总体方差。

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

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

以下查询运行相同的计算但将结果转换为小数值。

```
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)
```

数组函数

本节介绍 AWS Clean Rooms中支持的 SQL 数组函数。

主题

数组函数

使用给定元素创建一个数组。

语法

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

参数

expr1、expr2

除日期和时间类型之外的任何数据类型的表达式。参数不需要为相同的数据类型。

返回类型

数组函数返回一个包含表达式中元素的数组。

示例

以下示例显示了一个数值数组和一个不同数据类型的数组。

```
--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 函数

ARRAY_CONTAINS 函数可用于对数组数据结构执行基本的成员资格检查。当你需要检查数组中是否存在特定值时，ARRAY_CONTAINS 函数很有用。

语法

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

Arguments

array

要搜索的数组。

值

一种表达式，其类型与数组元素共享最不常见的类型。

返回类型

ARRAY_CONTAINS 函数返回一个布尔值。

如果值为空，则结果为空。

如果数组中的任何元素为 NULL，则如果值与任何其他元素都不匹配，则结果为 NULL。

示例

以下示例检查数组是否[1, 2, 3]包含该值4。由于数组[1, 2, 3] 不包含该值4，因此 array_contains 函数返回。false

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

以下示例检查数组是否[1, 2, 3]包含该值2。由于数组[1, 2, 3]确实包含该值2，因此 array_contains 函数返回。true

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

数组_DISTINCT 函数

ARRAY_DISTINCT 函数可用于从数组中删除重复的值。当您需要从数组中删除重复项并仅处理唯一元素时，ARRAY_DISTINCT 函数非常有用。在您想要在不受到重复值干扰的情况下对数据集执行操作或分析的情况下，这会很有用。

语法

```
array_distinct(array)
```

Arguments

array

一个数组表达式。

返回类型

ARRAY_DISTINCT 函数返回一个仅包含输入数组中唯一元素的数组。

示例

在此示例中，输入数组[1, 2, 3, null, 3]包含的重复值为3。该array_distinct函数删除此重复值，3并返回一个包含唯一元素的新数组:[1, 2, 3, null].

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

在此示例中，输入数组[1, 2, 2, 3, 3, 3]包含2和的重复值3。该array_distinct函数删除这些重复项，并返回一个包含唯一元素的新数组:[1, 2, 3].

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

数组_EXCEPT 函数

ARRAY_EXCEPT 函数将两个数组作为参数，并返回一个新数组，该数组仅包含第一个数组中存在的元素，而不包含第二个数组中存在的元素。

当您需要查找一个数组与另一个数组相比具有唯一性的元素时，ARRAY_EXCEPT 非常有用。在需要对数组执行类似集合的操作（例如找出两组数据之间的差异）的场景中，这可能很有用。

语法

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

Arguments

数组 1

具有可比元素的任何类型的数组。

数组 2

与 arr ay1 的元素共享最不常见类型的元素数组。

返回类型

ARRAY_EXCEPT 函数向 ar ray 1 返回一个类型匹配且没有重复项的数组。

示例

在此示例中，第一个数组[1, 2, 3]包含元素 1、2 和 3。第二个数组[2, 3, 4]包含元素 2、3 和 4。该array_except函数从第一个数组中删除元素 2 和 3，因为它们也存在于第二个数组中。生成的输出是数组[1]。

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

在此示例中，第一个数组[1, 2, 3]包含元素 1、2 和 3。第二个数组[1, 3, 5]包含元素 1、3 和 5。该array_except函数从第一个数组中删除元素 1 和 3，因为它们也存在于第二个数组中。生成的输出是数组[2]。

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

ARRAY_INTERSECT 函数

ARRAY_INTERSECT 函数将两个数组作为参数，并返回一个包含两个输入数组中存在的元素的新数组。当您需要查找两个数组之间的公共元素时，此函数很有用。在需要对数组执行类似集合的操作（例如查找两组数据之间的交集）的场景中，这可能很有用。

语法

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

Arguments

数组 1

具有可比元素的任何类型的数组。

数组 2

与 array1 的元素共享最不常见类型的元素数组。

返回类型

ARRAY_INTERSECT 函数向 array1 返回一个类型相匹配的数组，其中没有重复项，数组 1 和 array2 中都包含元素。

示例

在此示例中，第一个数组[1, 2, 3]包含元素 1、2 和 3。第二个数组[1, 3, 5]包含元素 1、3 和 5。ARRAY_INTERSECT 函数标识两个数组之间的公共元素，即 1 和 3。生成的输出数组为[1, 3]。

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

ARRAY_JOIN 函数

ARRAY_JOIN 函数有两个参数：第一个参数是要连接的输入数组。第二个参数是用于连接数组元素的分隔符字符串。当您需要将字符串数组（或任何其他数据类型）转换为单个串联字符串时，此函数很有用。这在您想要将值数组呈现为单个格式化字符串的场景中很有用，例如用于显示目的或用于进一步处理。

语法

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

Arguments

array

任何数组类型，但其元素都被解释为字符串。

分隔符

用于分隔连接的数组元素的 STRING。

NULL替换

用于在结果中表示空值的字符串。

返回类型

ARRAY_JOIN 函数返回一个字符串，其中数组的元素用分隔符分隔，并用空元素替换。`nullReplacement`如果省略，`nullReplacement`则`null`元素将被过滤掉。如果有任何参数NULL，则结果为NULL。

示例

在此示例中，ARRAY_JOIN 函数获取数组`['hello', 'world']`并使用分隔符`' '`（空格字符）连接元素。生成的输出是字符串`'hello world'`。

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

在此示例中，ARRAY_JOIN 函数获取数组['hello', null, 'world']并使用分隔符' '（空格字符）连接元素。该null值将替换为提供的替换字符串','（逗号）。生成的输出是字符串'hello , world'。

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

ARRAY_REMOVE 函数

ARRAY_REMOVE 函数有两个参数：第一个参数是将从中删除元素的输入数组。第二个参数是将从数组中删除的值。当你需要从数组中删除特定元素时，这个函数很有用。在需要对值数组执行数据清理或预处理的情况下，这可能很有用。

语法

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

Arguments

array

一个数组。

元素

一种与数组元素共享最不常见类型的表达式。

返回类型

ARRAY_REMOVE 函数返回与数组类型匹配的结果类型。如果要删除的元素是NULL，则结果为NULL。

示例

在此示例中，ARRAY_REMOVE 函数获取数组[1, 2, 3, null, 3]并删除所有出现的值 3。生成的输出是数组[1, 2, null]。

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

ARRAY_UNION 函数

ARRAY_UNION 函数将两个数组作为参数，并返回一个包含两个输入数组中唯一元素的新数组。当你需要合并两个数组并消除任何重复的元素时，这个函数很有用。在需要对数组执行类似集合的操作（例如在两组数据之间找到并集）的场景中，这可能会很有用。

语法

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

Arguments

数组 1

一个数组。

数组 2

与 array 1 相同类型的数组。

返回类型

ARRAY_UNION 函数返回一个与数组类型相同的数组。

示例

在此示例中，第一个数组[1, 2, 3]包含元素 1、2 和 3。第二个数组[1, 3, 5]包含元素 1、3 和 5。ARRAY_UNION 函数将两个数组中的唯一元素组合在一起，生成输出数组。[1, 2, 3, 5]T

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

“爆炸” 功能

EXPLODE 函数用于将包含数组或映射列的单行转换为多行，其中每行对应于数组或映射中的单个元素。

语法

```
explode(expr)
```

Arguments

expr

数组表达式或地图表达式。

返回类型

EXPLODE 函数返回一组行，其中每行代表输入数组或映射中的单个元素。

输出行的数据类型取决于输入数组或映射中元素的数据类型。

示例

以下示例采用单行数组 [10, 20] 并将其转换为两个单独的行，每行包含一个数组元素（10 和 20）。

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

在第一个示例中，输入数组直接作为参数传递给explode()。在此示例中，使用=>语法指定输入数组，其中明确提供了列名 (collection)。

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

这两种方法都是有效的，并且可以获得相同的结果，但是当你需要从更大的数据集中分解一列，而不仅仅是简单的数组文字时，第二种语法可能更有用。

Flatten 功能

FLATTEN 函数用于将嵌套数组结构 “扁平” 为单个平面数组。

语法

```
flatten(arrayOfArrays)
```

Arguments

arrayOfArrays

数组数组。

返回类型

FLATTEN 函数返回一个数组。

示例

在此示例中，输入是一个包含两个内部数组的嵌套数组，输出是一个包含内部数组所有元素的单个平面数组。FLATTEN 函数采用嵌套数组`[[1, 2], [3, 4]]`并将所有元素组合成一个数组`[1, 2, 3, 4]`。

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

条件表达式

在 SQL 中，条件表达式用于根据特定条件做出决策。它们允许您控制 SQL 语句的流并根据对一个或多个条件的评估返回不同的值或执行不同的操作。

AWS Clean Rooms 支持以下条件表达式：

主题

- CASE 条件表达式
- COALESCE 表达式
- 最大和最小表达式
- IF 表达式
- IS_NULL 表达式
- IS_NOT_NULL 表达式
- NVL 和 COALESCE 函数
- NVL2 函数
- NULLIF 函数

CASE 条件表达式

CASE 表达式是一种条件表达式，类似于其他语言中的语if/then/else句。CASE 用于指定存在多个条件时的结果。在 SQL 表达式有效的情况下使用 CASE，例如在 SELECT 命令中。

有两种类型的 CASE 表达式：简单和搜索。

- 在简单 CASE 表达式中，将一个表达式与一个值比较。在找到匹配项时，将应用 THEN 子句中的指定操作。如果未找到匹配项，则应用 ELSE 子句中的操作。
- 在搜索 CASE 表达式中，基于布尔表达式计算每个 CASE，而且 CASE 语句会返回第一个匹配的 CASE。如果在 WHEN 子句中未找到匹配，则返回 ELSE 子句中的操作。

语法

用于匹配条件的简单 CASE 语句：

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

用于计算每个条件的搜索 CASE 语句：

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

参数

expression

一个列名称或任何有效的表达式。

值

与该表达式比较的值，如数字常数或字符串。

result

计算表达式或布尔条件时返回的目标值或表达式。所有结果表达式的数据类型必须可转换为单一输出类型。

condition

计算结果为 true 或 false 的 Boolean 表达式。如果 condition 为 true，则 CASE 表达式的值是符合条件的结果，不处理 CASE 表达式的其余部分。如果 condition 为 false，则计算任何后续的

WHEN 子句。如果没有 WHEN 条件结果为 true，则 CASE 表达式的值是 ELSE 子句的结果。如果没有 ELSE 子句且没有条件为 true，则结果为 null。

示例

使用简单 CASE 表达式在针对 VENUE 表的查询中将 `New York City` 替换为 `Big Apple`。将所有其他城市名称替换为 `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
...
```

使用搜索 CASE 表达式来基于单个门票销售的 PRICEPAID 值分配组编号：

```
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 表达式

COALESCE 表达式返回列表中的第一个不为 null 的表达式的值。如果所有表达式为 null，则结果为 null。当找到非 null 值时，将不计算该列表中的剩余表达式。

如果您要在首选值缺失或为 null 时返回某些项的备份值，则此类表达式非常有用。例如，查询可能返回三个电话号码（手机、住宅或工作，按该顺序）之一，无论首先在表（非 null）中找到哪一个号码。

语法

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

示例

将 COALESCE 表达式应用于两列。

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

NVL 表达式的默认列名称为 COALESCE。以下查询将返回相同的结果。

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

最大和最小表达式

从包含任何数量的表达式的列表中返回最大值或最小值。

语法

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

参数

expression_list

表达式的逗号分隔的列表，如列名称。这些表达式都必须可转换为常见数据类型。忽略该列表中的 NULL 值。如果所有表达式的计算结果为 NULL，则结果为 NULL。

返回值

从所提供的表达式列表中返回最大值（对于 GREATEST）或最小值（对于 LEAST）。

示例

以下示例按字母顺序返回 `firstname` 或 `lastname` 的最高值。

```
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 表达式

IF 条件函数根据条件返回两个值中的一个。

此函数是 SQL 中常用的控制流语句，用于根据条件的评估做出决策并返回不同的值。这对于在查询中实现简单的 if-else 逻辑很有用。

语法

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

Arguments

expr1

被评估的条件或表达式。如果是`true`，则该函数将返回 expr 2 的值。如果 expr1 是*`false`*，则该函数将返回 expr3 的值。

expr2

如果 e xpr1 是，则计算并返回的表达式。`true`

expr3

如果 e xpr1 是，则计算并返回的表达式。`false`

返回值

如果`expr1`计算结果为`true`，则返回`expr2`；否则返回`expr3`。

示例

以下示例使用该`if()`函数根据条件返回两个值中的一个。正在评估的条件是`1 < 2`，也就是说`true`，因此返回第一个值`'a'`。

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

IS_NULL 表达式

IS_NULL条件表达式用于检查值是否为空。

此表达式是的同义词。IS NULL

语法

```
is_null(expr)
```

Arguments

expr

任何类型的表达式。

返回值

IS_NULL条件表达式返回布尔值。如果`expr1`为 NULL，则返回`true`，否则返回`false`。

示例

以下示例检查该值1是否为空，并返回布尔结果，true因为 1 是一个有效的非空值。

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

以下示例从squirrels表中选择id列，但仅选择年龄列所在的行null。

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

IS_NOT_NULL 表达式

IS_NOT_NULL条件表达式用于检查值是否不为空。

此表达式是的同义词。IS NOT NULL

语法

```
is_not_null(expr)
```

Arguments

expr

任何类型的表达式。

返回值

IS_NOT_NULL条件表达式返回布尔值。如果不expr1为 NULL，则返回true，否则返回false。

示例

以下示例检查该值1是否不为空，并返回布尔结果，true因为 1 是一个有效的非空值。

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

以下示例从squirrels表中选择id列，但仅选择年龄列不在的行null。

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

NVL 和 COALESCE 函数

返回表达式系列中不为 null 的第一个表达式的值。当找到非 null 值时，将不计算该列表中的剩余表达式。

NVL 与 COALESCE 相同。它们是同义词。本主题说明了其语法，并提供这两者的示例。

语法

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

用于 COALESCE 的语法是相同的：

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

如果所有表达式为 null，则结果为 null。

如果您要在主要值缺失或为 null 时返回次要值，则这些函数非常有用。例如，一个查询可能会返回前三个可用电话号码中的第一个：手机、家庭或工作号码。函数中表达式的顺序决定了计算结果的顺序。

参数

expression

一个要针对 null 状态进行计算的表达式，如列名称。

返回类型

AWS Clean Rooms 根据输入表达式确定返回值的数据类型。如果输入表达式的数据类型不是通用类型，则会返回错误。

示例

如果列表包含整数表达式，则该函数返回一个整数。

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

coalesce
--------------
```

```
12
```

此示例与前面的示例相同（不同之处在于它使用 NVL），返回相同的结果。

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

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

以下示例返回字符串类型。

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

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

以下示例会导致错误，因为表达式列表中的数据类型有变化。在这种情况下，列表中既有字符串类型，也有数字类型。

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

NVL2 函数

根据指定表达式的计算结果是 NULL 还是 NOT NULL 返回这两个值之一。

语法

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

参数

expression

一个要针对 null 状态进行计算的表达式，如列名称。

not_null_return_value

在 expression 的计算结果为 NOT NULL 时返回的值。not_null_return_value 值必须具有与 expression 相同的数据类型或可隐式转换为该数据类型。

null_return_value

在 expression 的计算结果为 NULL 时返回的值。null_return_value 值必须具有与 expression 相同的数据类型或可隐式转换为该数据类型。

返回类型

NVL2 返回类型按如下方式确定：

- 如果 not_null_return_value 或 null_return_value 为 null，则返回非 null 表达式的数据类型。

如果 not_null_return_value 和 null_return_value 都不为 null：

- 如果 not_null_return_value 和 null_return_value 具有相同的数据类型，则返回该数据类型。
- 如果 not_null_return_value 和 null_return_value 具有不同的数字数据类型，则返回最小的可兼容数字数据类型。
- 如果 not_null_return_value 和 null_return_value 具有不同的日期时间数据类型，则返回时间戳数据类型。
- 如果 not_null_return_value 和 null_return_value 具有不同的字符数据类型，则返回 not_null_return_value 的数据类型。
- 如果 not_null_return_value 和 null_return_value 具有混合的数字和非数字数据类型，则返回 not_null_return_value 的数据类型。

> **Important**
>
> 在最后两个示例中（其中返回 not_null_return_value 的数据类型），null_return_value 将隐式转换为该数据类型。如果数据类型不兼容，则该函数将失败。

使用说明

对于 NVL2，返回值将为 n ot_null_return_value 或 null_return_val ue 参数，以函数选择者为准，但数据类型为 not _null_return_valu e。

例如，假定 column1 为 NULL，则以下查询将返回相同的值。但是，DECODE 返回值数据类型将为 INTEGER， NVL2 返回值数据类型将为 VARCHAR。

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

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

示例

以下示例修改一些示例数据，然后计算两个字段以为用户提供相应的联系人信息：

```
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 函数

比较两个参数，并在两个参数相等时返回 null。如果它们不相等，则返回第一个参数。

语法

NULLIF 表达式比较两个参数并在两个参数相等时返回 null。如果它们不相等，则返回第一个参数。此表达式为 NVL 或 COALESCE 表达式的反向表达式。

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

参数

expression1，expression2

所比较的目标列或表达式。返回类型与第一个表达式的类型相同。

示例

在以下示例中，查询返回字符串 first，因为参数不相等。

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

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

在以下示例中，查询返回字符串 NULL，因为字符串文本参数相等。

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

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

在以下示例中，查询返回 1，因为整数参数不相等。

```
SELECT NULLIF(1, 2);

case
-------
1
```

在以下示例中，查询返回 NULL，因为整数参数相等。

```
SELECT NULLIF(1, 1);

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

在以下示例中，查询在 LISTID 和 SALESID 值匹配时返回 null：

```
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)
```

构造函数

SQL 构造函数是用于创建新数据结构（例如数组或地图）的函数。

它们接受一些输入值并返回一个新的数据结构对象。构造函数通常以它们创建的数据类型命名，例如 ARRAY 或 MAP。

构造函数不同于标量函数或聚合函数，后者对现有数据进行操作并返回单个值。构造函数用于创建新的数据结构，然后将其用于进一步的数据处理或分析。

AWS Clean Rooms 支持以下构造函数：

主题

- MAP 构造函数
- NAMED_STRUCT 构造函数
- STRUCT 构造函数

MAP 构造函数

MAP 构造函数使用给定的键/值对创建映射。

当你需要在 SQL 查询中以编程方式创建新的数据结构时，像 MAP 这样的构造函数非常有用。它们允许您构建复杂的数据结构，用于进一步的数据处理或分析。

语法

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

Arguments

key0

任何可比类型的表达式。所有 key0 必须共享最不常见的类型。

value0

任何类型的表达式。所有 ValuEn 都必须共享一种最不常见的类型。

返回值

MAP 函数返回一个 MAP，其中键为最不常见的 key0 类型，键入为最不常见的 valu e0 类型。

示例

以下示例创建了一个包含两个键值对的新地图：键与1.0值相关联。'2'密钥与3.0值相关联'4'。然后将生成的地图作为 SQL 语句的输出返回。

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

NAMED_STRUCT 构造函数

NAMED_STRUCT 构造函数使用给定的字段名和值创建一个结构。

当你需要在 SQL 查询中以编程方式创建新的数据结构时，像 NAMED_STRUCT 这样的构造函数非常有用。它们允许您构建复杂的数据结构，例如结构或记录，用于进一步的数据处理或分析。

语法

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

参数

名字1

字符串字面命名字段 1。

val1

任何类型的表达式，用于指定字段 1 的值。

返回值

NAMED_STRUCT 函数返回一个结构，其字段 1 与 val1 的类型相匹配。

示例

以下示例创建了一个包含三个命名字段的新结构："a"为该字段分配了值1。为该字段"b"分配了值。2.该字段"c"被分配了该值3。然后，生成的结构将作为 SQL 语句的输出返回。

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

STRUCT 构造函数

STRUCT 构造函数使用给定字段值创建一个结构。

当你需要在 SQL 查询中以编程方式创建新的数据结构时，像 STRUCT 这样的构造函数非常有用。它们允许您构建复杂的数据结构，例如结构或记录，用于进一步的数据处理或分析。

语法

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

参数

col1

一个列名称或任何有效的表达式。

返回值

STRUCT 函数返回一个结构，其中的字段 1 与 expr1 的类型相匹配。

如果参数被命名为引用，则使用这些名称来命名字段。否则，这些字段将命名为 ColN，其中 N 是该字段在结构中的位置。

示例

以下示例创建了一个包含三个字段的新结构：第一个字段的值为 1。第二个字段的值为 2。第三个字段的值为 3。默认情况下，生成的结构中的字段根据其在参数列表中的位置命名col1col2col3、和。然后，生成的结构将作为 SQL 语句的输出返回。

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

数据类型格式设置函数

使用数据类型格式设置函数，您可以将值从一种数据类型转换为另一种数据类型。对于这些函数，第一个参数始终是要进行格式设置的参数，第二个参数包含新格式的模板。

AWS Clean Rooms Spark SQL 支持多种数据类型格式化函数。

主题

- BASE64 函数
- CAST 函数
- DECODE 函数
- 编码功能
- HEX 函数
- STR_TO_MAP 函数
- TO_CHAR
- TO_DATE 函数
- TO_NUMBER
- UNBASE64 函数
- UNHEX 函数
- 日期时间格式字符串
- 数字格式字符串

BASE64 函数

该 BASE64 函数使用 MIME 的 Base64 传输编码将表达式转换为RFC2045 以 64 为基数的字符串。

语法

```
base64(expr)
```

参数

expr

二进制表达式或字符串，函数将其解释为二进制。

返回类型

STRING

示例

要将给定的字符串输入转换为其 Base64 编码的表示形式，请使用以下示例。结果是输入字符串 “Spark SQL” 的 Base64 编码表示形式，即 “u3bhcmsgu1fm”。

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

CAST 函数

CAST 函数将一种数据类型转换为另一种兼容的数据类型。例如，您可以将字符串转换为日期，或将数值类型转换为字符串。CAST 执行运行时转换，这意味着转换不会更改源表中值的数据类型。仅在查询上下文中对其进行更改。

某些数据类型需要使用 CAST 函数显式转换为其他数据类型。其他数据类型可以作为另一个命令的一部分进行隐式转换，而不必使用 CAST。请参阅类型兼容性和转换。

语法

使用以下两个等效的语法形式，将表达式从一种数据类型强制转换为另一种数据类型。

```
CAST ( expression AS type )
```

参数

expression

计算结果为一个或多个值的表达式，如列名称或文本。转换 null 值将返回 null。表达式不能包含空白或空字符串。

type

除二进制和二进制变化数据类型外，是支持的数据类型类型之一。

返回类型

CAST 返回 type 参数指定的数据类型。

> **Note**
>
> AWS Clean Rooms 如果您尝试执行有问题的转换（例如会丢失精度的十进制转换），则返回错误，如下所示：
>
> ```
> select 123.456::decimal(2,1);
> ```
>
> 或导致溢出的 INTEGER 转换：
>
> ```
> select 12345678::smallint;
> ```

示例

以下两个查询是等效的。它们都将小数值转换为整数：

```
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)
```

以下内容会产生类似的结果。它不需要示例数据即可运行：

```
select cast(162.00 as integer) as pricepaid;

pricepaid
-----------
162
(1 row)
```

在此示例中，时间戳列中的值将强制转换为日期，这会导致从每个结果中删除时间：

```
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)
```

如果您没有像前一个示例中所示的那样使用 CAST，则结果将包括时间：2008-02-18 02:36:48。

以下查询将可变字符数据强制转换为日期。它不需要示例数据即可运行。

```
select cast('2008-02-18 02:36:48' as date) as mysaletime;
```

```
mysaletime
--------------------
2008-02-18
(1 row)
```

在此示例中，日期列中的值将强制转换为时间戳：

```
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)
```

在与前面的示例类似的情况下，您可以使用 TO_CHAR 进一步控制输出格式。

在此示例中，整数将强制转换为字符串：

```
select cast(2008 as char(4));

bpchar
--------
2008
```

在此示例中，DECIMAL(6,3) 值将强制转换为 DECIMAL(4,1) 值：

```
select cast(109.652 as decimal(4,1));

numeric
---------
109.7
```

此示例显示了一个更复杂的表达式。SALES 表中的 PRICEPAID 列（DECIMAL(8,2) 列）将转换为 DECIMAL(38,2) 列，并且值将乘以 100000000000000000000：

```
select salesid, pricepaid::decimal(38,2)*100000000000000000000
as value from sales where salesid<10 order by salesid;


 salesid |           value
---------+----------------------------
       1 | 728000000000000000000000.00
       2 |  760000000000000000000000.00
       3 | 350000000000000000000000.00
       4 | 175000000000000000000000.00
       5 | 154000000000000000000000.00
       6 | 394000000000000000000000.00
       7 | 788000000000000000000000.00
       8 | 197000000000000000000000.00
       9 | 591000000000000000000000.00

(9 rows)
```

DECODE 函数

DECODE 函数与 ENCODE 函数相对应，后者用于使用特定的字符编码将字符串转换为二进制格式。DECODE 函数获取二进制数据，并使用指定的字符编码将其转换回可读的字符串格式。

当您需要处理存储在数据库中的二进制数据并需要将其以人类可读的格式呈现，或者需要在不同的字符编码之间转换数据时，此函数非常有用。

语法

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

Arguments

expr

用字符集编码的二进制表达式。

字符集

一个字符串表达式。

支持的字符集编码（不区分大小写）：'US-ASCII'、、'ISO-8859-1'、'UTF-8''UTF-16BE'、'UTF-16LE'和。'UTF-16'

返回类型

DECODE 函数返回一个字符串。

示例

以下示例有一个名为的表，messages其中有一个名为的列message_text，该列使用 UTF-8 字符编码以二进制格式存储消息数据。DECODE 函数将二进制数据转换回可读的字符串格式。此查询的输出是存储在消息表中的消息的可读文本，其中带有 ID123，使用'utf-8'编码从二进制格式转换为字符串。

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

编码功能

ENCODE 函数用于使用指定的字符编码将字符串转换为其二进制表示形式。

当您需要处理二进制数据或需要在不同的字符编码之间进行转换时，此函数非常有用。例如，在需要二进制存储的数据库中存储数据时，或者需要在使用不同字符编码的系统之间传输数据时，可以使用 ENCODE 函数。

语法

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

Arguments

str

要编码的字符串表达式。

字符集

指定编码的字符串表达式。

支持的字符集编码（不区分大小写）：'US-ASCII'、、'ISO-8859-1'、'UTF-8''UTF-16BE'、'UTF-16LE'和。'UTF-16'

返回类型

ENCODE 函数返回二进制。

示例

以下示例使用'utf-8'编码将字符串'abc'转换为其二进制表示形式，在本例中将返回原始字符串。这是因为'utf-8'编码是一种可变宽度的字符编码，可以用每个字符一个字节来表示整个 ASCII 字符集（包括字母'a''b'、和'c'）。因此，using 的'abc'二进制表示形式与原始字符串相同。'utf-8'

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

HEX 函数

HEX 函数将数值（整数或浮点数）转换为其相应的十六进制字符串表示形式。

十六进制是一种数字系统，它使用 16 个不同的符号（0-9 和 A-F）来表示数值。它通常用于计算机科学和编程中，以更紧凑和人类可读的格式表示二进制数据。

语法

```
hex(expr)
```

Arguments

expr

一个 BIGINT、BINARY 或 STRING 表达式。

返回类型

HEX 返回一个字符串。该函数返回参数的十六进制表示形式。

示例

以下示例将整数值 17 作为输入，并对其应用 HEX () 函数。输出为11，这是输入值的十六进制表示形式。17

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

以下示例将字符串'Spark_SQL'转换为其十六进制表示形式。输出为537061726B2053514C，这是输入字符串的十六进制表示形式。'Spark_SQL'

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

在此示例中，字符串 “spark_SQL” 的转换方式如下：

- 'S-> 53
- 'p'-> 70
- 'a'-> 61
- 'r'-> 72 '
- k'-> 6B
- '_'-> 20
- 'S-> 53
- 'Q'-> 51
- 'L'-> 4C

这些十六进制值的串联生成最终输出 “。537061726B2053514C"

STR_TO_MAP 函数

STR_TO_MAP 函数是一个转换函数。 string-to-map它将地图（或字典）的字符串表示形式转换为实际的地图数据结构。

当您需要在 SQL 中使用地图数据结构，但数据最初存储为字符串时，此函数非常有用。通过将字符串表示形式转换为实际地图，即可对地图数据执行操作和操作。

语法

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

Arguments

文本

表示地图的字符串表达式。

PairDelim

一个可选的 STRING 文字，用于指定如何分隔条目。默认为逗号 (',')。

keyValueDelim

一个可选的 STRING 文字，用于指定如何分隔每个键值对。它默认为冒号 (':')。

返回类型

STR_TO_MAP 函数返回键和值的字符串映射。pairDelim 和 pairDelim keyValueDelim都被视为正则表达式。

示例

以下示例采用输入字符串和两个分隔符参数，并将字符串表示形式转换为实际的地图数据结构。在此特定示例中，输入字符串'a:1,b:2,c:3'表示具有以下键值对的映射：'a'是键，'1'是值。 'b'是关键，'2'也是价值。 'c'是关键，'3'也是价值。分','隔符用于分隔键值对，分隔符用于':'分隔每对中的键和值。此查询的输出是:{"a":"1","b":"2","c":"3"}. 这是生成的地图数据结构，其中键是'a''b''c'、和，对应的值是'1''2'、和'3'。

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

以下示例演示 STR_TO_MAP 函数要求输入字符串采用特定格式，并正确分隔键值对。如果输入字符串与预期格式不匹配，则该函数仍会尝试创建地图，但结果值可能与预期不符。

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

TO_CHAR

TO_CHAR 将时间戳或数值表达式转换为字符串数据格式。

语法

```
TO_CHAR (timestamp_expression | numeric_expression , 'format')
```

参数

timestamp_expression

一个表达式，用于生成 TIMESTAMP 或 TIMESTAMPTZ 类型值或可隐式强制转换为时间戳的值。

numeric_expression

一个表达式，用于生成数字数据类型值或可隐式强制转换为数字类型的值。有关更多信息，请参阅数字类型。TO_CHAR 在数字串左侧插入空格。

> **Note**
>
> TO_CHAR 不支持 128 位 DECIMAL 值。

format

新值的格式。有关有效格式，请参阅日期时间格式字符串和数字格式字符串。

返回类型

VARCHAR

示例

以下示例将时间戳转换为一个具有日期和时间的值，格式为月份名称填充为九个字符、星期几的名称和当月的日期编号。

```
select to_char(timestamp '2009-12-31 23:15:59', 'MONTH-DY-DD-YYYY HH12:MIPM');
to_char
-------------------------
DECEMBER -THU-31-2009 11:15PM
```

以下示例将时间戳转换为具有这一年中日期编号的值。

```
select to_char(timestamp '2009-12-31 23:15:59', 'DDD');

to_char
-------------------------
365
```

以下示例将时间戳转换为这一周的 ISO 日期编号。

```
select to_char(timestamp '2022-05-16 23:15:59', 'ID');

to_char
-------------------------
1
```

以下示例从日期中提取月份名称。

```
select to_char(date '2009-12-31', 'MONTH');

to_char
------------------------
DECEMBER
```

以下示例将 EVENT 表中的每个 STARTTIME 值转换为由小时、分钟和秒组成的字符串。

```
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)
```

以下示例将整个时间戳值转换为不同的格式。

```
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)
```

以下示例将时间戳文本转换为字符串。

```
select to_char(timestamp '2009-12-31 23:15:59','HH24:MI:SS');
to_char
----------
23:15:59
(1 row)
```

以下示例将一个数字转换为末尾带负号的字符串。

```
select to_char(-125.8, '999D99S');
to_char
---------
125.80-
(1 row)
```

以下示例将一个数字转换为带货币符号的字符串。

```
select to_char(-125.88, '$S999D99');
to_char
---------
$-125.88
(1 row)
```

以下示例将一个数字转换为用尖括号将负数括起来的字符串。

```
select to_char(-125.88, '$999D99PR');
to_char
---------
$<125.88>
(1 row)
```

以下示例将一个数字转换为罗马数字字符串。

```
select to_char(125, 'RN');
to_char
---------
CXXV
(1 row)
```

以下示例显示一周中的某天。

```
SELECT to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS');
```

```
               to_char
-----------------------
Wednesday, 31 09:34:26
```

以下示例显示数字的序数后缀。

```
SELECT to_char(482, '999th');
               to_char
-----------------------
 482nd
```

以下示例将销售表中支付的价格减去佣金。差随后将向上舍入并转换为罗马数字，如 to_char 列中所示：

```
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)
```

以下示例向 to_char 列中显示的差值添加货币符号：

```
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)
```

以下示例列出了完成每次销售的世纪。

```
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)
```

以下示例将 EVENT 表中的每个 STARTTIME 值转换为由小时、分钟、秒和时区组成的字符串：

```
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)
```

以下示例显示了秒、毫秒和微秒的格式设置。

```
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 函数

TO_DATE 会将以字符串形式表示的日期转换为 DATE 数据类型。

语法

```
TO_DATE (date_str)
```

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

参数

date_str

可以转换为日期字符串的日期字符串或数据类型。

format

与 Spark 的日期时间模式相匹配的字符串文字。有关有效的日期时间模式，请参阅用于格式化和解析的日期时间模式。

返回类型

TO_DATE 将根据 format 值返回 DATE。

如果转换为格式 失败，则返回错误。

示例

以下 SQL 语句将日期 02 Oct 2001 转换为日期数据类型。

```
select to_date('02 Oct 2001', 'dd MMM yyyy');

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

以下 SQL 语句将字符串 20010631 转换为日期。

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

以下 SQL 语句将字符串 20010631 转换为日期：

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

结果为空值，因为六月份只有 30 天。

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

TO_NUMBER

TO_NUMBER 将字符串转换为数字（小数）值。

语法

```
to_number(string, format)
```

参数

string

要转换的字符串。格式必须是文本值。

format

第二个参数是指示应如何分析字符串以创建数字值的格式字符串。例如，格式 '99D999' 指定要转换的字符串包含五位数，其中小数点在第三位。例如，to_number('12.345','99D999') 将 12.345 作为数字值返回。有关有效格式的列表，请参阅数字格式字符串。

返回类型

TO_NUMBER 返回 DECIMAL 数。

如果转换为格式 失败，则返回错误。

示例

以下示例将字符串 12,454.8- 转换为数字：

```
select to_number('12,454.8-', '99G999D9S');

to_number
-----------
-12454.8
```

以下示例将字符串 $ 12,454.88 转换为数字：

```
select to_number('$ 12,454.88', 'L 99G999D99');

to_number
-----------
12454.88
```

以下示例将字符串 $ 2,012,454.88 转换为数字：

```
select to_number('$ 2,012,454.88', 'L 9,999,999.99');

to_number
-----------
2012454.88
```

UNBASE64 函数

该 UNBASE64 函数将参数从以 64 为基数的字符串转换为二进制。

Base64 编码通常用于以文本格式表示二进制数据（例如图像、文件或加密信息），这种格式可以安全地通过各种通信渠道（例如电子邮件、URL 参数或数据库存储）传输。

该 UNBASE64 函数允许您逆转此过程并恢复原始的二进制数据。在需要处理以 Base64 格式编码的数据的情况下，例如与外部系统集成或使用 Base64 作为数据传输机制时 APIs ，此类功能非常有用。

语法

```
unbase64(expr)
```

Arguments

expr

base64 格式的字符串表达式。

返回类型

BINARY

示例

在以下示例中，将 Base64 编码的字符串'U3BhcmsgU1FM'转换回原始字符串。'Spark SQL'

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

UNHEX 函数

UNHEX 函数将十六进制字符串转换回其原始字符串表示形式。

在需要处理以十六进制格式存储或传输的数据，并且需要恢复原始字符串表示形式以便进一步处理或显示的情况下，此函数非常有用。

UNHEX 函数是十六进制函数的对应函数。

语法

```
unhex(expr)
```

Arguments

expr

由十六进制字符组成的字符串表达式。

返回类型

UNHEX 返回一个二进制。

如果 expr 的长度为奇数，则丢弃第一个字符并用空字节填充结果。如果 e xpr 包含非十六进制字符，则结果为 NULL。

示例

以下示例通过同时使用 UNHEX () 和 DECODE () 函数，将十六进制字符串转换回其原始字符串表示形式。查询的第一部分使用 UNHEX () 函数将十六进制字符串 '537061726B2053514C' 转换为其二进制表示形式。查询的第二部分使用 DECODE () 函数使用 'UTF-8' 字符编码将从 UNHEX () 函数获得的二进制数据转换回字符串。查询的输出是原始字符串 'spark_SQL'，它被转换为十六进制，然后又转换为字符串。

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

日期时间格式字符串

您可以在以下常见场景中使用日期时间模式：

- 使用 CSV 和 JSON 数据源来解析和格式化日期时间内容时
- 使用诸如以下的函数在字符串类型与日期或时间戳类型之间进行转换时：
  - unix_timestamp
  - date_format
  - to_unix_timestamp
  - from_unixtime
  - to_date
  - to_timestamp
  - from_utc_timestamp
  - to_utc_timestamp

使用下表中的模式字母进行日期和时间戳的解析和格式化。

日期部分或时间部分	意义	示例
a	当天的上午或下午，显示为上午至下午	PM
D	一年中的某一天，以 3 位数字表示	189
d	月份中的某一天，以 2 位数的数字表示	28
E	一周中的某一天，以文字形式呈现	星期二 星期二
F	该月中一周中的某一天对齐，以 1 位数的数字表示	3
G	时代指示器，以文本形式呈现	AD Anno Domini
h	上午或下午的时钟时间，以 2 位数字表示	12
H	一天中的某一小时，以 0 到 23 之间的 2 位数字表示	0
k	一天中的时钟，以 1 到 24 之间的 2 位数字表示	1
K	上午或下午的时间，以 0 到 11 之间的 2 位数字表示	0
m	时分，以 2 位数的数字表示	30
M/L	一年中的月份，以月份表示	7 07

日期部分或时间部分	意义	示例
		七月 七月
O	与 UTC 的本地化区域偏移量	GMT+8 GMT+ 8:00 世界标准时间-08:00
Q/q	一年中的季度，以数字（1 到 4）或文本形式显示	3 03 Q3 第三季度
s	分钟秒数，以 2 位数的数字表示	55
S	一秒钟的分数，以分数形式呈现	978
V	时区标识符，以区域 ID 的形式显示	美洲/洛杉矶 Z 08:30
x	与 UTC 的区域偏移量（Offset-X）	+0000 -08 -0830 -08:30 -083015 -08:30:15

日期部分或时间部分	意义	示例
X	与 UTC 的区域偏移量；其中 Z 表示零	Z -08 -0830 -08:30 -083015 -08:30:15
y	年份，以年份表示	2020 20
z	时区名称，以文本形式显示	太平洋标准时间 PST
Z	与 UTC 的区域偏移量（偏移量-Z）	+0000 -0800 -08:00
'	转义为文本，以分隔符的形式呈现	不适用
''	单引号，以字面形式呈现	'
[	可选部分开始	不适用
]	可选章节结尾	不适用

模式字母的数量决定了格式类型：

文本格式

- 缩写形式使用 1-3 个字母（例如，“星期一” 表示星期一）

- 完整表格应恰好使用 4 个字母（例如，“星期一”）
- 不要使用 5 个或更多字母-这会导致错误

数字格式 (n)

- 值 n 表示允许的最大字母数
- 对于单字母图案：
  - 输出使用不带填充的最小位数
- 对于多个字母图案：
  - 输出用零填充以匹配字母计数的宽度
- 解析时，输入必须包含确切的位数

数字/文本格式

- 对于 3 个或更多字母，请遵循文本格式规则
- 对于较少的字母，请遵循数字格式规则

分数格式

- 使用 1-9 个 'S' 字符（例如 SSSSSS）
- 用于解析：
  - 接受 1 和 S 字符数之间的分数
- 要进行格式化：
  - 用零填充以匹配 S 字符的数量
- 支持高达 6 位数字，以实现微秒精度
- 可以解析纳秒但会截断多余的数字

年份格式

- 字母数设置填充的最小字段宽度
- 对于两个字母：
  - 打印最后两位数字
  - 解析 2000-2099 年之间的年份

- 对于少于四个字母（两个除外）：
  - 仅显示负年份的符号
- 不要使用 7 个或更多字母-这会导致错误

月份格式

- 使用 “M” 表示标准表单，使用 “L” 表示独立表单
- 单曲 “M” 或 “L”：
  - 显示不带填充的月份数字 1-12
- 'MM'或'LL'：
  - 显示带有填充的月份数字 01-12
- 'MM'：
  - 以标准格式显示缩写的月份名称
  - 必须是完整日期模式的一部分
- '哈哈'：
  - 以独立形式显示缩写的月份名称
  - 用于仅限月份的格式
- 'MMM'：
  - 以标准格式显示完整的月份名称
  - 用于日期和时间戳
- '哈哈'：
  - 以独立形式显示完整的月份名称
  - 用于仅限月份的格式

时区格式

- 上午至下午：仅使用 1 个字母
- 区域 ID (V)：仅使用 2 个字母
- 区域名称 (z):
  - 1-3 个字母：显示简称
  - 4 个字母：显示全名

- 不要使用 5 个或更多字母

偏移格式

- X 和 x：
  - 1 个字母：显示小时 (+01) 或小时分钟 (+0130)
  - 2 个字母：显示不带冒号的小时分钟 (+0130)
  - 3 个字母：显示带冒号的小时分钟 (+ 01:30)
  - 4 个字母：显示时 hour-minute-second不带冒号 (+013015)
  - 5 个字母： hour-minute-second用冒号显示 (+ 01:30:15)
  - X 使用 'Z' 表示零偏移
  - x 使用 '+00'、'+0000' 或 '+ 00:00 '作为零偏移量
- O:
  - 1 个字母：显示简写形式 (GMT+8)
  - 4 个字母：显示完整表格 (GMT+ 08:00)
- Z:
  - 1-3 个字母：显示不带冒号的小时分钟 (+0130)
  - 4 个字母：显示完整的本地化表单
  - 5 个字母： hour-minute-second用冒号显示

可选章节

- 使用方括号 [] 标记可选内容
- 您可以嵌套可选部分
- 所有有效数据都显示在输出中
- 输入可以省略整个可选部分

> **Note**
>
> 符号 'E'、'F'、'q' 和 'Q' 仅适用于日期时间格式（例如 date_format）。不要使用它们来解析日期时间（比如 to_timestamp）。

数字格式字符串

以下数字格式字符串适用于 TO_NUMBER 和 TO_CHAR 之类的函数。

- 有关将字符串格式化为数字的示例，请参阅TO_NUMBER。
- 有关将数字格式化为字符串的示例，请参阅TO_CHAR。

Format	描述
9	具有指定位数的数字值。
0	包含前导零的数字值。
.（句点），D	小数点。
,（逗号）	千位分隔符。
CC	世纪代码。例如，21 世纪从 2001-01-01 开始（仅受 TO_CHAR 支持）。
FM	填充模式。隐藏填补空格和零。
PR	尖括号中的负值。
S	锚定数字的符号。
L	指定位置中的货币符号。
G	组分隔符。
MI	小于 0 的数字的指定位置中的减号。
PL	大于 0 的数字的指定位置中的加号。
SG	指定位置中的加号或减号。
RN	1 到 3999 之间的罗马数字（仅受 TO_CHAR 支持）。
TH 或 th	序号后缀。不会转换小于零的分数或值。

日期和时间函数

日期和时间函数允许您对日期和时间数据执行各种操作，例如提取日期的一部分、执行日期计算、格式化日期和时间以及使用当前日期和时间。这些功能对于诸如数据分析、报告和涉及时态数据的数据处理之类的任务是必不可少的。

AWS Clean Rooms 支持以下日期和时间函数：

主题

ADD_MONTHS 函数

ADD_MONTHS 会将指定的月数添加到日期或时间戳值或表达式中。DATE_ADD 函数提供了类似的功能。

语法

```
ADD_MONTHS( {date | timestamp}, integer)
```

参数

date | timestamp

日期或时间戳列，或隐式转换为日期或时间戳的表达式。如果日期是该月的最后一天，或者如果产生的月份较短，则函数在结果中返回该月的最后一天。对于其他日期，结果包含与日期表达式相同的日期编号。

integer

正整数或负整数。使用负数从日期中减去月份。

返回类型

TIMESTAMP

示例

以下查询使用 TRUNC 函数内的 ADD_MONTHS 函数。TRUNC 函数从 ADD_MONTHS 的结果中删除一天中的时间。ADD_MONTHS 函数会为 CALDATE 列中的每个值添加 12 个月。

```
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)
```

以下示例演示 ADD_MONTHS 函数在具有不同天数的月份的日期上运行时的行为。

```
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 函数

CONVERT_TIMEZONE 将一个时区的时间戳转换为另一个时区的时间戳。该函数会自动根据夏令时调整。

语法

```
CONVERT_TIMEZONE ( ['source_timezone',] 'target_timezone', 'timestamp')
```

参数

source_timezone

（可选）当前时间戳的时区。默认值为 UTC。

target_timezone

新时间戳的时区。

timestamp

时间戳列或隐式转换为时间戳的表达式。

返回类型

TIMESTAMP

示例

以下示例将时间戳值从默认的 UTC 时区转换为 PST。

```
select convert_timezone('PST', '2008-08-21 07:23:54');

 convert_timezone
-----------------------
2008-08-20 23:23:54
```

以下示例将 LISTTIME 列中的时间戳值从默认 UTC 时区转换为 PST。尽管时间戳在夏令时间段内，但它会转换为标准时间，因为目标时区被指定为缩写 (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
```

以下示例将时间戳 LISTTIME 列从默认 UTC 时区转换为时区。 US/Pacific 目标时区使用时区名称，时间戳位于夏令时间段内，因此函数返回夏令时。

```
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
```

以下示例将时间戳字符串从 EST 转换为 PST：

```
select convert_timezone('EST', 'PST', '20080305 12:25:29');

 convert_timezone
-------------------
2008-03-05 09:25:29
```

以下示例将时间戳转换为美国东部标准时间，因为目标时区使用时区名称 (America/New_York)，并且时间戳在标准时间段内。

```
select convert_timezone('America/New_York', '2013-02-01 08:00:00');

 convert_timezone
---------------------
2013-02-01 03:00:00
(1 row)
```

以下示例将时间戳转换为美国东部夏令时，因为目标时区使用时区名称 (America/New_York)，并且时间戳在夏令时时间段内。

```
select convert_timezone('America/New_York', '2013-06-01 08:00:00');

 convert_timezone
---------------------
2013-06-01 04:00:00
(1 row)
```

以下示例演示了偏移的用法。

```
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 函数

CURRENT_DATE 以默认格式返回当前会话时区（默认为 UTC）中的日期：。 YYYY-MM-DD

> **Note**
>
> CURRENT_DATE 返回当前事务的开始日期，而不是当前语句的开始日期。考虑这样的场景，即您在 2008 年 1 月 10 日 23:59 开始一个包含多个语句的事务，而包含 CURRENT_DATE 的语句在 2008 年 2 月 10 日 00:00 运行。CURRENT_DATE 返回 10/01/08，而不是 10/02/08。

语法

```
CURRENT_DATE
```

返回类型

DATE

示例

以下示例返回当前日期（在函数运行 AWS 区域 的地方）。

```
select current_date;

   date
------------
2008-10-01
```

当前时间戳函数

CURRENT_TIMESTAMP 返回当前日期和时间，包括日期、时间以及（可选）毫秒或微秒。

当您需要获取当前日期和时间（例如，记录事件的时间戳、执行基于时间的计算或填充 date/time 列）时，此函数非常有用。

语法

```
current_timestamp()
```

返回类型

CURRENT_TIMESTAMP 函数返回一个日期。

示例

以下示例返回执行查询时的当前日期和时间，即 2020 年 4 月 25 日 15:49:11 .914（下午 3:49:11 .914）。

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

以下示例检索squirrels表中每行的当前日期和时间。

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

DATE_ADD 函数

返回起始日期之后的天数的日期。

语法

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

参数

开始日期

起始日期值。

天数

要添加的天数（整数）。正数加天数，负数减去天数。

返回类型

DATE

示例

以下示例为日期添加一天：

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

Result:
2016-07-31
```

以下示例添加了多天。

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

Result:
```

```
2016-08-04
```

使用说明

本文档适用于 Spark SQL 的 DATE_ADD 函数，与其他一些 SQL 变体相比，该函数提供了一个更简单的界面，用于向日期添加天数。要添加其他间隔，例如月或年，可能需要不同的函数。

DATE_DIFF 函数

DATE_DIFF 返回两个日期或时间表达式中日期部分的差值。

语法

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

参数

endDate

日期表达式。

startDate

日期表达式。

返回类型

BIGINT

具有 DATE 列的示例

以下示例查找两个文本日期值之间的差异（以周数为单位）。

```
select date_diff(week,'2009-01-01','2009-12-31') as numweeks;

numweeks
----------
52
(1 row)
```

以下示例查找两个文本日期值之间的差异，以小时为单位。如果您没有为日期提供时间值，则默认为 00:00:00。

```
select date_diff(hour, '2023-01-01', '2023-01-03 05:04:03');

date_diff
----------
53
(1 row)
```

以下示例查找两个文本 TIMESTAMETZ 值之间的差异，以天为单位。

```
Select date_diff(days, 'Jun 1,2008  09:59:59 EST', 'Jul 4,2008  09:59:59 EST')

date_diff
----------
33
```

以下示例查找表中同一行的两个日期之间的差异，以天为单位。

```
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)
```

以下示例查找过去日期和今天日期中的文本值之间的差异（以季度数为单位）。此示例假定当前日期为 2008 年 6 月 5 日。您可以可以用全名或缩写来命名日期部分。DATE_DIFF 函数的默认列名是 DATE_DIFF。

```
select date_diff(qtr, '1998-07-01', current_date);

date_diff
-----------
```

```
40
(1 row)
```

以下示例将 SALES 和 LISTING 表联接，以计算它们列出后多少天清单 1000 到 1005 的所有票证被售出。这些清单的最长销售等待时间为 15 天，最短等待时间不到一天（0 天）。

```
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)
```

此示例计算卖家等待所有票证销售的平均小时数。

```
select avg(date_diff(hours, listtime, saletime)) as avgwait
from sales, listing
where sales.listid = listing.listid;

avgwait
---------
465
(1 row)
```

具有 TIME 列的示例

下面的示例表 TIME_TEST 具有一个列 TIME_VAL（类型 TIME），其中插入了三个值。

```
select time_val from time_test;

time_val
---------------------
```

```
20:00:00
00:00:00.5550
00:58:00
```

以下示例查找 TIME_VAL 列与时间文本之间的小时数差异。

```
select date_diff(hour, time_val, time '15:24:45') from time_test;

 date_diff
-----------
        -5
        15
        15
```

以下示例查找两个文本时间值之间的分钟数差异。

```
select date_diff(minute, time '20:00:00', time '21:00:00') as nummins;

nummins
----------
60
```

具有 TIMETZ 列的示例

下面的示例表 TIMETZ_TEST 具有一个列 TIMETZ_VAL（类型 TIMETZ），其中插入了三个值。

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

以下示例查找 TIMETZ 文本与 timetz_val 之间的小时数差异。

```
select date_diff(hours, timetz '20:00:00 PST', timetz_val) as numhours from
 timetz_test;

numhours
----------
```

```
0
-4
1
```

以下示例查找两个文本 TIMETZ 值之间的小时数差异。

```
select date_diff(hours, timetz '20:00:00 PST', timetz '00:58:00 EST') as numhours;

numhours
----------
1
```

DATE_PART 函数

DATE_PART 从表达式中提取日期部分值。DATE_PART 是 PGDATE_PART 函数的同义词。

语法

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

参数

field

应该提取源代码的哪一部分，支持的字符串值与等效函数 EXTRACT 的字段相同。

source

应从中提取字段的日期或间隔列。

返回类型

如果字段为 “秒”，则为十进制 (8, 6)。在所有其他情况下，为整数。

示例

以下示例从日期值中提取一年中的某一天 (DOY)。输出显示，日期 “2019-08-12” 的一年中的某一天是。224这意味着 2019 年 8 月 12 日是 2019 年的第 224 天。

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

```
224
```

DATE_TRUNC 函数

DATE_TRUNC 函数根据您指定的日期部分（如小时、天或月）截断时间戳表达式或文字。

语法

```
date_trunc(format, datetime)
```

参数

format

表示要截断的单位的格式。有效格式如下所示：

- “YEAR”、“YYYY”、“YY”-截断到 ts 所在年的第一个日期，时间部分将为零
- “QUARTER”-截断到 ts 所在季度的第一个日期，时间部分将为零
- “月”、“MM”、“MON”-截断到 ts 所在月的第一个日期，时间部分将为零
- “WEEK”-截断到 ts 所在周的星期一，时间部分将为零
- “DAY”、“DD” — 将时间部分归零
- “HOUR”-用分数部分将分钟和秒归零
- “MINUTE”-用分数部分将秒归零
- “SECOND”-将第二部分归零
- “MILLISECOND”-将微秒归零
- “MICROSECOND”-一切都保持不变

ts

日期时间值

返回类型

返回截断为格式模型指定的单位的时间戳 ts

示例

以下示例将日期值截断为年初。输出显示，日期 “2015-03-05” 已被截断为 “2015-01-01”，即2015年初。

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

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

日间功能

DAY 函数返回日期/时间戳的月份中的某一天。

当您需要处理日期或时间戳的特定组件时，例如执行基于日期的计算、筛选数据或格式化日期值时，日期提取函数非常有用。

语法

```
day(date)
```

Arguments

date

日期或时间戳表达式。

返回值

DAY 函数返回一个整数。

示例

以下示例从输入日期中提取月份中的某一天 (30) '2009-07-30'。

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

以下示例从squirrels表的birthday列中提取月份中的某一天，并将结果作为 SELECT 语句的输出返回。此查询的输出将是一个日期值列表，squirrels表中每行一个，代表每只松鼠生日的月中的某一天。

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

“月日” 功能

DAYOFMONTH 函数返回的月份中的某一天 date/timestamp （该值介于 1 和 31 之间，具体取决于月份和年份）。

DAYOFMONTH 函数与 DAY 函数类似，但它们的名称略有不同，行为也略有不同。DAY 函数更常用，但是 DAYOFMONTH 函数可以用作替代函数。当您需要对包含日期或时间戳数据的表执行基于日期的分析或筛选时，例如提取日期的特定组成部分以供进一步处理或报告时，这种类型的查询可能很有用。

语法

```
dayofmonth(date)
```

Arguments

date

日期或时间戳表达式。

返回值

DAYOFMONTH 函数返回一个整数。

示例

以下示例从输入日期中提取月份中的某一天 (30) '2009-07-30'。

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

以下示例将 DAYOFMONTH 函数应用于birthday表的squirrels列。对于squirrels表中的每一行，将从该birthday列中提取月份中的某一天，并将其作为 SELECT 语句的输出返回。此查询的输出将是一个日期值列表，squirrels表中每行一个，代表每只松鼠生日的月中的某一天。

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

“一周日” 功能

DAYOFWEEK 函数将日期或时间戳作为输入，并以数字形式返回一周中的某天（1 代表星期日，2 代表星期一，...，7 代表星期六）。

当您需要处理日期或时间戳的特定组件时，例如执行基于日期的计算、筛选数据或格式化日期值时，此日期提取功能非常有用。

语法

```
dayofweek(date)
```

Arguments

date

日期或时间戳表达式。

返回值

DAYOFWEEK 函数返回一个整数，其中

1 = 星期日

2 = 星期一

3 = 星期二

4 = 星期三

5 = 星期四

6 = 星期五

7 = 星期六

示例

以下示例从该日期中提取一周中的某一天，即 5（代表星期四）。

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

以下示例从squirrels表的birthday列中提取星期几并将结果作为 SELECT 语句的输出返回。此查询的输出将是一周中的某天值列表，squirrels表中每行一个，代表每只松鼠生日的一周中的某一天。

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

“年中日” 功能

DAYOFYEAR 函数是一个日期提取函数，它以日期或时间戳作为输入并返回一年中的某一天（值介于 1 到 366 之间，具体取决于年份以及是否为闰年）。

当您需要处理日期或时间戳的特定组件时，例如执行基于日期的计算、筛选数据或格式化日期值时，此函数非常有用。

语法

```
dayofyear(date)
```

Arguments

date

日期或时间戳表达式。

返回值

DAYOFYEAR 函数返回一个整数（介于 1 到 366 之间，具体取决于年份以及是否为闰年）。

示例

以下示例从输入日期中提取一年中的某一天 (100) '2016-04-09'。

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

以下示例从squirrels表的birthday列中提取一年中的某一天，并将结果作为 SELECT 语句的输出返回。

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

EXTRACT 函数

EXTRACT 函数返回 TIMESTAMP、TIMESTAMPTZ、TIME 或 TIMETZ 值中的日期或时间部分。示例包括时间戳中的日、月、年、小时、分钟、秒、毫秒或微秒。

语法

```
EXTRACT(datepart FROM source)
```

参数

datepart

要提取的日期或时间的子字段，例如日、月、年、小时、分钟、毫秒或微秒。有关可能的值，请参阅日期或时间戳函数的日期部分。

source

计算结果为 TIMESTAMP、TIMESTAMPTZ、TIME 或 TIMETZ 数据类型的列或表达式。

返回类型

如果 source 值的计算结果为数据类型 TIMESTAMP、TIME 或 TIMETZ，则为 INTEGER。

如果 source 值的计算结果为数据类型 TIMESTAMPTZ，则为 DOUBLE PRECISION。

TIME 示例

下面的示例表 TIME_TEST 具有一个列 TIME_VAL（类型 TIME），其中插入了三个值。

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

以下示例从每个 time_val 中提取分钟数。

```
select extract(minute from time_val) as minutes from time_test;

minutes
-----------
         0
         0
         58
```

以下示例从每个 time_val 中提取小时数。

```
select extract(hour from time_val) as hours from time_test;

hours
-----------
         20
         0
         0
```

FROM_UTC_TIMESTAMP 函数

FROM_UTC_TIMESTAMP 函数将输入日期从 UTC（协调世界时）转换为指定的时区。

当您需要将日期和时间值从 UTC 转换为特定时区时，此函数非常有用。当处理来自世界不同地区且需要在适当的当地时间呈现的数据时，这一点可能很重要。

语法

```
from_utc_timestamp(timestamp, timezone
```

Arguments

timestamp

带有 UTC 时间戳的时间戳表达式。

timezone

一个 STRING 表达式，它是一个有效的时区，应将输入日期或时间戳转换为该时区。

返回值

FROM_UTC_TIMESTAMP 函数返回时间戳。

示例

以下示例将输入日期从 UTC 转换为指定的时区 (`'Asia/Seoul'`)，在本例中为 UTC 提前 9 小时。生成的输出是首尔时区的日期和时间，即`2016-08-31 09:00:00`。

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

```
 2016-08-31 09:00:00
```

小时功能

HOUR 函数是一个时间提取函数，它以时间或时间戳作为输入并返回小时分量（介于 0 和 23 之间的值）。

当您需要处理时间或时间戳的特定组成部分时，例如执行基于时间的计算、筛选数据或格式化时间值时，此时间提取功能非常有用。

语法

```
hour(timestamp)
```

Arguments

timestamp

时间戳表达式。

返回值

HOUR 函数返回一个整数。

示例

以下示例从输入时间戳'2009-07-30 12:58:59'中提取小时部分 (12)。

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

分钟功能

MINUTE 函数是一个时间提取函数，它以时间或时间戳作为输入并返回分钟分量（介于 0 到 60 之间的值）。

语法

```
minute(timestamp)
```

Arguments

timestamp

时间戳表达式或有效时间戳格式的字符串。

返回值

MINUTE 函数返回一个整数。

示例

以下示例从输入时间戳'2009-07-30 12:58:59'中提取分钟分量 (58)。

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

月份函数

MONTH 函数是一个时间提取函数，它以时间或时间戳作为输入并返回月份部分（介于 0 到 12 之间的值）。

语法

```
month(date)
```

Arguments

date

时间戳表达式或有效时间戳格式的字符串。

返回值

MONTH 函数返回一个整数。

示例

以下示例从输入时间戳'2016-07-30'中提取月份部分 (7)。

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

第二个函数

SECOND 函数是一个时间提取函数，它以时间或时间戳作为输入并返回第二个分量（介于 0 和 60 之间的值）。

语法

```
second(timestamp)
```

Arguments

timestamp

时间戳表达式。

返回值

第二个函数返回一个整数。

示例

以下示例从输入时间戳'2009-07-30 12:58:59'中提取第二个分量 (59)。

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

时间戳函数

TIMESTAMP 函数获取一个值（通常是一个数字）并将其转换为时间戳数据类型。

当您需要将表示时间或日期的数值转换为时间戳数据类型时，此函数非常有用。当您处理以数字格式存储的数据（例如 Unix 时间戳或纪元时间）时，这会很有帮助。

语法

```
timestamp(expr)
```

Arguments

expr

任何可以转换为 TIMESTAMP 的表达式。

返回值

时间戳函数返回时间戳。

示例

以下示例将数字 Unix 时间戳 (1632416400) 转换为其相应的时间戳数据类型：世界标准时间 2021 年 9 月 22 日下午 12:00:00。

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

TO_TIMESTAMP 函数

TO_TIMESTAMP 将 TIMESTAMP 字符串转换为 TIMESTAMPTZ。

语法

```
to_timestamp (timestamp)
```

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

参数

timestamp

可以转换为时间戳字符串的时间戳字符串或数据类型。

format

与 Spark 的日期时间模式相匹配的字符串文字。有关有效的日期时间模式，请参阅用于格式化和解析的日期时间模式。

返回类型

TIMESTAMP

示例

以下示例演示如何使用 TO_TIMESTAMP 函数将时间戳字符串转换为时间戳。

```
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
```

可以传递日期的 TO_TIMESTAMP 部分。其余日期部分设置为默认值。时间包括在输出中：

```
SELECT TO_TIMESTAMP('2017','YYYY');

to_timestamp
---------------------------
2017-01-01 00:00:00+00
```

以下 SQL 语句将字符串 '2011-12-18 24:38:15 '转换为时间戳。结果是时间戳落在第二天，因为小时数超过 24 小时：

```
select to_timestamp('2011-12-18 24:38:15', 'YYYY-MM-DD HH24:MI:SS');

to_timestamp
----------------------
2011-12-19 00:38:15+00
```

年份函数

YEAR 函数是一个日期提取函数，它以日期或时间戳作为输入并返回年份部分（一个四位数的数字）。

语法

```
year(date)
```

Arguments

date

日期或时间戳表达式。

返回值

YEAR 函数返回一个整数。

示例

以下示例从输入日期中提取年份部分 (2016) '2016-07-30'。

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

以下示例从squirrels表的birthday列中提取年份部分，并将结果作为 SELECT 语句的输出返回。此查询的输出将是一个年份值列表，squirrels表中每行一个，代表每只松鼠的生日年份。

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

日期或时间戳函数的日期部分

下表标识了作为以下函数参数接受的日期部分和时间部分的名称和缩写：

- DATE_ADD
- DATE_DIFF
- DATE_PART
- EXTRACT

日期部分或时间部分	缩写
millennium、millennia	mil、mils
century、centuries	c、cent、cents
decade、decades	dec、decs
纪元	epoch（由 EXTRACT 提供支持）
year、years	y、yr、yrs
quarter、quarters	qtr、qtrs
month、months	mon,、mons
week、weeks	w

日期部分或时间部分	缩写
星期几	dayofweek、dow、dw、weekday（由 DATE_PART 和 EXTRACT 函数 提供支持） 返回 0–6 的整数（星期日是第一个数）。 Note DOW 日期部分的运行方式与用于日期时间格式字符串的星期 (D) 日期部分不同。D 是基于 1–7 的整数，其中星期日是 1。有关更多信息，请参阅 日期时间格式字符串。
一年中的日期	dayofyear、doy、dy、yearday（由 EXTRACT提供支持）
day、days	d
hour、hours	h、hr、hrs
minute、minutes	m、min、mins
second、seconds	s、sec、secs
millisecond、milliseconds	ms、msec、msecs、msecond、mseconds、millisec、millisecs、millisecon
microsecond、microseconds	microsec、microsecs、microsecond、usecond、useconds、us、usec、usecs
timezone、timezone_hour、timezone_minute	由 EXTRACT 支持，仅用于带有时区的时间戳 (TIMESTAMPTZ)。

秒、毫秒和微秒导致的结果差异

当不同的日期函数指定秒、毫秒或微秒作为日期部分时，查询结果会出现细微差异：

- EXTRACT 函数仅返回指定日期部分的整数，忽略较高级别和较低级别的日期部分。如果指定的日期部分为秒，则结果中不包括毫秒和微秒。如果指定的日期部分为毫秒，则不包括秒和微秒。如果指定的日期部分为微秒，则不包括秒和毫秒。

- DATE_PART 函数返回时间戳的完整秒部分，无论指定的日期部分是什么，从而根据需要返回十进制值或整数。

CENTURY、EPOCH、DECADE 和 MIL 说明

CENTURY 或 CENTURIES

AWS Clean Rooms 将世纪解释为从年份 ## #1 开始并以年份结尾：###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

EPOCH 的 AWS Clean Rooms 实现与 1970-01-01 00:00:00.00.000 无关，与集群所在的时区无关。根据集群所在的时区，您可能需要按小时差来抵消结果。

DECADE 或 DECADES

AWS Clean Rooms 根据通用日历解释 “十年” 或 “十年” 的日期部分。例如，由于公历从第一年开始，因此第一个十年（第 1 个十年）是 0001-01-01 到 0009-12-31，而第二个十年（第 2 个十年）是 0010-01-01 到 0019-12-31。例如，十年 201 为 2000-01-01 - 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 或 MILS

AWS Clean Rooms 将 MIL 解释为从年 #001 的第一天开始，到一年的最后一天结束：#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)
```

加密和解密功能

加密和解密功能通过在可读的纯文本形式和不可读的密文形式之间进行转换，帮助 SQL 开发人员保护敏感数据免遭未经授权的访问或滥用。

AWS Clean Rooms Spark SQL 支持以下加密和解密功能：

主题

- AES_加密功能
- AES_DECRYPT 函数

AES_加密功能

AES_ENCRYPT 函数用于使用高级加密标准 (AES) 算法对数据进行加密。

语法

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

参数

expr

要加密的二进制值。

密钥

用于加密数据的密码。

支持 16、24 和 32 位的密钥长度。

mode

指定应使用哪种分组密码模式来加密消息。

有效模式：ECB（电子 CodeBook）、GCM（伽罗瓦/计数器模式）、CBC（密码块链接）。

填充

指定如何填充长度不是区块大小的倍数的消息。

有效值：PKCS、无、默认。

默认填充表示欧洲央行的 PKCS（公钥加密标准），GCM 为 NONE，CBC 为 PKCS。

支持的（模式、填充）组合是（'ECB'、'P KCS'）、（'GCM'、'NONE'）和（'CBC'、'PKCS'）。

iv

可选的初始化向量 (IV)。仅支持 CBC 和 GCM 模式。

有效值：GCM 长度为 12 字节，CBC 为 16 字节。

aad

可选的其他经过身份验证的数据 (AAD)。仅支持 GCM 模式。这可以是任何自由格式的输入，并且必须同时用于加密和解密。

返回类型

AES_ENCRYPT 函数在给定模式下使用 AES 返回带有指定填充的 e xpr 的加密值。

示例

以下示例演示如何使用 Spark SQL AES_ENCRYPT 函数使用指定的加密密钥安全地加密一串数据（在本例中为 “Spark” 一词）。然后对生成的密文进行 Base64 编码，使其更易于存储或传输。

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

以下示例演示如何使用 Spark SQL AES_ENCRYPT 函数使用指定的加密密钥安全地加密一串数据（在本例中为 “Spark” 一词）。然后，生成的密文以十六进制格式表示，这对于数据存储、传输或调试等任务非常有用。

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

以下示例演示如何使用 Spark SQL AES_ENCRYPT 函数使用指定的加密密钥、加密模式和填充模式安全地加密一串数据（在本例中为 “Spark SQL”）。然后对生成的密文进行 Base64 编码，使其更易于存储或传输。

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

AES_DECRYPT 函数

AES_DECRYPT 函数用于使用高级加密标准 (AES) 算法解密数据。

语法

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

参数

expr

要解密的二进制值。

密钥

用于解密数据的密码。

密码必须与最初用于生成加密值的密钥相匹配，并且长度必须为 16、24 或 32 字节。

mode

指定应使用哪种分组密码模式来解密消息。

有效模式：ECB、GCM、CBC。

填充

指定如何填充长度不是区块大小的倍数的消息。

有效值：PKCS、无、默认。

默认填充表示欧洲央行的 PKCS、GCM 的 NONE 和 CBC 的 PKCS。

aad

可选的其他经过身份验证的数据 (AAD)。仅支持 GCM 模式。这可以是任何自由格式的输入，并且必须同时用于加密和解密。

返回类型

在带填充的模式下使用 AES 返回解密后的 e xpr 值。

示例

以下示例演示如何使用 Spark SQL AES_ENCRYPT 函数使用指定的加密密钥安全地加密一串数据（在本例中为 “Spark” 一词）。然后对生成的密文进行 Base64 编码，使其更易于存储或传输。

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

以下示例演示了如何使用 Spark SQL AES_DECRYPT 函数来解密之前已加密和 Base64 编码的数据。解密过程需要正确的加密密钥和参数（加密模式和填充模式）才能成功恢复原始纯文本数据。

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

哈希函数

哈希函数是将数值输入值转换为另一个值的数学函数。

AWS Clean Rooms Spark SQL 支持以下哈希函数：

主题

MD5 函数

使用 MD5 加密哈希函数将长度可变的字符串转换为 32 个字符的字符串，该字符串是 128 位校验和的十六进制值的文本表示形式。

语法

```
MD5(string)
```

参数

string

一个长度可变的字符串。

返回类型

该 MD5 函数返回一个 32 个字符的字符串，该字符串是 128 位校验和的十六进制值的文本表示形式。

示例

以下示例显示了字符串“AWS Clean Rooms”的 128 位值：

```
select md5('AWS Clean Rooms');
md5
----------------------------------
f7415e33f972c03abd4f3fed36748f7a
(1 row)
```

SHA 函数

SHA1 函数的同义词。

请参阅SHA1 函数。

SHA1 函数

该 SHA1 函数使用 SHA1 加密哈希函数将长度可变的字符串转换为 40 个字符的字符串，该字符串是 160 位校验和的十六进制值的文本表示形式。

语法

SHA1 是的同义词。SHA 函数

```
SHA1(string)
```

参数

string

一个长度可变的字符串。

返回类型

该 SHA1 函数返回一个 40 个字符的字符串，该字符串是 160 位校验和的十六进制值的文本表示形式。

示例

以下示例返回单词“AWS Clean Rooms”的 160 位值：

```
select sha1('AWS Clean Rooms');
```

SHA2 函数

该 SHA2 函数使用 SHA2 加密哈希函数将长度可变的字符串转换为字符串。该字符串是具有指定位数的校验和的十六进制值的文本表示形式。

语法

```
SHA2(string, bits)
```

参数

string

一个长度可变的字符串。

integer

哈希函数中的位数。有效值为 0（与 256 相同）、224、256、384 和 512。

返回类型

该 SHA2 函数返回一个字符串，该字符串是校验和的十六进制值的文本表示形式，如果位数无效，则返回一个空字符串。

示例

以下示例返回单词“AWS Clean Rooms”的 256 位值：

```
select sha2('AWS Clean Rooms', 256);
```

xx HASH64 函数

xxhash64 函数返回参数的 64 位哈希值。

xxhash64 () 函数是一种非加密哈希函数，旨在实现快速和高效。它通常用于数据处理和存储应用程序，其中需要数据的唯一标识符，但不需要对数据的确切内容保密。

在 SQL 查询的上下文中，xxhash64 () 函数可以用于各种用途，例如：

- 为表中的一行生成唯一标识符
- 根据哈希值对数据进行分区
- 实现自定义索引或数据分发策略

具体用例将取决于应用程序的要求和正在处理的数据。

语法

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

Arguments

expr1

任何类型的表达式。

expr2

任何类型的表达式。

返回值

返回参数的 64 位哈希值 (BIGINT)。哈希种子是 42。

示例

以下示例根据提供的输入生成一个 64 位哈希值 (5602566077635097486)。第一个参数是字符串值，在本例中为 “Spark” 一词。第二个参数是一个包含单个整数值 123 的数组。第三个参数是一个整数值，代表哈希函数的种子。

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

超级日志函数

SQL 中的 HyperLogLog (HLL) 函数提供了一种高效估计大型数据集中唯一元素（基数）数量的方法，即使未存储实际的唯一元素集也是如此。

使用 HLL 函数的主要好处是：

- 存储效率：HLL 草图所需的内存比存储全套独特元素少得多，因此适合大型数据集。
- 分布式计算：HLL 草图可以跨多个数据源或处理节点进行组合，从而实现高效的分布式唯一计数估计。
- 近似结果：HLL 提供了近似的唯一计数估计，在精度和内存使用之间进行了可调整的权衡（通过精度参数）。

这些函数在需要估计唯一项目数量的场景中特别有用，例如在分析、数据仓库和实时流处理应用程序中。

AWS Clean Rooms 支持以下 HLL 函数。

主题

HLL_SKETCH_AGG 函数

HLL_SKETCH_AGG 聚合函数根据指定列中的值创建 HLL 草图。它返回封装输入表达式值的 HLLSKETCH 数据类型。

HLL_SKETCH_AGG 聚合函数适用于任何数据类型并忽略空值。

如果表中没有行或所有行都为 NULL，则生成的草图没有 `{"version":1,"logm":15,"sparse":{"indices":[],"values":[]}}` 之类的索引值对。

语法

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

参数

aggregate_expression

任何类型为 INT、BIGINT、STRING 或 BINARY 的表达式，将对其进行唯一计数。任何NULL值都将被忽略。

lgConfigk

一个介于 4 到 21 之间的可选整数常数，包括默认值 12。K 的 log-base-2，其中 K 是草图的桶或槽的数量。

返回类型

HLL_SKETCH_AGG 函数返回一个非空二进制缓冲区，其中包含由于消耗和聚合聚合组中的所有输入值而计算出的 HyperLogLog 草图。

示例

以下示例使用 HyperLogLog (HLL) 算法来估计col列中不同值的数量。该hll_sketch_agg(col, 12)函数聚合 col 列中的值，使用 12 的精度创建 HLL 草图。然后，该hll_sketch_estimate()函数用于根据生成的 HLL 草图估计不同值的数量。查询的最终结果是 3，它表示该col列中估计的不同值数量。在本例中，不同的值为 1、2 和 3。

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

以下示例还使用 HLL 算法来估计col列中不同值的数量，但它没有为 HLL 草图指定精度值。在这种情况下，它使用默认精度 14。该hll_sketch_agg(col)函数获取col列中的值并创建一个 HyperLogLog (HLL) 草图，这是一个紧凑的数据结构，可用于估计元素的不同数量。该hll_sketch_estimate(hll_sketch_agg(col))函数采用在上一步中创建的 HLL 草图，并计算col列中不同值计数的估计值。查询的最终结果是 3，它表示该col列中估计的不同值数量。在本例中，不同的值为 1、2 和 3。

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

HLL_SKETCH_ESTIMATE 函数

HLL_SKETCH_ESTIMATE 函数获取 HLL 草图并估计该草图所表示的唯一元素的数量。它使用 HyperLogLog (HLL) 算法计算给定列中唯一值数量的概率近似值，消耗先前由 HLL_SKETCH_AGG 函数生成的称为草图缓冲区的二进制表示形式，并将结果作为大整数返回。

HLL 草图算法提供了一种有效的方法来估计唯一元素的数量，即使对于大型数据集也是如此，而无需存储完整的唯一值集。

hll_union和hll_union_agg函数还可以通过消耗和合并这些缓冲区作为输入来将草图组合在一起。

语法

```
HLL_SKETCH_ESTIMATE (hllsketch_expression)
```

参数

hllsketch_expression

一个包含由 HLL_SKETCH_AGG 生成的草图的BINARY表达式

返回类型

HLL_SKETCH_ESTIMATE 函数返回一个 BIGINT 值，该值是输入草图表示的近似不同计数。

示例

以下示例使用 HyperLogLog (HLL) 草绘算法来估计列中值的基数（唯一计数）。col该hll_sketch_agg(col, 12)函数取col列并使用 12 位的精度创建 HLL 草图。HLL 草图是一种近似数据结构，可以有效地估计集合中唯一元素的数量。该hll_sketch_estimate()函数采用由创建的 HLL 草图，hll_sketch_agg并估计草图所表示的值的基数（唯一计数）。FROM VALUES (1), (1), (2), (2), (3) tab(col);生成一个包含 5 行的测试数据集，其中该col列包含值 1、1、2、2 和 3。此查询的结果是该col列中值的估计唯一计数，即 3。

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

以下示例与上一个示例的区别在于，hll_sketch_agg函数调用中未指定精度参数（12 位）。在这种情况下，将使用 14 位的默认精度，与之前使用 12 位精度的示例相比，这可能为唯一计数提供更准确的估计值。

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

HLL_UNION 函数

HLL_UNION 函数将两个 HLL 草图组合成一个统一的草图。它使用 HyperLogLog (HLL) 算法将两幅草图合并为一张草图。查询可以使用生成的缓冲区将近似的唯一计数计算为该hll_sketch_estimate函数的长整数。

语法

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

参数

expRN

一个包含由 HLL_SKETCH_AGG 生成的草图的BINARY表达式。

allowDifferentLgConfigK

一个可选的布尔表达式，用于控制是否允许合并两个具有不同 LGConfigk 值的草图。默认值为false。

返回类型

HLL_UNION 函数返回一个二进制缓冲区，其中包含组合输入表达式后计算出的 HyperLogLog 草图。当allowDifferentLgConfigK参数为时true，结果草图使用提供的两个lgConfigK值中较小的一个。

示例

以下示例使用 HyperLogLog (HLL) 草图算法来估计数据集中两列col1和col2值的唯一计数。

该hll_sketch_agg(col1)函数为col1列中的唯一值创建 HLL 草图。

该hll_sketch_agg(col2)函数为 col2 列中的唯一值创建 HLL 草图。

该hll_union(...)函数将步骤 1 和步骤 2 中创建的两个 HLL 草图组合成一个统一的 HLL 草图。

该hll_sketch_estimate(...)函数采用组合的 HLL 草图，并估计和的唯一值数量col1。col2

该FROM VALUES子句生成一个包含 5 行的测试数据集，其中col1包含值 1、1、2、2 和 3，col2包含值 4、4、5、5 和 6。

此查询的结果是两个col1和的估计唯一值计数col2，即 6。HLL 草图算法提供了一种有效的方法来估计唯一元素的数量，即使对于大型数据集也是如此，而无需存储完整的唯一值集。在此示例中，该hll_union函数用于组合来自两列的 HLL 草图，这样可以估计整个数据集的唯一计数，而不仅仅是单独估计每列的唯一计数。

```
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
```

以下示例与上一个示例的区别在于，hll_sketch_agg函数调用中未指定精度参数（12 位）。在这种情况下，将使用 14 位的默认精度，与之前使用 12 位精度的示例相比，这可能为唯一计数提供更准确的估计值。

```
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 函数

HLL_UNION_AGG 函数将多个 HLL 草图组合成一个统一的草图。它使用 HyperLogLog （HLL）算法将一组草图组合成一个草图。查询可以使用生成的缓冲区通过该hll_sketch_estimate函数计算近似的唯一计数。

语法

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

参数

expr

一个包含由 HLL_SKETCH_AGG 生成的草图的BINARY表达式。

allowDifferentLgConfigK

一个可选的布尔表达式，用于控制是否允许合并两个具有不同 LGConfigk 值的草图。默认值为false。

返回类型

HLL_UNION_AGG 函数返回一个二进制缓冲区，其中包含通过组合同一组的输入表达式而计算出的 HyperLogLog 草图。当`allowDifferentLgConfigK`参数为时`true`，结果草图使用提供的两个`lgConfigK`值中较小的一个。

示例

以下示例使用 HyperLogLog (HLL) 草图绘制算法来估计多个 HLL 草图的唯一值数量。

第一个示例估计数据集中值的唯一数量。

```
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
```

内部查询创建了两个 HLL 草图：

- 第一个 SELECT 语句使用单个值 1 创建草图。
- 第二个 SELECT 语句根据另一个单一值 1 创建草图，但精度为 20。

外部查询使用 HLL_UNION_AGG 函数将两个草图组合成一个草图。然后，它将 HLL_SKETCH_ESTIMATE 函数应用于此组合草图，以估计唯一的值数。

此查询的结果是该`col`列中值的估计唯一计数，即`1`。这意味着两个输入值 1 被认为是唯一的，即使它们具有相同的值。

第二个示例包含 HLL_UNION_AGG 函数的不同精度参数。在这种情况下，两个 HLL 草图都是以 14 位的精度创建的，这使得它们可以`hll_union_agg`与参数一起成功组合。`true`

```
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
```

查询的最终结果是估计的唯一计数，在本例中也是如此1。这意味着两个输入值 1 被认为是唯一的，即使它们具有相同的值。

JSON 函数

当您需要存储相对较小的一组键值对时，您可以通过以 JSON 格式存储数据来节省空间。由于 JSON 字符串可存储在单个列中，因此使用 JSON 可能比以表格格式存储数据更高效。

Example

例如，假设您有一个稀疏表，在此表中，您需要设置多个列来完整表示所有可能的属性。但大多数列值对任何给定行或任何给定列为 NULL。通过将 JSON 用于存储，您可能能够将行的数据以键值对的形式存储在单个 JSON 字符串中并删除稀疏填充的表列。

此外，您还可以轻松修改 JSON 字符串以存储其他键值对，而无需向表添加列。

我们建议慎用 JSON。若要存储较大的数据集，JSON 不是一个好的选择，因为将分散的数据存储在单个列中后，JSON 不会利用 AWS Clean Rooms 的列存储架构。

JSON 使用 UTF-8 编码的文本字符串，因此 JSON 字符串可存储为 CHAR 或 VARCHAR 数据类型。如果字符串包含多字节字符，则使用 VARCHAR。

JSON 字符串必须是根据以下规则正确设置格式的 JSON：

- 根级别的 JSON 可以是 JSON 对象或 JSON 数组。JSON 对象是用大括号括起的一组无序的键值对（由逗号分隔）。

  例如，`{"one":1, "two":2}`
- JSON 数组是用方括号括起的一组有序值（由逗号分隔）。

  以下是示例：`["first", {"one":1}, "second", 3, null]`
- JSON 数组使用从零开始的索引；数组中的第一个元素位于位置 0。在 JSON 键:值对中，键是用双引号括起的字符串。
- JSON 值可能为以下任一值：
  - JSON 对象
  - JSON 数组
  - 用双引号括起的字符串
  - 数字 (整数和浮点)

  - 布尔值
  - Null
- 空对象和空数组是有效的 JSON 值。
- JSON 字段区分大小写。
- 将忽略 JSON 结构元素之间的空格（如 `{ }, [ ]`）。

主题

- GET_JSON_OBJECT 函数
- TO_JSON 函数

GET_JSON_OBJECT 函数

GET_JSON_OBJECT 函数从中提取一个 json 对象。`path`

语法

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

Arguments

json_txt

包含格式良好的 JSON 的字符串表达式。

path

带有格式良好的 JSON 路径表达式的字符串文字。

返回值

返回一个字符串。

如果找不到对象，则返回 NULL。

示例

以下示例从 JSON 对象中提取一个值。第一个参数是一个 JSON 字符串，它表示具有单个键值对的简单对象。第二个参数是 JSON 路径表达式。$符号表示 JSON 对象的根，该.a部分指定我们要提取与 “a” 键关联的值。该函数的输出是 'b'，这是与输入 JSON 对象中的 a “” 键关联的值。

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

TO_JSON 函数

TO_JSON 函数将输入表达式转换为 JSON 字符串表示形式。该函数处理将不同的数据类型（例如数字、字符串和布尔值）转换为相应的 JSON 表示形式。

当您需要将结构化数据（例如数据库行或 JSON 对象）转换为更便携的、自我描述的格式（如 JSON）时，TO_JSON 函数非常有用。当您需要与其他需要使用 JSON 格式数据的系统或服务进行交互时，这可能特别有用。

语法

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

Arguments

expr

要转换为 JSON 字符串的输入表达式。它可以是值、列或任何其他有效的 SQL 表达式。

options

一组可选的配置选项，可用于自定义 JSON 转换过程。这些选项可能包括诸如空值的处理、数值的表示和特殊字符的处理之类的内容。

返回值

返回具有给定结构值的 JSON 字符串

示例

以下示例将命名结构（一种结构化数据）转换为 JSON 字符串。第一个参数(named_struct('a', 1, 'b', 2)) 是传递给to_json()函数的输入表达式。它创建一个包含两个字段的命名结构：值为 1 的 “a” 和值为 2 的 “b”。to_json () 函数将命名的结构作为其参数，并将其转换为 JSON 字符串表示形式。输出是{"a":1,"b":2}，这是表示命名结构的有效 JSON 字符串。

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

以下示例将包含时间戳值的命名结构转换为具有自定义时间戳格式的 JSON 字符串。第一个参数 (named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd'))) 创建一个命名结构，其单个字段 “time” 包含时间戳值。第二个参数 (map('timestampFormat', 'dd/MM/yyyy')) 使用单个键值对创建映射（键值字典），其中键为 'timestampFormat'，值为 "。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输出是{"time":"26/08/2015"}，它是一个带有单个字段 “时间” 的 JSON 字符串，其中包含所需的 “dd/MM/yyyy” 格式的时间戳值。

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

数学函数

本节介绍了 AWS Clean Rooms Spark SQL 中支持的数学运算符和函数。

主题

- 数学运算符符号
- ABS 函数
- ACOS 函数
- ASIN 函数
- ATAN 函数
- ATAN2 函数
- CBRT 函数
- CEILING（或 CEIL）函数
- COS 函数
- COT 函数
- DEGREES 函数
- DIV 函数
- EXP 函数
- FLOOR 函数
- LN 函数

- LOG 函数
- MOD 函数
- PI 函数
- POWER 函数
- RADIANS 函数
- 兰德函数
- RANDOM 函数
- ROUND 函数
- SIGN 函数
- SIN 函数
- SQRT 函数
- TRUNC 函数

数学运算符符号

下表列出了支持的数学运算符。

支持的运算符

操作符	描述	示例	结果
+	加	2 + 3	5
-	减	2 - 3	–1
*	乘	2 * 3	6
/	除	4 / 2	2
%	取模	5 % 4	1
^	幂	2.0 ^ 3.0	8

示例

为给定交易计算支付的佣金加 2.00 美元手续费：

```
select commission, (commission + 2.00) as comm
from sales where salesid=10000;

commission | comm
-----------+-------
28.05      | 30.05
(1 row)
```

为给定交易计算销售价格的 20%：

```
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;

pricepaid | twentypct
----------+-----------
187.00    |    37.400
(1 row)
```

根据持续增长模式预测票的销售量。在此示例中，子查询将返回 2008 年销售的票数。在此后 10 年，该结果将以 5% 的连续增长率呈指数增长。

```
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)
```

查找带有大于或等于 2000 的日期 ID 的销售的总支付价格和总佣金。然后将总支付价格减去总佣金。

```
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 函数

ABS 用于计算数字的绝对值，该数字可以是文本或计算结果为数字的表达式。

语法

```
ABS (number)
```

参数

number

数字或计算结果为数字的表达式。它可以是 SMALLINT、INTEGER、BIGINT、DECIMAL 或 FLOAT4 FLOAT8 类型。

返回类型

ABS 返回与其参数相同的数据类型。

示例

计算 -38 的绝对值：

```
select abs (-38);
abs
-------
38
(1 row)
```

计算 (14-76) 的绝对值：

```
select abs (14-76);
abs
```

```
-------
62
(1 row)
```

ACOS 函数

ACOS 是返回数字的反余弦的三角函数。返回值采用弧度形式且介于 0 和 PI 之间。

语法

```
ACOS(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 -1 的反余弦，请使用以下示例。

```
SELECT ACOS(-1);

+-------------------+
|       acos        |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ASIN 函数

ASIN 是返回数字的反正弦的三角函数。返回值采用弧度形式且介于 PI/2 和 -PI/2 之间。

语法

```
ASIN(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 1 的反正弦，请使用以下示例。

```
SELECT ASIN(1) AS halfpi;

+--------------------+
|       halfpi       |
+--------------------+
| 1.5707963267948966 |
+--------------------+
```

ATAN 函数

ATAN 是返回数字的反正切的三角函数。返回值采用弧度形式且介于 -PI 和 PI 之间。

语法

```
ATAN(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 1 的反正切并将其乘以 4，请使用以下示例。

```
SELECT ATAN(1) * 4 AS pi;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ATAN2 函数

ATAN2 是一个三角函数，它返回一个数除以另一个数字的反正切值。返回值采用弧度形式且介于 PI/2 和 -PI/2 之间。

语法

```
ATAN2(number1, number2)
```

参数

number1

DOUBLE PRECISION 数值。

number2

DOUBLE PRECISION 数值。

返回类型

DOUBLE PRECISION

示例

要返回 2/2 的反正切并将其乘以 4，请使用以下示例。

```
SELECT ATAN2(2,2) * 4 AS PI;
```

```
+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

CBRT 函数

CBRT 函数是计算数字的立方根的数学函数。

语法

```
CBRT (number)
```

参数

CBRT 将 DOUBLE PRECISION 数作为参数。

返回类型

CBRT 返回 DOUBLE PRECISION 数。

示例

计算为给定交易支付的佣金的立方根：

```
select cbrt(commission) from sales where salesid=10000;

cbrt
------------------
3.03839539048843
(1 row)
```

CEILING（或 CEIL）函数

CEILING 或 CEIL 函数用于将数字向上舍入到下一个整数。（FLOOR 函数将数字向下舍入到下一个整数）

语法

```
CEIL | CEILING(number)
```

参数

number

数字或计算结果为数字的表达式。它可以是 SMALLINT、INTEGER、BIGINT、DECIMAL 或 FLOAT4 FLOAT8 类型。

返回类型

CEILING 和 CEIL 返回与其参数相同的数据类型。

示例

计算为给定销售交易支付的佣金的上限：

```
select ceiling(commission) from sales
where salesid=10000;

ceiling
---------
29
(1 row)
```

COS 函数

COS 是返回数字的余弦的三角函数。返回值采用弧度形式且介于 -1 和 1 之间（含）。

语法

```
COS(double_precision)
```

参数

number

输入参数是双精度数。

返回类型

COS 函数返回双精度数。

示例

以下示例返回 0 的余弦：

```
select cos(0);
cos
-----
1
(1 row)
```

以下示例返回 PI 的余弦：

```
select cos(pi());
cos
-----
-1
(1 row)
```

COT 函数

COT 是返回数字的余切的三角函数。输入参数必须为非零。

语法

```
COT(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 1 的余切，请使用以下示例。

```
SELECT COT(1);
```

```
+--------------------+
|        cot         |
+--------------------+
| 0.6420926159343306 |
+--------------------+
```

DEGREES 函数

将用弧度表示的角度转换用度表示。

语法

```
DEGREES(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 0.5 弧度的等效度数，请使用以下示例。

```
SELECT DEGREES(.5);

+-------------------+
|      degrees      |
+-------------------+
| 28.64788975654116 |
+-------------------+
```

要将 PI 弧度转换为度数，请使用以下示例。

```
SELECT DEGREES(pi());
```

```
+---------+
| degrees |
+---------+
|     180 |
+---------+
```

DIV 函数

DIV 运算符返回除数除以股息的整数部分。

语法

```
dividend div divisor
```

参数

分红

计算结果为数值或间隔的表达式。

除数

匹配的间隔类型 dividend if 为间隔，否则为数字。

返回类型

BIGINT

示例

以下示例从松鼠表中选择了两列：一id列（包含每只松鼠的唯一标识符）和一calculated列（表示年龄列除以 2 的整数）。age div 2该age div 2计算对age列执行整数除法，实际上是将年龄向下舍入到最接近的偶数整数。例如，如果该age列包含诸如 3、5、7 和 10 之类的值，则该age div 2列将分别包含值 1、2、3 和 5。

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

在需要根据年龄范围对数据进行分组或分析的情况下，此查询非常有用，并且您希望通过将年龄值向下舍入到最接近的偶数整数来简化年龄值。生成的输出将提供squirrels表中每只松鼠的年龄除以 2。id

EXP 函数

EXP 函数实施数值表达式的指数函数，即以自然对数 e 为底数，对表达式求次方。EXP 函数是 LN 函数的反函数。

语法

```
EXP (expression)
```

参数

expression

表达式必须是 INTEGER、DECIMAL 或 DOUBLE PRECISION 数据类型。

返回类型

EXP 返回 DOUBLE PRECISION 数。

示例

使用 EXP 函数根据持续增长模式预测票的销售量。在此示例中，子查询将返回 2008 年销售的票数。该结果将乘以 EXP 函数的结果（指定了在接下来 10 年保持 7% 的持续增长率）。

```
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 函数

FLOOR 函数将数字向下舍入到下一个整数。

语法

```
FLOOR (number)
```

参数

number

数字或计算结果为数字的表达式。它可以是 SMALLINT、INTEGER、BIGINT、DECIMAL 或 FLOAT4 FLOAT8 类型。

返回类型

FLOOR 返回与其参数相同的数据类型。

示例

此示例显示在使用 FLOOR 函数之前和之后为给定的销售交易支付的佣金值。

```
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 函数

LN 函数返回输入参数的自然对数。

语法

```
LN(expression)
```

参数

expression

对其执行函数的目标列或表达式。

> **Note**
>
> 如果表达式引用了 AWS Clean Rooms 用户创建的表或 AWS Clean Rooms STL 或 STV 系统表，则此函数会返回某些数据类型的错误。

具有以下数据类型的表达式在引用了用户创建的表或系统表时将产生错误。

- BOOLEAN
- CHAR
- DATE
- DECIMAL 或 NUMERIC
- TIMESTAMP
- VARCHAR

具有以下数据类型的表达式可在用户创建的表以及 STL 或 STV 系统表上成功运行：

- BIGINT
- DOUBLE PRECISION
- INTEGER
- REAL
- SMALLINT

返回类型

LN 函数返回与表达式相同的类型。

示例

以下示例返回数字 2.718281828 的自然对数（即以 e 为底的对数）：

```
select ln(2.718281828);
```

```
ln
--------------------
0.9999999998311267
(1 row)
```

请注意，结果约等于 1。

此示例返回 USERS 表的 USERID 列中的值的自然对数：

```
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 函数

返回 with 的对数。expr base

语法

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

参数

expr

表达式必须具有整数、小数或浮点数据类型。

base

对数计算的基数。必须是双精度数据类型的正数（不等于 1）。

返回类型

LOG 函数返回双精度数。

示例

以下示例返回数字 100 的以 10 为底的对数：

```
select log(10, 100);
--------
2
(1 row)
```

MOD 函数

返回两个数字的余数，也称为取模 运算。将第一个参数除以第二个参数来计算结果。

语法

```
MOD(number1, number2)
```

参数

number1

第一个输入参数是 INTEGER、SMALLINT、BIGINT 或 DECIMAL 数。如果其中一个参数是 DECIMAL 类型，则另一参数也必须是 DECIMAL 类型。如果其中一个参数是 INTEGER，则另一参数可以是 INTEGER、SMALLINT 或 BIGINT。两个参数也都可以是 SMALLINT 或 BIGINT，但如果一个参数是 BIGINT，则另一个参数不能是 SMALLINT。

number2

第二个参数是 INTEGER、SMALLINT、BIGINT 或 DECIMAL 数。相同的数据类型规则与 number1 一样适用于 number2。

返回类型

有效的返回类型是 DECIMAL、INT、SMALLINT 或 BIGINT。如果两个参数属于相同的类型，MOD 函数的返回类型是与输入参数相同的数值类型。但是，如果其中一个输入参数是 INTEGER，返回类型也将是 INTEGER。

使用说明

您可以使用 % 作为取模运算符。

示例

以下示例返回一个数字除以另一个数字后的余数：

```
SELECT MOD(10, 4);

 mod
------
 2
```

以下示例返回一个小数结果：

```
SELECT MOD(10.5, 4);

 mod
------
 2.5
```

您可以转换参数值：

```
SELECT MOD(CAST(16.4 as integer), 5);

 mod
------
 1
```

通过将第一个参数除以 2 来检查该参数是否为偶数：

```
SELECT mod(5,2) = 0 as is_even;

 is_even
--------
 false
```

您可以使用 % 作为取模运算符：

```
SELECT 11 % 4 as remainder;
```

```
 remainder
-----------
 3
```

以下示例返回 CATEGORY 表中的奇数类别的信息：

```
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 函数

PI 函数返回 14 个小数位的 pi 值。

语法

```
PI()
```

返回类型

DOUBLE PRECISION

示例

要返回 pi 的值，请使用以下示例。

```
SELECT PI();
```

```
+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

POWER 函数

POWER 函数是让一个数值表达式自乘到另一个数值表达式的幂的指数函数。例如，2 的三次幂的计算公式为 POWER(2,3)，结果为 8。

语法

```
{POWER(expression1, expression2)
```

参数

expression1

要自乘的数值表达式。必须是 INTEGER、DECIMAL 或 FLOAT 数据类型。

expression2

让 expression1 自乘到的幂。必须是 INTEGER、DECIMAL 或 FLOAT 数据类型。

返回类型

DOUBLE PRECISION

示例

```
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 函数

RADIANS 函数将用度表示的角度转换为用弧度表示。

语法

```
RADIANS(number)
```

参数

number

输入参数是 DOUBLE PRECISION 数。

返回类型

DOUBLE PRECISION

示例

要返回 180 度的等效弧度，请使用以下示例。

```
SELECT RADIANS(180);

+-------------------+
|      radians      |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

兰德函数

RAND 函数生成一个介于 0 和 1 之间的随机浮点数。每次调用 RAND 函数时，都会生成一个新的随机数。

语法

```
RAND()
```

返回类型

随机返回双精度。

示例

以下示例为表中的每一行生成一列介于 0 和 1 之间的随机浮点数。squirrels生成的输出将是包含随机十进制值列表的单列，松鼠表中的每行都有一个值。

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

当您需要生成随机数时，例如模拟随机事件或在数据分析中引入随机性时，这种类型的查询非常有用。在该squirrels表的上下文中，它可以用来为每只松鼠分配随机值，然后将其用于进一步的处理或分析。

RANDOM 函数

RANDOM 函数生成介于 0.0（含）和 1.0（不含）之间的随机值。

语法

```
RANDOM()
```

返回类型

RANDOM 返回 DOUBLE PRECISION 数。

示例

1. 计算介于 0 和 99 之间的随机值。如果随机数为 0 - 1，此查询将生成 0 - 100 的随机值：

```
select cast (random() * 100 as int);

INTEGER
------
24
(1 row)
```

2. 检索 10 个项目的统一随机样本：

```
select *
from sales
order by random()
limit 10;
```

现在检索 10 个项目的随机样本，但选择与其价格成比例的项目。例如，价格是另一个两倍的项目在查询结果中出现的可能性是其两倍：

```
select *
from sales
order by log(1 - random()) / pricepaid
limit 10;
```

3. 此示例使用 SET 命令设置一个 SEED 值，以使 RANDOM 生成可预测的数字序列。

首先，返回三个 RANDOM 整数，而不先设置 SEED 值：

```
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)
```

现在，将 SEED 值设置为 .25，并返回 3 个以上的 RANDOM 数字：

```
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)
```

最后，将 SEED 值重置为 .25，并验证 RANDOM 是否返回与前三个调用相同的结果：

```
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 函数

ROUND 函数将数字舍入到最近的整数或小数。

ROUND 函数可以选择性地以整数形式包含另一个参数，指示在任意方向舍入到的小数位数。当您不提供第二个参数时，函数会舍入到最接近的整数。指定第二个参数 >n 时，函数将舍入为最接近的数字，其中精度为 n 个小数位。

语法

```
ROUND (number [ , integer ] )
```

参数

number

数字或计算结果为数字的表达式。它可以是十进制或 FLOAT8 类型。 AWS Clean Rooms 可以根据隐式转换规则转换其他数据类型。

integer（可选）

一个整数，指示任意方向四舍五入的小数位数。

返回类型

ROUND 返回与输入参数相同的数字数据类型。

示例

将为给定交易支付的佣金舍入到最近的整数。

```
select commission, round(commission)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    28
(1 row)
```

将为给定交易支付的佣金舍入到第一个小数位。

```
select commission, round(commission, 1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |  28.1
(1 row)
```

对于同一查询，请沿相反的方向扩展精度。

```
select commission, round(commission, -1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    30
(1 row)
```

SIGN 函数

SIGN 函数返回数字的符号（正或负）。SIGN 函数的结果为 1、-1 或 0，表示参数的符号。

语法

```
SIGN (number)
```

参数

number

数字或计算结果为数字的表达式。它可以是 DECIMALor FLOAT8 类型。 AWS Clean Rooms 可以根据隐式转换规则转换其他数据类型。

返回类型

SIGN 返回与输入参数相同的数字数据类型。如果输入为 DECIMAL，则输出为 DECIMAL(1,0)。

示例

要从 SALES 表中确定为给定交易支付的佣金的符号，请使用以下示例。

```
SELECT commission, SIGN(commission)
FROM sales WHERE salesid=10000;

+------------+------+
| commission | sign |
+------------+------+
|      28.05 |    1 |
```

```
+------------+------+
```

SIN 函数

SIN 是返回数字的正弦的三角函数。返回值介于 -1 与 1 之间。

语法

```
SIN(number)
```

参数

number

以弧度表示的 DOUBLE PRECISION 数值。

返回类型

DOUBLE PRECISION

示例

要返回 -PI 的正弦，请使用以下示例。

```
SELECT SIN(-PI());

+-------------------------+
|           sin           |
+-------------------------+
| -0.00000000000000012246 |
+-------------------------+
```

SQRT 函数

SQRT 函数返回数字值的平方根。平方根是一个乘以自身以得到给定值的数字。

语法

```
SQRT (expression)
```

参数

expression

表达式必须具有整数、小数或浮点数据类型。表达式可以包含函数。系统可能会执行隐式类型转换。

返回类型

SQRT 返回 DOUBLE PRECISION 数。

示例

以下示例返回数字的平方根。

```
select sqrt(16);

sqrt
---------------
4
```

以下示例执行隐式类型转换。

```
select sqrt('16');

sqrt
---------------
4
```

以下示例嵌套函数以执行更复杂的任务。

```
select sqrt(round(16.4));

sqrt
---------------
4
```

以下示例得出给定圆面积时的半径长度。例如，当给定以平方英寸为单位的面积时，它以英寸为单位计算半径。示例中的面积为 20。

```
select sqrt(20/pi());
```

这将返回值 5.046265044040321。

以下示例返回 SALES 表中 COMMISSION 值的平方根。COMMISSION 列是 DECIMAL 列。此示例说明如何在具有更复杂条件逻辑的查询中使用该函数。

```
select sqrt(commission)
from sales where salesid < 10 order by salesid;

sqrt
------------------
10.4498803820905
3.37638860322683
7.24568837309472
5.1234753829798
...
```

以下查询返回同一组 COMMISSION 值的平方根的舍入值。

```
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
...
```

有关中示例数据的更多信息 AWS Clean Rooms，请参阅示例数据库。

TRUNC 函数

TRUNC 函数将数字截断为前一个整数或小数。

TRUNC 函数可以选择性地以整数形式包含另一个参数，指示在任意方向舍入到的小数位数。当您不提供第二个参数时，函数会舍入到最接近的整数。当指定第二个参数 >n 时，函数将舍入为最接近的数字 >n 精度的小数位。此函数还会截断时间戳并返回日期。

语法

```
TRUNC (number [ , integer ] |
```

```
timestamp )
```

参数

number

数字或计算结果为数字的表达式。它可以是十进制或 FLOAT8 类型。 AWS Clean Rooms 可以根据隐式转换规则转换其他数据类型。

integer（可选）

一个整数，指示精度在任意方向的小数位数。如果未提供整数，数字将作为整数截断；如果指定了整数，数字将截断到指定的小数位。

timestamp

该函数也可返回时间戳中的日期。（要返回以 00:00:00 作为时间的时间戳值，请将函数结果强制转换为时间戳。）

返回类型

TRUNC 返回与第一个输入参数的数据类型相同的数据类型。对于时间戳，TRUNC 将返回日期。

示例

截断为给定销售交易支付的佣金。

```
select commission, trunc(commission)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   111

(1 row)
```

将同一佣金值截断到第一个小数位。

```
select commission, trunc(commission,1)
from sales where salesid=784;
```

```
commission | trunc
-----------+-------
    111.15 | 111.1

(1 row)
```

截断第二个参数为负值的佣金；111.15 向下舍入到 110。

```
select commission, trunc(commission,-1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   110
(1 row)
```

返回 SYSDATE 函数（返回时间戳）的结果的日期部分：

```
select sysdate;

timestamp
----------------------------
2011-07-21 10:32:38.248109
(1 row)

select trunc(sysdate);

trunc
------------
2011-07-21
(1 row)
```

将 TRUNC 函数应用于 TIMESTAMP 列。返回类型为日期。

```
select trunc(starttime) from event
order by eventid limit 1;

trunc
------------
2008-01-25
(1 row)
```

标量函数

本节介绍了 AWS Clean Rooms Spark SQL 中支持的标量函数。标量函数是一种将一个或多个值作为输入并返回单个值作为输出的函数。标量函数对单个行或元素进行操作，并为每个输入生成单个结果。

标量函数（例如 SIZE）与其他类型的 SQL 函数不同，例如聚合函数（计数、求和、平均值）和生成表的函数（分解、展平）。这些其他函数类型对多行进行操作或生成多行，而标量函数对单个行或元素起作用。

主题

大小函数

SIZE 函数将现有的数组、映射或字符串作为参数，并返回一个表示该数据结构大小或长度的单个值。它不会创建新的数据结构。它用于查询和分析现有数据结构的属性，而不是用于创建新的数据结构。

此函数对于确定数组中元素的数量或字符串的长度非常有用。在 SQL 中处理数组和其他数据结构时，它可能特别有用，因为它允许您获取有关数据大小或基数的信息。

语法

```
size(expr)
```

Arguments

expr

数组、映射或字符串表达式。

返回类型

SIZE 函数返回一个整数。

示例

在此示例中，将 SIZE 函数应用于数组`['b', 'd', 'c', 'a']`，它返回值`4`，即数组中元素的数量。

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

在此示例中，将 SIZE 函数应用于地图`{'a': 1, 'b': 2}`，它返回值`2`，即地图中键值对的数量。

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

在此示例中，将 SIZE 函数应用于字符串 `'hello world'`11，它返回值，即字符串中的字符数。

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

字符串函数

字符串函数用于处理和操作字符串或计算结果为字符串的表达式。当这些函数中的 string 参数为文本值时，该参数必须括在单引号中。支持的数据类型包括 CHAR 和 VARCHAR。

以下部分提供了支持的函数的函数名称、语法和描述。对字符串的所有偏移都从 1 开始。

主题

- || (串联) 运算符
- BTRIM 函数
- CONCAT 函数
- 格式字符串函数
- LEFT 和 RIGHT 函数
- LENGTH 函数
- LOWER 函数
- LPAD 和 RPAD 函数
- LTRIM 函数
- POSITION 函数
- REGEXP_COUNT 函数
- REGEXP_INSTR 函数
- REGEXP_REPLACE 函数

- REGEXP_SUBSTR 函数
- REPEAT 函数
- REPLACE 函数
- REVERSE 函数
- RTRIM 函数
- 分割功能
- SPLIT_PART 函数
- SUBSTRING 函数
- TRANSLATE 函数
- TRIM 函数
- UPPER 函数
- UUID 函数

|| (串联) 运算符

联接位于 || 符号的任意一侧的两个表达式并返回联接后的表达式。

串联运算符类似于 CONCAT 函数。

> **Note**
>
> 对于 CONCAT 函数和联接运算符，如果一个或多个表达式为 null，则联接的结果也为 null。

语法

```
expression1 || expression2
```

参数

expression1、expression2

两个参数都可以是长度固定或长度可变的字符串或表达式。

返回类型

|| 运算符返回字符串。字符串的类型与输入参数的类型相同。

示例

以下示例将 USERS 表中的 FIRSTNAME 和 LASTNAME 字段联接：

```
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)
```

要联接可能包含 null 值的列，请使用 NVL 和 COALESCE 函数 表达式。以下示例使用 NVL 在遇到 NULL 时返回 0。

```
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 函数

BTRIM 函数通过删除前导空格和尾随空格或删除与可选的指定字符串匹配的前导字符和尾随字符来剪裁字符串。

语法

```
BTRIM(string [, trim_chars ] )
```

参数

string

要剪裁的输入 VARCHAR 字符串。

trim_chars

该 VARCHAR 字符串包含要匹配的字符。

返回类型

BTRIM 函数返回 VARCHAR 字符串。

示例

以下示例从字符串 ' abc ' 中剪裁前导和尾随空格：

```
select '     abc    ' as untrim, btrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

以下示例从字符串 'xyzaxyzbxyzcxyz' 中删除前导和尾随 'xyz' 字符串。将删除前导和尾随的 'xyz'，但不会删除字符串内部的匹配字符。

```
select 'xyzaxyzbxyzcxyz' as untrim,
```

```
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | axyzbxyzc
```

以下示例从字符串 'setuphistorycassettes' 中删除与 trim_chars 列表 'tes' 中的任何字符相匹配的开头和结尾部分。在输入字符串开头或结尾部分，在 trim_chars 列表中未包含的其他字符之前出现的任何 t、e 或 s 都将被删除。

```
SELECT btrim('setuphistorycassettes', 'tes');

     btrim
-----------------
 uphistoryca
```

CONCAT 函数

CONCAT 函数将联接两个表达式并返回生成的表达式。要联接两个以上的表达式，请使用嵌套 CONCAT 函数。在两个表达式之间使用联接运算符（||）将生成与 CONCAT 函数相同的结果。

> **Note**
>
> 对于 CONCAT 函数和联接运算符，如果一个或多个表达式为 null，则联接的结果也为 null。

语法

```
CONCAT ( expression1, expression2 )
```

参数

expression1、expression2

两个参数可以是固定长度字符串、可变长度字符串、二进制表达式或计算结果为其中一个输入的表达式。

返回类型

CONCAT 返回一个表达式。表达式的数据类型与输入参数的数据类型相同。

如果输入表达式的类型不同，则 AWS Clean Rooms 尝试对其中一个表达式进行隐式类型转换。如果值无法转换，则会返回一个错误。

示例

以下示例联接两个字符文本：

```
select concat('December 25, ', '2008');

concat
------------------
December 25, 2008
(1 row)
```

以下查询（使用 || 运算符而不是 CONCAT）将生成相同的结果：

```
select 'December 25, '||'2008';

concat
------------------
December 25, 2008
(1 row)
```

以下示例使用两个 CONCAT 函数联接三个字符串：

```
select concat('Thursday, ', concat('December 25, ', '2008'));

concat
----------------------------
Thursday, December 25, 2008
(1 row)
```

要联接可能包含 null 值的列，请使用 NVL 和 COALESCE 函数。以下示例使用 NVL 在遇到 NULL 时返回 0。

```
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)
```

以下查询联接 VENUE 表中的 CITY 和 STATE 值：

```
select concat(venuecity, venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
-------------------
DenverCO
Kansas CityMO
East RutherfordNJ
LandoverMD
(4 rows)
```

以下查询使用嵌套 CONCAT 函数。该查询将联接 VENUE 表中的 CITY 和 STATE 值，但会使用逗号和空格分隔生成的字符串：

```
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)
```

格式字符串函数

FORMAT_STRING 函数通过使用提供的参数替换模板字符串中的占位符来创建格式化字符串。它从 printf 样式的格式字符串中返回格式化字符串。

FORMAT_STRING 函数的工作原理是将模板字符串中的占位符替换为作为参数传递的相应值。当您需要动态构造包含静态文本和动态数据的字符串时，例如在生成输出消息、报告或其他类型的信息文本时，这种类型的字符串格式可能很有用。FORMAT_STRING 函数提供了一种简洁易读的方式来创建这些类型的格式化字符串，从而更易于维护和更新生成输出的代码。

语法

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

Arguments

strfmt

一个字符串表达式。

obj

字符串或数字表达式。

返回类型

FORMAT_STRING 返回一个字符串。

示例

以下示例包含包含两个占位符的模板字符串：%d十进制（整数）值和%s字符串值。%d占位符被十进制（整数）值 (100) 替换，%s 占位符替换为字符串值 ("days")。输出是一个模板字符串，占位符由提供的参数替换:"Hello World 100 days".

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

LEFT 和 RIGHT 函数

这些函数返回指定数量的位于字符串最左侧或最右侧的字符。

该数量基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。

语法

```
LEFT ( string,  integer )
```

```
RIGHT ( string,  integer )
```

参数

string

任何字符串或计算结果为字符串的任何表达式。

integer

一个正整数。

返回类型

LEFT 和 RIGHT 返回 VARCHAR 字符串。

示例

以下示例返回介于 1000 和 1005 IDs 之间的事件名称中最左边的 5 个和最右边的 5 个字符：

```
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 函数

LOWER 函数

将字符串转换为小写。LOWER 支持 UTF-8 多字节字符，并且每个字符最多可以有 4 个字节。

语法

```
LOWER(string)
```

参数

string

输入参数是 VARCHAR 字符串（或任何其他可隐式转换为 VARCHAR 的数据类型，如 CHAR）。

返回类型

LOWER 函数返回与输入字符串具有相同数据类型的字符串。

示例

以下示例将 CATNAME 字段转换为小写：

```
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 和 RPAD 函数

这些函数根据指定长度在字符串前面或后面追加字符。

语法

```
LPAD (string1, length, [ string2 ])
```

```
RPAD (string1, length, [ string2 ])
```

参数

string1

一个字符串或计算结果为字符串的表达式，如字符列的名称。

length

一个用于定义函数结果的长度的整数。字符串的长度基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。如果 string1 的长度超过指定长度，它将被截断（在右侧）。如果 length 为负数，函数的结果将为空字符串。

string2

追加到 string1 前面或后面的一个或多个字符。此参数是可选的；如果未指定它，则使用空格。

返回类型

这些函数返回 VARCHAR 数据类型。

示例

将指定的一组事件名称截断到 20 个字符并在短于此长度的名称前面追加空格：

```
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)
```

将指定的一组事件名称截断到 20 个字符但在短于此长度的名称后面追加 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 函数

从字符串的开头剪裁几个字符。删除只包含剪裁字符列表中的字符的最长字符串。当修剪字符未出现在输入字符串中时，修剪即告完成。

语法

```
LTRIM( string [, trim_chars] )
```

参数

string

要剪裁的字符串列、表达式或字符串文本。

trim_chars

表示要从 string 的开头剪裁的字符的字符串列、表达式或字符串文本。如果未指定，则使用空格作为剪裁字符。

返回类型

LTRIM 函数返回与输入字符串（CHAR 或 VARCHAR）具有相同数据类型的字符串。

示例

以下示例从 `listime` 列中剪裁掉年份。字符串文本中的剪裁字符 `'2008-'` 表示要从左侧剪裁的字符。如果您使用剪裁字符 `'028-'`，则会获得相同的结果。

```
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
```

当 trim_chars 中的任意字符出现在 string 的开头时，LTRIM 都会予以删除。以下示例从 VENUENAME（VARCHAR 列）的开头剪裁字符“C”、“D”和“G”。

```
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
```

以下示例使用从 venueid 列中检索到的剪裁字符 2。

```
select ltrim('2008-01-24 06:43:29', venueid)
from venue where venueid=2;

ltrim
------------------
```

```
008-01-24 06:43:29
```

以下示例不剪裁任何字符，因为在 '0' 剪裁字符之前找到了 2。

```
select ltrim('2008-01-24 06:43:29', '0');

ltrim
-------------------
2008-01-24 06:43:29
```

以下示例使用默认的空格剪裁字符，从字符串的开头剪裁掉两个空格。

```
select ltrim('  2008-01-24 06:43:29');

ltrim
-------------------
2008-01-24 06:43:29
```

POSITION 函数

返回指定子字符串在字符串中的位置。

语法

```
POSITION(substring IN string )
```

参数

substring

要在 string 中搜索的子字符串。

string

要搜索的字符串或列。

返回类型

POSITION 函数返回与子字符串的位置对应的整数（从 1 开始，而不是从 0 开始）。此位置基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。

使用说明

如果在字符串中未找到子字符串，POSITION 将返回 0：

```
select position('dog' in 'fish');

position
----------
 0
(1 row)
```

示例

以下示例显示字符串 fish 在单词 dogfish 中的位置：

```
select position('fish' in 'dogfish');

position
----------
 4
(1 row)
```

以下示例返回 SALES 表中 COMMISSION 超过 999.00 的销售交易的数量：

```
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 函数

在字符串中搜索正则表达式模式并返回指示该模式在字符串中出现的次数的整数。如果未找到匹配项，此函数将返回 0。

语法

```
REGEXP_COUNT ( source_string, pattern [, position [, parameters ] ] )
```

参数

source_string

要搜索的字符串表达式（如列名称）。

pattern

表示正则表达式模式的字符串文本。

position

指示在 source_string 中开始搜索的位置的正整数。此位置基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。默认值为 1。如果 position 小于 1，则搜索从 source_string 的第一个字符开始。如果 position 大于 source_string 中字符的数量，则结果为 0。

参数

一个或多个字符串，指示函数与模式的匹配方式。可能的值包括：

- c – 执行区分大小写的匹配。默认情况下，使用区分大小写的匹配。
- i – 执行不区分大小写的匹配。
- p – 使用 Perl 兼容正则表达式 (PCRE) 方言解释模式。

返回类型

整数

示例

以下示例计算三个字母序列出现的次数。

```
SELECT regexp_count('abcdefghijklmnopqrstuvwxyz', '[a-z]{3}');

 regexp_count
 --------------
            8
```

以下示例计算顶级域名为 org 或 edu 的次数。

```
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
```

下面的示例计算字符串 FOX 的出现次数，使用不区分大小写的匹配。

```
SELECT regexp_count('the fox', 'FOX', 1, 'i');

 regexp_count
 --------------
            1
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例使用区分大小写的匹配计算此类单词的出现次数。

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'p');

 regexp_count
 --------------
            2
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的含义。此示例计算此类单词的出现次数，但与前面的示例不同，因为它使用了不区分大小写的匹配。

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'ip');

 regexp_count
 --------------
            3
```

REGEXP_INSTR 函数

在字符串中搜索正则表达式模式并返回指示匹配子字符串的开始位置的整数。如果未找到匹配项，此函数将返回 0。REGEXP_INSTR 与 函数相似，只不过前者可让您在字符串中搜索正则表达式模式。

语法

```
REGEXP_INSTR ( source_string, pattern [, position [, occurrence] [, option
 [, parameters ] ] ] ] )
```

参数

source_string

要搜索的字符串表达式（如列名称）。

pattern

表示正则表达式模式的字符串文本。

position

指示在 source_string 中开始搜索的位置的正整数。此位置基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。默认值为 1。如果 position 小于 1，则搜索从 source_string 的第一个字符开始。如果 position 大于 source_string 中字符的数量，则结果为 0。

出现

一个正整数，指示要使用的模式的匹配项。REGEXP_INSTR 会跳过第一个 occurrence -1 匹配项。默认值为 1。如果 occurrence 小于 1 或大于 source_string 中的字符串，则会忽略搜索，并且结果为 0。

option

一个值，指示是否返回匹配项的第一个字符的位置 (0) ，或匹配项结尾后第一个字符的位置 (1)。非零值与 1 相同。默认值是 0。

参数

一个或多个字符串，指示函数与模式的匹配方式。可能的值包括：

- c – 执行区分大小写的匹配。默认情况下，使用区分大小写的匹配。
- i – 执行不区分大小写的匹配。
- e – 使用子表达式提取子字符串。

  如果 pattern 包含一个子表达式，REGEXP_INSTR 会使用 pattern 中的第一个子表达式来匹配子字符串。REGEXP_INSTR 仅考虑第一个子表达式；其他子表达式会被忽略。如果模式没有子表达式，REGEXP_INSTR 会忽略“e”参数。

- p – 使用 Perl 兼容正则表达式 (PCRE) 方言解释模式。

返回类型

整数

示例

以下示例搜索作为域名的开头的 @ 字符并返回第一个匹配项的开始位置。

```
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
```

以下示例搜索单词 Center 的变体并返回第一个匹配项的开始位置。

```
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
```

以下示例使用不区分大小写的匹配逻辑找到字符串 FOX 第一次出现的起始位置。

```
SELECT regexp_instr('the fox', 'FOX', 1, 1, 0, 'i');

 regexp_instr
 --------------
            5
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例查找第二个此类单词的起始位置。

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'p');

 regexp_instr
 --------------
           21
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。本示例查找第二个此类单词的起始位置，但与前面的示例不同，因为它使用了不区分大小写的匹配。

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'ip');

 regexp_instr
 --------------
           15
```

REGEXP_REPLACE 函数

在字符串中搜索正则表达式模式并将该模式的每个匹配项替换为指定字符串。REGEXP_REPLACE 与 REPLACE 函数相似，只不过前者可让您在字符串中搜索正则表达式模式。

REGEXP_REPLACE 与 TRANSLATE 函数和 REPLACE 函数相似，只不过 TRANSLATE 进行多次单字符替换，REPLACE 一次性将整个字符串替换为其他字符串，而 REGEXP_REPLACE 可让您在字符串中搜索正则表达式模式。

语法

```
REGEXP_REPLACE ( source_string, pattern [, replace_string [ , position [, parameters
 ] ] ] )
```

参数

source_string

要搜索的字符串表达式（如列名称）。

pattern

表示正则表达式模式的字符串文本。

replace_string

将替换模式的每个匹配项的字符串表达式（如列名称）。默认值是空字符串 ("")。

position

指示在 source_string 中开始搜索的位置的正整数。此位置基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。默认值为 1。如果 position 小于 1，则搜索从 source_string 的第一个字符开始。如果 position 大于 source_string 中的字符数量，则结果为 source_string。

参数

一个或多个字符串，指示函数与模式的匹配方式。可能的值包括：

- c – 执行区分大小写的匹配。默认情况下，使用区分大小写的匹配。
- i – 执行不区分大小写的匹配。
- p – 使用 Perl 兼容正则表达式 (PCRE) 方言解释模式。

返回类型

VARCHAR

如果 pattern 或 replace_string 为 NULL，则返回 NULL。

示例

以下示例删除电子邮件地址中的 @ 和域名。

```
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
```

以下示例将使用此值替换电子邮件地址的域名：`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
```

下面的示例使用不区分大小写的匹配替换值 quick brown fox 内的字符串 FOX 的所有出现次数。

```
SELECT regexp_replace('the fox', 'FOX', 'quick brown fox', 1, 'i');

   regexp_replace
---------------------
 the quick brown fox
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例将此单词的每次出现替换为值 [hidden]。

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'p');

        regexp_replace
-------------------------------
 [hidden] plain A1234 [hidden]
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例将此单词的每次出现替换为值 [hidden]，但与前面的示例不同，它使用不区分大小写的匹配。

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'ip');

          regexp_replace
---------------------------------
```

```
[hidden] plain [hidden] [hidden]
```

REGEXP_SUBSTR 函数

通过在字符串中搜索正则表达式模式，返回字符串中的字符。REGEXP_SUBSTR 与 SUBSTRING 函数 函数相似，只不过前者可让您在字符串中搜索正则表达式模式。如果函数无法将正则表达式与字符串中的任何字符匹配，则返回一个空字符串。

语法

```
REGEXP_SUBSTR ( source_string, pattern [, position [, occurrence [, parameters ] ] ] )
```

参数

source_string

要搜索的字符串表达式。

pattern

表示正则表达式模式的字符串文本。

position

指示在 source_string 中开始搜索的位置的正整数。此位置基于字符数而不是字节数，这是为了将多字节字符作为单字符计数。默认值为 1。如果 position 小于 1，则搜索从 source_string 的第一个字符开始。如果 position 大于 source_string 中的字符数量，则结果为空字符串 ("")。

出现

一个正整数，指示要使用的模式的匹配项。REGEXP_SUBSTR 会跳过第一个 occurrence -1 匹配项。默认值为 1。如果 occurrence 小于 1 或大于 source_string 中的字符串，则会忽略搜索，并且结果为 NULL。

参数

一个或多个字符串，指示函数与模式的匹配方式。可能的值包括：

- c – 执行区分大小写的匹配。默认情况下，使用区分大小写的匹配。
- i – 执行不区分大小写的匹配。
- e – 使用子表达式提取子字符串。

如果 pattern 包含一个子表达式，REGEXP_SUBSTR 会使用 pattern 中的第一个子表达式来匹配子字符串。子表达式是模式中用括号括起的表达式。例如，模式 `'This is a (\\w+)'` 将

第一个表达式与字符串 'This is a ' 后接一个字词进行匹配。此时不返回模式，带 e 参数的 REGEXP_SUBSTR 仅返回子表达式内的字符串。

REGEXP_SUBSTR 仅考虑第一个子表达式；其他子表达式会被忽略。如果模式没有子表达式，REGEXP_SUBSTR 会忽略“e”参数。

- p – 使用 Perl 兼容正则表达式 (PCRE) 方言解释模式。

返回类型

VARCHAR

示例

以下示例返回电子邮件地址中 @ 字符和域扩展名之间的部分。

```
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
```

以下示例使用不区分大小写的匹配返回与字符串 FOX 的第一次出现相对应的输入部分。

```
SELECT regexp_substr('the fox', 'FOX', 1, 1, 'i');

 regexp_substr
---------------
 fox
```

以下示例返回以小写字母开头的输入的第一部分。这在功能上与不带 c 参数的同一 SELECT 语句相同。

```
SELECT regexp_substr('THE SECRET CODE IS THE LOWERCASE PART OF 1931abc0EZ.', '[a-z]+',
 1, 1, 'c');
```

```
 regexp_substr
---------------
 abc
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例返回与第二个此类单词相对应的输入部分。

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'p');

 regexp_substr
---------------
 a1234
```

以下示例使用用 PCRE 方言编写的模式来定位至少包含一个数字和一个小写字母的单词。它使用 ?= 运算符，它在 PCRE 中具有特定的前瞻含义。此示例返回与第二个此类单词相对应的输入部分，但与前面的示例不同，因为它使用了不区分大小写的匹配。

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'ip');

 regexp_substr
---------------
 A1234
```

以下示例使用子表达式，通过不区分大小写的匹配来查找与模式 'this is a (\\w+)' 匹配的第二个字符串。它返回括号内的子表达式。

```
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 函数

将字符串重复指定的次数。如果输入参数为数字，REPEAT 会将其视为字符串。

语法

```
REPEAT(string, integer)
```

参数

string

第一个输入参数是要重复的字符串。

integer

第二个参数是指示字符串重复次数的整数。

返回类型

REPEAT 函数返回字符串。

示例

以下示例将重复 CATEGORY 表中 CATID 列的值三次：

```
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 函数

将现有字符串中一组字符的所有匹配项替换为其他指定字符。

REPLACE 与 TRANSLATE 函数和 REGEXP_REPLACE 函数相似，只不过 TRANSLATE 进行多次单字符替换，REGEXP_REPLACE 可让您在字符串中搜索正则表达式模式，而 REPLACE 一次性将整个字符串替换为其他字符串。

语法

```
REPLACE(string1, old_chars, new_chars)
```

参数

string

要搜索的 CHAR 或 VARCHAR 字符串

old_chars

要替换的 CHAR 或 VARCHAR 字符串。

new_chars

用于替换 old_string 的新 CHAR 或 VARCHAR 字符串。

返回类型

VARCHAR

如果 old_chars 或 new_chars 为 NULL，则将返回 NULL。

示例

以下示例将 CATGROUP 字段中的字符串 Shows 转换为 Theatre：

```
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 函数

REVERSE 函数对字符串运行并以反向顺序返回字符。例如，`reverse('abcde')` 将返回 edcba。此函数适用于数字和日期数据类型以及字符数据类型；但在大多数情况下，它对于字符串具有实用价值。

语法

```
REVERSE ( expression )
```

参数

expression

一个表达式，带有表示字符反转目标的字符、日期、时间戳或数字数据类型。所有表达式均可隐式转换为可变长度的字符串。将忽略定宽字符串中的尾随空格。

返回类型

REVERSE 返回 VARCHAR。

示例

从 USERS 表中选择 5 个不同的城市名称及其对应的反转名称：

```
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)
```

选择五笔销售额 IDs 及其相应的反向 IDs 转换作为字符串：

```
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 函数

RTRIM 函数从字符串的末尾剪裁指定的一组字符。删除只包含剪裁字符列表中的字符的最长字符串。当修剪字符未出现在输入字符串中时，修剪即告完成。

语法

```
RTRIM( string, trim_chars )
```

参数

string

要剪裁的字符串列、表达式或字符串文本。

trim_chars

表示要从 string 的结尾剪裁的字符的字符串列、表达式或字符串文本。如果未指定，则使用空格作为剪裁字符。

返回类型

与 string 参数具有相同的数据类型的字符串。

示例

以下示例从字符串 ' abc ' 中剪裁前导和尾随空格：

```
select '     abc    ' as untrim, rtrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    |    abc
```

以下示例从字符串 'xyzaxyzbxyzcxyz' 中删除尾随字符串 'xyz'。将删除尾随的 'xyz'，但不会删除字符串内部的匹配字符。

```
select 'xyzaxyzbxyzcxyz' as untrim,
rtrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | xyzaxyzbxyzc
```

以下示例从字符串 'setuphistorycassettes' 中删除与 trim_chars 列表 'tes' 中的任何字符相匹配的结尾部分。在输入字符串结尾部分，在 trim_chars 列表中未包含的其他字符之前出现的任何 t、e 或 s 都将被删除。

```
SELECT rtrim('setuphistorycassettes', 'tes');

     rtrim
-----------------
 setuphistoryca
```

以下示例从 VENUENAME 的末尾剪裁字符“Park”（如果有）：

```
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
```

请注意，当字符 P、a、r 或 k 中的任意一个出现在 VENUENAME 的末尾时，RTRIM 都会予以删除。

分割功能

SPLIT 函数允许您从较大的字符串中提取子字符串并将其作为数组处理。当您需要根据特定的分隔符或模式将字符串分解为单个组件时，SPLIT 函数非常有用。

语法

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

Arguments

str

要拆分的字符串表达式。

regex

表示正则表达式的字符串。正则表达式字符串应为 Java 正则表达式。

limit

一个整数表达式，用于控制应用正则表达式的次数。

- limit > 0：结果数组的长度不会超过限制，并且结果数组的最后一个条目将包含最后一个匹配的正则表达式之外的所有输入。
- limit <= 0：将尽可能多地应用正则表达式，并且生成的数组可以是任何大小。

返回类型

SPLIT 函数返回一个数组<STRING>。

If`limit > 0`：结果数组的长度不会超过限制，并且结果数组的最后一个条目将包含最后一个匹配的正则表达式之外的所有输入。

If`limit <= 0`: regex 将尽可能多地被应用，并且生成的数组可以是任何大小。

示例

在此示例中，SPLIT 函数会在遇到字符'A''B'、或'C'（由正则表达式模式指定'[ABC]'）'oneAtwoBthreeC'的任何地方拆分输入字符串。结果输出是一个由四个元素组成的数组："one"、"two""three"、和一个空字符串""。

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

SPLIT_PART 函数

用指定的分隔符拆分字符串，并返回指定位置的部分内容。

语法

```
SPLIT_PART(string, delimiter, position)
```

参数

string

要拆分的字符串列、表达式或字符串文本。字符串可以是 CHAR 或 VARCHAR。

分隔符

分隔符字符串指示输入 string 的部分。

如果 delimiter 是文本，则将其括在单引号中。

position

要返回的 string 部分的位置（从 1 算起）。必须是大于 0 的整数。如果 position 大于字符串部分的数量，SPLIT_PART 将返回空字符串。如果在字符串中未找到分隔符，则返回的值包含指定部分的内容，它可能是整个字符串或一个空值。

返回类型

CHAR 或 VARCHAR 字符串，与 string 参数相同。

示例

以下示例使用 $ 分隔符，将字符串文本拆分为多个部分，并返回第二部分。

```
select split_part('abc$def$ghi','$',2)

split_part
----------
def
```

以下示例使用 $ 分隔符，将字符串文本拆分为多个部分。它返回一个空字符串，因为找不到部分 4。

```
select split_part('abc$def$ghi','$',4)

split_part
----------

```

以下示例使用 # 分隔符，将字符串文本拆分为多个部分。它返回整个字符串，也就是第一部分，因为找不到分隔符。

```
select split_part('abc$def$ghi','#',1)

split_part
------------
abc$def$ghi
```

以下示例将时间戳字段 LISTTIME 拆分为年、月和日组成部分。

```
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
```

以下示例选择 LISTTIME 时间戳字段并在 '-' 字符处拆分它以获取月（LISTTIME 字符串的第二部分），然后计算每个月的条目数：

```
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 函数

按指定的开始位置返回子字符串子集。

如果输入的是字符串，字符的开始位置和数量基于字符数而不是字节数，这是为了将多字节字符作为单个字符计数。如果输入的是二进制表达式，则开始位置和提取的子字符串基于字节。您无法指定负长度，但可指定负开始位置。

语法

```
SUBSTRING(charactestring FROM start_position [ FOR numbecharacters ] )
```

```
SUBSTRING(charactestring, start_position, numbecharacters )
```

```
SUBSTRING(binary_expression, start_byte, numbebytes )
```

```
SUBSTRING(binary_expression, start_byte )
```

参数

字符串

要搜索的字符串。非字符数据类型将视为字符串。

start_position

字符串中开始提取的位置，从 1 开始。start_position 基于字符数而不是字节数，这是为了将多字节字符作为单个字符计数。此数字可以为负。

数字字符

要提取的字符的数量（子字符串的长度）。nu mbecharacters 基于字符数，而不是字节数，因此多字节字符计为单个字符。此数字不能为负。

start_byte

二进制表达式中开始提取的位置，从 1 开始。此数字可以为负。

数字字节

要提取的字节的数量（子字符串的长度）。此数字不能为负。

返回类型

VARCHAR

字符串的使用说明

以下示例返回以第 6 个字符开头的 4 字符字符串。

```
select substring('caterpillar',6,4);
substring
-----------
pill
(1 row)
```

如果 start_position + num becharacters 超过字符串的长度，则 SUBSTRING 会返回一个从起始位置开始直到字符串结尾的子字符串。例如：

```
select substring('caterpillar',6,8);
substring
-----------
```

```
pillar
(1 row)
```

如果 start_position 为负或 0，SUBSTRING 函数将返回从长度为 start_position + numbecharacters -1 的字符串的第一个字符开始的子字符串。例如：

```
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)
```

如果 start_position + numbecharacters -1 小于或等于零，SUBSTRING 将返回空字符串。例如：

```
select substring('caterpillar',-5,4);
substring
-----------

(1 row)
```

示例

以下示例返回 LISTING 表的 LISTTIME 字符串中的月份：

```
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)
```

以下示例与上述示例相同，但使用 FROM...FOR 选项：

```
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)
```

您无法使用 SUBSTRING 以可预测的方式提取可能包含多字节字符的字符串的前缀，因为您需要根据字节数（而不是字符数）指定多字节字符串的长度。要基于以字节为单位的长度提取字符串的开始部分，您可将字符串强制转换为 VARCHAR(byte_length) 以截断字符串，其中 byte_length 是必需长度。以下示例提取字符串 `'Fourscore and seven'` 的前 5 个字节。

```
select cast('Fourscore and seven' as varchar(5));

varchar
-------
Fours
```

以下示例返回出现在输入字符串 `Silva, Ana` 中最后一个空格之后的名字 `Ana`。

```
select reverse(substring(reverse('Silva, Ana'), 1, position(' ' IN reverse('Silva,
 Ana'))))
```

```
 reverse
-----------
 Ana
```

TRANSLATE 函数

对于给定表达式，将指定字符的所有匹配项替换为指定替代项。现有字符将按其在 characters_to_replace 和 characters_to_substitute 参数中的位置映射到替换字符。如果在 characters_to_replace 参数中指定的字符多于在 characters_to_substitute 参数中指定的字符，返回值中将省略 characters_to_replace 参数中的额外字符。

TRANSLATE 与 REPLACE 函数和 REGEXP_REPLACE 函数相似，只不过 REPLACE 将整个字符串替换为其他字符串，REGEXP_REPLACE 可让您在字符串中搜索正则表达式模式，而 TRANSLATE 进行多次单字符替换。

如果任何参数为空，则返回 NULL。

语法

```
TRANSLATE ( expression, characters_to_replace, characters_to_substitute )
```

参数

expression

要转换的表达式。

characters_to_replace

一个包含要替换的字符的字符串。

characters_to_substitute

一个字符串，其中包含要替换其他字符的字符。

返回类型

VARCHAR

示例

以下示例将替换字符串中的多个字符：

```
select translate('mint tea', 'inea', 'osin');

translate
-----------
most tin
```

以下示例将列中的所有值的 at (@) 符号替换为句点：

```
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
```

以下示例将空格替换为下划线并去掉列中的所有值的句点：

```
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 函数

通过删除前导空格和尾随空格或删除与可选的指定字符串匹配的前导字符和尾随字符来剪裁字符串。

语法

```
TRIM( [ BOTH ] [ trim_chars FROM ] string
```

参数

trim_chars

（可选）要从字符串剪裁的字符数。如果忽略此参数，则剪裁空白区域。

string

要剪裁的字符串。

返回类型

TRIM 函数返回 VARCHAR 或 CHAR 字符串。如果您将 TRIM 函数与 SQL 命令一起使用，则会将结果 AWS Clean Rooms 隐式转换为 VARCHAR。如果您将 SELECT 列表中的 TRIM 函数用于 SQL 函数，则 AWS Clean Rooms 不会隐式转换结果，并且可能需要执行显式转换以避免出现数据类型不匹配错误。有关显式转换的信息，请参阅CAST 函数函数。

示例

以下示例从字符串 ' abc ' 中剪裁前导和尾随空格：

```
select '     abc    ' as untrim, trim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

以下示例将删除将字符串 "dog" 括起的双引号：

```
select trim('"' FROM '"dog"');

btrim
-------
dog
```

当 trim_chars 中的任意字符出现在 string 的开头时，TRIM 都会予以删除。以下示例从 VENUENAME（VARCHAR 列）的开头剪裁字符“C”、“D”和“G”。

```
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 函数

将字符串转换为大写。UPPER 支持 UTF-8 多字节字符，并且每个字符最多可以有 4 个字节。

语法

```
UPPER(string)
```

参数

string

输入参数是 VARCHAR 字符串（或任何其他可隐式转换为 VARCHAR 的数据类型，如 CHAR）。

返回类型

UPPER 函数返回与输入字符串具有相同数据类型的字符串。

示例

以下示例将 CATNAME 字段转换为大写：

```
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 函数

UUID 函数生成通用唯一标识符 (UUID)。

UUIDs 是全球唯一标识符，通常用于为各种目的提供唯一标识符，例如：

- 识别数据库记录或其他数据实体。
- 为文件、目录或其他资源生成唯一的名称或密钥。
- 跨分布式系统跟踪和关联数据。
- 为网络数据包、软件组件或其他数字资产提供唯一标识符。

UUID 函数生成的 UUID 值是唯一的，而且概率非常高，即使在分布式系统中和长时间内也是如此。UUIDs 通常使用当前时间戳、计算机的网络地址以及其他随机或伪随机数据的组合生成，从而确保生成的每个 UUID 极不可能与任何其他 UUID 发生冲突。

在 SQL 查询的上下文中，UUID 函数可用于为插入到数据库中的新记录生成唯一标识符，或者为数据分区、索引或其他需要唯一标识符的目的提供唯一密钥。

> **Note**
>
> UUID 函数是不确定的。

语法

```
uuid()
```

Arguments

UUID 函数不带任何参数。

返回类型

UUID 返回通用唯一标识符 (UUID) 字符串。该值以规范 UUID 的 36 个字符的字符串形式返回。

示例

以下示例生成通用唯一标识符 (UUID)。输出是一个 36 个字符的字符串，表示通用唯一标识符。

```
SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

与隐私相关的功能

AWS Clean Rooms 提供的功能可帮助您遵守以下规范的隐私相关合规性。

- 全球隐私平台 (GPP) — 互动广告局 (IAB) 的一项规范，它为在线隐私和数据使用建立了全球标准化框架。有关 GPP 技术规范的更多信息，请参阅上的 GitHub全球隐私平台文档。
- 透明度和同意框架（TCF）——GPP的关键组成部分，于2020年推出，它提供了一个标准化的技术框架，以帮助公司遵守欧盟《通用数据保护条例》（GDPR）等隐私法规。TCF 使客户能够同意或拒绝同意收集和处理数据。有关 TCF 技术规格的更多信息，请参阅上的 TCF 文档。 GitHub

主题

consent_gpp_v1_decode 函数

该consent_gpp_v1_decode函数用于解码全球隐私平台 (GPP) v1 同意数据。它将编码后的同意字符串作为输入，并返回解码后的同意数据，其中包括有关用户隐私偏好和同意选择的信息。此功能在处理包含 GPP v1 用户意见征求信息的数据时非常有用，因为它允许您以结构化格式访问和分析同意数据。

语法

```
consent_gpp_v1_decode(gpp_string)
```

参数

gpp_string

编码后的 GPP v1 同意字符串。

返回值

返回的字典包括以下键值对：

- version：使用的 GPP 规范版本（当前为 1）。
- cmpId：对同意字符串进行编码的同意管理平台 (CMP) 的 ID。
- cmpVersion：对同意字符串进行编码的 CMP 版本。
- consentScreen：CMP 用户界面中用户提供同意的屏幕的 ID。
- consentLanguage: 同意信息的语言代码。
- vendorListVersion：使用的供应商列表版本。
- publisherCountryCode：出版商的国家/地区代码。
- purposeConsent：代表用户同意的目的的整数列表。
- purposeLegitimateInterest： IDs 以透明方式传达用户合法利益的目的清单。

- specialFeatureOptIns：代表用户选择使用的特殊功能的整数列表。
- vendorConsent：用户已 IDs 同意的供应商列表。
- vendorLegitimateInterest：已透明地传达用户合法利益的供应商 IDs 名单。

示例

以下示例采用单个参数，即编码后的同意字符串。它会返回一本包含解码后的同意数据的字典，包括有关用户隐私偏好、同意选择和其他元数据的信息。

```
SELECT * FROM consent_gpp_v1_decode('ABCDEFGHIJK');
```

返回的同意数据的基本结构包括有关同意字符串版本、CMP（同意管理平台）详细信息、用户出于不同目的和供应商的同意和合法利益选择以及其他元数据的信息。

```
{
    "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 函数

该consent_tcf_v2_decode函数用于解码透明度和同意框架 (TCF) v2 同意数据。它将编码后的同意字符串作为输入，并返回解码后的同意数据，其中包括有关用户隐私偏好和同意选择的信息。此功能在处理包含 TCF v2 用户意见征求信息的数据时非常有用，因为它允许您以结构化格式访问和分析同意数据。

语法

```
consent_tcf_v2_decode(tcf_string)
```

参数

tcf_string

编码后的 TCF v2 同意字符串。

返回值

该consent_tcf_v2_decode函数返回一个字典，其中包含来自透明度和同意框架 (TCF) v2 同意字符串的已解码同意数据。

返回的字典包括以下键值对：

核心细分市场

- version：使用的 TCF 规范版本（当前为 2）。
- created：用户意见征求字符串的创建日期和时间。
- lastUpdated：用户意见征求字符串上次更新的日期和时间。
- cmpId：对同意字符串进行编码的同意管理平台 (CMP) 的 ID。
- cmpVersion：对同意字符串进行编码的 CMP 版本。
- consentScreen：CMP 用户界面中用户提供同意的屏幕的 ID。
- consentLanguage: 同意信息的语言代码。
- vendorListVersion：使用的供应商列表版本。
- tcfPolicyVersion：用户意见征求字符串所基于的 TCF 政策版本。
- isServiceSpecific：一个布尔值，表示同意是针对特定服务还是适用于所有服务。
- useNonStandardStacks：表示是否使用非标准堆栈的布尔值。
- specialFeatureOptIns：代表用户选择使用的特殊功能的整数列表。
- purposeConsent：代表用户同意的目的的整数列表。
- purposesLITransparency：代表用户提供合法利益透明度的目的的整数列表。
- purposeOneTreatment：一个布尔值，表示用户是否请求了“目的为一的处理”（也就是说，所有目的都一视同仁）。
- publisherCountryCode：出版商的国家/地区代码。
- vendorConsent：用户已 IDs 同意的供应商列表。
- vendorLegitimateInterest：已透明地传达用户合法利益的供应商 IDs 名单。

- pubRestrictionEntry：发布者限制列表。此字段包含目的 ID、限制类型和 IDs 受该目的限制的供应商列表。

已披露的供应商细分

- disclosedVendors：代表已向用户披露的供应商的整数列表。

出版商目的细分

- pubPurposesConsent：整数列表，代表用户已同意的发布商特定用途。
- pubPurposesLITransparency：代表出版商特定目的的整数列表，用户为之提供合法利益透明度。
- customPurposesConsent：代表用户同意的自定义目的的整数列表。
- customPurposesLITransparency：一个整数列表，代表用户为其提供合法利益透明度的自定义目的。

这些详细的同意数据可用于了解和尊重用户在处理个人数据时的隐私偏好。

示例

以下示例采用单个参数，即编码后的同意字符串。它会返回一本包含解码后的同意数据的字典，包括有关用户隐私偏好、同意选择和其他元数据的信息。

```
from aws_clean_rooms.functions import consent_tcf_v2_decode

consent_string = "CO1234567890abcdef"
consent_data = consent_tcf_v2_decode(consent_string)

print(consent_data)
```

返回的同意数据的基本结构包括有关同意字符串版本、CMP（同意管理平台）详细信息、用户出于不同目的和供应商的同意和合法利益选择以及其他元数据的信息。

```
    /** 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],
};
```

窗口函数

通过使用窗口函数，您可以更高效地创建分析业务查询。窗口函数运行于分区或结果集的“窗口”上，并为该窗口中的每个行返回一个值。相比之下，非窗口函数执行与结果集中的每个行相关的计算。与聚合结果行的分组函数不同，窗口函数在表的表达式中的保留所有行。

使用该窗口中的行集中的值计算返回的值。对于表中的每一行，窗口定义一组用于计算其他属性的行。窗口使用窗口规范（OVER 子句）进行定义并基于以下三个主要概念：

- 窗口分区，构成了行组（PARTITION 子句）
- 窗口排序，定义了每个分区中行的顺序或序列（ORDER BY 子句）
- 窗口框架，相对于每个行进行定义以进一步限制行集（ROWS 规范）

窗口函数是在查询中执行的最后一组操作（最后的 ORDER BY 子句除外）。所有联接和所有 WHERE、GROUP BY 和 HAVING 子句均在处理窗口函数前完成。因此，窗口函数只能显示在选择列表或 ORDER BY 子句中。您可以在一个具有不同框架子句的查询中使用多个窗口函数。您还可以在其他标量表达式（如 CASE）中使用窗口函数。

窗口函数语法摘要

窗口函数遵循标准语法，如下所示。

```
function (expression) OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list [ frame_clause ] ] )
```

其中，function 是本部分介绍的函数之一。

expr_list 如下所示。

```
expression | column_name [, expr_list ]
```

order_list 如下所示。

```
expression | column_name [ ASC | DESC ]
[ NULLS FIRST | NULLS LAST ]
[, order_list ]
```

frame_clause 如下所示。

```
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 }}
```

参数

函数

窗口函数。有关详细信息，请参阅各个函数描述。

OVER

定义窗口规范的子句。OVER 子句是窗口函数必需的，并可区分窗口函数与其他 SQL 函数。

PARTITION BY expr_list

（可选）PARTITION BY 子句将结果集细分为分区，与 GROUP BY 子句很类似。如果存在分区子句，则为每个分区中的行计算该函数。如果未指定任何分区子句，则一个分区包含整个表，并为整个表计算该函数。

排名函数 DENSE_RANK、NTILE、RANK 和 ROW_NUMBER 需要全局比较结果集中的所有行。使用 PARTITION BY 子句时，查询优化程序可通过根据分区跨多个切片分布工作负载来并行运行每个聚合。如果不存在 PARTITION BY 子句，则必须在一个切片上按顺序运行聚合步骤，这可能对性能产生显著的负面影响，特别是对于大型集群。

AWS Clean Rooms不支持 PARTITION BY 子句中的字符串文字。

ORDER BY order_list

（可选）窗口函数将应用于每个分区中根据 ORDER BY 中的顺序规范排序的行。此 ORDER BY 子句与 frame_clause 中的 ORDER BY 子句不同且完全不相关。ORDER BY 子句可在没有 PARTITION BY 子句的情况下使用。

对于排名函数，ORDER BY 子句确定排名值的度量。对于聚合函数，分区的行必须在为每个框架计算聚合函数之前进行排序。有关窗口函数的更多信息，请参阅 窗口函数。

顺序列表中需要列标识符或计算结果为列标识符的表达式。常数和常数表达式都不可用作列名称的替代。

NULLS 值将被视为其自己的组，并根据 NULLS FIRST 或 NULLS LAST 选项进行排序和排名。默认情况下，按 ASC 顺序最后对 NULL 值进行排序和排名，按 DESC 顺序首先对 NULL 值进行排序和排名。

AWS Clean Rooms在 ORDER BY 子句中不支持字符串文字。

如果省略 ORDER BY 子句，则行的顺序是不确定的。

> **Note**
>
> 在任何并行系统中AWS Clean Rooms，例如，当 ORDER BY 子句不生成数据的唯一和总体顺序时，行的顺序是不确定的。也就是说，如果 ORDER BY 表达式生成重复的值（部分排序），则这些行的返回顺序可能因运行而异。AWS Clean Rooms反过来，窗口函数可能返回意外的或不一致的结果。有关更多信息，请参阅 窗口函数的唯一数据排序。

column_name

执行分区或排序操作所依据的列的名称。

ASC | DESC

一个定义表达式的排序顺序的选项，如下所示：

- ASC：升序（例如，按数值的从低到高的顺序和字符串的从 A 到 Z 的顺序）。如果未指定选项，则默认情况下将按升序对数据进行排序。
- DESC：降序（按数值的从高到低的顺序和字符串的从 Z 到 A 的顺序）。

NULLS FIRST | NULLS LAST

指定是应首先对 NULL 值进行排序（非 null 值之前）还是最后对 NULL 值进行排序（非 null 值之后）的选项。默认情况下，按 ASC 顺序最后对 NULLS 进行排序和排名，按 DESC 顺序首先对 NULLS 进行排序和排名。

frame_clause

对于聚合函数，框架子句在使用 ORDER BY 时进一步优化函数窗口中的行集。它使您可以包含或排除已排序结果中的行集。框架子句包括 ROWS 关键字和关联的说明符。

frame 子句不适用于排名函数。同时，在聚合函数的 OVER 子句中未使用 ORDER BY 子句时不需要框架子句。如果 ORDER BY 子句用于聚合函数，则需要显式框架子句。

未指定 ORDER BY 子句时，隐式框架是无界的：等同于 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。

ROWS

此子句通过从当前行中指定物理偏移来定义窗口框架。

此子句指定当前行中的值将并入的当前窗口或分区中的行。它使用指定行位置的参数，行位置可位于当前行之前或之后。所有窗口框架的参考点为当前行。当窗口框架向前滑向分区中时，每个行会依次成为当前行。

框架可以是一组超过并包括当前行的行。

```
{UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW}
```

或者可以是两个边界之间的一组行。

```
BETWEEN
{ UNBOUNDED PRECEDING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
```

```
AND
{ UNBOUNDED FOLLOWING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
```

UNBOUNDED PRECEDING 指示窗口从分区的第一行开始；offset PRECEDING 指示窗口开始于等同于当前行之前的偏移值的行数。UNBOUNDED PRECEDING 是默认值。

CURRENT ROW 指示窗口在当前行开始或结束。

UNBOUNDED FOLLOWING 指示窗口在分区的最后一行结束；offset FOLLOWING 指示窗口结束于等同于当前行之后的偏移值的行数。

offset 标识当前行之前或之后的实际行数。在这种情况下，offset 必须为计算结果为正数值的常数。例如，5 FOLLOWING 将在当前行之后的第 5 行结束框架。

其中，未指定 BETWEEN，框架受当前行隐式限制。例如，ROWS 5 PRECEDING 等于 ROWS BETWEEN 5 PRECEDING AND CURRENT ROW。同时，ROWS UNBOUNDED FOLLOWING 等于 ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING。

> **Note**
>
> 您无法指定起始边界大于结束边界的框架。例如，您无法指定以下任一框架。
>
> ```
> between 5 following and 5 preceding
> between current row and 2 preceding
> between 3 following and current row
> ```

窗口函数的唯一数据排序

如果窗口函数的 ORDER BY 子句不生成数据的唯一排序和总排序，则行的顺序是不确定的。如果 ORDER BY 表达式生成重复的值（部分排序），则这些行的返回顺序可能会在多次运行中有所不同。在这种情况下，窗口函数还可能返回意外的或不一致的结果。

例如，以下查询在多次运行中返回了不同的结果。出现这些不同的结果是因为 order by dateid 未生成 SUM 窗口函数的数据的唯一排序。

```
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
...
```

在这种情况下，向该窗口函数添加另一个 ORDER BY 列可解决此问题。

```
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
...
```

支持的函数

AWS Clean RoomsSpark SQL 支持两种类型的窗口函数：聚合和排名。

以下是支持的聚合函数：

- CUME_DIST 开窗函数
- DENSE_RANK 窗口函数

- 第一个窗口功能
- FIRST_VALUE 窗口函数
- LAG 窗口函数
- 最后一个窗口函数
- LAST_VALUE 窗口函数
- LEAD 窗口函数

以下是支持的排名函数：

- DENSE_RANK 窗口函数
- PERCENT_RANK 开窗函数
- RANK 窗口函数
- ROW_NUMBER 窗口函数

窗口函数示例的示例表

您可以通过每个函数描述找到特定的窗口函数示例。其中一些示例使用一个名为 WINSALES 的表，该表包含 11 行，如下表所示。

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

SALESID	DATEID	SELLERID	BUYERID	QTY	QTY_SHIPPED
30003	4/18/2004	3	B	15	
30004	4/18/2004	3	B	20	
30007	9/7/2004	3	C	30	

CUME_DIST 开窗函数

计算某个窗口或分区中某个值的累积分布。假定升序排序，则使用以下公式确定累积分布：

```
count of rows with values <= x / count of rows in the window or partition
```

其中，x 等于 ORDER BY 子句中指定的列的当前行中的值。以下数据集说明了此公式的使用：

```
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
```

返回值范围介于 0 和 1（含 1）之间。

语法

```
CUME_DIST ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

参数

OVER

一个指定窗口分区的子句。OVER 子句不能包含窗口框架规范。

PARTITION BY partition_expression

可选。一个设置 OVER 子句中每个组的记录范围的表达式。

ORDER BY order_list

用于计算累积分布的表达式。该表达式必须具有数字数据类型或可隐式转换为 1。如果省略 ORDER BY，则所有行的返回值为 1。

如果 ORDER BY 未生成唯一顺序，则行的顺序是不确定的。有关更多信息，请参阅 窗口函数的唯一数据排序。

返回类型

FLOAT8

示例

以下示例计算每个卖家的销量的累积分布：

```
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
```

有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

DENSE_RANK 窗口函数

DENSE_RANK 窗口函数基于 OVER 子句中的 ORDER BY 表达式确定一组值中的一个值的排名。如果存在可选的 PARTITION BY 子句，则为每个行组重置排名。带符合排名标准的相同值的行接收相同的排名。DENSE_RANK 函数与 RANK 存在以下一点不同：如果两个或两个以上的行结合，则一系列排名的值之间没有间隔。例如，如果两个行的排名为 1，则下一个排名则为 2。

您可以在同一查询中包含带有不同的 PARTITION BY 和 ORDER BY 子句的排名函数。

语法

```
DENSE_RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

参数

( )

该函数没有参数，但需要空括号。

OVER

适用于 DENSE_RANK 函数的窗口子句。

PARTITION BY expr_list

可选。一个或多个定义窗口的表达式。

ORDER BY order_list

可选。排名值基于的表达式。如果未指定 PARTITION BY，则 ORDER BY 使用整个表。如果省略 ORDER BY，则所有行的返回值为 1。

如果 ORDER BY 未生成唯一顺序，则行的顺序是不确定的。有关更多信息，请参阅 窗口函数的唯一数据排序。

返回类型

INTEGER

示例

以下示例按销量对表进行排序（按降序顺序），并将紧密排名和常规排名分配给每个行。在应用窗口函数结果后，对结果进行排序。

```
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)
```

在同一查询中一起使用 DENSE_RANK 和 RANK 函数时，记下已分配给同一组行的排名的差异。有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

以下示例按 SELLERID 对表进行分区，按数量对每个分区进行排序（按降序顺序），并为每个行分配紧密排名。在应用窗口函数结果后，对结果进行排序。

```
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)
```

有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

第一个窗口功能

给定一组有序的行，FIRST 返回指定表达式相对于窗口框架中第一行的值。

有关选择框架中最后一行的信息，请参阅 最后一个窗口函数。

语法

```
FIRST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

参数

expression

对其执行函数的目标列或表达式。

IGNORE NULLS

当此选项与 FIRST 一起使用时，该函数返回框架中第一个不是 NULL 的值（如果所有值都为 NULL，则返回 NULL）。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

引入函数的窗口子句。

PARTITION BY expr_list

依据一个或多个表达式定义函数的窗口。

ORDER BY order_list

对每个分区中的行进行排序。如果未指定 PARTITION BY 子句，则 ORDER BY 对整个表进行排序。如果指定 ORDER BY 子句，则还必须指定 frame_clause。

FIRST 函数的结果取决于数据的排序。在以下情况下，结果是不确定的：

- 当未指定 ORDER BY 子句且一个分区包含一个表达式的两个不同的值时
- 当表达式的计算结果为对应于 ORDER BY 列表中同一值的不同值时。

frame_clause

如果 ORDER BY 子句用于聚合函数，则需要显式框架子句。框架子句优化函数窗口中的行集，包含或排除已排序结果中的行集。框架子句包括 ROWS 关键字和关联的说明符。请参阅窗口函数语法摘要。

返回类型

这些函数支持使用原始AWS Clean Rooms数据类型的表达式。返回类型与 expression 的数据类型相同。

示例

以下示例返回 VENUE 表中每个场地的座位数，同时按容量对结果进行排序（从高到低）。FIRST 函数用于选择与框架中第一行相对应的场地名称：在本例中为座位数最多的那一排。按州对结果进行分区，以便当 VENUESTATE 值发生更改时，会选择一个新的第一个值。窗口框架是无界的，因此为每个分区中的每个行选择相同的第一个值。

对于加利福利亚，Qualcomm Stadium 具有最大座位数 (70561)，此名称是 CA 分区中所有行的第一个值。

```
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 窗口函数

在提供一组已排序行的情况下，FIRST_VALUE 返回有关窗口框架中的第一行的指定表达式的值。

有关选择框架中最后一行的信息，请参阅 LAST_VALUE 窗口函数。

语法

```
FIRST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

参数

expression

对其执行函数的目标列或表达式。

IGNORE NULLS

将此选项与 FIRST_VALUE 结合使用时，该函数返回不为 NULL 的框架中的第一个值（如果所有值为 NULL，则返回 NULL）。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

引入函数的窗口子句。

PARTITION BY expr_list

依据一个或多个表达式定义函数的窗口。

ORDER BY order_list

对每个分区中的行进行排序。如果未指定 PARTITION BY 子句，则 ORDER BY 对整个表进行排序。如果指定 ORDER BY 子句，则还必须指定 frame_clause。

FIRST_VALUE 函数的结果取决于数据的排序。在以下情况下，结果是不确定的：

- 当未指定 ORDER BY 子句且一个分区包含一个表达式的两个不同的值时
- 当表达式的计算结果为对应于 ORDER BY 列表中同一值的不同值时。

frame_clause

如果 ORDER BY 子句用于聚合函数，则需要显式框架子句。框架子句优化函数窗口中的行集，包含或排除已排序结果中的行集。框架子句包括 ROWS 关键字和关联的说明符。请参阅窗口函数语法摘要。

返回类型

这些函数支持使用原始AWS Clean Rooms数据类型的表达式。返回类型与 expression 的数据类型相同。

示例

以下示例返回 VENUE 表中每个场地的座位数，同时按容量对结果进行排序（从高到低）。FIRST_VALUE 函数用于选择与框架中的第一行对应的场地的名称：在这种情况下，为座位数最多的行。按州对结果进行分区，以便当 VENUESTATE 值发生更改时，会选择一个新的第一个值。窗口框架是无界的，因此为每个分区中的每个行选择相同的第一个值。

对于加利福利亚，Qualcomm Stadium 具有最大座位数 (70561)，此名称是 CA 分区中所有行的第一个值。

```
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 窗口函数

LAG 窗口函数返回位于分区中当前行的上方（之前）的某个给定偏移量位置的行的值。

语法

```
LAG (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

参数

value_expr

对其执行函数的目标列或表达式。

offset

一个可选参数，该参数指定要返回其值的当前行前面的行数。偏移量可以是常量整数或计算结果为整数的表达式。如果未指定偏移量，则AWS Clean Rooms使用1作为默认值。偏移量为 0 表示当前行。

IGNORE NULLS

一种可选规范，用于指示在确定要使用哪一行时AWS Clean Rooms应跳过空值。如果未列出 IGNORE NULLS，则包含 Null 值。

> **Note**
>
> 您可以使用 NVL 或 COALESCE 表达式将 null 值替换为另一个值。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

指定窗口分区和排序。OVER 子句不能包含窗口框架规范。

PARTITION BY window_partition

一个可选参数，该参数设置 OVER 子句中每个组的记录范围。

ORDER BY window_ordering

对每个分区中的行进行排序。

LAG 窗口函数支持使用任何AWS Clean Rooms数据类型的表达式。返回类型与 value_expr 的类型相同。

示例

以下示例显示已售给买家 ID 为 3 的买家的票数以及买家 3 的购票时间。要将每个销售与买家 3 的上一销售进行比较，查询要返回每个销售的上一销量。由于 1/16/2008 之前未进行购买，则第一个上一销量值为 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)
```

最后一个窗口函数

给定一组有序的行，LAST 函数返回相对于帧中最后一行的表达式值。

有关选择框架中第一行的信息，请参阅 第一个窗口功能。

语法

```
LAST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

参数

expression

对其执行函数的目标列或表达式。

IGNORE NULLS

该函数返回不为 NULL 的框架中的最后一个值（如果所有值为 NULL，则返回 NULL）。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

引入函数的窗口子句。

PARTITION BY expr_list

依据一个或多个表达式定义函数的窗口。

ORDER BY order_list

对每个分区中的行进行排序。如果未指定 PARTITION BY 子句，则 ORDER BY 对整个表进行排序。如果指定 ORDER BY 子句，则还必须指定 frame_clause。

结果取决于数据的排序。在以下情况下，结果是不确定的：

- 当未指定 ORDER BY 子句且一个分区包含一个表达式的两个不同的值时
- 当表达式的计算结果为对应于 ORDER BY 列表中同一值的不同值时。

frame_clause

如果 ORDER BY 子句用于聚合函数，则需要显式框架子句。框架子句优化函数窗口中的行集，包含或排除已排序结果中的行集。框架子句包括 ROWS 关键字和关联的说明符。请参阅窗口函数语法摘要。

返回类型

这些函数支持使用原始AWS Clean Rooms数据类型的表达式。返回类型与 expression 的数据类型相同。

示例

以下示例返回 VENUE 表中每个场地的座位数，同时按容量对结果进行排序（从高到低）。LAST 函数用于选择与框架中最后一行相对应的场地名称：在本例中为座位数最少的那一行。按州对结果进行分区，以便当 VENUESTATE 值发生更改时，会选择一个新的最后一个值。窗口框架是无界的，因此为每个分区中的每个行选择相同的最后一个值。

对于加利福利亚，为该分区中的每个行返回 `Shoreline Amphitheatre`，因为它具有最小座位数 (`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 窗口函数

在提供一组已排序行的情况下，LAST_VALUE 函数返回有关框架中最后一行的表达式的值。

有关选择框架中第一行的信息，请参阅 FIRST_VALUE 窗口函数。

语法

```
LAST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

参数

expression

对其执行函数的目标列或表达式。

IGNORE NULLS

该函数返回不为 NULL 的框架中的最后一个值（如果所有值为 NULL，则返回 NULL）。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

引入函数的窗口子句。

PARTITION BY expr_list

依据一个或多个表达式定义函数的窗口。

ORDER BY order_list

对每个分区中的行进行排序。如果未指定 PARTITION BY 子句，则 ORDER BY 对整个表进行排序。如果指定 ORDER BY 子句，则还必须指定 frame_clause。

结果取决于数据的排序。在以下情况下，结果是不确定的：

- 当未指定 ORDER BY 子句且一个分区包含一个表达式的两个不同的值时
- 当表达式的计算结果为对应于 ORDER BY 列表中同一值的不同值时。

frame_clause

如果 ORDER BY 子句用于聚合函数，则需要显式框架子句。框架子句优化函数窗口中的行集，包含或排除已排序结果中的行集。框架子句包括 ROWS 关键字和关联的说明符。请参阅窗口函数语法摘要。

返回类型

这些函数支持使用原始AWS Clean Rooms数据类型的表达式。返回类型与 expression 的数据类型相同。

示例

以下示例返回 VENUE 表中每个场地的座位数，同时按容量对结果进行排序（从高到低）。LAST_VALUE 函数用于选择与框架中的最后一行对应的场地的名称：在本例中，为座位数最少的行。按州对结果进行分区，以便当 VENUESTATE 值发生更改时，会选择一个新的最后一个值。窗口框架是无界的，因此为每个分区中的每个行选择相同的最后一个值。

对于加利福利亚，为该分区中的每个行返回 Shoreline Amphitheatre，因为它具有最小座位数 (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 窗口函数

LEAD 窗口函数返回位于分区中当前行的下方（之后）的某个给定偏移量位置的行的值。

语法

```
LEAD (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

参数

value_expr

对其执行函数的目标列或表达式。

offset

一个可选参数，该参数指定要返回其值的当前行后面的行数。偏移量可以是常量整数或计算结果为整数的表达式。如果未指定偏移量，则AWS Clean Rooms使用1作为默认值。偏移量为 0 表示当前行。

IGNORE NULLS

一种可选规范，用于指示在确定要使用哪一行时AWS Clean Rooms应跳过空值。如果未列出 IGNORE NULLS，则包含 Null 值。

> **Note**
>
> 您可以使用 NVL 或 COALESCE 表达式将 null 值替换为另一个值。

RESPECT NULLS

表示在确定AWS Clean Rooms要使用哪一行时应包含空值。如果您未指定 IGNORE NULLS，则默认情况下不支持 RESPECT NULLS。

OVER

指定窗口分区和排序。OVER 子句不能包含窗口框架规范。

PARTITION BY window_partition

一个可选参数，该参数设置 OVER 子句中每个组的记录范围。

ORDER BY window_ordering

对每个分区中的行进行排序。

LEAD 窗口函数支持使用任何AWS Clean Rooms数据类型的表达式。返回类型与 value_expr 的类型相同。

示例

以下示例提供了 SALES 表中于 2008 年 1 月 1 日与 1 月 2 日已售票的事件的佣金以及为后续销售中售票所付的佣金。

```
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 开窗函数

计算给定行的百分比排名。使用以下公式确定百分比排名：

```
(x - 1) / (the number of rows in the window or partition - 1)
```

其中，x 为当前行的排名。以下数据集说明了此公式的使用：

```
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
```

返回值范围介于 0 和 1（含 1）之间。任何集合中的第一行的 PERCENT_RANK 均为 0。

语法

```
PERCENT_RANK ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

参数

( )

该函数没有参数，但需要空括号。

OVER

一个指定窗口分区的子句。OVER 子句不能包含窗口框架规范。

PARTITION BY partition_expression

可选。一个设置 OVER 子句中每个组的记录范围的表达式。

ORDER BY order_list

可选。用于计算百分比排名的表达式。该表达式必须具有数字数据类型或可隐式转换为 1。如果省略 ORDER BY，则所有行的返回值为 0。

如果 ORDER BY 未生成唯一顺序，则行的顺序是不确定的。有关更多信息，请参阅 窗口函数的唯一数据排序。

返回类型

FLOAT8

示例

以下示例计算每个卖家的销售数量的百分比排名：

```
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
```

有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

RANK 窗口函数

RANK 窗口函数基于 OVER 子句中的 ORDER BY 表达式确定一组值中的一个值的排名。如果存在可选的 PARTITION BY 子句，则为每个行组重置排名。排名标准值相等的行将获得相同的排名。 AWS Clean Rooms将并列的行数与并列的排名相加，以计算下一个等级，因此排名可能不是连续的数字。例如，如果两个行的排名为 1，则下一个排名则为 3。

RANK 与 DENSE_RANK 窗口函数存在以下一点不同：对于 DENSE_RANK 来说，如果两个或两个以上的行结合，则一系列排名的值之间没有间隔。例如，如果两个行的排名为 1，则下一个排名则为 2。

您可以在同一查询中包含带有不同的 PARTITION BY 和 ORDER BY 子句的排名函数。

语法

```
RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
```

```
)
```

参数

( )

该函数没有参数，但需要空括号。

OVER

适用于 RANK 函数的窗口子句。

PARTITION BY expr_list

可选。一个或多个定义窗口的表达式。

ORDER BY order_list

可选。定义排名值基于的列。如果未指定 PARTITION BY，则 ORDER BY 使用整个表。如果省略 ORDER BY，则所有行的返回值为 1。

如果 ORDER BY 未生成唯一顺序，则行的顺序是不确定的。有关更多信息，请参阅 窗口函数的唯一数据排序。

返回类型

INTEGER

示例

以下示例按销量对表进行排序（预设情况下按升序顺序），并为每个行分配一个排名。排名值 1 为排名最高的值。在应用窗口函数结果后，对结果进行排序：

```
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)
```

请注意，此示例中的外部 ORDER BY 子句包括第 2 列和第 1 列，以确保每次运行此查询时AWS Clean Rooms返回排序一致的结果。例如，销售额为 IDs 10001 和 10006 的行具有相同的 QTY 和 RNK 值。按列 1 对最后的结果集进行排序可确保行 10001 始终在 10006 之前。有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

在下面的示例中，将窗口函数的顺序倒转 (`order by qty desc`)。现在，最高排名值将应用于最大的 QTY 值。

```
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)
```

有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

以下示例按 SELLERID 对表进行分区，按数量对每个分区进行排序（按降序顺序），并为每个行分配排名。在应用窗口函数结果后，对结果进行排序。

```
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 窗口函数

基于 OVER 子句中的 ORDER BY 表达式确定一组行中当前行的序号（从 1 开始计数）。如果存在可选的 PARTITION BY 子句，则为每组行重置序号。ORDER BY 表达式中具有相同值的行以非确定性的方式接收不同的行号。

语法

```
ROW_NUMBER () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

参数

( )

该函数没有参数，但需要空括号。

OVER

适用于 ROW_NUMBER 函数的窗口子句。

PARTITION BY expr_list

可选。一个或多个定义 ROW_NUMBER 函数的表达式。

ORDER BY order_list

可选。定义行数基于的列的表达式。如果未指定 PARTITION BY，则 ORDER BY 使用整个表。

如果 ORDER BY 未生成唯一顺序或被省略，则行的顺序是不确定的。有关更多信息，请参阅 窗口函数的唯一数据排序。

返回类型

BIGINT

示例

以下示例按 SELLERID 对表进行分区并按 QTY 对每个分区进行排序（按升序顺序），然后为每个行分配一个行号。在应用窗口函数结果后，对结果进行排序。

```
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)
```

有关 WINSALES 表的说明，请参阅窗口函数示例的示例表。

AWS Clean Rooms Spark SQL 条件

条件是包含一个或多个表达式和逻辑运算符的语句，计算结果为 true、false 或 unknown。条件有时也称为“谓词”。

语法

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
| IN_condition
```

> **Note**
>
> 所有字符串比较和 LIKE 模式匹配项均区分大小写。例如，“A”和“a”不匹配。但是，您可通过使用 ILIKE 谓词执行不区分大小写的模式匹配。

AWS Clean Rooms Spark SQL 中支持以下 SQL 条件。

主题

比较运算符

比较条件阐明两个值之间的逻辑关系。所有比较条件都是具有布尔值返回类型的二进制运算符。

AWS Clean Rooms Spark SQL 支持下表中描述的比较运算符。

运算符	语法	描述
!	!expression	逻辑NOT运算符。用于否定布尔表达式，这意味着它返回与表达式值相反的值。 的！ 运算符也可以与其他逻辑运算符（例如 AND 和 OR）组合使用，以创建更复杂的布尔表达式。
<	a < b	小于比较运算符。用于比较两个值并确定左边的值是否小于右边的值。
>	a > b	大于比较运算符。用于比较两个值并确定左边的值是否大于右边的值。
<=	a <= b	小于或等于比较运算符。用于比较两个值并返回左边的值`true`是否小于或等于右边的值，`false`否则返回。
>=	a >= b	大于或等于比较运算符。用于比较两个值并确定左边的值是否大于或等于右边的值。
=	a = b	相等比较运算符，它比较两个值并返回它们`true`是否相等，`false`否则返回。
<> 或 !=	a <> b 或 a != b	不等于比较运算符，它比较两个值，`true`如果两个值不相等，则返回，`false`否则返回。

运算符	语法	描述
==	a == b	标准相等比较运算符，它比较两个值并返回它们true是否相等，false否则返回。 Note 比较字符串值时，== 运算符区分大小写。如果需要执行不区分大小写的比较，则可以在比较之前使用 UPPER () 或 LOWER () 之类的函数将值转换为相同的大小写。

示例

下面是比较条件的一些简单示例：

```
a = 5
a < b
min(x) >= 5
qtysold = any (select qtysold from sales where dateid = 1882
```

以下查询返回所有当前未觅食的松鼠的 id 值。

```
SELECT id FROM squirrels
WHERE !is_foraging
```

以下查询返回 VENUE 表中座位数超过 1 万的场地：

```
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)
```

此示例从 USERS 表中选择喜欢摇滚音乐的用户 (USERID)：

```
select userid from users where likerock = 't' order by 1 limit 5;

userid
--------
3
5
6
13
16
(5 rows)
```

此示例从 USERS 表中选择不清楚是否喜欢摇滚音乐的用户 (USERID)：

```
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
```

具有 TIME 列的示例

下面的示例表 TIME_TEST 具有一个列 TIME_VAL（类型 TIME），其中插入了三个值。

```
select time_val from time_test;

time_val
--------------------
20:00:00
00:00:00.5550
00:58:00
```

以下示例从每个 timetz_val 中提取小时数。

```
select time_val from time_test where time_val < '3:00';
   time_val
---------------
 00:00:00.5550
 00:58:00
```

以下示例比较两种时间文本。

```
select time '18:25:33.123456' = time '18:25:33.123456';
 ?column?
----------
 t
```

具有 TIMETZ 列的示例

下面的示例表 TIMETZ_TEST 具有一个列 TIMETZ_VAL（类型 TIMETZ），其中插入了三个值。

```
select timetz_val from timetz_test;

timetz_val
```

```
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

下面的示例仅选择小于 3:00:00 UTC 的 TIMETZ 值。将值转换为 UTC 后进行比较。

```
select timetz_val from timetz_test where timetz_val < '3:00:00 UTC';

   timetz_val
---------------
 00:00:00.5550+00
```

以下示例比较两种 TIMETZ 文本。比较时忽略时区。

```
select time '18:25:33.123456 PST' < time '19:25:33.123456 EST';

 ?column?
----------
 t
```

逻辑条件

逻辑条件组合两个条件的结果以生成一个结果。所有逻辑条件都是具有布尔值返回类型的二进制运算符。

语法

```
expression
{ AND | OR }
expression
NOT expression
```

逻辑条件使用具有三个值的布尔逻辑，其中 null 值表示未知关系。下表描述逻辑条件的结果，其中 E1 和 E2 表示表达式：

E1	E2	E1 AND E2	E1 OR E2	NOT E2
TRUE	TRUE	TRUE	TRUE	FALSE

E1	E2	E1 AND E2	E1 OR E2	NOT E2
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	

NOT 运算符先于 AND 计算，而 AND 运算符先于 OR 运算符计算。使用的任何圆括号可优先于此默认计算顺序。

示例

以下示例将返回 USERS 表中用户同时喜欢拉斯维加斯和运动的 USERID 和 USERNAME：

```
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)
```

下一个示例将返回 USERS 表中用户喜欢拉斯维加斯或运动或同时喜欢这二者的 USERID 和 USERNAME。此查询将返回上例中的所有输出以及只喜欢拉斯维加斯或运动的用户。

```
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)
```

以下查询使用圆括号将 OR 条件括起来以查找纽约或加利福尼亚演出过 Macbeth 的场地：

```
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
...
```

删除此示例中的圆括号将更改逻辑和查询的结果。

以下示例使用 NOT 运算符：

```
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
...
```

以下示例使用一个 NOT 条件并后跟一个 AND 条件：

```
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)
```

模式匹配条件

模式匹配运算符在字符串中搜索条件表达式中指定的模式，并根据是否找到匹配项返回 true 或 false。AWS Clean Rooms Spark SQL 使用以下方法进行模式匹配：

- LIKE 表达式

LIKE 运算符将字符串表达式（如列名称）与使用通配符 %（百分比）和 _（下划线）的模式进行比较。LIKE 模式匹配始终涵盖整个字符串。LIKE 执行区分大小写的匹配。

主题

- LIKE
- RLIKE

LIKE

LIKE 运算符将字符串表达式（如列名称）与使用通配符 %（百分比）和 _（下划线）的模式进行比较。LIKE 模式匹配始终涵盖整个字符串。若要匹配字符串中任意位置的序列，模式必须以百分比符号开始和结尾。

LIKE 区分大小写。

语法

```
expression [ NOT ] LIKE | pattern [ ESCAPE 'escape_char' ]
```

参数

expression

有效的 UTF-8 字符表达式（如列名称）。

LIKE

LIKE 执行区分大小写的模式匹配。要为多字节字符执行不区分大小写的模式匹配，请将 expression 上的 LOWER 函数和带有 LIKE 函数的 pattern 一起使用。

与比较谓词（例如 = 和 <>）相比，LIKE 谓词不会隐式忽略尾随空格。要忽略尾随空格，请使用 RTRIM 或者将 CHAR 列显式强制转换为 VARCHAR。

该~~运算符等同于 LIKE。此外，!~~运算符等同于 NOT LIKE。

pattern

具有要匹配的模式的有效的 UTF-8 字符表达式。

escape_char

将对模式中的元字符进行转义的字符表达式。默认为两个反斜杠 ('\\')。

如果 pattern 不包含元字符，则模式仅表示字符串本身；在此情况下，LIKE 的行为与等于运算符相同。

其中一个字符表达式可以是 CHAR 或 VARCHAR 数据类型。如果它们不同， AWS Clean Rooms 会将 pattern 转换为 expression 的数据类型。

LIKE 支持下列模式匹配元字符：

操作符	描述
%	匹配任意序列的零个或多个字符。
_	匹配任何单个字符。

示例

下表显示使用 LIKE 的模式匹配的示例：

表达式	返回值
'abc' LIKE 'abc'	True
'abc' LIKE 'a%'	True
'abc' LIKE '_B_'	False
'abc' LIKE 'c%'	False

以下示例查找名称以“E”开头的所有城市：

```
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
...
```

以下示例查找姓中包含“ten”的用户：

```
select distinct lastname from users
where lastname like '%ten%' order by lastname;
lastname
-------------
Christensen
Wooten
...
```

以下示例查找第三和第四个字符为 “ea” 的城市 。：

```
select distinct city from users where city like '__EA%' order by city;
city
-------------
Brea
Clearwater
Great Falls
Ocean City
Olean
Wheaton
(6 rows)
```

以下示例使用默认转义字符串（\\）搜索包含“start_”（文本 `start` 后跟下划线 _)的字符串：

```
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)
```

以下示例指定“^”作为转义字符，然后使用该转义字符搜索包含“start_”（文本 `start` 后跟下划线 _)的字符串：

```
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

RLIKE 运算符允许您检查字符串是否与指定的正则表达式模式匹配。

`true`如果 str 匹配则返回`regexp`，`false`否则返回。

语法

```
rlike(str, regexp)
```

Arguments

str

字符串表达式

regexp

字符串表达式。正则表达式字符串应为 Java 正则表达式。

字符串文字（包括正则表达式模式）在我们的 SQL 解析器中是未转义的。例如，要匹配 “\ abc”，正则表达式的正则表达式可以是 “^\ abc$”。

示例

以下示例将spark.sql.parser.escapedStringLiterals配置参数的值设置为true。此参数特定于 Spark SQL 引擎。Spark SQL 中的spark.sql.parser.escapedStringLiterals参数控制 SQL 解析器如何处理转义的字符串文字。设置为时true，解析器会将字符串文字中的反斜杠字符 (\) 解释为转义字符，从而允许您在字符串值中包含特殊字符，例如换行符、制表符和引号。

```
SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
```

例如，使用spark.sql.parser.escapedStringLiterals=true，您可以在 SQL 查询中使用以下字符串文字：

```
SELECT 'Hello, world!\n'
```

换行符\n将在输出中解释为字面换行符。

以下示例执行正则表达式模式匹配。第一个参数传递给 RLIKE 运算符。它是一个表示文件路径的字符串，其中实际的用户名被替换为'****'模式。第二个参数是用于匹配的正则表达式模式。输出 (true) 表示第一个字符串 ('%SystemDrive%\Users****') 与正则表达式模式 ('%SystemDrive%\\Users.*') 匹配。

```
SELECT rlike('%SystemDrive%\Users\John', '%SystemDrive%\Users.*');
true
```

BETWEEN 范围条件

BETWEEN 条件使用关键字 BETWEEN 和 AND 测试表达式是否包含在某个值范围中。

语法

```
expression [ NOT ] BETWEEN expression AND expression
```

表达式可以是数字、字符或日期时间数据类型，但它们必须是可兼容的。此范围包含起始值。

示例

第一个示例计算有多少个事务登记了 2、3 或 4 票证的销售：

```
select count(*) from sales
where qtysold between 2 and 4;

count
--------
104021
(1 row)
```

范围条件包含开始和结束值。

```
select min(dateid), max(dateid) from sales
where dateid between 1900 and 1910;

min  | max
-----+-----
1900 | 1910
```

范围条件中的第一个表达式必须是较小的值，第二个表达式必须是较大的值。在以下示例中，由于表达式的值，将始终返回零行：

```
select count(*) from sales
where qtysold between 4 and 2;

count
-------
0
(1 row)
```

但是，应用 NOT 修饰符将反转逻辑并生成所有行的计数：

```
select count(*) from sales
where qtysold not between 4 and 2;

count
--------
172456
(1 row)
```

以下查询将返回拥有 20000 到 50000 个座位的场馆的列表：

```
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)
```

以下示例演示了如何为日期值使用 BETWEEN：

```
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
```

请注意，尽管 BETWEEN 的范围包括在内，但日期默认具有 00:00:00 的时间值。示例查询中唯一有效的 1 月 3 日行是 saletime 为 `1/3/2008 00:00:00` 的行。

Null 条件

这些区域有：NULL 当值缺失或未知时，条件测试是否为空。

语法

```
expression IS [ NOT ] NULL
```

参数

expression

任何表达式（如列）。

IS NULL

当表达式的值为 null 时为 true；当表达式具有一个值时，为 false。

IS NOT NULL

当表达式的值为 null 时为 false；当表达式具有一个值时，为 true。

示例

此示例指示 SALES 表的 QTYSOLD 字段中包含 null 的次数：

```
select count(*) from sales
where qtysold is null;
count
-------
0
(1 row)
```

EXISTS 条件

EXISTS 条件测试子查询中是否存在行，并在子查询返回至少一个行时返回 true。如果指定 NOT，此条件将在子查询未返回任何行时返回 true。

语法

```
[ NOT ] EXISTS (table_subquery)
```

参数

EXISTS

当 table_subquery 返回至少一行时，为 true。

NOT EXISTS

当 table_subquery 未返回任何行时，为 true。

table_subquery

计算结果为包含一个或多个列和一个或多个行的表的子查询。

示例

此示例针对具有任何类型的销售的日期返回所有日期标识符，一次返回一个日期：

```
select dateid from date
where exists (
select 1 from sales
where date.dateid = sales.dateid
)
order by dateid;

dateid
--------
1827
1828
1829
...
```

IN 条件

IN 条件测试一组值或一个子查询中的成员身份值。

语法

```
expression [ NOT ] IN (expr_list | table_subquery)
```

参数

expression

数字、字符或日期时间表达式，针对 expr_list 或 table_subquery 进行计算，必须是与列表或子查询的数据类型兼容的。

expr_list

一个或多个逗号分隔的表达式，或一组或多组逗号分隔的表达式（用括号限定）。

table_subquery

一个子查询，计算结果为具有一行或多行的表，但在其选择列表中限制为一列。

IN | NOT IN

如果表达式是表达式列表或查询的成员，则 IN 将返回 true。如果表达式不是成员，NOT IN 将返回 true。在下列情况下，IN 和 NOT IN 将返回 NULL 并且不会返回任何行：如果 expression 生成 null；或者，如果没有匹配的 expr_list 或 table_subquery 值并且至少一个比较行生成 null。

示例

下列条件仅对列出的值有效：

```
qtysold in (2, 4, 5)
date.day in ('Mon', 'Tues')
date.month not in ('Oct', 'Nov', 'Dec')
```

优化大型 IN 列表

为了优化查询性能，包含 10 个以上的值的 IN 列表将在内部作为标量数组计算。少于 10 个值的 IN 列表将作为一系列 OR 谓词计算。SMALLINT、INTEGER、BIGINT、REAL、DOUBLE PRECISION、BOOLEAN、CHAR、VARCHAR、DATE、TIMESTAMP 和 TIMESTAMPTZ 数据类型均支持此优化。

查看查询的 EXPLAIN 输出以查看此优化的效果。例如：

```
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)
```

查询嵌套数据

AWS Clean Rooms 提供对关系和嵌套数据的与 SQL 兼容的访问权限。

AWS Clean Rooms 访问嵌套数据时，使用点符号和数组下标进行路径导航。它还启用 FROM 子句项用于遍历数组并用于取消嵌套操作。以下主题描述了不同的查询模式，这些模式将array/struct/map数据类型的使用与路径和数组导航、取消嵌套和联接相结合。

主题

导航

AWS Clean Rooms 允许分别使用方[...]括号和点符号导航到数组和结构。此外，您还可以使用点记法将导航混合到结构中，使用括号符号将数组混合到结构中。

Example

例如，以下示例查询假定 `c_orders` 数组数据列是一个具有结构的数组，并且属性名为 `o_orderkey`。

```
SELECT cust.c_orders[0].o_orderkey FROM customer_orders_lineitem AS cust;
```

您可以在所有类型的查询中使用点和括号符号，例如筛选、联接和聚合。您可以在通常存在列引用的查询中使用这些符号。

Example

以下示例使用筛选结果的 SELECT 语句。

```
SELECT count(*) FROM customer_orders_lineitem WHERE c_orders[0].o_orderkey IS NOT NULL;
```

Example

以下示例在 GROUP BY 和 ORDER BY 子句中使用括号和点导航：

```
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;
```

取消嵌套查询

要取消嵌套查询，请 AWS Clean Rooms 启用对数组的迭代。它通过使用查询的 FROM 子句导航数组来实现这一点。

Example

使用前面的示例，以下示例对 c_orders 的属性值进行迭代。

```
SELECT o FROM customer_orders_lineitem c, c.c_orders o;
```

取消嵌套语法是 FROM 子句的扩展。在标准 SQL 中，FROM 子句 x （AS） y 表示 y 迭代关系 x 中的每个元组。在这种情况下，x 指的是关系，而 y 指的是关系 x 的别名。同样，使用 FROM 子句项 x （AS） y 进行取消嵌套的语法表示 y 迭代数组表达式 x 中的每个值。在这种情况下，x 是一个数组表达式，而 y 是 x 的别名。

左侧操作数也可以使用点和括号表示法进行常规导航。

Example

在上一个示例中：

- customer_orders_lineitem c 是对 customer_order_lineitem 基表的迭代
- c.c_orders o 是对 c.c_orders array 的迭代

要迭代作为数组中的数组的 o_lineitems 属性，必须添加多个子句。

```
SELECT o, l FROM customer_orders_lineitem c, c.c_orders o, o.o_lineitems l;
```

AWS Clean Rooms 在使用遍历数组时，还支持使用数组索引 AT 关键字。子句 x AS y AT z 迭代数组 x 并生成字段 z，即数组索引。

Example

以下示例演示数组索引的工作原理：

```
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

以下示例对标量数组进行迭代：

```
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

以下示例对多个级别的数组进行迭代。该示例使用多个 unnest 子句来迭代到最内层的数组。这些区域有：f.multi_level_arrayAS 数组迭multi_level_array代。阵列 AS 元素是对里面数组的迭代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)
```

宽松语义

预设情况下，在导航无效时，嵌套数据值的导航操作返回 null，而不是返回错误。如果嵌套数据值不是对象，或者嵌套数据值是一个对象，但不包含查询中使用的属性名称，则对象导航无效。

Example

例如，以下查询访问嵌套数据列 `c_orders` 中的无效属性名称：

```
SELECT c.c_orders.something FROM customer_orders_lineitem c;
```

如果嵌套数据值不是数组或数组索引超出界限，则数组导航返回 null。

Example

以下查询返回 null，因为 `c_orders[1][1]` 超出了界限。

```
SELECT c.c_orders[1][1] FROM customer_orders_lineitem c;
```

自检类型

嵌套数据列支持返回有关该值的类型和其他类型信息的检查函数。AWS Clean Rooms 支持以下用于嵌套数据列的布尔函数：

- 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

如果输入值为 null，所有这些函数都返回 false。IS_SCALAR、IS_OBJECT 和 IS_ARRAY 是相互排斥的，涵盖除 null 之外的所有可能的值。要推断出与数据对应的类型，请 AWS Clean Rooms 使用 JSON_TYPEOF 函数，该函数返回嵌套数据值的类型（顶层），如以下示例所示：

```
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
```

AWS Clean Rooms SQL 参考的文档历史记录

下表介绍了《 AWS Clean Rooms SQL 参考》的文档版本。

如需有关此文档的更新通知，您可以订阅 RSS 源。要订阅 RSS 更新，您必须为当前使用的浏览器启用 RSS 插件。

变更	说明	日期
Spark SQL 支持提示	AWS Clean Rooms Spark SQL 支持查询提示，以优化查询性能并降低计算成本。	2026 年 1 月 20 日
Spark SQL 支持缓存表	AWS Clean Rooms Spark SQL 支持 CACHE TABLE 命令，该命令允许客户缓存现有表或根据查询结果创建和缓存新表，以提高查询性能。	2025 年 10 月 22 日
Spark SQL 支持第一个和最后一个窗口函数	AWS Clean Rooms Spark SQL 支持以下窗口函数：第一个和最后一个。	2025 年 6 月 12 日
Spark SQL 函数文档更新	仅限文档更新，以准确反映支持的 Spark SQL 函数。删除了 25 个不支持的函数的文档，包括 <=> 运算符、类似于、LISTAGG 和 ARRAY_INSERT。将函数名称从 DATEADD 更正为 DATE_ADD，将 DATEDIFF 改为 DATE_DIFF，将 ISNOTNULL 改为 IS_NULL，将 ISNOTNULL 更正为 IS_NOT_NULL。修复了 DATE_PART 示例中的一个错字。	2025 年 5 月 20 日

AWS Clean Rooms 火花 SQL	现在，客户可以使用 Spark SQL 分析引擎支持的某些 SQL 条件、函数、命令和约定来运行查询。	2024 年 10 月 29 日
SQL 命令和 SQL 函数-更新	添加了 JOIN 子句、EXCEPT 集合运算符、CASE 条件表达式以及以下函数的示例：ANY_VALUE、NVL 和 COALESCE、NULLIF、CAST、CONVERT、CONVERT_TIMEZONE、EXTRACT、MOD、SIGN、CONCAT、FIRST_VALUE 和 LAST_VALUE。	2024 年 2 月 28 日
SQL 函数 — 更新	AWS Clean Rooms 现在支持以下 SQL 函数：数组、SUPER 和 VARBYTE。现在支持以下数学函数：ACOS、ASIN、ATAN、COT、 ATAN2 DEXP、PI、POW、RADIANS 和 SIN。现在支持以下 JSON 函数：CAN_JSON_PARSE、JSON_PARSE 和 JSON_SERIALIZE。	2023 年 10 月 6 日
支持嵌套数据类型	AWS Clean Rooms 现在支持嵌套数据类型。	2023 年 8 月 30 日
SQL 命名规则 — 更新	仅限文档的更改，以明确保留的列名。	2023 年 8 月 16 日
正式发布	S AWS Clean Rooms QL 参考现已正式发布。	2023 年 7 月 31 日

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。