SQL リファレンス

AWS Clean Rooms

AWS Clean Rooms: SQL リファレンス

Table of Contents

での SQL の概要AWS Clean Rooms

AWS Clean Rooms SQL リファレンスへようこそ。

AWS Clean Roomsは、業界標準の構造化クエリ言語 (SQL) を中心に構築されています。SQL は、データベースとデータベースオブジェクトの操作に使用するコマンドと関数で構成されるクエリ言語です。また SQL は、データ型、式、およびリテラルに関するルールも適用します。

以下のトピックでは、この SQL リファレンスで使用される規則と命名規則に関する一般的な情報を提供します。

トピック

- SQL リファレンスの規則
- SQL の命名規則
- SQL エンジンによるデータ型サポート

以下のセクションでは、 で使用できるリテラル、データ型、SQL コマンド、SQL 関数のタイプ、および SQL 条件について説明しますAWS Clean Rooms。

- AWS Clean Rooms Spark SQL

詳細についてはAWS Clean Rooms、AWS Clean Rooms「 ユーザーガイド」とAWS Clean Rooms「 API リファレンス」を参照してください。

SQL リファレンスの規則

このセクションでは、SQL の式、コマンド、および関数の構文を記述する際に使用される規則について説明します。

文字	説明
CAPS	大文字の単語はキーワードです。
[ ]	角括弧はオプションの引数を示します。角括弧に複数の引数が含まれる場合は、任意の個数の引数を選択で

文字	説明
	きることを示します。さらに、複数の行に角括弧で囲まれた引数がある場合、 パーサーは、それらの引数が構文の順番どおりに出現するものと想定します。
{}	中括弧は、括弧内の引数の 1 つを選択する必要があることを示します。
|	縦線は、どちらかの引数を選択できることを示します。
イタリック	イタリック体の単語は、プレースホルダーを示します。イタリック体の単語の場所に適切な値を挿入する必要があります。
...	省略符号は、先行する要素の繰り返しが可能であることを示します。
'	一重引用符に囲まれた単語は、引用符の入力が必要であることを示します。

SQL の命名規則

以下のセクションでは、 AWS Clean Roomsでの SQL の命名規則について説明します。

トピック

設定済みテーブルの関連付け名と列

クエリを行えるメンバーは、設定済みテーブルの関連付け名をクエリでテーブル名として使用できます。設定済みテーブルの関連付け名と設定済みテーブルの列には、クエリでエイリアスを使用できます。

設定済みテーブルの関連付け名、設定済みテーブルの列名、エイリアスには、以下の命名規則が適用されます。

- 英数字、アンダースコア (_)、またはハイフン (-) のみを使用する必要がありますが、先頭または末尾にハイフンを使用することはできません。
  - (カスタム分析ルールのみ) ドル記号 ($) を使用できますが、ドル引用符で囲まれた文字列定数に続くパターンは使用できません。

    ドル引用符付けされた文字列定数は、次のもので構成されます。

    - ドル記号 ($)
    - 0 文字以上の省略可能な「タグ」
    - もう 1 つのドル記号
    - 文字列の内容を構成する任意の一連の文字
    - ドル記号 ($)
    - ドル引用符の先頭と同じタグ
    - ドル記号

    例: `$$invalid$$`
- 連続するハイフン (-) 文字を含めることはできません。
- 以下のプレフィックスで始めることはできません。

  `padb_`, `pg_`, `stcs_`, `stl_`, `stll_`, `stv_`, `svcs_`, `svl_`, `svv_`, `sys_`, `systable_`
- バックスラッシュ文字 (\)、引用符 (')、または二重引用符で囲まれていないスペースを含めることはできません。
- アルファベット以外の文字で始まる場合は、二重引用符 (" ") で囲む必要があります。
- ハイフン (-) が含まれる場合は、二重引用符 (" ") で囲む必要があります。
- 1 ~ 127 文字の長さにする必要があります。
- 予約語は二重引用符 (" ") で囲む必要があります。
- 次の列名は (引用符を使用して AWS Clean Rooms も) 予約できません。
  - oid
  - tableoid
  - xmin
  - cmin
  - xmax
  - cmax
  - ctid

予約語

以下は、 の予約語のリストです AWS Clean Rooms。

AES128	DELTA32KDESC	LEADING	PRIMARY
AES256ALL	DISTINCT	LEFTLIKE	RAW
ALLOWOVER WRITEANALYSE	DO	LIMIT	READRATIO
ANALYZE	DISABLE	LOCALTIME	RECOVERRE FERENCES
AND	ELSE	LOCALTIMESTAMP	REJECTLOG
ANY	EMPTYASNU LLENABLE	LUN	RESORT
ARRAY	ENCODE	LUNS	RESPECT
AS	ENCRYPT	LZO	RESTORE
ASC	ENCRYPTIONEND	LZOP	RIGHTSELECT
AUTHORIZATION	EXCEPT	MINUS	SESSION_USER
AZ64	EXPLICITFALSE	MOSTLY16	SIMILAR
BACKUPBETWEEN	FOR	MOSTLY32	SNAPSHOT
BINARY	FOREIGN	MOSTLY8NATURAL	SOME
BLANKSASN ULLBOTH	FREEZE	NEW	SYSDATESYSTEM
BYTEDICT	FROM	NOT	TABLE
BZIP2CASE	FULL	NOTNULL	TAG
CAST	GLOBALDICT256	NULL	TDES

CHECK	GLOBALDIC T64KGRANT	NULLSOFF	TEXT255
COLLATE	GROUP	OFFLINEOFFSET	TEXT32KTHEN
COLUMN	GZIPHAVING	OID	TIMESTAMP
CONSTRAINT	IDENTITY	OLD	TO
CREATE	IGNOREILIKE	ON	TOPTRAILING
CREDENTIA LSCROSS	IN	ONLY	TRUE
CURRENT_DATE	INITIALLY	OPEN	TRUNCATEC OLUMNSUNION
CURRENT_TIME	INNER	OR	UNIQUE
CURRENT_T IMESTAMP	INTERSECT	ORDER	UNNEST
CURRENT_USER	INTERVAL	OUTER	USING
CURRENT_U SER_IDDEFAULT	INTO	OVERLAPS	VERBOSE
DEFERRABLE	IS	PARALLELP ARTITION	WALLETWHEN
DEFLATE	ISNULL	PERCENT	WHERE
DEFRAG	JOIN	PERMISSIONS	WITH
DELTA	LANGUAGE	PIVOTPLACING	WITHOUT

SQL エンジンによるデータ型サポート

AWS Clean Rooms は、複数の SQL エンジンとダイアレクトをサポートしています。これらの実装全体のデータ型システムを理解することは、データのコラボレーションと分析を成功させるために不

可欠です。次の表は、 AWS Clean Rooms SQL、Snowflake SQL、Spark SQL における同等のデータ型を示しています。

数値データ型

数値型は、正確な整数から近似浮動小数点値まで、さまざまな種類の数値を表します。数値型の選択は、ストレージ要件と計算精度の両方に影響します。整数型はバイトサイズによって異なりますが、10 進数型と浮動小数点型では精度とスケールのオプションが異なります。

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
8 バイト整数	BIGINT	サポートされていません	BIGINT、LONG	-9,223,372,036,854,775,808 から 9,223,372,036,854,775,807 までの符号付き整数。
4 バイト整数	INT	サポートされていません	INT, INTEGER	-2,147,483,648 から 2,147,483,647 までの符号付き整数
2 バイト整数	SMALLINT	サポートされていません	SMALLINT、SHORT	-32,768 ~ 32,767 の符号付き整数
1 バイト整数	サポートされません	サポートされません	TINYINT、BYTE	-128 ~ 127 の符号付き整数
倍精度浮動小数点数	DOUBLE、DOUBLE PRECISION	FLOAT、FLOAT4, FLOAT8、DOUBLE、DOUBLE	DOUBLE	8 バイトの倍精度浮動小数点数

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
		PRECISION、REAL		
単一精度浮動小数点数	REAL、FLOAT	サポートされていません	FLOAT	4 バイトの単精度浮動小数点数
10 進数 (固定精度)	DECIMAL	DECIMAL、NUMERIC、NUMBER **Note** Snowflake は、小幅の正確な数値型 (INT、BIGINT、SMALLINT など) を NUMBER に自動的にエイリアスします。	DECIMAL、NUMERIC、	任意精度符号付き小数
10 進数 (精度付き)	DECIMAL (p)	DECIMAL (p)、NUMBER (p)	DECIMAL (p)	固定精度の 10 進数

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
10 進数 (スケール付き)	DECIMAL(p,s)	DECIMAL (p,s)、NUMBER (p,s)	DECIMAL(p,s)	スケール付きの固定精度の 10 進数

ブールデータ型

ブール型は、単純な true/false 論理値を表します。これらのタイプは SQL エンジン間で一貫性があり、一般的にフラグ、条件、論理オペレーションに使用されます。

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
ブール値	BOOLEAN	BOOLEAN	BOOLEAN	true/false 値を表します

日付と時刻のデータ型

日付と時刻のタイプは、さまざまなレベルの精度とタイムゾーン認識で、時間データを処理します。これらのタイプは、日付、時刻、タイムスタンプの保存にさまざまな形式をサポートし、タイムゾーン情報を包含または除外するオプションがあります。

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
日付	DATE	DATE	DATE	タイムゾーンのない日付値 (年、月、日)
Time	TIME	サポートされません	サポートされません	UTC でのタイムゾーンなしの時刻

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
TZ での時間	TIMETZ	サポートされません	サポートされません	UTC でのタイムゾーンを含む時刻
タイムスタンプ	TIMESTAMP	TIMESTAMP、TIMESTAMP_NTZ	TIMESTAMP_NTZ	Timestamp without time zone Note NTZ は「タイムゾーンなし」を示します
TZ によるタイムスタンプ	TIMESTAMPTZ	TIMESTAMP_LTZ	TIMESTAMP、TIMESTAMP_LTZ	ローカルタイムゾーンのタイムスタンプ Note LTZ は「ローカルタイムゾーン」を示します

文字データ型

文字タイプはテキストデータを保存し、固定長オプションと可変長オプションの両方を提供します。これらのタイプは、テキスト文字列とバイナリデータを処理します。ストレージの割り当てを制御するためのオプションの長さ仕様があります。

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
固定長文字	CHAR	CHAR、CHARACTER	CHAR、CHARACTER	固定長のキャラクタ文字列
長さの固定長文字	CHAR(n)	CHAR(n)、CHARACTER(n)	CHAR(n)、CHARACTER(n)	指定された長さの固定長文字列
可変長文字	VARCHAR	VARCHAR、STRING、TEXT	VARCHAR、STRING	可変長の文字列
長さの可変長文字	VARCHAR(n)	VARCHAR(n)、STRING(n)、TEXT(n)	VARCHAR(n)	長さ制限付きの可変長文字列
バイナリ	VARBYTE	BINARY、VARBINARY	BINARY	バイナリバイトシーケンス
長さのバイナリ	VARBYTE(n)	サポートされません	サポートされません	長さ制限付きのバイナリバイトシーケンス

構造化データ型

構造化型を使用すると、複数の値を 1 つのフィールドに結合することで、複雑なデータ整理が可能になります。これには、順序付けられたコレクションの配列、キーと値のペアのマップ、名前付きフィールドを持つカスタムデータ構造を作成するための構造体が含まれます。

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
配列	ARRAY<type>	ARRAY (タイプ)	ARRAY<type>	同じタイプの要素の順序付けられたシーケンス Note 配列タイプには同じタイプの要素が含まれている必要があります
マッピング	MAP<key,value>	MAP (キー、値)	MAP<key,value>	キーと値のペアのコレクション Note マップタイプには同じタイプの要素が含まれている必

データ型	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	説明
				要があります
Struct	STRUCT< field1: type1、field2: type2>	OBJECT(field1 type1、field2 type2)	STRUCT< field1: type1、field2: type2 >	指定されたタイプの名前付きフィールドを持つ構造 Note 構造化型の構文は実装によって若干異なる場合があります
Super	SUPER	サポートされません	サポートされません	複合型を含むすべてのデータ型をサポートする柔軟な型

AWS Clean Rooms Spark SQL

AWS Clean Rooms Spark SQL は、データ型、式、リテラルの使用に関するルールを適用します。

Spark SQL の詳細については、 AWS Clean Rooms AWS Clean Rooms 「 ユーザーガイド」とAWS Clean Rooms 「 API リファレンス」を参照してください。

以下のトピックでは、 AWS Clean Rooms Spark SQL でサポートされているリテラル、データ型、コマンド、関数、および条件について説明します。

トピック

リテラル

リテラルまたは定数は固定データ値であり、一連の文字または数値定数から構成されます。

AWS Clean Rooms Spark SQL は、次のようないくつかのタイプのリテラルをサポートしています。

- 数値リテラル。整数、10 進数、および浮動小数点数に使用されます。
- 文字列、文字列、または文字定数とも呼ばれる文字リテラルは、文字列値を指定するために使用されます。
- 日時データ型で使用される、日付、時刻、およびタイムスタンプのリテラル。詳細については、「日付、時刻、およびタイムスタンプのリテラル」を参照してください。
- 間隔リテラル。詳細については、「間隔リテラル」を参照してください。
- ブールリテラル。詳細については、「ブールリテラル」を参照してください。
- Null リテラル (null 値を指定するために使用されます)。
- Unicode 一般カテゴリ (Cc) から、TAB、CARRIAGE RETURN (CR)、LINE FEED (LF) の Unicode 制御文字のみがサポートされます。

AWS Clean Rooms Spark SQL は SELECT 句の文字列リテラルへの直接参照をサポートしていませんが、CAST などの関数内で使用できます。

+ (連結) 演算子

数値リテラル、文字列リテラル、日時リテラル、間隔リテラルを連結します。+ 記号の両側にリテラルがあり、+ 記号の両側の入力に基づいて異なる型を返します。

構文

```
numeric + string
```

```
date + time
```

```
date + timetz
```

引数の順序は逆にすることができます。

引数

######

数値を表現するリテラルまたは定数は、整数または浮動小数点数になり得ます。

#######

文字列、キャラクタ文字列、または文字定数。

date

DATE の列、あるいは暗黙的に DATE に変換される式。

time

TIME の列、あるいは暗黙的に TIME に変換される式。

timetz

TIMETZ の列、あるいは暗黙的に TIMETZ に変換される式。

例

次のサンプルテーブル TIME_TEST には、3 つの値が挿入された列 TIME_VAL (TIME 型) があります。

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

データ型

Spark SQL AWS Clean Rooms が保存または取得する各値には、関連付けられたプロパティの固定セットを持つデータ型があります。データ型はテーブルの作成時に宣言されます。データ型は、列または引数に含めることができる値セットを制限します。

次の表に、Spark SQL AWS Clean Rooms で使用できるデータ型を示します。

データ型名	データ型	エイリアス	説明
配列	the section called “ネスト型”	該当しない	配列のネストされたデータ型
BIGINT	the section called “数値型”	該当しない	符号付き 8 バイト整数
BINARY	the section called “バイナリタイプ”	該当しない	バイトシーケンス値
BOOLEAN	the section called “ブール型”	BOOL	論理ブール演算型 (true/false)
BYTE	the section called “数値型”	該当しない	1 バイトの符号付き整数、-128 ~ 127
CHAR	the section called “文字型”	CHARACTER	固定長のキャラクタ文字列
DATE	the section called “日時型”	該当しない	カレンダー日付 (年、月、日)

データ型名	データ型	エイリアス	説明
DECIMAL	the section called “数値型”	NUMERIC	精度の選択が可能な真数
FLOAT	the section called “数値型”	FLOAT8、倍精度	倍精度浮動小数点数
INTEGER	the section called “数値型”	INT	符号付き 4 バイト整数
INTERVAL	the section called “日時型”	該当しない	日単位の順序または年単位の順序の時間
LONG	the section called “数値型”	該当しない	8 バイトの符号付き整数
MAP	the section called “ネスト型”	該当なし	マップのネストされたデータ型
REAL	the section called “数値型”	FLOAT4	単精度浮動小数点数
SHORT	the section called “数値型”	該当しない	2 バイトの符号付き整数。
SMALLINT	the section called “数値型”	該当しない	符号付き 2 バイト整数
STRUCT	the section called “ネスト型”	該当しない	構造体のネストされたデータ型
TIMESTAMP_LTZ	the section called “日時型”	該当しない	ローカルタイムゾーンの時刻
TIMESTAMP_NTZ	the section called “日時型”	該当しない	タイムゾーンのない時刻
TINYINT	the section called “数値型”	該当しない	1 バイトの符号付き整数、-128 ~ 127

データ型名	データ型	エイリアス	説明
VARCHAR	the section called “文字型”	CHARACTER VARYING	ユーザーによって定義された制限を持つ可変長キャラクタ文字列

> **Note**
>
> ARRAY、STRUCT、MAP のネストされたデータ型は現在、カスタム分析ルールでのみ有効になっています。詳細については、「ネスト型」を参照してください。

マルチバイト文字

VARCHAR データ型では、最大 4 バイトの UTF-8 マルチバイト文字をサポートします。5 バイト以上の文字はサポートされていません。マルチバイト文字を含む VARCHAR 列のサイズを計算するには、文字数と 1 文字当たりのバイト数を掛けます。例えば、文字列に漢字が 4 文字含まれ、各文字のサイズが 3 バイトである場合は、文字列を格納するのに VARCHAR(12) 列が必要です。

VARCHAR データ型は、次に示す無効な UTF-8 コードポイントをサポートしていません:

`0xD800 – 0xDFFF(バイトシーケンス:ED A0 80 – ED BF BF)`

CHAR データ型は、マルチバイト文字をサポートしていません。

数値型

数値データ型には、整数型、10 進数型、および浮動小数点数型などがあります。

トピック

- 整数型
- DECIMAL 型または NUMERIC 型
- 浮動小数点型
- 数値に関する計算

整数型

次のデータ型を使用して、さまざまな範囲の整数を保存します。各データ型の許容範囲の外にある値を格納することはできません。

名前	ストレージ	範囲
SMALLINT	2 バイト	-32768 ~ +32767
SHORT	2 バイト	-32768 ~ +32767
INTEGER または INT	4 バイト	-2147483648 ~ +2147483647
BIGINT	8 バイト	-9223372036854775808 ~ 9223372036854775807
LONG	8 バイト	-9223372036854775808 ~ 9223372036854775807

DECIMAL 型または NUMERIC 型

DECIMAL データ型または NUMERIC データ型を使用し、ユーザー定義の精度で値を格納します。DECIMAL キーワードと NUMERIC キーワードは同じように使用できます。このドキュメントでは、このデータ型を表す用語として decimal を優先的に使用します。numeric という用語は一般的に整数、10 進数、および浮動小数点のデータ型を称する場合に使用します。

ストレージ	範囲
可変。非圧縮の DECIMAL 型の場合は最大 128 ビット。	最大で 38 桁の精度を持つ、128 ビットの符号付き整数。

テーブル内に DECIMAL 列を定義するには、*precision* と *scale* を次のように指定します。

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

precision

値全体での有効な桁の合計。小数点の両側の桁数。例えば、数値 48.2891 の場合は精度が 6、スケールが 4 となります。指定がない場合、デフォルトの精度は 18 です。最大精度は 38 です。

入力値で小数点の左側の桁数が、列の精度から列のスケールを引いて得られた桁数を超えている場合、入力値を列にコピー (または挿入も更新も) することはできません。このルールは、列の定義を外れるすべての値に適用されます。例えば、numeric(5,2) 列の値の許容範囲は、-999.99～999.99 です。

scale

小数点の右側に位置する、値の小数部における小数の桁数です。整数のスケールはゼロです。列の仕様では、スケール値は精度値以下である必要があります。指定がなければ、デフォルトのスケールは 0 です。最大スケールは 37 です。

テーブルにロードされた入力値のスケールが列のスケールより大きい場合、値は指定されたスケールに丸められます。SALES テーブルの PRICEPAID 列が DECIMAL(8,2) 列である場合を例にとります。DECIMAL(8,4) の値を PRICEPAID 列に挿入すると、値のスケールは 2 に丸められます。

```
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)
```

ただし、テーブルから選択された値の明示的なキャストの結果は丸められません。

> **Note**
>
> DECIMAL(19,0) 列に挿入できる最大の正の値は、9223372036854775807 (2^{63} -1) です。最大の負の値は -9223372036854775807 です。例えば、値 9999999999999999999 (19 桁の 9 の並び) の挿入を試みると、オーバーフローエラーが発生します。小数点の位置に関係なく、 AWS Clean Rooms が DECIMAL 数として表現できる最大の文字列は

9223372036854775807 です。例えば、DECIMAL(19,18) 列にロードできる最大値は 9.223372036854775807 です。
これらの規則は、次の理由によるものです。

- 精度の有効桁数が 19 桁以下の DECIMAL 値は、内部で 8 バイトの整数として格納されます。
- 精度の有効桁数が 20 ~ 38 桁の DECIMAL 値は、16 バイトの整数として格納されます。

128 ビットの DECIMAL または NUMERIC の列の使用に関する注意事項

アプリケーションがそのような精度を必要とすることが明確でない限り、最大精度を DECIMAL 列に任意に割り当てないでください。128 ビット値は、64 ビット値の 2 倍のディスク容量を使用するので、クエリの実行時間が長くなる可能性があります。

浮動小数点型

可変精度の数値を格納するには、REAL および DOUBLE PRECISION のデータ型を使用します。これらのデータ型は非正確型です。すなわち、一部の値が近似値として格納されるため、特定の値を格納して返すと若干の相違が生じる場合があります。正確な格納および計算が必要な場合は (金額の場合など)、DECIMAL データ型を使用します。

REAL は、浮動小数点演算に関する標準規格 IEEE 754 に従って単精度浮動小数点形式を表します。精度は約 6 桁で、範囲は約 1E-37 から 1E+37 です。このデータ型を FLOAT4 として指定することもできます。

DOUBLE PRECISION は、IEEE 標準 754 の 2 進浮動小数点演算に従って倍精度浮動小数点形式を表します。精度は約 15 桁で、範囲は約 1E-307 から 1E+308 です。このデータ型を FLOAT または FLOAT8 として指定することもできます。

数値に関する計算

では AWS Clean Rooms、計算は、加算、減算、乗算、除算の二項数演算を指します。このセクションでは、このような演算の予期される戻り型について、さらに DECIMAL データ型が必要な場合に精度とスケールを決定するのに適用される特定の計算式について説明します。

クエリ処理中に数値の計算が行われる場合には、計算が不可能であるためにクエリが数値オーバーフローエラーを返すといった状況が発生することがあります。さらに、算出される値のスケールが、変化したり予期せぬものであったりする状況が発生することもあります。一部の演算については、明示

的なキャスト (型の上位変換) または AWS Clean Rooms 設定パラメータを使用してこれらの問題を解決できます。

SQL 関数を使用した同様の計算の結果の詳細については、「AWS Clean Rooms Spark SQL 関数」を参照してください。

計算の戻り型

でサポートされている一連の数値データ型を考慮すると AWS Clean Rooms、次の表は、加算、減算、乗算、除算オペレーションで予想される戻り型を示しています。表の左側の最初の列は計算の 1 番目のオペランドを示し、一番上の行は 2 番目のオペランドを示します。

オペランド 1	オペランド 2	戻り型
SMALLINT または SHORT	SMALLINT または SHORT	SMALLINT または SHORT
SMALLINT または SHORT	INTEGER	INTEGER
SMALLINT または SHORT	BIGINT	BIGINT
SMALLINT または SHORT	DECIMAL	DECIMAL
SMALLINT または SHORT	FLOAT4	FLOAT8
SMALLINT または SHORT	FLOAT8	FLOAT8
INTEGER	INTEGER	INTEGER
INTEGER	BIGINT または LONG	BIGINT または LONG
INTEGER	DECIMAL	DECIMAL
INTEGER	FLOAT4	FLOAT8
INTEGER	FLOAT8	FLOAT8
BIGINT または LONG	BIGINT または LONG	BIGINT または LONG
BIGINT または LONG	DECIMAL	DECIMAL
BIGINT または LONG	FLOAT4	FLOAT8

オペランド 1	オペランド 2	戻り型
BIGINT または LONG	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL
DECIMAL	FLOAT4	FLOAT8
DECIMAL	FLOAT8	FLOAT8
FLOAT4	FLOAT8	FLOAT8
FLOAT8	FLOAT8	FLOAT8

DECIMAL の計算結果の精度とスケール

次の表に、算術演算で DECIMAL 型の結果が返されるときに算出結果の精度とスケールに適用されるルールの概要を示します。この表では、 p1と は計算の最初のオペランドの精度とスケールs1を表します。 p2と は 2 番目のオペランドの精度とスケールs2を表します。(これらの計算に関係なく、結果の最大精度は 38、結果の最大スケールは 38 です)。

オペレーション	結果の精度とスケール
+ または -	スケール = `max(s1,s2)` 精度 = `max(p1-s1,p2-s2)+1+scale`
*	スケール = `s1+s2` 精度 = `p1+p2+1`
/	スケール = `max(4,s1+p2-s2+1)` 精度 = `p1-s1+ s2+scale`

例えば、SALES テーブルの PRICEPAID 列と COMMISSION 列は両方とも DECIMAL(8,2) 列です。PRICEPAID を COMMISSION で除算する場合 (または COMMISSION を PRICEPAID で除算する場合)、計算式は次のように適用されます。

```
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)
```

次の計算は、UNION、INTERSECT、EXCEPT などの集合演算子および COALESCE や DECODE などの関数を使用して DECIMAL 値に対して演算を実行した場合に、結果として生じる精度とスケールを計算するための汎用的なルールです。

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

例えば、DECIMAL(7,2) 列が 1 つ含まれる DEC1 テーブルと、DECIMAL(15,3) 列が 1 つ含まれる DEC2 テーブルを結合して DEC3 テーブルを作成します。DEC3 のスキーマを確認すれば、列が NUMERIC(15,3) 列になることが分かります。

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

上記の例では、計算式は次のように適用されます。

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

Scale = max(2,3) = 3

Result = DECIMAL(15,3)
```

除算演算に関する注意事項

除算演算の場合、ゼロ除算を行うとエラーが返されます。

精度とスケールが計算されると、スケール制限 100 が適用されます。算出された結果スケールが 100 より大きい場合、除算結果は次のようにスケーリングされます。

- 精度 = precision - (scale - max_scale)
- スケール = max_scale

算出された精度が最大精度 (38) より高い場合、精度は 38 に引き下げられ、スケールは `max(38 + scale - precision), min(4, 100))` で計算された結果となります。

オーバーフロー条件

すべての数値計算についてオーバーフローが調べられます。精度が 19 以下の DECIMAL データは 64 ビットの整数として格納されます。精度が 19 を上回る DECIMAL データは 128 ビットの整数として格納されます。すべての DECIMAL 値の最大精度は 38 であり、最大スケールは 37 です。値がこれらの制限を超えるとオーバーフローエラーが発生します。このルールは中間結果セットと最終結果セットの両方に適用されます。

- 明示的なキャストでは、特定のデータ値がキャスト関数で指定されたリクエストされた精度またはスケールに適合しない場合、ランタイムオーバーフローエラーが発生します。例えば、SALES テーブルの PRICEPAID 列 (DECIMAL(8,2) 列) からの値をすべてキャストできるとは限らないので、結果として DECIMAL(7,3) を返すことはできません。

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

  このエラーは、PRICEPAID 列に含まれる大きな値のいくつかをキャストできないために発生します。
- 乗算演算によって得られる結果では、結果スケールが各オペランドのスケールを足し算した値となります。例えば、両方のオペランドとも 4 桁のスケールを持っている場合、結果としてスケールは 8 桁となり、小数点の左側には 10 桁のみが残ります。したがって、有効スケールを持つ 2 つの大きな数値を乗算する場合は、比較的簡単にオーバーフロー状態に陥ります。

INTEGER 型および DECIMAL 型での数値計算

計算式のオペランドの一方が INTEGER データ型を持っており、他方のオペランドが DECIMAL データ型を持っている場合、INTEGER オペランドは DECIMAL として暗黙的にキャストされます。

- SMALLINT または SHORT は DECIMAL(5,0) としてキャストされます
- INTEGER は、DECIMAL(10,0) としてキャストされます。
- BIGINT または LONG は DECIMAL (19,0) としてキャストされます

例えば、DECIMAL(8,2) 列の SALES.COMMISSION と、SMALLINT 列の SALES.QTYSOLD を乗算する場合、この計算は次のようにキャストされます。

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

文字型

文字データ型には、CHAR (文字) や VARCHAR (可変文字) などがあります。

トピック

- CHAR または CHARACTER
- VARCHAR または CHARACTER VARYING
- 末尾の空白の重要性

CHAR または CHARACTER

固定長の文字列を格納するには、CHAR または CHARACTER を使用します。これらの文字列は空白で埋められるので、CHAR(10) 列は常に 10 バイトのストレージを占有します。

```
char(10)
```

長さの指定がない場合 CHAR 列は、CHAR(1) 列になります。

CHAR および VARCHAR のデータ型は、文字単位でなくバイト単位で定義されます。CHAR 列にはシングルバイト文字のみを含めることができます。したがって、CHAR(10) 列には、最大 10 バイト長の文字列を含めることができます。

名前	ストレージ	範囲 (列の幅)
CHAR または CHARACTER	文字列の長さ。末尾の空白を含む (存在する場合)。	4096 バイト

VARCHAR または CHARACTER VARYING

一定の制限を持つ可変長の文字列を格納するには、VARCHAR 列または CHARACTER VARYING 列を使用します。これらの文字列は空白で埋められないので、VARCHAR(120) 列は、最大で 120 個のシングルバイト文字、60 個の 2 バイト文字、40 個の 3 バイト文字、または 30 個の 4 バイト文字で構成されます。

```
varchar(120)
```

VARCHAR データ型は、文字ではなくバイト単位で定義されます。VARCHAR にはマルチバイト文字 (1 文字あたり最大で 4 バイトまで) を含めることができます。例えば、VARCHAR(12) 列には、シングルバイト文字なら 12 個、2 バイト文字なら 6 個、3 バイト文字なら 4 個、4 バイト文字なら 3 個含めることができます。

名前	ストレージ	範囲 (列の幅)
VARCHAR または CHARACTER VARYING	4 バイト + 文字の合計バイト数 (ここで、各文字は 1 ~ 4 バイト)。	65535 バイト (64K -1)

末尾の空白の重要性

CHAR と VARCHAR のデータ型は両方とも、最大 n バイト長の文字列を格納できます。それよりも長い文字列をこれらの型の列に格納しようとすると、エラーが発生します。ただし、余分な文字がすべてスペース (空白) であれば、文字列は最大長に切り捨てられます。文字列の長さが最大長よりも短い場合、CHAR 値は空白で埋められますが、VARCHAR 値では空白なしで文字列を格納します。

CHAR 値の末尾の空白はいつも意味的に重要であるとは限りません。末尾の空白は、LENGTH 計算を含めないで 2 つの CHAR 値を比較するときは無視され、CHAR 値を別の文字列型に変換するときは削除されます。

VARCHAR および CHAR の値の末尾の空白は、値が比較されるとき、意味的に重要でないものとして扱われます。

長さの計算によって返される VARCHAR キャラクタ文字列の長さには末尾の空白が含められます。固定長のキャラクタ文字列の長さを計算する場合、末尾の空白はカウントされません。

日時型

日時データ型には、DATE、TIME、TIMESTAMP_LTZ、TIMESTAMP_NTZ などがあります。

トピック

- DATE

- TIMESTAMP_LTZ
- TIMESTAMP_NTZ
- 日時型を使用する例
- 日付、時刻、およびタイムスタンプのリテラル
- 間隔リテラル
- 間隔のデータ型とリテラル

DATE

タイムスタンプなしで単純にカレンダー日付だけを保存するには DATE データ型を使用します。

名前	ストレージ	範囲	解決策
DATE	4 バイト	4713 BC ~ 294276 AD	1 日

TIMESTAMP_LTZ

TIMESTAMP_LTZ データ型を使用して、日付、時刻、ローカルタイムゾーンを含む完全なタイムスタンプ値を保存します。

TIMESTAMP は、セッションローカルタイムゾーンを持つフィールド `year`、`month`、`day`、`second`、、`hourminute`および の値で構成される値を表します。`timestamp` 値は絶対時点を表します。

Spark の TIMESTAMP は、TIMESTAMP_LTZ および TIMESTAMP_NTZ バリエーションのいずれかに関連付けられたユーザー指定のエイリアスです。デフォルトのタイムスタンプタイプは、設定 を介して TIMESTAMP_LTZ (デフォルト値) または TIMESTAMP_NTZ として設定できます`spark.sql.timestampType`。

TIMESTAMP_NTZ

TIMESTAMP_NTZ データ型を使用して、日付、時刻を含む完全なタイムスタンプ値をローカルタイムゾーンなしで保存します。

TIMESTAMP は、フィールド `year`、、`month`、`day`、`minute`、および `hour`の値で構成される値を表します`second`。すべてのオペレーションは、タイムゾーンを考慮せずに実行されます。

Spark の TIMESTAMP は、TIMESTAMP_LTZ および TIMESTAMP_NTZ バリエーションのいずれかに関連付けられたユーザー指定のエイリアスです。デフォルトのタイムスタンプタイプは、設定 を介して TIMESTAMP_LTZ (デフォルト値) または TIMESTAMP_NTZ として設定できますspark.sql.timestampType。

日時型を使用する例

以下の例は、 AWS Clean Roomsでサポートされている日時型の使用方法を示しています。

日付の例

次の例では、形式が異なる複数の日付を挿入して、出力を表示します。

```
select * from datetable order by 1;

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

DATE 列にタイムスタンプ値を挿入した場合、時刻部分が無視され、日付のみロードされます。

時間の例

次の例では、形式の異なる複数の TIME および TIMETZ 値を挿入して、出力を表示します。

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

日付、時刻、およびタイムスタンプのリテラル

以下は、 AWS Clean Rooms Spark SQL でサポートされている日付、時刻、タイムスタンプリテラルを操作するためのルールです。

日付

次の表は、 AWS Clean Rooms テーブルにロードできるリテラル日付値の有効な例である入力日付を示しています。デフォルトの MDY DateStyle モードが有効であると想定されます。このモードでは、1999-01-08 や 01/02/00 などの文字列で月の値が日の値より前にあることを意味します。

> **Note**
>
> 日付またはタイムスタンプのリテラルをテーブルにロードするには、それらのリテラルを引用符で囲む必要があります。

入力日付	完全な日付
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
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 (日付の 3 桁部分は 001 ~ 366 である必要があります)

Times

次の表は、 AWS Clean Rooms テーブルにロードできるリテラル時間値の有効な例である入力時間を示しています。

入力時間	説明 (時刻部分)
04:05:06.789	午前 4 時 5 分 6.789 秒
04:05:06	午前 4 時 5 分 6 秒

入力時間	説明（時刻部分）
04:05	午前 4 時 5 分ちょうど
040506	午前 4 時 5 分 6 秒
午前 4 時 5 分	午前 4 時 5 分ちょうど、午前はオプション
午後 4 時 5 分	午後 4 時 5 分ちょうど。時値は 12 未満である必要があります。
16:05	午後 4 時 5 分ちょうど

特殊な日時値

次の表に示す特殊な値は、日時リテラルとして、および日付関数に渡す引数として使用できます。このような特殊な値は、一重引用符を必要とし、クエリの処理時に正規のタイムスタンプ値に変換されます。

特殊な値	説明
now	現在のトランザクションの開始時間に等しく、マイクロ秒の精度を持つタイムスタンプを返します。
today	該当する日付に等しく、時刻部分がゼロのタイムスタンプを返します。
tomorrow	該当する日付に等しく、時刻部分がゼロのタイムスタンプを返します。
yesterday	該当する日付に等しく、時刻部分がゼロのタイムスタンプを返します。

次の例は、 nowと が DATE_ADD 関数todayを操作する方法を示しています。

```
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)
```

間隔リテラル

以下は、Spark SQL で AWS Clean Rooms サポートされている間隔リテラルを操作するためのルールです。

`12 hours` または `6 weeks` など、特定の期間を識別するには、間隔リテラルを使用します。これらの間隔リテラルは、日時式を必要とする条件および計算の中で使用できます。

> **Note**
>
> AWS Clean Rooms テーブルの列に INTERVAL データ型を使用することはできません。

間隔は、INTERVAL キーワードに数量およびサポートされている日付部分を組み合わせて表現します。例えば、`INTERVAL '7 days'` または `INTERVAL '59 minutes'` のようになります。複数の数量および単位をつなぎ合わせることで、より正確な間隔を作成できます (例えば、`INTERVAL '7 days, 3 hours, 59 minutes'`)。各単位の省略形および複数形もサポートされています。例えば、`5 s`、`5 second`、および `5 seconds` は等価な間隔です。

日付部分を指定しない場合、間隔値は秒数を示します。数量値は小数として指定できます (例えば、`0.5 days`)。

例

次の例に、さまざまな間隔値を使用した一連の計算を示します。

以下の例は、指定された日付に 1 秒を追加します。

```
select caldate + interval '1 second' as dateplus from date
```

```
where caldate='12-31-2008';
dateplus
---------------------
2008-12-31 00:00:01
(1 row)
```

以下の例は、指定された日付に 1 分を追加します。

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

以下の例は、指定された日付に 3 時間と 35 分を追加します。

```
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)
```

以下の例は、指定された日付に 52 週を追加します。

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

以下の例は、指定された日付に 1 週、1 時間、1 分、および 1 秒を追加します。

```
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)
```

以下の例は、指定された日付に 12 時間 (半日) を追加します。

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

以下の例では、2023 年 3 月 31 日から 4 か月を引いて、結果が 2022 年 11 月 30 日になります。計算では、1 か月の日数を考慮します。

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

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

間隔のデータ型とリテラル

間隔のデータ型を使用して、seconds、minutes、hours、days、months、years などの単位で期間を保存できます。間隔のデータ型とリテラルは、日付とタイムスタンプへの間隔の追加、間隔の合計、日付またはタイムスタンプからの間隔の減算など、日時の計算に使用できます。間隔リテラルは、テーブル内の間隔データ型列への入力値として使用できます。

間隔データ型の構文

期間を年数と月数で保存する間隔データ型を指定するには:

```
INTERVAL year_to_month_qualifier
```

期間を日数、時間数、分数、および秒数で保存する間隔データ型を指定するには:

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

間隔リテラルの構文

間隔リテラルを指定して、期間を年数と月数で定義するには:

```
INTERVAL quoted-string year_to_month_qualifier
```

間隔リテラルを指定して、期間を日数、時間数、分数、および秒数で定義するには:

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

引数

quoted-string

数量と日時単位を入力文字列として指定する正または負の数値を指定します。quoted-string に数値のみが含まれている場合 AWS Clean Rooms 、 は year_to_month_qualifier または day_to_second_qualifier から単位を決定します。例えば、`'23' MONTH` は `1 year 11 months`、`'-2' DAY` は `-2 days 0 hours 0 minutes 0.0 seconds`、`'1-2' MONTH` は `1 year 2 months`、`'13 day 1 hour 1 minute 1.123 seconds' SECOND` は `13 days 1 hour 1 minute 1.123 seconds` を表します。間隔の出力形式の詳細については、「間隔スタイル」を参照してください。

year_to_month_qualifier

間隔の範囲を指定します。修飾子を使用して、修飾子より小さい時間単位の間隔を作成すると、は間隔の小さい部分 AWS Clean Rooms を切り捨てて破棄します。year_to_month_qualifier の有効な値は次のとおりです。

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

day_to_second_qualifier

間隔の範囲を指定します。修飾子を使用して、修飾子より小さい時間単位の間隔を作成すると、は間隔の小さい部分 AWS Clean Rooms を切り捨てて破棄します。day_to_second_qualifier の有効な値は次のとおりです。

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

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

INTERVAL リテラルの出力は、指定された最小の INTERVAL コンポーネントで切り捨てられます。たとえば、MINUTE 修飾子を使用する場合、 AWS Clean Rooms は MINUTE より小さい時間単位を破棄します。

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

結果の値は '1 day 01:01:00' で切り捨てられます。

Fractional_precision

間隔で使用できる小数点以下の桁数を指定するオプションのパラメータ。fractional_precision 引数は、間隔に SECOND が含まれている場合にのみ指定する必要があります。例えば、SECOND(3) は、1.234 秒など、小数点以下 3 桁のみを許可する間隔を作成します。小数点以下の最大桁数は 6 です。

セッション設定 interval_forbid_composite_literals は、YEAR TO MONTH と DAY TO SECOND の両方のパートを使用して間隔を指定している場合に、エラーを返すかどうかを決定します。

区間演算

間隔の値を他の日時値と一緒に使用して算術演算を実行できます。次の表は、使用可能な演算と、各演算の結果であるデータ型を示しています。

> **Note**
>
> date と timestamp の両方の結果を生成できる演算は、式に関連する最小時間単位に基づいて行われます。例えば、interval を date に追加すると、YEAR TO MONTH 間隔の場合、結果は date になり、DAY TO SECOND 間隔の場合、結果はタイムスタンプになります。

最初のオペランドが interval の演算は、指定された 2 番目のオペランドに対して次の結果を生成します。

オペレーター	日付	タイムスタンプ	間隔	数値
-	該当なし	該当なし	間隔	該当なし
+	日付	日付/タイムスタンプ	間隔	該当なし
*	該当なし	該当なし	該当なし	間隔
/	該当なし	該当なし	該当なし	間隔

最初のオペランドが date の演算は、指定された 2 番目のオペランドに対して次の結果を生成します。

オペレーター	日付	タイムスタンプ	間隔	数値
-	数値	間隔	日付/タイムスタンプ	日付
+	該当なし	該当なし	該当なし	該当なし

最初のオペランドが timestamp の演算は、指定された 2 番目のオペランドに対して次の結果を生成します。

オペレーター	日付	タイムスタンプ	間隔	数値
-	数値	間隔	タイムスタンプ	タイムスタンプ
+	該当なし	該当なし	該当なし	該当なし

間隔スタイル

- postgres — PostgreSQL スタイルに従います。これがデフォルトです。
- postgres_verbose — PostgreSQL の詳細スタイルに従います。

- sql_standard — SQL 標準間隔リテラルスタイルに従います。

次のコマンドは、間隔スタイルを sql_standard に設定します。

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

postgres 出力形式

postgres 間隔スタイルの出力形式は、次のとおりです。各数値は負の値にすることができます。

```
'<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 出力形式

postgres_verbose 構文は postgres と似ていますが、postgres_verbose 出力には時間の単位も含まれています。

```
'[@] <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 出力形式

間隔値 year to month は次のようにフォーマットされます。間隔の前にマイナス記号を指定すると、間隔が負の値になり、間隔全体に適用されます。

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

間隔値 day to second は次のようにフォーマットされます。

```
'[-]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
```

間隔データ型の例

以下の例は、テーブルでの INTERVAL データ型の使用方法を示しています。

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

間隔リテラルの例

以下の例は、間隔スタイルを postgres に設定して実行します。

次の例は、1 年の INTERVAL リテラルを作成する方法を示しています。

```
select INTERVAL '1' YEAR

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

修飾子を超える quoted-string を指定すると、残りの時間単位が間隔から切り捨てられます。次の例では、13 か月の間隔は 1 年と 1 か月になりますが、YEAR 修飾子により残りの 1 か月が除外されます。

```
select INTERVAL '13 months' YEAR

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

間隔文字列を下回る修飾子を使用すると、残った単位が含まれます。

```
select INTERVAL '13 months' MONTH
```

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

間隔で精度を指定すると、小数桁数が指定された精度まで切り捨てられます。

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

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

精度を指定しない場合、 は最大精度 6 AWS Clean Rooms を使用します。

```
select INTERVAL '1.23456789' SECOND

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

次の例では、範囲指定した間隔を作成する方法を示します。

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

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

修飾子は、指定する単位を指定します。たとえば、次の例では、前の例と同じ「2:2」の引用符で囲まれた文字列を使用していますが、 AWS Clean Rooms は修飾子のために異なる時間単位を使用していることを認識します。

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

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

各単位の略語と複数形もサポートされています。例えば、5s、5 second、5 seconds は同じ間隔です。サポートされている単位は、年、月、日、時間、分、秒です。

```
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
```

修飾子構文を使用しない間隔リテラルの例

> **Note**
>
> 以下の例は、YEAR TO MONTH または DAY TO SECOND 修飾子を使用しない間隔リテラルを示しています。修飾子を使用した間隔リテラルの推奨例については、「間隔のデータ型とリテラル」を参照してください。

12 hours または 6 months など、特定の期間を識別するには、間隔リテラルを使用します。これらの間隔リテラルは、日時式を必要とする条件および計算の中で使用できます。

間隔リテラルは、INTERVAL キーワードに数量およびサポートされている日付パートを組み合わせて表現します。例えば、INTERVAL '7 days' または INTERVAL '59 minutes' のようになります。複数の数量および単位をつなぎ合わせることで、より正確な間隔を作成できます (例えば、INTERVAL '7 days, 3 hours, 59 minutes')。各単位の省略形および複数形もサポートされています。例えば、5 s、5 second、および 5 seconds は等価な間隔です。

日付部分を指定しない場合、間隔値は秒数を示します。数量値は小数として指定できます (例えば、0.5 days)。

次の例に、さまざまな間隔値を使用した一連の計算を示します。

以下は、指定された日付に 1 秒を追加します。

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

以下は、指定された日付に 1 分を追加します。

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

以下では、指定された日付に 3 時間と 35 分を追加します。

```
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)
```

以下は、指定された日付に 52 週を追加します。

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

以下では、指定された日付に 1 週、1 時間、1 分、および 1 秒を追加します。

```
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)
```

以下は、指定された日付に 12 時間 (半日) を追加します。

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

以下では、2023 年 2 月 15 日から 4 か月を引いて、結果が 2022 年 10 月 15 日になります。

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

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

以下では、2023 年 3 月 31 日から 4 か月を引いて、結果が 2022 年 11 月 30 日になります。計算では、1 か月の日数を考慮します。

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

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

ブール型

シングルバイト列に true 値および false 値を格納するには、BOOLEAN データ型を使用します。次の表に、ブール値の取り得る 3 つの状態と、各状態をもたらすリテラル値について説明します。入力文字列に関係なく、ブール列では、true の場合は「t」を、false の場合は「f」を格納および出力します。

州	有効なリテラル値	ストレージ
True	TRUE 't' 'true' 'y' 'yes' '1'	1 バイト
False	FALSE 'f' 'false' 'n' 'no' '0'	1 バイト
不明	NULL	1 バイト

IS 比較を使用すると、ブール値を WHERE 句の述語としてのみチェックできます。SELECT リストのブール値に対して IS 比較を使用することはできません。

例

BOOLEAN 列を使用すれば、CUSTOMER テーブル内の顧客ごとに「アクティブ/非アクティブ」状態を格納できます。

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

この例では、以下のクエリによって、スポーツは好きだが映画は好きでないユーザーが USERS テーブルから選択されます。

```
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)
```

次の例では、ロックミュージックを好むかどうか不明なユーザーが USERS テーブルから選択されます。

```
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)
```

次の例では、SELECT リストで IS 比較を使用しているため、エラーが返されます。

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

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

次の例は、IS 比較ではなく SELECT リストで等号比較 (=) を使用しているため成功します。

```
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    |
```

ブールリテラル

次のルールは、Spark SQL で AWS Clean Rooms サポートされているブールリテラルを操作するためのものです。

ブールリテラルを使用して、 TRUEや などのブール値を指定しますFALSE。

構文

```
TRUE | FALSE
```

例

次の例は、指定された値が TRUE の列を示しています。

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

バイナリタイプ

BINARY データ型を使用して、解釈されていない固定長のバイナリデータを保存および管理し、特定のユースケースに効率的なストレージおよび比較機能を提供します。

BINARY データ型は、保存されるデータの実際の長さに関係なく、固定バイト数を保存します。通常、最大長は 255 バイトです。

BINARY は、画像、ドキュメント、その他のタイプのファイルなど、未解釈の未加工のバイナリデータを保存するために使用されます。データは、文字のエンコードや解釈を行うことなく、指定されたとおりに保存されます。BINARY 列に保存されているバイナリデータは、文字エンコーディングまたは照合ルールではなく、実際のバイナリ値に基づいてbyte-by-byte比較およびソートされます。

次のクエリ例は、文字列 のバイナリ表現を示しています"abc"。文字列の各文字は ASCII コードで 16 進形式で表されます。「a」は 0x61、「b」は 0x62、「c」は 0x63 です。組み合わせると、これらの 16 進値はバイナリ表現 を形成します"616263"。

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

ネスト型

AWS Clean Rooms は、ネストされたデータ型、特に AWS Glue STRUCT、ARRAY、MAP 列型を含むデータを含むクエリをサポートします。ネストされたデータ型をサポートするのはカスタム分析ルールのみです。

特に、ネストされたデータ型は、SQL データベースのリレーショナルデータモデルの厳密な表形式の構造に準拠していません。

ネストされたデータ型には、データ内の個別のエンティティを参照するタグが含まれます。配列やネストされた構造体、JSON などのシリアル化形式に関連付けられているその他の複雑な構造体などの複雑な値を含めることができます。ネストされたデータ型は、個々のネストされたデータ型のフィールドまたはオブジェクトで最大 1 MB のデータをサポートします。

トピック

- ARRAY タイプ
- MAP タイプ
- STRUCT タイプ
- ネストされたデータ型の例

ARRAY タイプ

ARRAY タイプを使用して、 タイプの一連の要素で構成される値を表しますelementType。

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

を使用してcontainsNull、ARRAY タイプの要素にnull値を含めることができるかどうかを示します。

MAP タイプ

MAP タイプを使用して、キーと値のペアのセットで構成される値を表します。

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

keyType: キーのデータ型

valueType: 値のデータ型

キーにnull値を含めることはできません。を使用してvalueContainsNull、MAP タイプの値にnull値を含めることができるかどうかを示します。

STRUCT タイプ

STRUCT タイプを使用して、一連の StructFields (フィールド) で記述された構造を持つ値を表します。

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

StructField(name、dataType、nullable): StructType のフィールドを表します。

dataType: データ型 a フィールド

name: フィールドの名前

を使用してnullable、これらのフィールドの値にnull値を含めることができるかどうかを示します。

ネストされたデータ型の例

struct<given:varchar, family:varchar> 型の場合、given と family という 2 つの属性名があり、それぞれが varchar 値に対応しています。

array<varchar> 型では、配列は varchar のリストとして指定されます。

array<struct<shipdate:timestamp, price:double>> 型は、struct<shipdate:timestamp, price:double> 型の要素のリストを参照します。

map データ型は、structs の array のように動作し、配列内の各要素の属性名は key で表され、その属性名が value にマップされます。

Example

例えば、map<varchar(20), varchar(20)> データ型は array<struct<key:varchar(20), value:varchar(20)>> として扱われ、key と value は、基になるデータ内のマップの属性を表しています。

が配列と構造へのナビゲーション AWS Clean Rooms を有効にする方法については、「」を参照してくださいナビゲーション。

がクエリの FROM 句を使用して配列を移動することで配列の反復 AWS Clean Rooms を有効にする方法については、「」を参照してくださいネストされていないクエリ。

型の互換性と変換

以下のトピックでは、 AWS Clean Rooms Spark SQL での型変換ルールとデータ型互換性の仕組みについて説明します。

トピック

- 互換性
- 互換性と変換に関する全般的なルール
- 暗黙的な変換型

互換性

データ型のマッチング、リテラル値および定数のデータ型とのマッチングは、以下のようなさまざまなデータベース操作で発生します。

- テーブルにおけるデータ操作言語 (DML) オペレーション
- UNION、INTERSECT、および EXCEPT のクエリ
- CASE 式

- LIKE や IN など、述語の評価
- データの比較または抽出を行う SQL 関数の評価
- 算術演算子との比較

これらの操作の結果は、型変換ルールおよびデータ型の互換性に左右されます。互換性は、特定の値と特定のデータ型との 1 対 1 のマッチングが必ずしも必要でないことを暗示しています。一部のデータ型は互換性があるため、暗黙変換、または強制が可能です。詳細については、「暗黙的な変換型」を参照してください。データ型に互換性がない場合は、明示変換関数を使用することにより、値をあるデータ型から別のデータ型に変換することが可能な場合があります。

互換性と変換に関する全般的なルール

次に示す互換性と変換に関するルールに注意してください。

- 一般に、同じデータ型のカテゴリに属するデータ型 (各種の数値データ型) は互換性があり、暗黙的に変換することができます。

  例えば、暗黙的な変換では、10 進値を整数列に変換できます。10 進値は整数に四捨五入されます。または、2008 のような数値を日付から抽出し、その値を整数列に挿入することができます。
- 数値データ型では、範囲外の値を挿入しようとしたときに発生するオーバーフロー条件を適用します。例えば、精度が 5 桁の 10 進値は、4 桁の精度で定義された 10 進列に適合しません。整数や 10 進数の全体が切り捨てられることはありませんが、10 進値の小数部分は、適宜、四捨五入される場合があります。ただし、テーブルから選択された値の明示的なキャストの結果は丸められません。
- 各種キャラクタ文字列型には互換性があります。シングルバイトデータを含む VARCHAR 列の文字列と CHAR 列の文字列は互換性があり、暗黙的に変換することができます。マルチバイトデータを含む VARCHAR 文字列には互換性がありません。また、キャラクタ文字列については、文字列が適切なリテラル値であれば、日付、時間、タイムスタンプ、または数値に変換できます。先頭と末尾のスペースはすべて無視されます。逆に、日付、時間、タイムスタンプ、または数値は、固定長または可変長の文字列に変換することができます。

> **Note**
>
> 数値型にキャストするキャラクタ文字列には、数字の文字表現が含まれている必要があります。例えば、文字列 '1.0' または '5.9' を 10 進値にキャストすることはできますが、文字列 'ABC' はいずれの数値型にもキャストできません。

- DECIMAL 値を文字列と比較すると、 AWS Clean Rooms は文字列を DECIMAL 値に変換しようとします。他のすべての数値と文字列を比較する場合、数値は文字列に変換されます。反対方向の変換 (文字列を整数に変換する、DECIMAL 値を文字列に変換するなど) を実行するには、CAST 関数などの明示的な関数を使用します。
- 64 ビットの DECIMAL または NUMERIC の値を上位の精度に変換するには、CAST や CONVERT などの明示的な変換関数を使用する必要があります。

暗黙的な変換型

暗黙的な変換には、2 つのタイプがあります。

- 割り当てにおける暗黙的な変換 (INSERT コマンドまたは UPDATE コマンドで値を設定するなど)
- 式における暗黙的な変換 (WHERE 句で比較を実行するなど)

次の表に、割り当てまたは式において暗黙的に変換できるデータ型を一覧表示します。これらの変換は、明示的な変換関数を使用して実行することもできます。

変換元の型	変換先の型
BIGINT	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER
	REAL (FLOAT4)
	SMALLINT または SHORT
	VARCHAR
CHAR	VARCHAR
DATE	CHAR

変換元の型	変換先の型
	VARCHAR
	TIMESTAMP
	TIMESTAMPTZ
DECIMAL (NUMERIC)	BIGINT または LONG
	CHAR
	DOUBLE PRECISION (FLOAT8)
	INTEGER (INT)
	REAL (FLOAT4)
	SMALLINT または SHORT
	VARCHAR
DOUBLE PRECISION (FLOAT8)	BIGINT または LONG
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	REAL (FLOAT4)
	SMALLINT または SHORT
	VARCHAR
INTEGER (INT)	BIGINT または LONG
	BOOLEAN
	CHAR

変換元の型	変換先の型
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	SMALLINT または SHORT
	VARCHAR
REAL (FLOAT4)	BIGINT または LONG
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	SMALLINT または SHORT
	VARCHAR
SMALLINT	BIGINT または LONG
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER (INT)
	REAL (FLOAT4)
	VARCHAR
TIME	VARCHAR

変換元の型	変換先の型
	TIMETZ

> **Note**
>
> DATE、TIME、TIMESTAMP_LTZ、TIMESTAMP_NTZ、または文字列間の暗黙的な変換では、現在のセッションタイムゾーンが使用されます。
> VARBYTE データ型は、暗黙的に他のデータ型に変換できません。詳細については、「CAST 関数」を参照してください。

AWS Clean Rooms Spark SQL コマンド

Spark SQL では、 AWS Clean Rooms 次の SQL コマンドがサポートされています。

トピック

- キャッシュテーブル
- ヒント
- SELECT

キャッシュテーブル

CACHE TABLE コマンドは、既存のテーブルのデータをキャッシュするか、クエリ結果を含む新しいテーブルを作成してキャッシュします。

> **Note**
>
> キャッシュされたデータはクエリ全体で保持されます。

構文、引数、およびいくつかの例は、Apache Spark SQL リファレンスから取得されます。

構文

CACHE TABLE コマンドは、次の 3 つの構文パターンをサポートしています。

> **許可されていない出力列のクエリ制約と CACHE TABLE**
>
> カスタム分析ルールで許可されていない出力列の制約は、キャッシュされたテーブルに適用されます。キャッシュされたテーブルは、SELECT 句で許可されていない出力列を参照できません。クエリの後続の部分で許可されていない出力列制約を持つ列を使用するには、キャッシュされたテーブルを共通テーブル式 (CTE) に変換します。

AS (括弧なし): クエリ結果に基づいて新しいテーブルを作成してキャッシュします。

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

AS と括弧の場合: 関数は最初の構文と似ていますが、括弧を使用してクエリを明示的にグループ化します。

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

AS なし: SELECT ステートメントを使用してキャッシュする行をフィルタリングし、既存のテーブルをキャッシュします。

```
CACHE TABLE cache_table_identifier query;
```

コードの説明は以下のとおりです。

- すべてのステートメントはセミコロン (;) で終わる必要があります
- query は通常、SELECT ステートメントです。
- AS では、クエリの周囲の括弧はオプションです。
- AS キーワードはオプションです

パラメータ

cache_table_identifier

キャッシュされたテーブルの名前。オプションのデータベース名修飾子を含めることができます。

AS

クエリ結果から新しいテーブルを作成してキャッシュするときに使用されるキーワード。

query

キャッシュするデータを定義する SELECT ステートメントまたはその他のクエリ。

例

次の例では、キャッシュされたテーブルはクエリ全体に対して保持されます。キャッシュ後、*cache_table_identifier* を参照する後続のクエリは、*sourceTable* から再計算または読み取りを行うのではなく、キャッシュされたバージョンから読み取ります。これにより、頻繁にアクセスされるデータのクエリパフォーマンスを向上させることができます。

クエリ結果からフィルタリングされたテーブルを作成してキャッシュする

最初の例は、クエリ結果から新しいテーブルを作成してキャッシュする方法を示しています。このコマンドは、SELECTステートメントの周囲に括弧を付けずに ASキーワードを使用します。ステータスがcache_table_identifier「」である「」の行のみを含むsourceTable「」という名前の新しいテーブルが作成されますactive'。クエリを実行し、結果を新しいテーブルに保存し、新しいテーブルの内容をキャッシュします。キャッシュされたデータを使用するには、元の 'sourceTable' は変更されず、後続のクエリは 'cache_table_identifier' を参照する必要があります。

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

括弧で囲まれた SELECT ステートメントを使用してクエリ結果をキャッシュする

2 番目の例は、 SELECT ステートメントの周囲に括弧を使用して、クエリの結果を指定された名前 (cache_table_identifier) の新しいテーブルとしてキャッシュする方法を示しています。このコマンドは、ステータスがcache_table_identifier「」である「」の行のみを含むsourceTable「」という名前の新しいテーブルを作成しますactive'。クエリを実行し、結果を新しいテーブルに保存し、新しいテーブルの内容をキャッシュします。元の 'sourceTable' は変更されません。キャッシュされたデータを使用するには、後続のクエリでcache_table_identifier「」を参照する必要があります。

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

フィルター条件を使用して既存のテーブルをキャッシュする

3 番目の例は、別の構文を使用して既存のテーブルをキャッシュする方法を示しています。この構文は、'AS' キーワードと括弧を省略し、通常、新しいテーブルを作成するのではなく、'cache_table_identifier' という名前の既存のテーブルから指定された行をキャッシュします。SELECT ステートメントは、キャッシュする行を決定するフィルターとして機能します。

> **Note**
>
> この構文の正確な動作は、データベースシステムによって異なります。必ず、特定の AWS サービスに適した構文を確認してください。

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

ヒント

SQL 分析のヒントは、 のクエリ実行戦略をガイドする最適化ディレクティブを提供するため AWS Clean Rooms、クエリのパフォーマンスを向上させ、コンピューティングコストを削減できます。ヒントは、Spark 分析エンジンが実行計画を生成する方法を提案します。

構文

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

ヒントは、コメント形式の構文を使用して SQL クエリに埋め込まれ、SELECT キーワードの直後に配置する必要があります。

サポートされているヒントタイプ

AWS Clean Rooms は、結合ヒントとパーティション化ヒントの 2 つのカテゴリのヒントをサポートしています。

トピック

- 結合ヒント
- パーティション分割のヒント

結合ヒント

結合ヒントは、クエリ実行の結合戦略を提案します。構文、引数、およびいくつかの例は、Apache Spark SQL リファレンスから入手できます。

ブロードキャスト

ブロードキャスト結合 AWS Clean Rooms を使用することをお勧めします。ヒントを含む結合側は、autoBroadcastJoinThreshold に関係なくブロードキャストされます。結合の両側にブロードキャストヒントがある場合、サイズが小さいもの (統計に基づく) がブロードキャストされます。

エイリアス: BROADCASTJOIN、MAPJOIN

パラメータ: テーブル識別子 (オプション)

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

シャッフルソートマージ結合 AWS Clean Rooms を使用することをお勧めします。

エイリアス: SHUFFLE_MERGE、MERGEJOIN

パラメータ: テーブル識別子 (オプション)

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;
```

SHUFFLE_HASH

シャッフルハッシュ結合 AWS Clean Rooms の使用を提案します。両側にシャッフルハッシュヒントがある場合、クエリオプティマイザはビルド側として小さい側 (統計に基づく) を選択します。

パラメータ: テーブル識別子 (オプション)

例:

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

SHUFFLE_REPLICATE_NL

ネストされたループ結合にshuffle-and-replicate AWS Clean Rooms を使用することをお勧めします。

パラメータ: テーブル識別子 (オプション)

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 のヒントのトラブルシューティング

次の表は、SparkSQL でヒントが適用されない一般的なシナリオを示しています。詳細については、「the section called “考慮事項と制限事項”」を参照してください。

ユースケース	クエリの例
テーブルリファレンスが見つかりません	`SELECT /*+ BROADCAST(fake_table) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

ユースケース	クエリの例
結合オペレーションに参加していないテーブル	`SELECT /*+ BROADCAST(s) */ * FROM students s WHERE s.age > 25;`
ネストされたサブクエリのテーブルリファレンス	`SELECT /*+ BROADCAST(s) */ * FROM employees e INNER JOIN (SELECT * FROM students s WHERE s.age > 20) sub ON e.eid = sub.sid;`
テーブル参照の代わりに列名	`SELECT /*+ BROADCAST(e.eid) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
必須パラメータのないヒント	`SELECT /*+ BROADCAST */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
テーブルエイリアスではなくベーステーブル名	`SELECT /*+ BROADCAST(employees) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

パーティション分割のヒント

パーティショニングヒントは、エグゼキューターノード間のデータ分散を制御します。複数のパーティショニングヒントを指定すると、複数のノードが論理プランに挿入されますが、左端のヒントはオプティマイザによって選択されます。

COALESCE

パーティションの数を指定されたパーティションの数に減らします。

パラメータ: 数値 (必須) - 1 ~ 2147483647 の正の整数である必要があります

例:

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

REPARTITION

指定されたパーティショニング式を使用して、指定された数のパーティションにデータを再パーティション化します。ラウンドロビン分散を使用します。

パラメータ :

- 数値 (オプション) - パーティションの数。1 ~ 2147483647 の正の整数である必要があります
- 列識別子 (オプション) - パーティション分割する列。これらの列は入力スキーマに存在する必要があります。
- 両方を指定した場合、数値が最初に来る必要があります

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;
```

REPARTITION_BY_RANGE

指定された列で範囲パーティショニングを使用して、指定された数のパーティションにデータを再パーティション化します。

パラメータ :

- 数値 (オプション) - パーティションの数。1～2147483647 の正の整数である必要があります
- 列識別子 (オプション) - パーティション分割する列。これらの列は入力スキーマに存在する必要があります。
- 両方を指定した場合、数値が最初に来る必要があります

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;
```

バランス

クエリ結果の出力パーティションのバランスを再調整して、すべてのパーティションが適切なサイズになるようにします (小さすぎず、大きすぎません) 。これはベスト工フォートオペレーションです。スキューがある場合、スキューされたパーティションを AWS Clean Rooms 分割して大きくなりすぎないようにします。このヒントは、ファイルが小さすぎたり大きすぎたりしないように、クエリの結果をテーブルに書き込む必要がある場合に役立ちます。

パラメータ :

- 数値 (オプション) - パーティションの数。1～2147483647 の正の整数である必要があります
- 列識別子 (オプション) - 列は SELECT 出力リストに表示される必要があります
- 両方を指定した場合、数値が最初に来る必要があります

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;
```

複数のヒントを組み合わせる

1 つのクエリで複数のヒントを指定するには、カンマで区切ります。

```
-- 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;
```

考慮事項と制限事項

- ヒントは最適化の提案であり、コマンドではありません。クエリオプティマイザは、リソースの制約または実行条件に基づいてヒントを無視することがあります。
- ヒントは、CreateAnalysisTemplate API と StartProtectedQuery APIs。
- ヒントは SELECT キーワードの直後に配置する必要があります。
- 名前付きパラメータはヒントではサポートされず、例外がスローされます。
- REPARTITION amd REPARTITION_BY_RANGE ヒントの列名は、入力スキーマに存在する必要があります。
- REBALANCE ヒントの列名は SELECT 出力リストに表示される必要があります。

- 数値パラメータは 1～2147483647 の正の整数である必要があります。1e1 のような科学的表記はサポートされていません
- ヒントは、差分プライバシー SQL クエリではサポートされていません。
- SQL クエリのヒントは、PySpark ジョブではサポートされていません。PySpark ジョブで実行プランのディレクティブを提供するには、データフレーム API を使用します。詳細については、「Apache Spark DataFrame API Docs」を参照してください。

SELECT

SELECT コマンドは、テーブルおよびユーザー定義関数から行を返します。

AWS Clean Rooms Spark SQL では、次の SELECT SQL コマンド、句、およびセット演算子がサポートされています。

トピック

構文、引数、およびいくつかの例は、Apache Spark SQL リファレンスから取得されます。

SELECT list

SELECT list は、クエリに返させる列、関数、および式を指定します。このリストは、クエリの出力を表しています。

構文

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

パラメータ

DISTINCT

1 つまたは複数の列の一致する値に基づいて、結果セットから重複する行を削除するオプション。

expression

クエリによって参照されるテーブル内に存在する 1 つまたは複数の列から構成される式。式には、SQL 関数を含めることができます。例えば、次のようになります。

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

AS column_alias

最終的な結果セットに使われる列のテンポラリ名。AS キーワードはオプションです。例えば、次のようになります。

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

シンプルな列名ではない式に対して、エイリアスを指定しない場合、結果セットはその列に対してデフォルト名を適用します。

> **Note**
>
> エイリアスは、ターゲットリストで定義された直後に認識されます。同じターゲットリストの後に定義された他の式でエイリアスを使用することはできません。

WITH 句

WITH 句は、クエリ内の SELECT リストに先行するオプション句です。WITH 句は、1 つまたは複数の common_table_expressions を定義します。各共通テーブル式 (CTE) は、ビュー定義に似ている

一時テーブルを定義します。これらの一時テーブルは、FROM 句で参照できます。それらは、所属するクエリが実行されている間にのみ使用されます。WITH 句内の各 CTE は、テーブル名、列名のオプションリスト、およびテーブルに対して評価を実行するクエリ表現 (SELECT ステートメント) を指定します。

WITH 句のサブクエリは、単一のクエリ実行中に、使用可能なテーブルを効率的に定義します。SELECT ステートメントの本文内でサブクエリを使用することで、すべてのケースで同じ結果を実現できますが、WITH 句のサブクエリの方が、読み書きが簡単になることがあります。可能な場合は、複数回参照される、WITH 句のサブクエリは、一般的な副次式として最適化されます。つまり、一度 WITH サブクエリを評価すると、その結果を再利用することができるということです。(一般的な副次式は、WITH 句内で定義される副次式に制限されない点に注意してください。)

構文

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

common_table_expression は、非再帰的にすることもできます。非再帰形式は次のとおりです。

```
CTE_table_name AS ( query )
```

パラメータ

common_table_expression

FROM 句 で参照できる一時テーブルを定義し、それが属するクエリの実行中にのみ使用されます。

CTE_table_name

WITH 句のサブクエリの結果を定義する一時テーブルの一意な名前。単一の WITH 句内で重複する名前を使用することはできません。各サブクエリには、FROM 句 で参照可能なテーブル名を付ける必要があります。

query

が AWS Clean Rooms サポートする SELECT クエリ。「SELECT」を参照してください。

使用に関する注意事項

次の SQL ステートメントで WITH 句を使用できます。

- SELECT、 WITH、UNION、UNION ALL、INTERSECT、INTERSECT ALL、EXCEPT、または EXCEPT ALL

WITH 句を含んでいるクエリの FROM 句が、WITH 句によって定義されたテーブルを参照していない場合、含まれている WITH 句は無視された上でクエリは通常どおり実行されます。

WITH 句のサブクエリで定義されたテーブルは、WITH 句が開始した SELECT クエリの範囲でのみ参照可能です。例えば、このようなテーブルは、SELECT リスト、WHERE 句、または HAVING 句内のサブクエリの FROM 句で参照できます。サブクエリ内で WITH 句を使用し、メインクエリまたは別のサブクエリの FROM 句でそのテーブルを参照することはできません。このクエリパターンを使用すると、WITH 句のテーブルに対して、`relation table_name doesn't exist`という形式のエラーメッセージが発生します。

WITH 句のサブクエリ内で、別の WITH 句を指定することはできません。

WITH 句のサブクエリによって定義されたテーブルに対して、前方参照を作成することはできません。例えば次のクエリでは、テーブル W1 の定義内でテーブル W2 への前方参照を設定しているため、エラーが帰されます。

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

例

次の例では、WITH 句を含む最もシンプルなケースを示します。VENUECOPY という名前の WITH クエリは、VENUE テーブルからすべての行を選択します。次にメインクエリでは、VENUECOPY からすべての行を選択します。VENUECOPY テーブルは、このクエリの有効期間中だけ存在します。

```
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)
```

次の例では、VENUE_SALES と TOP_VENUES という名前の 2 つのテーブルを生成する WITH 句を示します。2 番目の WITH 句テーブルは最初のテーブルから選択します。次に、メインクエリブロックの WHERE 句には、TOP_VENUES テーブルを制約するサブクエリが含まれています。

```
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)
```

次の 2 つの例は、WITH 句サブクエリに基づいた、テーブル参照の範囲に関するルールをデモンストレーションしています。最初のクエリは実行されますが、2 番目のクエリは予想どおりのエラーが発生して失敗します。最初のクエリには、メインクエリの SELECT リスト内に WITH 句サブクエリが存在します。WITH 句によって定義されるテーブル (HOLIDAYS) は、SELECT リストのサブクエリの FROM 句で参照されます。

```
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)
```

2 番目のクエリは SELECT リストのサブクエリ内だけでなく、メインクエリ内の HOLIDAYS テーブルを参照しようとしたため、失敗しました。メインクエリの参照は範囲外です。

```
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 句

クエリ内の FROM 句は、データの選択下のテーブル参照 (テーブル、ビュー、サブクエリ) を一覧表示します。複数のテーブル参照が一覧表示されている場合、FROM 句または WHERE 句のいずれかの適切な構文を使って、テーブル参照を結合する必要があります。結合基準を指定していない場合、クエリはクロス結合 (デカルト積) として処理されます。

トピック

- 構文
- パラメータ
- 使用に関する注意事項

構文

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

ここで table_reference は、次のいずれかになります。

```
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
```

パラメータ

with_subquery_table_name

WITH 句 のサブクエリで定義されるテーブル。

table_name

テーブルまたはビューの名前。

alias

テーブルまたはビューの一時的な代替名。エイリアスは、サブクエリから生成されたテーブルに対して提供する必要があります。他のテーブル参照では、エイリアスはオプションです。AS

キーワードは常にオプションです。テーブルエイリアスは、WHERE 句など、クエリの別の部分のテーブルを識別するため、便利なショートカットを提供します。

例えば、次のようになります。

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

テーブルエイリアスを定義した場合、クエリでそのテーブルを参照するにはエイリアスを使用する必要があります。

例えば、クエリが SELECT "tbl"."col" FROM "tbl" AS "t" の場合、テーブル名は基本的に上書きされているため、クエリは失敗します。この場合の有効なクエリは SELECT "t"."col" FROM "tbl" AS "t" です。

column_alias

テーブルまたはビュー内の列に対する一時的な代替名。

subquery

テーブルに対して評価を実行するクエリ式。テーブルは、クエリの有効期間中のみ存在し、通常は名前またはエイリアスが与えられます。ただし、エイリアスは必須ではありません。また、サブクエリから生成されたテーブルに対して、列名を定義することもできます。サブクエリの結果を他のテーブルに結合する場合、および列をクエリ内のどこかで選択または拘束する場合、列のエイリアスの命名は重要です。

サブクエリには ORDER BY 句が含まれることがありますが、LIMIT または OFFSET 句も併せて指定しない場合、この句には効力がありません。

NATURAL

2 つのテーブル内で同じ名前を付けられた列のペアをすべて結合列として、自動的に使用する結合を定義します。明示的な結合条件は必要ありません。例えば、CATEGORY と EVENT の両方のテーブルに CATID という名前の列が存在する場合、これらのテーブルの NATURAL 結合は CATID 列による結合です。

> **Note**
>
> NATURAL 結合を指定しても、結合対象のテーブルに同じ名前の列ペアが存在しない場合、クエリはデフォルト設定のクロス結合になります。

join_type

以下のいずれかの結合タイプを指定します。

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

クロス結合は非限定の結合で、2 つの表のデカルト積を返します。

内部結合と外部結合は限定結合です。これらの結合は、FROM 句の ON または USING 構文、または WHERE 句条件を使った (Natural 結合での) 黙示的な結合です。

内部結合は、結合条件、また結合列のリストに基づいて、一致する行だけを返します。外部結合は、同等の内部結合が返すすべての行に加え、「左側の」表、「右側の」表、または両方の表から一致しない行を返します。左の表は最初に一覧表示された表で、右の表は 2 番目に一覧表示された表です。一致しない行には、出力列のギャップを埋めるため、NULL が含まれます。

ON join_condition

結合列を ON キーワードに続く条件として記述する、結合タイプの指定。次に例を示します。

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

USING (join_column [, ...])

結合列をかっこで一覧表示する結合の指定タイプ。複数の結合列を指定する場合、カンマによって区切ります。USING キーワードは、リストより前に指定する必要があります。例:

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

使用に関する注意事項

列を結合するには、データ型に互換性がなければなりません。

NATURAL または USING 結合は、中間結果セットの結合列の各ペアの一方だけを保持します。

ON 構文を使った結合は、中間結果セットの両方の結合列を保持します。

WITH 句 も参照してください。

JOIN 句

SQL JOIN 句は、共通のフィールドに基づいて 2 つ以上のテーブルのデータを結合するために使用されます。結果は、指定した結合方法によって変わる場合もあります。左と右の外部結合は、もう一方のテーブルに一致するものが見つからない場合に、結合したどちらかのテーブルの値を保持します。

JOIN タイプと結合条件の組み合わせによって、最終結果セットに含まれる行が決まります。SELECT 句と WHERE 句は、返される列と行のフィルタリング方法を制御します。さまざまな JOIN タイプとそれらを効果的に使用する方法を理解することは、複数のテーブルからのデータを柔軟かつ強力な方法で組み合わせることができるため、SQL の重要なスキルです。

構文

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

パラメータ

SELECT column1、column2、...、columnN

結果セットに含める列。JOIN に関係するテーブルのいずれかまたは両方から列を選択できます。

FROM テーブル 1

JOIN オペレーションの最初の (左) テーブル。

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

実行する JOIN のタイプ。JOIN または INNER JOIN は、両方のテーブルで一致する値を持つ行のみを返します。

LEFT [OUTER] JOIN は、左側のテーブルからすべての行を返し、右側のテーブルから一致する行を返します。

RIGHT [OUTER] JOIN は、右側のテーブルからすべての行を返し、左側のテーブルから一致する行を返します。

FULL [OUTER] JOIN は、一致があるかどうかにかかわらず、両方のテーブルのすべての行を返します。

CROSS JOIN は、2 つのテーブルから行のデカルト積を作成します。

ON table1.column = table2.column

2 つのテーブルの行の一致方法を指定する結合条件。結合条件は、1 つ以上の列に基づくことができます。

WHERE 条件:

指定された条件に基づいて、結果セットをさらにフィルタリングするために使用できるオプションの句。

例

次の例は、USING 句が含まれている 2 つのテーブル間の結合です。この場合、listid と eventid の各列が結合列として使用されます。結果は 5 行に制限されます。

```
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
```

結合の種類

INNER

これはデフォルトの結合タイプです。両方のテーブル参照で一致する値を持つ行を返します。

INNER JOIN は、SQL で使用される結合の最も一般的なタイプです。これは、共通の列または列のセットに基づいて複数のテーブルのデータを組み合わせる強力な方法です。

構文:

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

次のクエリは、顧客と注文テーブルの間に一致する customer_id 値があるすべての行を返します。結果セットには、customer_id、name、order_id、および order_date 列が含まれます。

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

次のクエリは、LISTING テーブルと SALES テーブル間の内部結合です (JOIN キーワードを除く)。ここで、LISTING テーブルの LISTID は 1~5 です。このクエリは、LISTING テーブル (左のテーブル) と SALES テーブル (右のテーブル) の LISTID 列の値と一致します。結果は、LISTID 1、4、および 5 が基準に一致することを示しています。

```
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
```

次の例は、ON 句を含む内部結合です。この場合、NULL 行は返されません。

```
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
```

次のクエリは、FROM 句の 2 つのサブクエリを内部結合したものです。このクエリは、イベント (コンサートとショー) の異なるカテゴリのチケットの販売数と売れ残り数を検出します。この FROM 句サブクエリはテーブルサブクエリです。これらは、複数の列と行を返すことができます。

```
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 [外部]

左のテーブルリファレンスからすべての値と右のテーブルリファレンスから一致した値を返します。一致しない場合は NULL を追加します。左外部結合とも呼ばれます。

左 (1 番目) テーブルのすべての行と右 (2 番目) テーブルの一致する行を返します。右側のテーブルに一致がない場合、結果セットには右側のテーブルの列の NULL 値が含まれます。OUTER キーワードは省略でき、結合は単純に LEFT JOIN として記述できます。LEFT OUTER JOIN の反対は RIGHT OUTER JOIN で、右テーブルのすべての行と左テーブルの一致する行を返します。

構文:

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

次のクエリは、顧客テーブルのすべての行と、注文テーブルの一致する行を返します。顧客が注文がない場合でも、結果セットにはその顧客の情報と order_id 列と order_date 列の NULL 値が含まれます。

```
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;
```

次のクエリは左外部結合です。左と右の外部結合は、もう一方のテーブルに一致するものが見つからない場合に、結合したどちらかのテーブルの値を保持します。左のテーブルと右のテーブルは、構文に一覧表示されている最初と 2 番目のテーブルです。NULL 値は、結果セットの「ギャップ」を埋めるために使われます。このクエリは、LISTING テーブル (左のテーブル) と SALES テーブル (右のテーブル) の LISTID 列の値と一致します。結果は、LISTIDs 2 と 3 が売上につながっていないことを示しています。

```
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]

右側のテーブルリファレンスからすべての値と左側のテーブルリファレンスから一致した値を返します。一致しない場合は NULL を追加します。右外部結合とも呼ばれます。

右 (秒) テーブルのすべての行と、左 (最初の) テーブルの一致する行が返されます。左のテーブルに一致がない場合、結果セットには左のテーブルの列の NULL 値が含まれます。OUTER キーワードは省略でき、結合は単に RIGHT JOIN として記述できます。RIGHT OUTER JOIN の反対は LEFT OUTER JOIN で、左側のテーブルのすべての行と右側のテーブルの一致する行を返します。

構文:

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

次のクエリは、顧客テーブルのすべての行と、注文テーブルの一致する行を返します。顧客が注文がない場合でも、結果セットにはその顧客の情報と order_id 列と order_date 列の NULL 値が含まれます。

```
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;
```

次のクエリは右外部結合です。このクエリは、LISTING テーブル (左のテーブル) と SALES テーブル (右のテーブル) の LISTID 列の値と一致します。結果は、LISTID 1、4、および 5 が基準に一致することを示しています。

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

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

FULL [外部]

両方のリレーションからすべての値を返します。一致しない側に NULL 値を追加します。完全外部結合とも呼ばれます。

一致するかどうかにかかわらず、左テーブルと右テーブルの両方からすべての行が返されます。一致する行がない場合、結果セットには、一致する行がないテーブルの列の NULL 値が含まれます。OUTER キーワードは省略でき、結合は単に FULL JOIN として記述できます。FULL OUTER JOIN は、LEFT OUTER JOIN や RIGHT OUTER JOIN よりも一般的には使用されませんが、一致するものがない場合でも、両方のテーブルのすべてのデータを表示する必要がある特定のシナリオで役立ちます。

構文:

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

次のクエリは、顧客テーブルと注文テーブルの両方のすべての行を返します。顧客が注文がない場合でも、結果セットにはその顧客の情報と order_id 列と order_date 列の NULL 値が含まれます。注文に関連付けられた顧客がない場合、結果セットにはその注文が含まれ、customer_id 列と名前列の値が NULL になります。

```
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;
```

次のクエリは完全結合です。完全結合は、もう一方のテーブルに一致するものが見つからない場合に、結合したテーブルの値を保持します。左のテーブルと右のテーブルは、構文に一覧表示されている最初と 2 番目のテーブルです。NULL 値は、結果セットの「ギャップ」を埋めるために使われます。このクエリは、LISTING テーブル (左のテーブル) と SALES テーブル (右のテーブル) の LISTID 列の値と一致します。結果は、LISTIDs 2 と 3 が売上につながっていないことを示しています。

```
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
```

次のクエリは完全結合です。このクエリは、LISTING テーブル (左のテーブル) と SALES テーブル (右のテーブル) の LISTID 列の値と一致します。セールスがない行 (LISTID 2 および 3) のみが結果に表示されます。

```
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] セミ

右側と一致するテーブルリファレンスの左側から値を返します。左半結合とも呼ばれます。

右 (秒) テーブルに一致する行を持つ左 (最初の) テーブルの行のみを返します。右側のテーブルの列は返されません。左側のテーブルの列のみが返されます。LEFT SEMI JOIN は、2 番目のテーブルからデータを返すことなく、1 つのテーブルで一致する行を別のテーブルで検索する場合に便利です。LEFT SEMI JOIN は、IN 句または EXISTS 句でサブクエリを使用するよりも効率的な方法です。

構文:

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

次のクエリは、注文テーブルに少なくとも 1 つの注文がある顧客について、顧客テーブルの customer_id 列と名前列のみを返します。結果セットには、注文テーブルの列は含まれません。

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

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

CROSS JOIN

2 つのリレーションのデカルト積を返します。つまり、結果セットには、条件やフィルターを適用せずに、2 つのテーブルの行の可能な組み合わせがすべて含まれます。

CROSS JOIN は、顧客情報と製品情報の組み合わせをすべて表示するレポートを作成する場合など、2 つのテーブルからデータの組み合わせをすべて生成する必要がある場合に役立ちます。CROSS JOIN は、ON 句に結合条件がないため、他の結合タイプ (INNER JOIN、LEFT JOIN など) とは異なります。結合条件は CROSS JOIN には必要ありません。

構文:

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

次のクエリは、顧客テーブルと製品テーブルからの customer_id、customer_name、product_id、product_name の可能なすべての組み合わせを含む結果セットを返します。顧客テーブルに 10 行があり、製品テーブルに 20 行がある場合、CROSS JOIN の結果セットには 10 x 20 = 200 行が含まれます。

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

次のクエリは、LISTING テーブルと SALES テーブルのクロス結合またはデカルト結合で、結果を制限する述語が使用されています。このクエリは、両方のテーブルの LISTID 1、2、3、4、および 5 について、SALES テーブルと LISTING テーブルの LISTID 列の値と一致します。結果は、20 行が基準に一致することを示しています。

```
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
```

アンチ結合

左のテーブルリファレンスから、右のテーブルリファレンスと一致しない値を返します。左アンチ結合とも呼ばれます。

ANTI JOIN は、あるテーブルで一致する行を別のテーブルで検索する場合に便利です。

構文:

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

次のクエリは、注文していないすべての顧客を返します。

```
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

2 つのリレーションの行が、一致する名前を持つすべての列の等価で暗黙的に一致することを指定します。

2 つのテーブル間で同じ名前とデータ型の列が自動的に照合されます。ON 句で結合条件を明示的に指定する必要はありません。2 つのテーブル間の一致するすべての列を結果セットに結合します。

NATURAL JOIN は、結合するテーブルに同じ名前とデータ型の列がある場合に便利です。ただし、一般的には、より明示的な INNER JOIN を使用することをお勧めします。結合条件をより明確でわかりやすいものにする ON 構文。

構文:

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

次の例は、次の列を持つ employeesと の departments2 つのテーブル間の自然な結合です。

- employees テーブル: employee_id、first_name、last_name、 department_id
- departments テーブル: department_id、 department_name

次のクエリは、department_id列に基づいて、2 つのテーブル間で一致するすべての行の名、姓、部門名を含む結果セットを返します。

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

次の例は、2 つのテーブル間の自然結合です。この場合、listid、sellerid、eventid、および dateid の各列は、両方のテーブルで同じ名前とデータ型を持つため、結合列として使用されます。結果は 5 行に制限されます。

```
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 句

WHERE 句には、テーブルの結合またはテーブル内の列への述語の適用のいずれかを実行する条件が含まれています。テーブルは、WHERE 句または FROM 句のいずれかの適切な構文を使うことで結合できます。外部結合基準は、FROM 句で指定する必要があります。

構文

```
[ WHERE condition ]
```

condition

結合条件やテーブル列に関する述語など、ブール値結果に関する検索条件 次の例は有効な join の条件です。

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

次の例は、テーブル内の列の有効な条件です。

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

条件は単純にすることも複雑することもできます。複雑な条件の場合、かっこを使って、論理ユニットを分離します。次の例では、結合条件をかっこによって囲みます。

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

使用に関する注意事項

WHERE 句でエイリアスを使って、SELECT リスト式を参照することができます。

WHERE 句内の集計関数の結果を制限することはできません。その目的には、HAVING 句を使用してください。

WHERE 句内で制限されている列は、FROM 句内のテーブル参照から生成する必要があります。

例

次のクエリは、SALES テーブルと EVENT テーブルの結合条件、EVENTNAME 列に関する述語、STARTTIME 列に関する 2 つの述語など、複数の WHERE 句制限の組み合わせを使用します。

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

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

VALUES 句

VALUES 句は、テーブルを参照することなく、クエリ内で一連の行値を直接提供するために使用されます。

VALUES 句は、次のシナリオで使用できます。

- INSERT INTO ステートメントで VALUES 句を使用して、テーブルに挿入される新しい行の値を指定できます。
- VALUES 句を単独で使用して、テーブルを参照することなく、一時的な結果セットまたはインラインテーブルを作成できます。
- VALUES 句を WHERE、ORDER BY、LIMIT などの他の SQL 句と組み合わせて、結果セットの行をフィルタリング、ソート、または制限できます。

この句は、永続的なテーブルを作成または参照することなく、SQL ステートメントで小さなデータセットを直接挿入、クエリ、または操作する必要がある場合に特に便利です。これにより、各行の列名と対応する値を定義できるため、個別のテーブルを管理するオーバーヘッドなしで、一時的な結果セットを作成したり、データをその場で挿入したりできます。

構文

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

パラメータ

expression

値を生成する 1 つ以上の値、演算子、SQL 関数の組み合わせを指定する式。

table_alias

オプションの列名リストで一時名を指定するエイリアス。

例

次の例では、インラインテーブル、2 つの列を持つ一時テーブルのような結果セット、col1およびを作成しますcol2。結果セットの 1 行には1、それぞれ値 "one"と が含まれます。クエリの SELECT * FROM部分は、この一時結果セットからすべての列と行を取得するだけです。VALUES 句は列名を明示的に指定しないため、列名 (col1 および col2) はデータベースシステムによって自動的に生成されます。

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

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

カスタム列名を定義する場合は、VALUES 句の後に AS 句を使用して次のように定義できます。

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

これにより、デフォルトの nameおよび ではなくid、列名 col1および を含む一時的な結果セットが作成されますcol2。

GROUP BY 句

GROUP BY 句は、クエリのグループ化列を特定します。クエリが SUM、AVG、COUNT などの標準関数を使って集計する場合、グループ化列を宣言する必要があります。SELECT 式に集計関数が含まれている場合、集計関数に含まれていない SELECT 式の列はすべて GROUP BY 句に含まれている必要があります。

詳細については、「AWS Clean Rooms Spark SQL 関数」を参照してください。

構文

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

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

パラメータ

expr

列または式のリストは、クエリの SELECT リストの非集計式のリストと一致する必要があります。例えば、次のシンプルなクエリを考慮してみます。

```
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)
```

このクエリでは、選択されたリストは 2 つの集計式で構成されています。最初の式は SUM 関数を使用し、2 番目の式は COUNT 関数を使用します。残りの 2 つの例 (LISTID と EVENTID) は、グループ化列として宣言する必要があります。

GROUP BY 句の式は、序数を使用することで、SELECT リストを参照することもできます。例えば、前の例は、次のように短縮できます。

```
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

集計拡張機能の ROLLUP を使用すると、1 つのステートメントで複数の GROUP BY 操作を実行できます。集計拡張機能および関連する関数の詳細については、「集計拡張機能」を参照してください。

集計拡張機能

AWS Clean Rooms は、1 つのステートメントで複数の GROUP BY オペレーションの作業を実行するための集約拡張機能をサポートしています。

GROUPING SETS

1 つのステートメントで 1 つ以上のグループ化セットを計算します。グループ化セットとは、1 つの GROUP BY 句のセットで、クエリの結果セットをグループ化できる 0 個以上の列のセットです。GROUP BY GROUPING SETS は、異なる列でグループ化された 1 つの結果セットに対して UNION ALL クエリを実行することに相当します。例えば、GROUP BY GROUPING SETS((a), (b)) は、GROUP BY a UNION ALL GROUP BY b と同等です。

次の例では、製品のカテゴリと販売された製品の種類の両方に従ってグループ化された注文テーブルの製品のコストを返します。

```
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

前の列が後続の列の親と見なされる階層を前提としています。ROLLUP は、指定された列ごとにデータをグループ化し、グループ化された行に加えて、グループ化列の全レベルの合計を表す追加の小計行を返します。例えば、GROUP BY ROLLUP((a), (b)) を使用すると、b が a のサブセクションであると仮定して、最初に a でグループ化された結果セットを返し、次に b でグループ化された結果セットを返すことができます。また、ROLLUP では、列をグループ化せずに結果セット全体を含む行を返します。

GROUP BY ROLLUP((a), (b)) は、GROUP BY GROUPING SETS((a,b), (a), ()) と同等です。

次の例では、最初にカテゴリ別にグループ化された注文テーブルの製品のコストを返し、次にカテゴリが細分化された製品を返します。

```
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

指定した列ごとにデータをグループ化し、グループ化された行に加えて、グループ化列の全レベルの合計を表す追加の小計行を返します。CUBE は ROLLUP と同じ行を返しますが、ROLLUP の対象とならないグループ列のすべての組み合わせで小計行を追加します。例えば、GROUP BY CUBE ((a), (b)) を使用すると、b が a のサブセクションであると仮定して、最初に a でグループ化された結果セットを返し、次に b でグループ化された結果セット、さらに b のみでグループ化された結果セットを返すことができます。また、CUBE では、列をグループ化せずに結果セット全体を含む行を返します。

GROUP BY CUBE((a), (b)) は GROUP BY GROUPING SETS((a, b), (a), (b), ()) と同等です。

次の例では、最初にカテゴリ別にグループ化された注文テーブルの製品のコストを返し、次にカテゴリが細分化された製品を返します。前述の ROLLUP の例とは異なり、このステートメントはグループ化列のすべての組み合わせの結果を返します。

```
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 句

HAVING 句は、クエリが返す中間グループ結果セットに条件を適用します。

構文

```
[ HAVING condition ]
```

例えば、SUM 関数の結果を制限できます。

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

HAVING 条件は、すべての WHERE 句条件が適用され、GROUP BY オペレーションが完了してから適用されます。

条件自体は、WHERE 句の条件と同じ形式になります。

使用に関する注意事項

- HAVING 句条件内で参照される列は、グループ化列または集計関数の結果を参照する列のいずれかでなければなりません。
- HAVING 句では、以下の項目を指定することはできません。
  - SELECT リスト項目を参照する序数。序数が使用できるのは、GROUP BY 句または ORDER BY 句だけです。

例

次のクエリは、すべてのイベントに対するチケットの合計販売を名前別に計算し、販売合計が 800,000 ドルに達しなかったイベントを削除します。HAVING 条件は、SELECT リスト内の集計関数の結果に適用されます。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)
```

次のクエリは、同じような結果セットを計算します。ただしこの場合、SELECT リスト sum(qtysold) で指定されていない集計に対して HAVING 条件が適用されます。2,000 枚を超えるチケットを販売しなかったイベントは、最終結果から削除されます。

```
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)
```

セット演算子

セット演算子は、2 つの個別のクエリ式の結果を比較およびマージするために使用されます。

AWS Clean Rooms Spark SQL は、次の表に示す以下のセット演算子をサポートしています。

設定演算子
INTERSECT
すべて解釈する
EXCEPT
ALL を除く
UNION
UNION ALL

例えば、ウェブサイトで購入者と販売者の両方を兼ねているが、ユーザー名が別々の列または表に格納されているユーザーを確認するには、これら 2 種類のユーザーの積集合を求めます。購入者ではあるが販売者ではないウェブユーザーを確認するには、EXCEPT 演算子を使用すると、2 つユーザーリストの差を見つけることができます。役割とは無関係に、すべてのユーザーのリストを作成する場合、UNION 演算子を使用できます。

> **Note**
>
> ORDER BY、LIMIT、SELECT TOP、および OFFSET 句は、UNION、UNION ALL、INTERSECT、および EXCEPT セット演算子によってマージされたクエリ式では使用できません。

トピック

構文

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

パラメータ

subquery1、subquery2

選択リストの形式で、UNION、UNION ALL、INTERSECT、INTERSECT ALL、EXCEPT、または EXCEPT ALL 演算子に続く 2 番目のクエリ式に対応するクエリ式。2 つの式は、互換性のあるデータ型の出力列を同数含んでいる必要があります。そうでない場合、2 つの結果セットの比較とマージはできません。設定オペレーションでは、異なるカテゴリのデータ型間の暗黙的な変換は許可されません。詳細については、「型の互換性と変換」を参照してください。

クエリ式の数を上限なしに含むクエリを構築して、そのクエリを任意の組み合わせで UNION、INTERSECT、および EXCEPT 演算子に関連付けることができます。例えば、テーブル T1、T2、および T3 に互換性のある列セットが含まれていると想定した場合、次のクエリ構造は有効です。

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

UNION [ALL | DISTINCT]

行が片方の式から生成されたか、両方の式から生成されたかにかかわらず、2 つのクエリ式からの行を返す演算を設定します。

交差 [すべて | 個別]

2 つのクエリ式から生成される行を返す演算を設定します。両方の式によって返されない行は破棄されます。

[ALL | DISTINCT] を除く

2 つのクエリ式の一方から生成される行を返す演算を設定します。結果を制限するため、行は最初の結果テーブルに存在し、2 番目のテーブルには存在しない必要があります。

EXCEPT ALL は結果行から重複を削除しません。

MINUS と EXCEPT はまったく同じ意味です。

セット演算の評価の順番

UNION および EXCEPT セット演算子は左結合です。優先順位の決定でかっこを指定しなかった場合、これらのセット演算子の組み合わせは、左から右に評価されます。例えば、次のクエリでは、T1 と T2 の UNION が最初に評価され、次に UNION の結果に対して、EXCEPT 演算が実行されます。

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

同じクエリ内で演算子の組み合わせを使用した場合、INTERSECT 演算子は UNION および EXCEPT よりも優先されます。例えば、次のクエリは T2 と T3 の積集合を評価し、その結果を T1 を使って結合します。

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

かっこを追加することで、評価の順番を変更することができます。次のケースでは、T1 と T2 の結合結果と T3 の積集合を求めます。このクエリでは異なる結果が生成されます。

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

使用に関する注意事項

- セット演算クエリの結果で返される列名は、最初のクエリ式のテーブルからの列名 (またはエイリアス) です。これらの列名は、列内の値はセット演算子の両方のテーブルから生成されるという点

で誤解を生む可能性があるため、結果セットには意味のある エイリアスを付けることをお勧めします。

- セット演算子クエリが 10 進数の結果を返した場合、同じ精度とスケールで対応する結果列を返すように奨励されます。例えば、T1.REVENUE が DECIMAL(10,2) 列で T2.REVENUE が DECIMAL(8,4) 列の次のクエリでは、DECIMAL(12,4) への結果も 10 進数であることが奨励されます。

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

スケールは 4 になります。2 つの列の最大のスケールです。精度は 12 です。T1.REVENUE は小数点の左側に 8 桁必要であるからです (12 - 4 = 8)。このような奨励により、UNION の両側からのすべての値が結果に適合します。64 ビットの値の場合、最大結果精度は 19 で、最大結果スケールは 18 です。128 ビットの値の場合、最大結果精度は 38 で、最大結果スケールは 37 です。

結果のデータ型が AWS Clean Rooms 精度とスケール制限を超えると、クエリはエラーを返します。

- 集合演算で 2 行が同一として扱われるのは、対応する列のペアごとに、2 つのデータ値が等しいまたはどちらも NULL である場合です。例えば、テーブル T1 と T2 の両方に 1 つの列と 1 つの行が含まれていて、両方のテーブルでその行が NULL の場合、これらのテーブルに INTERSECT 演算に実行すると、その行が返されます。

UNION クエリの例

次の UNION クエリでは、SALES テーブルの行が、LISTING テーブルの行とマージされます。各テーブルからは 3 つの互換性のある列が選択されます。この場合、対応する列には同じ名前とデータ型が与えられます。

```
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
```

次の例では、どのクエリ式が結果セットの各行を生成したかを確認できるように、UNION クエリの出力にリテラル値を追加する方法を示します。このクエリは、最初のクエリ式からの行を (販売者を意味する)「B」として識別し、2 番目のクエリ式からの行を (購入者を意味する)「S」として識別します。

このクエリは 10,000 ドル以上のチケット取引の販売者と購入者を識別します。UNION 演算子の両側の 2 つのクエリ式の違いは、SALES テーブルの結合列だけです。

```
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
```

次の例では、重複行が検出された場合、その重複行を結果に保持する必要があるため、UNION ALL 演算子を使用します。一連の特定イベント ID では、クエリは各イベントに関連付けられているセールスごとに 0 行以上の行を返し、そのイベントのリスティングごとに 0 行または 1 行を返します。イベント ID は、LISTING テーブルと EVENT テーブルの各行に対して一意ですが、SALES テーブルのイベント ID とリスティング ID の同じ組み合わせに対して、複数のセールスが存在することがあります。

結果セットの 3 番目の列は、行のソースを特定します。その行が SALES テーブルからの行だった場合、SALESROW 列に "Yes" というマークが付きます。(SALESROW は SALES.LISTID のエイリアスです。) その行が LISTING テーブルからの行だった場合、SALESROW 列に "No" というマークが付きます。

この場合、リスティング 500、イベント 7787 の結果セットは、3 つの行から構成されます。つまり、このリスティングとイベントの組み合わせに対して、3 つの異なる取引が実行されたということです。他の 2 つのリスティング (501 と 502) では販売はありません。このため、これらのリスト ID に対してクエリが生成した唯一の行は LISTING テーブル (SALESROW = 'No') から生成されます。

```
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
```

ALL キーワードを付けずに同じクエリを実行した場合、結果には、セールス取引の 1 つだけが保持されます。

```
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
```

UNION ALL クエリの例

次の例では、重複行が検出された場合、その重複行を結果に保持する必要があるため、UNION ALL 演算子を使用します。一連の特定イベント ID では、クエリは各イベントに関連付けられているセールスごとに 0 行以上の行を返し、そのイベントのリスティングごとに 0 行または 1 行を返します。イベント ID は、LISTING テーブルと EVENT テーブルの各行に対して一意ですが、SALES テーブルのイベント ID とリスティング ID の同じ組み合わせに対して、複数のセールスが存在することがあります。

結果セットの 3 番目の列は、行のソースを特定します。その行が SALES テーブルからの行だった場合、SALESROW 列に "Yes" というマークが付きます。(SALESROW は SALES.LISTID のエイリアスです。) その行が LISTING テーブルからの行だった場合、SALESROW 列に "No" というマークが付きます。

この場合、リスティング 500、イベント 7787 の結果セットは、3 つの行から構成されます。つまり、このリスティングとイベントの組み合わせに対して、3 つの異なる取引が実行されたということです。他の 2 つのリスティング (501 と 502) では販売はありません。このため、これらのリスト ID に対してクエリが生成した唯一の行は LISTING テーブル (SALESROW = 'No') から生成されます。

```
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
```

ALL キーワードを付けずに同じクエリを実行した場合、結果には、セールス取引の 1 つだけが保持されます。

```
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
```

INTERSECT クエリの例

次の例を最初の UNION の例と比較してみます。2 つの例の違いは使われたセット演算子だけですが、結果は大きく異なります。1 つの行だけが同じになります。

```
235494 |    23875 |    8771
```

これは、両方のテーブルから検出された 5 つの行の制限された結果の唯一の行です。

```
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
```

次のクエリでは、3 月にニューヨーク市とロサンジェルスの両方で発生した (チケットが販売された) イベントを検索します。2 つのクエリ式の違いは、VENUECITY 列の制約です。

```
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
```

EXCEPT クエリの例

データベースの CATEGORY テーブルには、次の 11 行が含まれています。

```
 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)
```

CATEGORY_STAGE テーブル (ステージングテーブル) には、1 つの追加行が含まれていると想定します。

```
 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)
```

2 つのテーブル間の違いを返します。つまり、CATEGORY テーブル内の行ではなく、CATEGORY_STAGE テーブル内の行が返されるということです。

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

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

次の同等のクエリでは、同義語の MINUS を使用します。

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

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

SELECT 式の順番を逆にすると、クエリは行を返しません。

ORDER BY 句

ORDER BY 句は、クエリの結果セットをソートします。

> **Note**
>
> 最も外側の ORDER BY 式には、SELECT リストにある列だけが含まれている必要があります。

トピック

構文

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

パラメータ

expression

クエリ結果のソート順序を定義する式。SELECT リストの 1 つ以上の列から構成されています。結果は、バイナリ UTF-8 順序付けに基づいて返されます。以下を指定することもできます。

- SELECT リストエントリの位置 (SELECT リストが存在しない場合は、テーブルの列の位置) を表す序数
- SELECT リストエントリを定義するエイリアス

ORDER BY 句に複数の式が含まれる場合、結果セットは、最初の式に従ってソートされ、次の最初の式の値と一致する値を持つ行に 2 番目の式が適用されます。以降同様の処理が行われます。

ASC | DESC

次のように、式のソート順を定義するオプション:

- ASC: 昇順 (数値の場合は低から高、文字列の場合は「A」から「Z」など) オプションを指定しない場合、データはデフォルトでは昇順にソートされます。
- DESC: 降順 (数値の場合は高から低、文字列の場合は「Z」から「A」)。

NULLS FIRST | NULLS LAST

NULL 値を NULL 以外の値より先に順序付けするか、NULL 以外の値の後に順序付けするかを指定するオプション。デフォルトでは、NULL 値は昇順ではソートされて最後にランク付けされ、降順ではソートされて最初にランク付けされます。

LIMIT number | ALL

クエリが返すソート済みの行数を制御するオプション。LIMIT 数は正の整数でなければなりません。最大値は 2147483647 です。

LIMIT 0 は行を返しません。この構文は、(行を表示せずに) クエリが実行されているかを確認したり、テーブルから列リストを返すためのテスト目的で使用できます。列リストを返すために LIMIT 0 を使用した場合、ORDER BY 句は重複です。デフォルトは LIMIT ALL です。

OFFSET start

行を返す前に、start の前の行数をスキップするよう指定するオプション。OFFSET 数は正の整数でなければなりません。最大値は 2147483647 です。LIMIT オプションを合わせて使用すると、OFFSET 行は、返される LIMIT 行のカウントを開始する前にスキップされます。LIMIT オプションを使用しない場合、結果セット内の行の数が、スキップされる行の数だけ少なくなります。OFFSET 句によってスキップされた行もスキャンされる必要はあるため、大きい OFFSET 値を使用することは一般的に非効率的です。

使用に関する注意事項

ORDER BY 句を使用すると、次の動作が予想されます。

- ヌル値は他のすべての値よりも「高い」と見なされます。デフォルトの昇順のソートでは、NULL 値は最後に表示されます。この動作を変更するには、NULLS FIRST オプションを使用します。
- クエリに ORDER BY 句が含まれていない場合、結果行が返す行の順番は予想不能です。同じクエリを 2 回実行した場合に、結果セットが返される順番が異なることがあります。

- LIMIT オプションと OFFSET オプションは、ORDER BY 句なしに使用できます。ただし、整合性のある行セットを返すには、これらのオプションを ORDER BY と組み合わせて使用してください。
- のような並列システムでは AWS Clean Rooms、ORDER BY が一意の順序を生成しない場合、行の順序は非決定的です。つまり、ORDER BY 式が重複した値を生成する場合、それらの行の戻り順序は他のシステムや の実行ごとに異なる場合があります AWS Clean Rooms 。
- AWS Clean Rooms は、ORDER BY 句の文字列リテラルをサポートしていません。

ORDER BY の例

CATEGORY テーブルの 11 の行をすべて、2 番目の列 (CATGROUP) でソートして返します。CATGROUP の値が同じ結果の場合、CATDESC 列の値は文字列の長さによってソートされます。次に CATID 列と CATNAME 列でソートされます。

```
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)
```

SALES テーブルの選択した列を、QTYSOLD 値の高い順にソートして返します。結果を上位の 10 行に制限します。

```
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
```

LIMIT 0 構文を使用することで、列リストを返し、行を返しません。

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

サブクエリの例

次の例は、サブクエリが SELECT クエリに適合するさまざまな方法を示しています。サブクエリの使用に関する別の例については、「例」を参照してください。

SELECT リストのサブクエリ

次の例には、SELECT リストのサブクエリが含まれています。このサブクエリはスカラー値であり、1 つの列と 1 つの値のみを返します。外部クエリから返される行の結果ごとに、このサブクエリが繰り返されます。このクエリは、サブクエリが計算した Q1SALES 値を、外部クエリが定義する、2008 年の他の 2 つの四半期 (第 2 と第 3) のセールス値と比較します。

```
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 句のサブクエリ

次の例には、WHERE 句にテーブルサブクエリが含まれます。このサブクエリは複数の行を生成します。この場合、その行には列が 1 つだけ含まれていますが、テーブルサブクエリには他のテーブルと同様、複数の列と行が含まれていることがあります。

このクエリは、最大販売チケット数の観点でトップ 10 の販売会社を検索します。トップ 10 のリストは、チケットカウンターが存在する都市に住んでいるユーザーを削除するサブクエリによって制限されます。このクエリは、メインクエリ内の結合としてサブクエリを作成するなど、さまざまな方法で作成できます。

```
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 句のサブクエリ

「WITH 句」を参照してください。

相関性のあるサブクエリ

次の例の WHERE 句には、相関性のあるサブクエリが含まれています。このタイプのサブクエリには、サブクエリの列と他のクエリが生成した列の間に相関性があります。この場合、相関は where

s.listid=l.listid となります。外部クエリが生成する各行に対してサブクエリが実行され、行が適正か適正でないかが判断されます。

```
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)
```

サポートされていない相関サブクエリのパターン

クエリのプランナーは、MPP 環境で実行する相関サブクエリの複数のパターンを最適化するため、「サブクエリ相関解除」と呼ばれるクエリ再生成メソッドを使用します。相関サブクエリのいくつかのタイプ AWS Clean Rooms は、関連付けを解除できず、サポートされないパターンに従います。次の相関参照を含んでいるクエリがエラーを返します。

- クエリブロックをスキップする相関参照 (「スキップレベル相関参照」とも呼ばれています) 例えば、次のクエリでは、相関参照とスキップされるブロックを含むブロックは、NOT EXISTS 述語によって接続されます。

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

  このケースでスキップされたブロックは、LISTING テーブルに対するサブクエリです。相関参照は、EVENT テーブルと SALES テーブルを関係付けます。
- 外部クエリで ON 句の一部であるサブクエリからの相関参照:

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

ON 句には、サブクエリの SALES から外部クエリの EVENT への相関参照が含まれています。

- AWS Clean Rooms システムテーブルへの Null センシティブな相関参照。例えば、次のようになります。

```
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);
```

- ウィンドウ関数を含んでいるサブクエリ内からの相関参照。

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

- 相関サブクエリの結果に対する、GROUP BY 列の参照。次に例を示します。

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

- 集計関数と GROUP BY 句のあり、IN 述語によって外部クエリに接続されているサブクエリからの相関参照。(この制限は、MIN と MAX 集計関数には適用されません。) 例 :

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

AWS Clean Rooms Spark SQL 関数

AWS Clean Rooms Spark SQL は、次の SQL 関数をサポートしています。

トピック

集計関数

Spark SQL AWS Clean Rooms の集計関数は、行のグループに対して計算またはオペレーションを実行し、単一の値を返すために使用されます。データ分析や要約タスクに不可欠です。

AWS Clean Rooms Spark SQL は、次の集計関数をサポートしています。

トピック

ANY_VALUE 関数

ANY_VALUE 関数は、入力式の値から任意の値を非決定的に返します。この関数は、入力式で行が返されない場合に NULL を返すことができます。

構文

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

引数

expression

関数が動作するターゲット列または式。式は、以下に示すデータ型の 1 つを取ります。

isIgnoreNull

関数が NULL 以外の値のみを返すかどうかを決定するブール値。

戻り値

同じデータ型を expression として返します。

使用に関する注意事項

列の ANY_VALUE 関数を指定するステートメントに 2 番目の列参照も含まれている場合、2 番目の列は GROUP BY 句に含めるか、集計関数に含める必要があります。

例

次の例では、eventname が Eagles である任意の dateid のインスタンスを返します。

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

結果は、以下のとおりです。

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

次の例では、eventname が Eagles または Cold War Kids である任意の dateid のインスタンスを返します。

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

結果は、以下のとおりです。

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

APPROX COUNT_DISTINCT 関数

APPROX COUNT_DISTINCT を使用すると、列またはデータセット内の一意の値の数を効率的に見積もることができます。

構文

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

引数

expr

一意の値の数を推定する式または列。

1 つの列、複雑な式、または列の組み合わせを指定できます。

relativeSD

見積りの望ましい相対標準偏差を指定するオプションのパラメータ。

これは 0 ~ 1 の値で、推定値の最大許容相対誤差を表します。relativeSD 値が小さいほど、より正確ですが推定が遅くなります。

このパラメータを指定しない場合、デフォルト値 (通常は約 0.05 または 5%) が使用されます。

戻り値

HyperLogLog++ による推定基数を返します。 relativeSD は、許容される最大相対標準偏差を定義します。

例

次のクエリは、col1列内の一意の値の数を推定し、相対標準偏差は 1% (0.01) です。

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

次のクエリでは、col1列に 3 つの一意の値 (値 1、2、3) があると推定します。

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

APPROX PERCENTILE 関数

APPROX PERCENTILE は、データセット全体をソートすることなく、特定の式または列のパーセンタイル値を推定するために使用されます。この関数は、正確なパーセンタイル計算を実行する計算

オーバーヘッドなしで、大規模なデータセットの分布をすばやく把握したり、パーセンタイルベースのメトリクスを追跡したりするシナリオに役立ちます。ただし、速度と精度のトレードオフを理解し、ユースケースの特定の要件に基づいて適切なエラー耐性を選択することが重要です。

構文

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

引数

expr

パーセンタイル値を推定する式または列。

1 つの列、複雑な式、または列の組み合わせを指定できます。

percentile

推定するパーセンタイル値。0 ~ 1 の値で表されます。

たとえば、0.5 は 50 パーセンタイル (中央値) に対応します。

精度

パーセンタイル推定値の望ましい精度を指定するオプションのパラメータ。これは 0 ~ 1 の値で、推定値の最大許容相対誤差を表します。accuracy 値が小さいほど、より正確ですが推定が遅くなります。このパラメータを指定しない場合、デフォルト値 (通常は約 0.05 または 5%) が使用されます。

戻り値

数値または ANSI 間隔列の col の近似パーセンタイルを返します。これは、順序付けられた col 値 (最小から最大にソート) の最小値であり、col 値の割合が 値より小さいか、その値と等しくならないようにします。

パーセンテージの値は 0.0 から 1.0 の間でなければなりません。精度パラメータ (デフォルト: 10000) は、メモリのコストで近似精度を制御する正の数値リテラルです。

精度の値が大きいほど精度が向上します。 `1.0/accuracy`は近似の相対誤差です。

percentage が配列の場合、割合配列の各値は 0.0 から 1.0 の間である必要があります。この場合、は指定されたパーセンテージ配列で列列列の近似パーセンタイル配列を返します。

例

次のクエリは、response_time列の 95 パーセンタイルを推定し、最大相対誤差は 1% (0.01) です。

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

次のクエリは、tabテーブル内のcol列の 50 パーセンタイル値、40 パーセンタイル値、および 10 パーセンタイル値を推定します。

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

次のクエリは、col 列の値の 50 パーセンタイル (中央値) を推定します。

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

AVG 関数

AVG 関数は、入力式の値の平均 (算術平均) を返します。AVG 関数は数値に対してはたらき、NULL 値は無視します。

構文

```
AVG (column)
```

引数

column

関数の対象となる列。列は、以下に示すデータ型の 1 つを取ります。

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

データ型

AVG 関数でサポートされる引数の型は、SMALLINT、INTEGER、BIGINT、DECIMAL、および DOUBLE です。

AVG 関数でサポートされる戻り値の型は次のとおりです。

- 整数型の引数の場合は BIGINT
- 浮動小数点の引数の場合は DOUBLE
- 他の引数型については、式と同じデータ型を返します。

DECIMAL 引数を使用した AVG 関数の結果のデフォルト精度は 38 です。結果のスケールは、引数のスケールと同じです。例えば、DEC(5,2) 列の AVG は DEC(38,2) データ型を返します。

例

SALES テーブルから、取引ごとの平均販売数量を求めます。

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

BOOL_AND 関数

BOOL_AND 関数は、単一のブール値、整数値、または式に対して動作します。この関数は、BIT_AND 関数と BIT_OR 関数に同様の口ジックを適用します。この関数の戻り値の型はブール値 (`true` または `false`) です。

セットの値がすべて true の場合、BOOL_AND 関数は `true` (`t`) を返します。いずれかの値が false の場合、関数は `false` (`f`) を返します。

構文

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

引数

expression

関数の対象となる列または式。この式には BOOLEAN または整数のデータ型がある必要があります。関数の戻り値の型は BOOLEAN です。

DISTINCT | ALL

引数 DISTINCT を指定すると、この関数は結果を計算する前に指定された式から重複した値をすべて削除します。引数 ALL 指定すると、この関数は重複する値をすべて保持します。ALL がデフォルトです。

例

ブール関数をブール式または整数式のどちらかに対して使用できます。

例えば、次のクエリは複数のブール列のある TICKIT データベースの標準 USERS テーブルから結果を返します。

BOOL_AND 関数は 5 行すべてに `false` を返します。これら各州のすべてのユーザーがスポーツを好きというわけではありません。

```
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 関数

BOOL_OR 関数は、単一のブール値、整数値、または式に対して動作します。この関数は、BIT_AND 関数と BIT_OR 関数に同様の口ジックを適用します。この関数の戻り値の型はブール値 (`true`、`false`、または NULL) です。

セットの値が `true` の場合、BOOL_OR 関数は `true` (t) を返します。セットの値が `false` の場合、関数は `false` (f) を返します。値が不明な場合は NULL を返すことができます。

構文

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

引数

expression

関数の対象となる列または式。この式には BOOLEAN または整数のデータ型がある必要があります。関数の戻り値の型は BOOLEAN です。

DISTINCT | ALL

引数 DISTINCT を指定すると、この関数は結果を計算する前に指定された式から重複した値をすべて削除します。引数 ALL 指定すると、この関数は重複する値をすべて保持します。ALL がデフォルトです。

例

ブール関数はブール式または整数式のどちらかで使用できます。例えば、次のクエリは複数のブール列のある TICKIT データベースの標準 USERS テーブルから結果を返します。

BOOL_OR 関数は 5 行すべてに `true` を返します。これらの州のそれぞれにおいて少なくとも 1 人のユーザーがスポーツを好きです。

```
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)
```

次の例は、NULL を返します。

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

CARDINALITY 関数

CARDINALITY 関数は、ARRAY 式または MAP 式 (expr) のサイズを返します。

この関数は、配列のサイズまたは長さを見つけるのに役立ちます。

構文

```
cardinality(expr)
```

引数

expr

ARRAY 式または MAP 式。

戻り値

配列またはマップ (INTEGER) のサイズを返します。

が に設定されている場合、falseまたは が に設定されている場合sizeOfNull、関数enabledはnull 入力NULLを返しますtrue。

それ以外の場合、関数は null 入力-1を返します。デフォルト設定では、関数は null 入力-1に対してを返します。

例

次のクエリは、指定された配列の基数または要素の数を計算します。配列 ('b', 'd', 'c', 'a') には 4 つの要素があるため、このクエリの出力は になります4。

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

COLLECT_LIST 関数

COLLECT_LIST 関数は、一意でない要素のリストを収集して返します。

このタイプの関数は、行のセットから複数の値を単一の配列またはリストデータ構造に収集する場合に便利です。

> **Note**
>
> 収集された結果の順序は行の順序に依存するため、関数は非決定的です。シャッフル操作の実行後に非決定的になる可能性があります。

構文

```
collect_list(expr)
```

引数

expr

任意のタイプの式。

戻り値

引数タイプの ARRAY を返します。配列内の要素の順序は非決定的です。

NULL 値は除外されます。

DISTINCT が指定されている場合、関数は一意の値のみを収集し、collect_set 集計関数のシノニムです。

例

次のクエリは、col 列からすべての値をリストに収集します。VALUES 句は、3 行のインラインテーブルを作成するために使用されます。各行の列列はそれぞれ 1、2、1 です。次に、collect_list() 関数を使用して、col 列のすべての値を 1 つの配列に集約します。この SQL ステートメントの出力は配列 となり[1,2,1]、入力データに表示された順序で col 列のすべての値が含まれます。

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

COLLECT_SET 関数

COLLECT_SET 関数は、一意の要素のセットを収集して返します。

この関数は、重複を含めずに、行のセットからすべての個別の値を 1 つのデータ構造に収集する場合に便利です。

> **Note**
>
> 収集された結果の順序は行の順序に依存するため、関数は非決定的です。シャッフル操作の実行後に非決定的になる可能性があります。

構文

```
collect_set(expr)
```

引数

expr

MAP を除く任意のタイプの式。

戻り値

引数タイプの ARRAY を返します。配列内の要素の順序は非決定的です。

NULL 値は除外されます。

例

次のクエリは、col 列からすべての一意の値をセットに収集します。VALUES 句は、3 行のインラインテーブルを作成するために使用されます。各行の列列はそれぞれ 1、2、1 です。次に、collect_set()関数を使用して、col 列のすべての一意の値を 1 つのセットに集約します。この SQL ステートメントの出力は[1,2]、col 列の一意の値を含むセット になります。重複する値 1 は、結果に 1 回のみ含まれます。

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

COUNT および COUNT DISTINCT 関数

COUNT 関数は、式で定義された行をカウントします。COUNT DISTINCT 関数は、列または式内の個別の非 NULL 値の数を計算します。この関数は、カウントを行う前に指定された式から重複した値をすべて削除します。

構文

```
COUNT (DISTINCT column)
```

引数

column

関数の対象となる列。

データ型

COUNT 関数および COUNT DISTINCT 関数は、引数のデータ型をすべてサポートしています。

COUNT DISTINCT 関数は BIGINT を返します。

例

フロリダ州のユーザーをすべてカウントします。

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

EVENT テーブルから一意の会場 ID をすべてカウントします。

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

COUNT 関数

COUNT 関数は式で定義された行をカウントします。

COUNT 関数には 3 つのバリエーションがあります。

- COUNT(*) は null を含むかどうかにかかわらず、ターゲットテーブルのすべての行をカウントします。
- COUNT (expression) は、特定の列または式にある Null 以外の値を持つ行数を計算します。

- COUNT (DISTINCT expression) は、列または式にある Null 以外の一意な値の数を計算します。

構文

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

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

引数

expression

関数の対象となる列または式。COUNT 関数は引数のデータ型をすべてサポートしています。

DISTINCT | ALL

引数 DISTINCT を指定すると、この関数はカウントを行う前に指定された式から重複した値をすべて削除します。引数 ALL を指定すると、この関数はカウントに使用する式から重複する値をすべて保持します。ALL がデフォルトです。

戻り型

COUNT 関数は BIGINT を返します。

例

フロリダ州のユーザーをすべてカウントします。

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

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

EVENT テーブルからすべてのイベント名をカウントします。

```
select count(eventname) from event;

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

EVENT テーブルからすべてのイベント名をカウントします。

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

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

EVENT テーブルから一意の会場 ID をすべてカウントします。

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

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

4 枚より多いチケットをまとめて販売した販売者ごとの回数をカウントします。販売者 ID で結果をグループ化します。

```
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 関数

MAX 関数は行のセットの最大値を返します。DISTINCT または ALL が使用される可能性がありますが、結果には影響しません。

構文

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

引数

expression

関数の対象となる列または式。式は任意の数値データ型です。

DISTINCT | ALL

引数 DISTINCT を指定すると、この関数は最大値を計算する前に指定された式から重複した値をすべて削除します。引数 ALL を指定すると、この関数は最大値を計算する式から重複する値をすべて保持します。ALL がデフォルトです。

データ型

同じデータ型を expression として返します。

例

すべての販売から最高支払価格を検索します。

```
select max(pricepaid) from sales;

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

すべての販売からチケットごとの最高支払価格を検索します。

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

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

MEDIAN 関数

構文

```
MEDIAN ( median_expression )
```

引数

median_expression

関数の対象となる列または式。

MIN 関数

MIN 関数は行のセットの最小値を返します。DISTINCT または ALL が使用される可能性がありますが、結果には影響しません。

構文

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

引数

expression

関数の対象となる列または式。式は任意の数値データ型です。

DISTINCT | ALL

引数 DISTINCT を指定すると、この関数は最小値を計算する前に指定された式から重複した値をすべて削除します。引数 ALL を指定すると、この関数は最小値を計算する式から重複する値をすべて保持します。ALL がデフォルトです。

データ型

同じデータ型を expression として返します。

例

すべての販売から最低支払価格を検索します。

```
select min(pricepaid) from sales;

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

すべての販売からチケットごとの最低支払価格を検索します。

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

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

PERCENTILE 関数

PERCENTILE 関数は、最初にcol列内の値をソートし、次に指定された で値を検索することで、正確なパーセンタイル値を計算するために使用しますpercentage。

PERCENTILE 関数は、正確なパーセンタイル値を計算する必要があり、計算コストがユースケースで許容できる場合に役立ちます。APPROX_PERCENTILE 関数よりも正確な結果が得られますが、特に大規模なデータセットでは時間がかかる場合があります。

対照的に、APPROX_PERCENTILE 関数は、指定されたエラー耐性を持つパーセンタイル値の推定値を提供できる、より効率的な代替手段であり、絶対精度よりも速度が優先度が高いシナリオに適しています。

構文

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

引数

コール

パーセンタイル値を計算する式または列。

パーセンテージ

計算するパーセンタイル値。0 ~ 1 の値で表されます。

たとえば、0.5 は 50 パーセンタイル (中央値) に対応します。

頻度

col 列内の各値の頻度または重みを指定するオプションのパラメータ。指定した場合、関数は各値の頻度に基づいてパーセンタイルを計算します。

戻り値

数値または ANSI 間隔の列列の正確なパーセンタイル値を指定されたパーセンテージで返します。

パーセンテージの値は 0.0 から 1.0 の間でなければなりません。

頻度の値は正の整数である必要があります

例

次のクエリは、 col列の値の 30% 以上の値を検索します。値は 0 と 10 であるため、30 パーセンタイルはデータの 30% 以上の値であるため、3.0 です。

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

SKEWNESS 関数

SKEWNESS 関数は、グループの値から計算された歪度値を返します。

歪みは、データセット内の非対称性または対称性の欠如を記述する統計的尺度です。データディストリビューションの形状に関する情報を提供します。

この関数は、データセットの統計プロパティを理解し、さらなる分析や意思決定を通知するのに役立ちます。

構文

```
skewness(expr)
```

引数

expr

数値に評価される式。

戻り値

DOUBLE を返します。

DISTINCT が指定されている場合、関数は一意の expr 値のセットでのみ動作します。

例

次のクエリは、 col列の値の歪みを計算します。この例では、 VALUES句を使用して 4 行のインラインテーブルを作成します。各行の列のcol値は -10、-20、100、1000 です。次に、 skewness( )関数を使用してcol列の値の歪度を計算します。結果 1.1135657469022011 は、データの歪みの程度と方向を表します。正の歪値は、データが右側に歪んでいて、値の大部分が分散の左側に集中していることを示します。負の歪値は、データが左側に歪んでいて、値の大部分がディストリビューションの右側に集中していることを示します。

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

次のクエリは、col 列の値の歪度を計算します。前の例と同様に、 VALUES句は 4 行のインラインテーブルを作成するために使用されます。各行には -1000、-100、10、20 の値が 1 列colあります。次に、 skewness( )関数を使用してcol列の値の歪度を計算します。結果 -1.1135657469022011 は、データの歪みの程度と方向を表します。この場合、負の歪値は、データが左側に歪んでいて、値の大部分が分布の右側に集中していることを示します。

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

STDDEV_SAMP および STDDEV_POP 関数

STDDEV_SAMP および STDDEV_POP 関数は、数値のセットの標本標準偏差と母集団標準偏差 (整数、10 進数、または浮動小数点) を返します。STDDEV_SAMP 関数の結果は、値の同じセットの標本分散の平方根に相当します。

STDDEV_SAMP および STDDEV は、同じ関数のシノニムです。

構文

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

式には数値データ型が必要です。式のデータ型にかかわらず、この関数の戻り値の型は倍精度の数値です。

> **Note**
>
> 標準偏差は浮動小数点演算を使用して計算されます。これはわずかに不正確である可能性があります。

使用に関する注意事項

標本標準偏差 (STDDEV または STDDEV_SAMP) が 1 つの値で構成される式に対して計算される場合、関数の結果は 0 ではなく、NULL になります。

例

次のクエリは VENUE テーブルの VENUESEATS 列の値の平均、続いて値の同じセットの標本標準偏差と母集団標準偏差を返します。VENUESEATS は INTEGER 列です。結果のスケールは 2 桁に減らされます。

```
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)
```

次のクエリは SALES テーブルの COMMISSION 列に標本標準偏差を返します。COMMISSION は DECIMAL 列です。結果のスケールは 10 桁に減らされます。

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

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

次のクエリは整数として COMMISSION 列の標本標準偏差をキャストします。

```
select cast(stddev(commission) as integer)
```

```
from sales;

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

次のクエリは COMMISSION 列に標本標準偏差と標本分散の平方根の両方を返します。これらの計算の結果は同じです。

```
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 および SUM DISTINCT 関数

SUM 関数は入力列または式の値の合計を返します。SUM 関数は数値に対してはたらき、NULL 値は無視します。

SUM DISTINCT 関数は合計を計算する前に指定された式から重複した値をすべて削除します。

構文

```
SUM (DISTINCT column )
```

引数

column

関数の対象となる列。列は任意の数値データ型です。

例

SALES テーブルから、支払われたすべての手数料の合計を求めます。

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

SALES テーブルから、支払われたすべての重複しない手数料の合計を求めます。

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

VAR_SAMP および VAR_POP 関数

VAR_SAMP および VAR_POP 関数は、数値のセットの標本分散と母集団分散 (整数、10 進数、または浮動小数点) を返します。VAR_SAMP 関数の結果は、値の同じセットの 2 乗の標本標準偏差に相当します。

VAR_SAMP および VARIANCE は同じ関数のシノニムです。

構文

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

この式は整数、10 進数、または浮動小数点数データ型である必要があります。式のデータ型にかかわらず、この関数の戻り値の型は倍精度の数値です。

> **Note**
>
> これらの関数の結果は、それぞれのクラスターの設定に応じて、データウェアハウスクラスターによって変わる場合があります。

使用に関する注意事項

標本分散 (VARIANCE または VAR_SAMP) が 1 つの値で構成される式に対して計算される場合、関数の結果は 0 ではなく、NULL になります。

例

次のクエリは LISTING テーブルの NUMTICKETS 列の四捨五入した標本分散と母集団分散を返します。

```
select avg(numtickets),
```

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

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

次のクエリは同じ計算を実行しますが、10 進値の結果をキャストします。

```
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)
```

配列関数

このセクションでは、 AWS Clean Roomsでサポートされている SQL の配列関数について説明します。

トピック

- ARRAY 関数
- ARRAY_CONTAINS 関数
- ARRAY_DISTINCT 関数
- ARRAY_EXCEPT 関数
- ARRAY_INTERSECT 関数
- ARRAY_JOIN 関数
- ARRAY_REMOVE 関数
- ARRAY_UNION 関数
- EXPLODE 関数
- FLATTEN 関数

ARRAY 関数

指定された要素を持つ配列を作成します。

構文

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

引数

expr1、expr2

日付型と時刻型を除く任意のデータ型の式。引数は同じデータ型である必要はありません。

戻り型

配列関数は、式内の要素を含む ARRAY を返します。

例

次の例は、数値の配列と、さまざまなデータ型の配列を示しています。

```
--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 関数

ARRAY_CONTAINS 関数を使用して、配列データ構造に対して基本的なメンバーシップチェックを実行できます。ARRAY_CONTAINS 関数は、配列内に特定の値が存在するかどうかを確認する必要がある場合に便利です。

構文

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

引数

array

検索する ARRAY。

value

配列要素と最も一般的でない型を共有する型を持つ式。

戻り型

ARRAY_CONTAINS 関数は BOOLEAN を返します。

値が NULL の場合、結果は NULL です。

配列内のいずれかの要素が NULL の場合、値が他の要素と一致しない場合、結果は NULL になります。

例

次の例では、配列に値 [1, 2, 3]が含まれているかどうかを確認します4。配列 [1, 2, 3] には値が含まれていないため4、array_contains 関数は を返しますfalse。

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

次の例では、配列に値 [1, 2, 3]が含まれているかどうかを確認します2。配列[1, 2, 3]には値が含まれているため2、array_contains 関数は を返しますtrue。

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

ARRAY_DISTINCT 関数

ARRAY_DISTINCT 関数を使用して、配列から重複した値を削除できます。ARRAY_DISTINCT 関数は、配列から重複を削除し、一意の要素のみを操作する必要がある場合に便利です。これは、繰り返

し値が干渉されることなくデータセットに対してオペレーションまたは分析を実行するシナリオに役立ちます。

構文

```
array_distinct(array)
```

引数

array

ARRAY 式。

戻り型

ARRAY_DISTINCT 関数は、入力配列の一意の要素のみを含む ARRAY を返します。

例

この例では、入力配列に の重複値[1, 2, 3, null, 3]が含まれています3。array_distinct 関数はこの重複した値を削除3し、一意の要素 を持つ新しい配列を返します[1, 2, 3, null]。

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

この例では、入力配列に 2と の重複した値[1, 2, 2, 3, 3, 3]が含まれています3。array_distinct 関数はこれらの重複を削除し、一意の要素 を持つ新しい配列を返します[1, 2, 3]。

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

ARRAY_EXCEPT 関数

ARRAY_EXCEPT 関数は 2 つの配列を引数として受け取り、最初の配列に存在する要素のみを含む新しい配列を返しますが、2 番目の配列は含みません。

ARRAY_EXCEPT は、ある配列と別の配列に固有の要素を見つける必要がある場合に便利です。これは、2 つのデータセットの違いを見つけるなど、配列に対してセットのようなオペレーションを実行する必要がある場合に役立ちます。

構文

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

引数

array1

同等の要素を持つ任意のタイプの ARRAY。

array2

array1 の要素と最小共通タイプを共有する要素の ARRAY。

戻り型

ARRAY_EXCEPT 関数は、重複することなく、配列 1 に一致するタイプの ARRAY を返します。

例

この例では、最初の配列に要素 1、2、3 [1, 2, 3]が含まれています。2 番目の配列には、要素 2、3、および 4 [2, 3, 4]が含まれています。このarray_except関数は、2 番目の配列にも存在するため、最初の配列から要素 2 と 3 を削除します。結果の出力は配列 です[1]。

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

この例では、最初の配列に要素 1、2、3 [1, 2, 3]が含まれています。2 番目の配列には、要素 1、3、5 [1, 3, 5]が含まれます。array_except 関数は、2 番目の配列にも存在するため、最初の配列から要素 1 と 3 を削除します。結果の出力は配列 です[2]。

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

ARRAY_INTERSECT 関数

ARRAY_INTERSECT 関数は 2 つの配列を引数として受け取り、両方の入力配列に存在する要素を含む新しい配列を返します。この関数は、2 つの配列間で共通の要素を見つける必要がある場合に便利です。これは、2 つのデータセット間の交差を見つけるなど、配列に対してセットのようなオペレーションを実行する必要がある場合に役立ちます。

構文

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

引数

array1

同等の要素を持つ任意のタイプの ARRAY。

array2

array1 の要素と最小共通タイプを共有する要素の ARRAY。

戻り型

ARRAY_INTERSECT 関数は、配列 1 と配列 2 の両方に含まれる重複や要素なしで、配列 1 に一致するタイプの ARRAY を返します。

例

この例では、最初の配列に要素 1、2、3 [1, 2, 3]が含まれています。2 番目の配列には、要素 1、3、5 [1, 3, 5]が含まれます。ARRAY_INTERSECT 関数は、1 と 3 の 2 つの配列間の共通要素を識別します。結果の出力配列は です[1, 3]。

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

ARRAY_JOIN 関数

ARRAY_JOIN 関数は 2 つの引数を取ります。最初の引数は、結合される入力配列です。2 番目の引数は、配列要素の連結に使用される区切り文字列です。この関数は、文字列の配列 (またはその他のデータ型) を 1 つの連結文字列に変換する必要がある場合に便利です。これは、表示目的や今後の処理など、値の配列を単一の形式の文字列として表示する場合に役立ちます。

構文

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

引数

array

任意の ARRAY 型、ただしその要素は文字列として解釈されます。

delimiter

連結された配列要素を分離するために使用される STRING。

nullReplacement

結果で NULL 値を表すために使用される STRING。

戻り型

ARRAY_JOIN 関数は、配列の要素が区切り文字で区切られ、null 要素が に置き換えられる STRING を返しますnullReplacement。nullReplacement を省略すると、null要素は除外されます。引数が の場合NULL、結果は ですNULL。

例

この例では、ARRAY_JOIN 関数は配列['hello', 'world']を取得し、区切り文字 ' ' (スペース文字) を使用して要素を結合します。結果の出力は文字列 です'hello world'。

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

この例では、ARRAY_JOIN 関数は配列['hello', null, 'world']を取得し、区切り文字 ' ' (スペース文字) を使用して要素を結合します。null 値は、指定された置換文字列 ',' (カンマ) に置き換えられます。結果の出力は文字列 です'hello , world'。

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

ARRAY_REMOVE 関数

ARRAY_REMOVE 関数は 2 つの引数を取ります。最初の引数は、要素が削除される入力配列です。2 番目の引数は、配列から削除される値です。この関数は、配列から特定の要素を削除する必要がある場合に便利です。これは、値の配列に対してデータクリーニングまたは前処理を実行する必要がある場合に役立ちます。

構文

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

引数

array

ARRAY。

element

配列の要素と最も一般的でない型を共有する型の式。

戻り型

ARRAY_REMOVE 関数は、配列のタイプに一致する結果タイプを返します。削除する要素が の場合NULL、結果は ですNULL。

例

この例では、ARRAY_REMOVE 関数は 配列[1, 2, 3, null, 3]を取得し、値 3 のすべての出現を削除します。結果の出力は配列 です[1, 2, null]。

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

ARRAY_UNION 関数

ARRAY_UNION 関数は 2 つの配列を引数として受け取り、両方の入力配列の一意の要素を含む新しい配列を返します。この関数は、2 つの配列を組み合わせて重複要素を排除する必要がある場合に便利です。これは、2 つのデータセット間の結合を見つけるなど、配列に対してセットのようなオペレーションを実行する必要がある場合に役立ちます。

構文

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

引数

array1

ARRAY。

array2

array1 と同じタイプの ARRAY。

戻り型

ARRAY_UNION 関数は、配列と同じタイプの ARRAY を返します。

例

この例では、最初の配列に要素 1、2、3 [1, 2, 3]が含まれています。2 番目の配列には、要素 1、3、5 [1, 3, 5]が含まれます。ARRAY_UNION 関数は、両方の配列の一意の要素を組み合わせて、出力配列 を生成します[1, 2, 3, 5]。T

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

EXPLODE 関数

EXPLODE 関数は、配列またはマップ列を持つ単一の行を複数の行に変換するために使用されます。各行は配列またはマップの単一の要素に対応します。

構文

```
explode(expr)
```

引数

expr

配列式またはマップ式。

戻り型

EXPLODE 関数は行のセットを返します。各行は入力配列またはマップの 1 つの要素を表します。

出力行のデータ型は、入力配列またはマップ内の要素のデータ型によって異なります。

例

次の例では、単一行配列 [10, 20] を取得し、それぞれ配列要素 (10 と 20) の 1 つを含む 2 つの別々の行に変換します。

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

最初の例では、入力配列が引数として に直接渡されましたexplode( )。この例では、入力配列は =>構文を使用して指定されます。ここで、列名 (collection) は明示的に指定されます。

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

どちらのアプローチも有効であり、同じ結果が得られますが、2 番目の構文は、単純な配列リテラルではなく、より大きなデータセットから列を爆発させる必要がある場合に便利です。

FLATTEN 関数

FLATTEN 関数は、ネストされた配列構造を単一のフラット配列に「フラット化」するために使用されます。

構文

```
flatten(arrayOfArrays)
```

引数

arrayOfArrays

配列の配列。

戻り型

FLATTEN 関数は配列を返します。

例

この例では、入力は 2 つの内部配列を持つネストされた配列であり、出力は内部配列のすべての要素を含む単一のフラット配列です。FLATTEN 関数は、ネストされた配列 [[1, 2], [3, 4]]を取得し、すべての要素を 1 つの配列 に結合します[1, 2, 3, 4]。

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

条件式

SQL では、条件式を使用して、特定の条件に基づいて決定を行います。これにより、SQL ステートメントのフローを制御し、1 つ以上の条件の評価に基づいて異なる値を返したり、異なるアクションを実行したりできます。

AWS Clean Rooms は、次の条件式をサポートしています。

トピック

- CASE 条件式
- COALESCE 式
- GREATEST 式と LEAST 式
- IF expression
- IS_NULL 式
- IS_NOT_NULL 式
- NVL および COALESCE 関数
- NVL2 関数
- NULLIF 関数

CASE 条件式

CASE 式は条件式であり、他の言語で使われる if/then/else ステートメントと似ています。CASE は、複数の条件がある場合に結果を指定するために使用されます。SELECT コマンドでなど、SQL 式が有効な場合に CASE を使用してください。

2 種類の CASE 式 (簡易および検索) があります。

- 簡易 CASE 式では、式は値と比較されます。一致が検出された場合、THEN 句で指定されたアクションが適用されます。一致が検出されない場合、ELSE 句のアクションが適用されます。
- 検索 CASE 式では、CASE ごとにブール式に基づいて検証され、CASE ステートメントは最初を一致する CASE を返します。WHEN 句で検出されない場合、ELSE 句のアクションが返されます。

構文

条件を満たすために使用される簡易 CASE ステートメント

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

各条件を検証するために使用する検索 CASE ステートメント

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

引数

expression

列名または有効な式。

value

数値定数または文字列などの式を比較する値。

result

式またはブール条件が検証されるときに返されるターゲット値または式。すべての結果式のデータタイプは、単一の出力タイプに変換できる必要があります。

condition

true または false に評価される Boolean 式。条件が true の場合、CASE 式の値は条件に続く結果であり、残りの CASE 式は処理されません。条件が false の場合、後続の WHEN 句はすべて評価されます。WHEN 条件の結果がどれも true ではない場合、CASE 式の値が ELSE 句の結果になります。ELSE 句が省略され、どの条件も true ではない場合、結果は Null となります。

例

簡易 CASE 式を使用し、VENUE テーブルに対するクエリで `New York City` を `Big Apple` に置換します。その他すべての都市名を `other` に置換します。

```
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
...
```

検索 CASE 式を使用し、それぞれのチケット販売の PRICEPAID 値に基づいてグループ番号を割り当てます。

```
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 式

COALESCE 式は、Null ではないリストの最初の式の値を返します。すべての式が null の場合、結果は null になります。null 以外の値が見つかると、リスト内の残りの式は検証されません。

このタイプの式は、優先する値がないか Null の場合に何かにバックアップ値を返すときに役に立ちます。例えば、クエリは 3 つの電話番号 (携帯、自宅、職場の順) のうち、いずれかテーブルで最初に検出された 1 つを返す可能性があります (Null でない)。

構文

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

例

COALESCE 式を 2 つの列に適用します。

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

NVL 式のデフォルトの列名は COALESCE です。次のクエリは同じ結果を返します。

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

GREATEST 式と LEAST 式

任意の数の式のリストから最大値または最小値を返します。

構文

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

パラメータ

expression_list

列名などの式のカンマ区切りリスト。式はすべて一般的なデータ型に変換可能である必要があります。リスト内の NULL 値は無視されます。すべての式が NULL と評価された場合、結果は NULL になります。

戻り値

指定された式のリストから最大 (GREATEST の場合) または最小 (LEAST の場合) の値を返します。

例

次の例は、アルファベット順で最も高い firstname または lastname の値を返します。

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

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

IF expression

IF 条件関数は、条件に基づいて 2 つの値のいずれかを返します。

この関数は、条件の評価に基づいて決定を行い、異なる値を返すために SQL で使用される一般的なコントロールフローステートメントです。これは、クエリ内に単純な if-else ロジックを実装するのに役立ちます。

構文

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

引数

expr1

評価される条件または式。の場合true、関数は expr2 の値を返します。expr1 が の場合false、関数は expr3 の値を返します。

expr2

expr1 が の場合に評価および返される式true。

expr3

expr1 が の場合に評価および返される式false。

戻り値

が にexpr1評価された場合true、 は を返しexpr2、それ以外の場合は を返しますexpr3。

例

次の例では、 if()関数を使用して、条件に基づいて 2 つの値のいずれかを返します。評価される条件は です1 < 2。これは であるためtrue、最初の値'a'が返されます。

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

IS_NULL 式

IS_NULL 条件式は、値が null かどうかをチェックするために使用されます。

この式は のシノニムですIS NULL。

構文

```
is_null(expr)
```

引数

expr

任意のタイプの式。

戻り値

IS_NULL 条件式はブール値を返します。expr1 が NULL の場合、 は を返しtrue、それ以外の場合は を返しますfalse。

例

次の例では、値が 1 null かどうかをチェックし、1 が有効な NULL 以外の値trueであるため、ブール値を返します。

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

次の例では、squirrelsテーブルからid列を選択しますが、経過時間列が である行に対してのみ選択しますnull。

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

IS_NOT_NULL 式

IS_NOT_NULL 条件式は、値が null でないかどうかを確認するために使用されます。

この式は のシノニムですIS NOT NULL。

構文

```
is_not_null(expr)
```

引数

expr

任意のタイプの式。

戻り値

IS_NOT_NULL 条件式はブール値を返します。expr1 が NULL でない場合、 は を返しtrue、それ以外の場合は を返しますfalse。

例

次の例では、値が null でないかどうかをチェックし、1 1が有効な NULL 以外の値trueであるため、ブール値を返します。

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

次の例では、squirrelsテーブルからid列を選択しますが、経過時間列が ではない行に対してのみ選択しますnull。

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

NVL および COALESCE 関数

一連の式の中で、Null 以外の最初の式の値を返します。Null 以外の値が見つかると、リスト内の残りの式は評価されません。

NVL は COALESCE と同じです。これらはシノニムです。このトピックでは、両方の構文について説明し、例を示します。

構文

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

COALESCE の構文は同じです。

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

すべての式が null の場合、結果は null になります。

これらの関数は、プライマリ値がないか Null の場合にセカンダリ値を返すときに役に立ちます。例えば、クエリを実行すると、使用可能な 3 つの電話番号 (携帯、自宅、職場) のうち最初の電話番号が返されることがあります。関数内の式の順序によって、評価の順序が決まります。

引数

expression

Null ステータスが評価される列名などの式。

戻り型

AWS Clean Rooms は、入力式に基づいて返される値のデータ型を決定します。入力式のデータ型に共通の型がない場合は、エラーが返されます。

例

リストに整数式が含まれている場合、関数は整数を返します。

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

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

この例は前の例と同じですが、NVL を使用して、同じ結果が返される点が異なります。

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

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

次の例は、文字列型を返します。

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

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

次の例では、式リストのデータ型が異なるため、エラーになります。この場合、リストには文字列型と数値型の両方があります。

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

NVL2 関数

指定された式の結果が NULL か NOT NULL かに基づいて、2 つの値のいずれかを返します。

構文

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

引数

expression

Null ステータスが評価される列名などの式。

not_null_return_value

expression が NOT NULL に評価された場合に返される値。not_null_return_value 値は、expression と同じデータ型を持つか、そのデータ型に暗黙的に変換可能である必要があります。

null_return_value

expression が NULL に評価される場合に値が返されます。null_return_value 値は、expression と同じデータ型を持つか、そのデータ型に暗黙的に変換可能である必要があります。

戻り型

NVL2 の戻り値の型は、次のように決まります。

- not_null_return_value または null_return_value が null の場合、not-null 式のデータ型が返されます。

not_null_return_value と null_return_value がどちらも null である場合

- not_null_return_value と null_return_value のデータ型が同じ場合、そのデータ型が返されます。
- not_null_return_value と null_return_value の数値データ型が異なる場合、互換性を持つ最小の数値データ型が返されます。
- not_null_return_value と null_return_value の日時データ型が異なる場合、タイムスタンプデータ型が返されます。
- not_null_return_value と null_return_value の文字データ型が異なる場合、not_null_return_value のデータ型が返されます。
- not_null_return_value と null_return_value で数値データ型と数値以外のデータ型が混合している場合、not_null_return_value のデータ型が返されます。

Important

not_null_return_value のデータ型が返される最後の 2 つのケースでは、null_return_value がそのデータ型に暗黙的にキャストされます。データ型に互換性がない場合、関数は失敗します。

使用に関する注意事項

NVL2 では、not_null_return_value または null_return_value パラメータのうち、どちらか関数により選択された方の値が戻り値に含められますが、データ型については not_null_return_value のデータ型が含められます。

例えば、column1 が NULL の場合、次のクエリは同じ値を返します。ただし、DECODE の戻り値のデータ型は INTEGER となり、NVL2 の戻り値のデータ型は VARCHAR になります。

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

例

次の例では、一部のサンプルデータを変更し、2 つのフィールドを評価してユーザーに適切な連絡先情報を提供します。

```
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 関数

2 つの引数を比較し、引数が等しい場合は Null を返します。等しくない場合、最初の引数が返されます。

構文

NULLIF 式は 2 つの引数を比較し、引数が等しい場合に Null を返します。等しくない場合、最初の引数が返されます。この式は NVL または COALESCE 式の逆です。

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

引数

expression1, expression2

比較対象の列または式。戻り値の型は、最初の式の型と同じです。

例

次の例では、引数が等しくないため、クエリは `first` 文字列を返します。

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

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

次の例では、文字列リテラル引数が等しいため、クエリは NULL を返します。

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

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

次の例では、整数の引数が等しくないため、クエリは 1 を返します。

```
SELECT NULLIF(1, 2);
```

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

次の例では、整数の引数が等しいため、クエリは NULL を返します。

```
SELECT NULLIF(1, 1);

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

次の例に示すクエリは、LISTID 値と SALESID 値が一致する場合に Null を返します。

```
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)
```

コンストラクター関数

SQL コンストラクタ関数は、配列やマップなどの新しいデータ構造を作成するために使用される関数です。

いくつかの入力値を取得し、新しいデータ構造オブジェクトを返します。コンストラクタ関数は通常、ARRAY や MAP など、作成するデータ型にちなんで命名されます。

コンストラクタ関数は、スカラー関数や集計関数とは異なり、既存のデータに対して動作し、単一の値を返します。コンストラクタ関数は、新しいデータ構造を作成するために使用されます。このデータ構造は、さらなるデータ処理や分析に使用できます。

AWS Clean Rooms は、次のコンストラクタ関数をサポートしています。

トピック

MAP コンストラクタ関数

MAP コンストラクタ関数は、指定されたキーと値のペアを持つマップを作成します。

MAP などのコンストラクタ関数は、SQL クエリ内でプログラムで新しいデータ構造を作成する必要がある場合に役立ちます。これにより、さらなるデータ処理や分析に使用できる複雑なデータ構造を構築できます。

構文

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

引数

キー0

同等の任意のタイプの式。すべての key0 は、最も一般的でないタイプを共有する必要があります。

値0

任意のタイプの式。すべての valueN は、最小共通タイプを共有する必要があります。

戻り値

MAP 関数は、最も一般的でないタイプの key0 として入力されたキーと、最も一般的でないタイプの value0 として入力された値を持つ MAP を返します。

例

次の例では、2 つのキーと値のペアを持つ新しいマップを作成します。キー `1.0`は値 に関連付けられています`'2'`。キー`3.0`は値 に関連付けられます`'4'`。その後、結果のマップが SQL ステートメントの出力として返されます。

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

NAMED_STRUCT コンストラクタ関数

NAMED_STRUCT コンストラクタ関数は、指定されたフィールド名と値を持つ構造体を作成します。

NAMED_STRUCT などのコンストラクタ関数は、SQL クエリ内でプログラムで新しいデータ構造を作成する必要がある場合に役立ちます。これにより、構造体やレコードなどの複雑なデータ構造を構築できます。これは、さらなるデータ処理や分析に使用できます。

構文

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

引数

名前1

STRING リテラル命名フィールド 1。

val1

フィールド 1 の値を指定する任意のタイプの式。

戻り値

NAMED_STRUCT 関数は、val1 のタイプに一致するフィールド 1 を持つ構造体を返します。

例

次の例では、3 つの名前付きフィールドを持つ新しい構造体を作成します。 フィールド"a"には値が割り当てられます1。フィールドには 値が"b"割り当てられます2.。 フィールド"c"には 値が割り当てられます3。結果として得られる構造体は、SQL ステートメントの出力として返されます。

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

STRUCT コンストラクタ関数

STRUCT コンストラクタ関数は、指定されたフィールド値を持つ構造体を作成します。

STRUCT などのコンストラクタ関数は、SQL クエリ内でプログラムで新しいデータ構造を作成する必要がある場合に役立ちます。これにより、構造体やレコードなどの複雑なデータ構造を構築できます。これは、さらなるデータ処理や分析に使用できます。

構文

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

引数

col1

列名または有効な式。

戻り値

STRUCT 関数は、expr1 のタイプに一致する field1 を持つ構造体を返します。

引数が参照という名前の場合、名前はフィールドの名前に使用されます。それ以外の場合、フィールドの名前は colN です。ここで、N は構造体内のフィールドの位置です。

例

次の例では、3 つのフィールドを持つ新しい構造体を作成します。最初のフィールドには値 1 が割り当てられます。2 番目のフィールドには値 2 が割り当てられます。3 番目のフィールドには値 3 が割り当てられます。デフォルトでは、結果の構造体のフィールドにはcol1、引数リスト内の位置に基づいて、col2、col3、および という名前が付けられます。結果として得られる構造体は、SQL ステートメントの出力として返されます。

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

データ型フォーマット関数

データ型フォーマット関数を使用すると、値のデータ型を変換できます。これらの各関数では必ず、最初の引数にはフォーマットする値を指定し、2 番目の引数には新しい形式のテンプレートを指定します。

AWS Clean Rooms Spark SQL は、複数のデータ型フォーマット関数をサポートしています。

トピック

- BASE64 関数
- CAST 関数
- DECODE 関数
- ENCODE 関数
- HEX 関数
- STR_TO_MAP 関数
- TO_CHAR
- TO_DATE 関数
- TO_NUMBER
- UNBASE64 関数
- UNHEX 関数
- 日時形式の文字列
- 数値形式の文字列

BASE64 関数

BASE64 関数は、MIME の RFC2045 Base64 転送エンコーディングを使用して式を base 64 文字列に変換します。

構文

```
base64(expr)
```

引数

expr

関数が BINARY として解釈する BINARY 式または STRING。

戻り型

STRING

例

指定された文字列入力を Base64 でエンコードされた表現に変換するには、次の例を使用します。その結果、入力文字列「Spark SQL」の Base64 エンコード表現がU3BhcmsgU1FM」になります。

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

CAST 関数

CAST 関数は、1 つのデータ型を互換性のある別のデータ型に変換します。例えば、文字列を日付に変換したり、数値型を文字列に変換したりできます。CAST はランタイム変換を実行します。つまり、変換によってソーステーブル内の値のデータ型は変更されません。クエリのコンテキストでのみ変更されます。

特定のデータ型では、CAST 関数を使用して他のデータ型に明示的に変換する必要があります。他のデータ型は、CAST を使用せずに、別のコマンドの一部として暗黙的に変換できます。「型の互換性と変換」を参照してください。

構文

式のデータ型を別のデータ型に変換するには、次の 2 つの同等な構文フォームのいずれかを使用します。

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

引数

expression

1 つ以上の値 (列名、値など) に評価される式。null 値を変換すると、null が返されます。式に、空白または空の文字列を含めることはできません。

type

BINARY データ型 および BINARY VARYING データ型を除く、サポートされている の 1 つ。

戻り型

CAST は、type 引数で指定されたデータ型を返します。

> **Note**
>
> AWS Clean Rooms は、次のような精度を失った DECIMAL 変換など、問題のある変換を実行しようとすると、エラーを返します。

```
select 123.456::decimal(2,1);
```

また、次の INTEGER 変換では、オーバーフローが生じます。

```
select 12345678::smallint;
```

例

次の 2 つのクエリは同等です。どちらも 10 進値を整数に変換します。

```
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)
```

次の場合も同様の結果が得られます。サンプルデータの実行は必要ありません。

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

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

次の例では、タイムスタンプ列の値を日付に変換して、各結果から時刻を削除します。

```
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)
```

前のサンプルで示した CAST を使用しなかった場合、結果には 2008-02-18 02:36:48 という時刻が含まれます。

次のクエリは、可変文字データを日付に変換します。実行にサンプルデータは必要ありません。

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

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

次の例では、日付列の値をタイムスタンプに変換します。

```
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)
```

前のサンプルのような場合は、TO_CHAR を使用して出力形式をさらに制御できます。

次の例では、整数が文字列に変換されます。

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

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

次の例では、DECIMAL(6,3) 値を DECIMAL(4,1) 値に変換します。

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

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

この例は、より複雑な式を示しています。SALES テーブル内の PRICEPAID 列 (DECIMAL(8,2) 列) を DECIMAL(38,2) 列に変換し、結果の値を 100000000000000000000 で乗算します。

```
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 関数

DECODE 関数は ENCODE 関数と同等であり、特定の文字エンコードを使用して文字列をバイナリ形式に変換するために使用されます。DECODE 関数はバイナリデータを取得し、指定された文字エンコードを使用して読み取り可能な文字列形式に変換します。

この関数は、データベースに保存されているバイナリデータを操作し、人間が読める形式で表示する必要がある場合や、異なる文字エンコード間でデータを変換する必要がある場合に便利です。

構文

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

引数

expr

文字セットでエンコードされた BINARY 式。

文字セット

STRING 式。

サポートされている文字セットエンコーディング (大文字と小文字を区別しない): 'US-ASCII'、'ISO-8859-1'、'UTF-8'、'UTF-16BE'、'UTF-16LE'、および 'UTF-16'。

戻り型

DECODE 関数は STRING を返します。

例

次の例では、UTF-8 文字エンコーディングを使用してメッセージデータをバイナリ形式でmessage_text保存する という列messagesを持つ というテーブルがあります。DECODE 関数は、バイナリデータを読み取り可能な文字列形式に変換します。このクエリの出力は、メッセージテーブルに保存されているメッセージの読み取り可能なテキストで、ID は で123、 エン'utf-8'コードを使用してバイナリ形式から文字列に変換されます。

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

ENCODE 関数

ENCODE 関数は、指定された文字エンコードを使用して文字列をバイナリ表現に変換するために使用されます。

この関数は、バイナリデータを使用する必要がある場合や、異なる文字エンコーディング間で変換する必要がある場合に便利です。たとえば、バイナリストレージを必要とするデータベースにデータを保存する場合や、異なる文字エンコードを使用するシステム間でデータを転送する必要がある場合に、ENCODE 関数を使用できます。

構文

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

引数

str

エンコードされる STRING 式。

文字セット

エンコードを指定する STRING 式。

サポートされている文字セットエンコーディング (大文字と小文字を区別しない): 'US-ASCII'、'ISO-8859-1'、'UTF-8'、'UTF-16BE'、'UTF-16LE'、および 'UTF-16'。

戻り型

ENCODE 関数は BINARY を返します。

例

次の例では、エン'utf-8'コードを使用して文字列を'abc'バイナリ表現に変換します。この場合、元の文字列が返されます。これは、エン'utf-8'コードが可変幅の文字エンコードであり'a'、文字ごとに 1 バイトを使用して ASCII 文字セット全体 (、'b'、 を含む'c') を表すことができるためです。したがって、 'abc'を使用する のバイナリ表現'utf-8'は、元の文字列と同じです。

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

HEX 関数

HEX 関数は、数値 (整数または浮動小数点数) を対応する 16 進文字列表現に変換します。

16 進数は、16 個の異なる記号 (0～9 および A～F) を使用して数値を表す数値システムです。コンピュータサイエンスやプログラミングで一般的に使用され、バイナリデータをよりコンパクトで人間が読める形式で表現します。

構文

```
hex(expr)
```

引数

expr

BIGINT、BINARY、または STRING 式。

戻り型

HEX は STRING を返します。関数は、引数の 16 進数表現を返します。

例

次の例では、整数値 17 を入力として受け取り、それに HEX() 関数を適用します。出力は です。これは11入力値 の 16 進数表現です17。

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

次の例では、文字列を 16 進数表現に変換'Spark_SQL'します。出力は です。これは537061726B2053514C入力文字列 の 16 進数表現です'Spark_SQL'。

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

この例では、文字列「Spark_SQL」は次のように変換されます。

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

これらの 16 進値の連結は、最終的な出力「」になります537061726B2053514C"。

STR_TO_MAP 関数

STR_TO_MAP 関数は、string-to-map変換関数です。マップ (またはディクショナリ) の文字列表現を実際のマップデータ構造に変換します。

この関数は、SQL でマップデータ構造を操作する必要がある場合に便利ですが、データは最初は文字列として保存されます。文字列表現を実際のマップに変換することで、マップデータのオペレーションと操作を実行できます。

構文

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

引数

text

マップを表す STRING 式。

pairDelim

エントリを分離する方法を指定するオプションの STRING リテラル。デフォルトはカンマ () です ' , '。

keyValueDelim

各キーと値のペアを分離する方法を指定するオプションの STRING リテラル。デフォルトではコロン () になります ' : '。

戻り型

STR_TO_MAP 関数は、キーと値の両方について STRING の MAP を返します。pairDelim と keyValueDelim はどちらも正規表現として扱われます。

例

次の例では、入力文字列と 2 つの区切り文字引数を取得し、文字列表現を実際のマップデータ構造に変換します。この特定の例では、入力文字列はキーと値のペアを持つマップ'a:1,b:2,c:3'を表します。 'a' はキー、 '1' は値です。 'b'はキー、 '2'は値です。 はキー、 は 値です。 'c' はキー、 '3'は 値です。',' 区切り記号を使用してキーと値のペアを区切り、':'区切り記号を使用して各ペア内のキーと値を区切ります。このクエリの出力は です{"a":"1","b":"2","c":"3"}。これは結果のマップデータ構造で、キーは 'a'、、'b'および 'c'で、対応する値は '1'、'2'、および です'3'。

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

次の例は、STR_TO_MAP 関数が入力文字列が特定の形式で、キーと値のペアが正しく区切られていることを想定していることを示しています。入力文字列が想定された形式と一致しない場合、関数は引き続きマップの作成を試みますが、結果の値は想定どおりではない可能性があります。

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

TO_CHAR

TO_CHAR は、タイムスタンプまたは数値式を文字列データ形式に変換します。

構文

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

引数

timestamp_expression

TIMESTAMP 型または TIMESTAMPTZ 型の値、またはタイムスタンプに暗黙的に強制変換できる値を生成する式。

numeric_expression

数値データ型の値、または数値型に暗黙的に強制変換できる値が生成される式。詳細については、「数値型」を参照してください。TO_CHAR は、数文字列の左側にスペースを挿入します。

> **Note**
>
> TO_CHAR は、128 ビットの DECIMAL 値をサポートしません。

format

新しい値の形式。有効な形式については、「日時形式の文字列」および「数値形式の文字列」を参照してください。

戻り型

VARCHAR

例

次の例では、月の名前を 9 文字に埋め込み、曜日の名前と日付の数字を指定した形式でタイムスタンプを日付と時刻の値に変換します。

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

次の例では、タイムスタンプを日付の番号の値に変換します。

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

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

次の例では、タイムスタンプを曜日の番号の値に変換します。

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

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

以下の例では、日付から月を抽出しています。

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

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

次の例は、EVENT テーブル内の各 STARTTIME 値を、時、分、および秒から成る文字列に変換します。

```
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)
```

次の例は、タイムスタンプの値全体を別の形式に変換します。

```
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)
```

次の例は、タイムスタンプリテラルをキャラクタ文字列に変換します。

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

```
to_char
----------
23:15:59
(1 row)
```

次の例では、数字を負の記号を末尾に付けた文字列に変換します。

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

次の例では、数字を通貨記号付きの文字列に変換します。

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

次の例では、負の数字に角括弧を使用して、数字を文字列に変換します。

```
select to_char(-125.88, '$999D99PR');
to_char
---------
$<125.88>
(1 row)
```

次の例では、数字をローマ字の文字列に変換します。

```
select to_char(125, 'RN');
to_char
---------
CXXV
(1 row)
```

次の例では、曜日を表示します。

```
SELECT to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS');
```

```
               to_char
-----------------------
Wednesday, 31 09:34:26
```

次の例では、数に対して序数のサフィックスを表示します。

```
SELECT to_char(482, '999th');
               to_char
-----------------------
 482nd
```

次の例では、販売テーブル内の支払い価格からコミッションを減算します。誤差は四捨五入されて、ローマ数字に変換され、to_char 列に表示されます。

```
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)
```

次の例では、さまざまな値に通貨記号を追加して、to_char 列に表示します。

```
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)
```

次の例では、各販売が行われた世紀を一覧で示します。

```
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)
```

次の例は、EVENT テーブル内の各 STARTTIME 値を、時、分、秒、およびタイムゾーンから成る文字列に変換します。

```
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)
```

以下の例では、秒、ミリ秒、マイクロ秒の形式を示しています。

```
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 関数

TO_DATE は、文字列で表記された日付を DATE データ型に変換します。

構文

```
TO_DATE (date_str)
```

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

引数

date_str

日付文字列または日付文字列にキャストできるデータ型。

format

Spark の日時パターンに一致する文字列リテラル。有効な日時パターンについては、「フォーマットと解析の日時パターン」を参照してください。

戻り型

TO_DATE は、format の値に応じて DATE を返します。

フォーマットへの変換が失敗すると、エラーが返されます。

例

次の SQL ステートメントは、日付 02 Oct 2001 を日付データ型に変換します。

```
select to_date('02 Oct 2001', 'dd MMM yyyy');

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

次の SQL ステートメントは、文字列 20010631 を日付に変換します。

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

次の SQL ステートメントは、文字列 20010631 を日付に変換します。

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

6 月には 30 日しかないため、結果は null 値です。

```

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

TO_NUMBER

TO_NUMBER は、文字列を数値 (10 進) に変換します。

構文

```
to_number(string, format)
```

引数

string

変換する文字列。形式はリテラル値である必要があります。

format

2 番目の引数は、数値を作成するために文字列を解析する方法を示す書式文字列です。例えば、形式 '99D999' では、変換する文字列が 5 つの数字で構成され、3 番目の位置に小数点が挿入されます。たとえば、to_number('12.345','99D999')は数値として 12.345 を返します。有効な形式の一覧については、「数値形式の文字列」を参照してください。

戻り型

TO_NUMBER は DECIMAL 型の数値を返します。

フォーマットへの変換が失敗すると、エラーが返されます。

例

次の例では、文字列 12,454.8- を数値に変換します。

```
select to_number('12,454.8-', '99G999D9S');

to_number
-----------
-12454.8
```

次の例では、文字列 $ 12,454.88 を数値に変換します。

```
select to_number('$ 12,454.88', 'L 99G999D99');

to_number
-----------
12454.88
```

次の例では、文字列 $ 2,012,454.88 を数値に変換します。

```
select to_number('$ 2,012,454.88', 'L 9,999,999.99');

to_number
-----------
2012454.88
```

UNBASE64 関数

UNBASE64 関数は、引数をベース 64 文字列からバイナリに変換します。

Base64 エンコーディングは、さまざまな通信チャネル (E メール、URL パラメータ、データベースストレージなど) を介した送信に安全なテキスト形式でバイナリデータ (イメージ、ファイル、暗号化された情報など) を表すために一般的に使用されます。

UNBASE64 関数を使用すると、このプロセスを逆にして元のバイナリデータを復元できます。このタイプの機能は、データ転送メカニズムとして Base64 を使用する外部システムや APIs と統合する場合など、Base64 形式でエンコードされたデータを操作する必要がある場合に役立ちます。

構文

```
unbase64(expr)
```

引数

expr

base64 形式の STRING 式。

戻り型

BINARY

例

次の例では、Base64-encodedされた文字列'U3BhcmsgU1FM'は元の文字列 に変換されます'Spark SQL'。

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

UNHEX 関数

UNHEX 関数は、16 進数の文字列を元の文字列表現に変換します。

この関数は、16 進形式で保存または送信されたデータを操作する必要があり、さらに処理または表示するために元の文字列表現を復元する必要がある場合に役立ちます。

UNHEX 関数は、HEX 関数のカウンターパートです。

構文

```
unhex(expr)
```

引数

expr

16 進数文字の STRING 式。

戻り型

UNHEX は BINARY を返します。

expr の長さが奇数の場合、最初の文字は破棄され、結果は null バイトで埋められます。expr に 16 進数以外の文字が含まれている場合、結果は NULL です。

例

次の例では、UNHEX() 関数と DECODE() 関数を一緒に使用して、16 進数の文字列を元の文字列表現に変換します。クエリの最初の部分では、UNHEX() 関数を使用して 16 進文字列 '537061726B2053514C' をバイナリ表現に変換します。クエリの 2 番目の部分では、DECODE() 関数を使用して、UNHEX() 関数から取得したバイナリデータを文字列に変換し、'UTF-8' 文字エンコーディングを使用します。クエリの出力は、16 進数に変換され、文字列に戻された元の文字列「Spark_SQL」です。

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

日時形式の文字列

以下の一般的なシナリオでは、日時パターンを使用できます。

- CSV および JSON データソースを使用して日時コンテンツを解析およびフォーマットする場合
- 次のような関数を使用して文字列型と日付型またはタイムスタンプ型の間で変換する場合:
  - unix_timestamp
  - date_format
  - to_unix_timestamp
  - from_unixtime

- to_date
- to_timestamp
- from_utc_timestamp
- to_utc_timestamp

日付とタイムスタンプの解析と書式設定には、次の表のパターン文字を使用します。

日付部分または時刻部分	意味	例
ある	当日の午前 または午後、午前 - 午後として表示	PM
D	3 桁の数字で表示される曜日	189
d	2 桁の数字で表示される日付	28
E	テキストとして表示される曜日	火 火曜日
F	1 桁の数字で表示される、その月の曜日の一致	3
G	テキストとして表示される Era インジケータ	AD アンノ・ドミニ
h	午前または午後の時刻。2 桁の数字で表示	12
H	0～23 の 2 桁の数字で表示される時刻	0
k	1～24 の 2 桁の数字で表示される時刻	1
K	0～11 の 2 桁の数字で表示される午前または午後の時間	0

日付部分または時刻部分	意味	例
m	2 桁の数字で表示される分	30
M/L	月として表示される年月	7 07 7 月 7 月
O	UTC からのローカライズゾーンオフセット	GMT+8 GMT+8:00 UTC-08:00
Q/Q	数値 (1 ~ 4) またはテキストで表示される、年四半期	3 03 Q3 第 3 四半期
s	秒、2 桁の数字で表示	55
S	分数として表示される 1 秒の分数	978
V	zone-id として表示されるタイムゾーン識別子	America/Los_Angeles Z 08:30

日付部分または時刻部分	意味	例
x	UTC からのゾーンオフセット (offset-X)	+0000 -08 -0830 -08:30 -083015 -08:30:15
X	UTC からのゾーンオフセット。Z は 0 の場合	Z -08 -0830 -08:30 -083015 -08:30:15
y	年、年として表示	2020 20
z	タイムゾーン名、テキストとして表示	太平洋標準時 PST
Z	UTC からのゾーンオフセット (offset-Z)	+0000 -0800 -08:00
'	区切り文字として表示されるテキストのエスケープ	該当なし

日付部分または時刻部分	意味	例
"	リテラルとして表示される一重引用符	'
[	オプションのセクション開始	該当なし
]	オプションのセクション終了	該当なし

パターン文字の数によって形式タイプが決まります。

テキスト形式

- 省略形には 1~3 文字を使用します (月曜日には「月」など) 。
- 完全な形式 (「月曜日」など) には 4 文字のみを使用してください。
- 5 文字以上は使用しないでください。エラーが発生します。

数値形式 (n)

- 値 n は、許可される最大文字数を表します。
- 単一文字パターンの場合:
  - 出力はパディングなしで最小桁数を使用します
- 複数の文字パターンの場合:
  - 出力は文字数の幅に合わせてゼロで埋められます
- 解析する場合、入力には正確な桁数が含まれている必要があります

数値/テキスト形式

- 3 文字以上の場合は、テキスト形式ルールに従います。
- 文字数を減らすには、数値形式ルールに従います。

フラクション形式

- 1~9 文字の「S」文字を使用する (SSSSSS など)
- 解析の場合:

  - 1 から S 文字数までの小数を受け入れる
- フォーマットの場合:
  - S 文字数と一致するゼロのパッド
- マイクロ秒の精度で最大 6 桁をサポート
- ナノ秒を解析できますが、余分な桁は切り捨てます

年形式

- 文字数はパディングの最小フィールド幅を設定します
- 2 文字の場合:
  - 最後の 2 桁を出力します
  - 2000-2099 の間の年を解析する
- 4 文字未満の場合 (2 文字を除く):
  - 負の年のみの符号を表示します
- 7 文字以上は使用しないでください。エラーが発生します。

月の形式

- 標準フォームには「M」、スタンドアロンフォームには「L」を使用します。
- 単一の「M」または「L」:
  - パディングなしで月番号 1～12 を表示します
- 「MM」または「LL」:
  - パディング付きの月番号 01～12 を表示します
- 'MMM':
  - 略名を標準形式で表示する
  - 完全な日付パターンの一部である必要があります
- 'LLL':
  - 省略された月名をスタンドアロン形式で表示する
  - 月のみの書式設定に使用
- 「MMMM」:
  - 月名を標準形式で表示する

  - 日付とタイムスタンプに を使用する
- 'LLLL':
  - 完全な月名をスタンドアロン形式で表示する
  - 月のみの書式設定に使用

タイムゾーン形式

- am-pm: 1 文字のみを使用する
- ゾーン ID (V): 2 文字のみを使用する
- ゾーン名 (z):
  - 1 ~ 3 文字: 短縮名を表示します
  - 4 文字: フルネームを表示します
  - 5 文字以上は使用しないでください

オフセット形式

- X と x:
  - 1 文字: 時間 (+01) または時間/分 (+0130) を表示
  - 2 文字: コロンなしの時間/分を表示 (+0130)
  - 3 文字: コロンで時間/分を表示 (+01:30)
  - 4 文字: コロンなしhour-minute-secondを表示 (+013015)
  - 5 文字: hour-minute-secondをコロンで表示 (+01:30:15)
  - X はゼロオフセットに「Z」を使用します
  - x はゼロオフセットに「+00」、「+0000」、または「+00:00」を使用します
- O:
  - 1 文字: ショートフォームを表示 (GMT+8)
  - 4 文字: フルフォームを表示 (GMT+08:00)
- Z:
  - 1 ~ 3 文字: コロンなしの時間/分を表示 (+0130)
  - 4 文字: 完全なローカライズされたフォームを表示します
  - 5 文字: hour-minute-secondとコロンを表示

オプションのセクション

- 角括弧 [ ] を使用してオプションのコンテンツをマークする
- オプションのセクションをネストできます
- すべての有効なデータが出力に表示されます
- 入力はオプションセクション全体を省略できます

> **Note**
>
> 記号「E」、「F」、「q」、「Q」は、日時形式 (date_format など) でのみ機能します。日時解析 (to_timestamp など) には使用しないでください。

数値形式の文字列

以下の数値形式の文字列は、TO_NUMBER や TO_CHAR などの関数に適用されます。

- 文字列を数値の形式にする例については、「TO_NUMBER」を参照してください。
- 数字をも文字列の形式にする例については、「TO_CHAR」を参照してください。

形式	説明
9	指定された桁数の数値。
0	先頭に 0 が付いた数値。
. (ピリオド)、D	小数点。
, (カンマ)	桁区切り文字。
CC	世紀コード。例えば、21 世紀は 2001-01-01 から始まる (TO_CHAR のみでサポートされる)。
FM	フルモード。パディングとして使用されている空白とゼロを非表示にします。
PR	山括弧で囲まれた負の値。

形式	説明
S	数値にアンカーされる符号。
L	指定位置に挿入される貨幣記号。
G	グループ区切り文字。
MI	0 未満の数値の指定位置に挿入される負符号。
PL	0 より大きい数値の指定位置に挿入される正符号。
SG	指定位置に挿入される正または負符号。
RN	1 ~ 3999 のローマ数字 (TO_CHAR のみでサポートされる)。
TH または th	序数のサフィックス。0 未満の小数は変換されません。

日付および時刻関数

日付と時刻関数を使用すると、日付の一部の抽出、日付計算の実行、日付と時刻の書式設定、現在の日付と時刻の操作など、日付と時刻のデータに対して幅広いオペレーションを実行できます。これらの関数は、データ分析、レポート、時間的データを含むデータ操作などのタスクに不可欠です。

AWS Clean Rooms では、次の日付および時刻関数がサポートされています。

トピック

- ADD_MONTHS 関数
- CONVERT_TIMEZONE 関数
- CURRENT_DATE 関数
- CURRENT_TIMESTAMP 関数
- DATE_ADD 関数
- DATE_DIFF 関数
- DATE_PART 関数

- DATE_TRUNC 関数
- DAY 関数
- DAYOFMONTH 関数
- DAYOFWEEK 関数
- DAYOFYEAR 関数
- EXTRACT 関数
- FROM_UTC_TIMESTAMP 関数
- HOUR 関数
- MINUTE 関数
- MONTH 関数
- SECOND 関数
- TIMESTAMP 関数
- TO_TIMESTAMP 関数
- YEAR 関数
- 日付関数またはタイムスタンプ関数の日付部分

ADD_MONTHS 関数

ADD_MONTHS は日付またはタイムスタンプの値または式に、指定された月数を加算します。DATE_ADD 関数は同様の機能を提供します。

構文

```
ADD_MONTHS( {date | timestamp}, integer)
```

引数

date | timestamp

日付またはタイムスタンプの列、あるいは暗黙的に日付またはタイムスタンプに変換される式。date がその月の最終日である場合、または結果の月が短い場合、関数は結果に月の最終日を返します。その他の日付の場合、結果には date 式と同じ日数が含まれます。

integer

正または負の整数。負の数を使用し、日付から月を削除します。

戻り型

TIMESTAMP

例

次のクエリは、TRUNC 関数内の ADD_MONTHS 関数を使用します。TRUNC 関数は、ADD_MONTHS の結果から日付の時刻を削除します。ADD_MONTHS 関数は CALDATE 列の値ごとに 12 か月を追加します。

```
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)
```

次の例は、ADD_MONTHS 関数が異なる日数の月を持つ日付で実行される動作を示しています。

```
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 関数

CONVERT_TIMEZONE は、タイムスタンプのタイムゾーンを別のタイムゾーンに変換します。この関数は夏時間に合わせて自動的に調整されます。

構文

```
CONVERT_TIMEZONE ( ['source_timezone',] 'target_timezone', 'timestamp')
```

引数

source_timezone

(オプション) 現在のタイムスタンプのタイムゾーン。デフォルトは UTC です。

target_timezone

新しいタイムスタンプのタイムゾーン。

timestamp

タイムスタンプの列、あるいは暗黙的にタイムスタンプに変換される式。

戻り型

TIMESTAMP

例

次の例は、タイムスタンプ値をデフォルトの UTC タイムゾーンから PST に変換します。

```
select convert_timezone('PST', '2008-08-21 07:23:54');

 convert_timezone
------------------------
2008-08-20 23:23:54
```

次の例は、LISTTIME 列のタイムスタンプ値をデフォルトの UTC タイムゾーンから PST に変換します。タイムスタンプが夏時間の期間内であっても、変換後のタイムゾーンが略名 (PST) で指定されているため、標準時間に変換されます。

```
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
```

次の例は、タイムスタンプの LISTTIME 列をデフォルトの UTC タイムゾーンから US/Pacific タイムゾーンに変換します。変換後のタイムゾーンはタイムゾーン名で指定されており、タイムスタンプは夏時間の期間内であるため、この関数は夏時間を返します。

```
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
```

次の例は、タイムスタンプの文字列を EST から PST に変換します。

```
select convert_timezone('EST', 'PST', '20080305 12:25:29');

 convert_timezone
-------------------
2008-03-05 09:25:29
```

次の例は、変換後のタイムゾーンがタイムゾーン名 (America/New_York) で指定されており、タイムスタンプが標準時間の期間内にあるため、タイムスタンプを米国東部標準時に変換します。

```
select convert_timezone('America/New_York', '2013-02-01 08:00:00');

 convert_timezone
---------------------
2013-02-01 03:00:00
(1 row)
```

次の例は、変換後のタイムゾーンがタイムゾーン名 (America/New_York) で指定されており、タイムスタンプが夏時間の期間内にあるため、タイムスタンプを米国東部夏時間に変換します。

```
select convert_timezone('America/New_York', '2013-06-01 08:00:00');

 convert_timezone
---------------------
2013-06-01 04:00:00
(1 row)
```

次の例は、オフセットの使用を示しています。

```
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 関数

CURRENT_DATE は、現在のセッションのタイムゾーン (デフォルトは UTC) の日付をデフォルト形式 YYYY-MM-DD で返します。

> **Note**
>
> CURRENT_DATE は、現在のステートメントの開始日ではなく、現在のトランザクションの開始日を返します。複数のステートメントを含むトランザクションを 2008 年 10 月 1 日 23:59 に開始し、CURRENT_DATE を含むステートメントが 2008 年 10 月 2 日 00:00 に実行されるシナリオを考えてみましょう。CURRENT_DATE は 10/02/08 ではなく、10/01/08 を返します。

構文

```
CURRENT_DATE
```

戻り型

DATE

例

次の例では、現在の日付 (関数 AWS リージョン が実行される) を返します。

```
select current_date;
```

```
   date
------------
2008-10-01
```

CURRENT_TIMESTAMP 関数

CURRENT_TIMESTAMP は、日付、時刻、および (オプションで) ミリ秒またはマイクロ秒を含む現在の日付と時刻を返します。

この関数は、イベントのタイムスタンプの記録、時間ベースの計算の実行、日付/時刻列の入力など、現在の日付と時刻を取得する必要がある場合に便利です。

構文

```
current_timestamp()
```

戻り型

CURRENT_TIMESTAMP 関数は DATE を返します。

例

次の例では、クエリが実行された時点の現在の日時を返します。2020 年 4 月 25 日、15:49:11.914 (午後 3:49:11.914)。

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

次の の例では、squirrelsテーブルの各行の現在の日付と時刻を取得します。

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

DATE_ADD 関数

start_date から num_days の日付を返します。

構文

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

引数

start_date

開始日の値。

num_days

追加する日数 (整数) 。正の数値は日数を加算し、負の数値は日数を減算します。

戻り型

DATE

例

次の例では、日付に 1 日を追加します。

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

Result:
2016-07-31
```

次の例では、複数日を追加します。

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

Result:
2016-08-04
```

使用に関する注意事項

このドキュメントは、Spark SQL の DATE_ADD 関数を対象としています。この関数は、他の SQL バリアントと比較して、日付に日数を追加するシンプルなインターフェイスを提供します。月や年などの他の間隔を追加するには、異なる関数が必要になる場合があります。

DATE_DIFF 関数

DATE_DIFF は、2 つの日付または時刻式の日付部分の差を返します。

構文

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

引数

endDate

DATE 式。

startDate

DATE 式。

戻り型

BIGINT

DATE 列の例

次の例では、2 つの日付リテラル値の間の差 (週単位) を取得します。

```
select date_diff(week,'2009-01-01','2009-12-31') as numweeks;

numweeks
----------
52
(1 row)
```

次の例は、2 つの日付リテラル値の差 (時間単位) を検出します。日付の時刻値を指定しなかった場合、デフォルトで 00:00:00 に設定されます。

```
select date_diff(hour, '2023-01-01', '2023-01-03 05:04:03');

date_diff
----------
53
(1 row)
```

次の例は、2 つの日付リテラル TIMESTAMETZ 値の差 (日単位) を検出します。

```
Select date_diff(days, 'Jun 1,2008  09:59:59 EST', 'Jul 4,2008  09:59:59 EST')

date_diff
----------
```

```
33
```

次の例は、テーブルの同じ行の 2 つの日付の差 (日単位) を検出します。

```
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)
```

次の例では、過去のリテラル値と今日の日付の間の差 (四半期単位) を取得します。この例では、現在の日付を 2008 年 6 月 5 日とします。完全名または略名で日付部分に名前を付けることができます。DATE_DIFF 関数のデフォルトの列名は DATE_DIFF です。

```
select date_diff(qtr, '1998-07-01', current_date);

date_diff
-----------
40
(1 row)
```

次の例では、SALES テーブルと LISTING テーブルを結合し、リスト 1000 から 1005 に対してチケットをリストしてから何日後に販売されたかを計算します。これらのリストの販売を最長の待機期間は 15 日であり、最小は 1 日より短いです (0 日)。

```
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)
```

この例は、販売者が任意およびすべてのチケット販売を待機する平均時間を計算します。

```
select avg(date_diff(hours, listtime, saletime)) as avgwait
from sales, listing
where sales.listid = listing.listid;

avgwait
---------
465
(1 row)
```

TIME 列の例

次のテーブルの TIME_TEST の例には、3 つの値が挿入された列 TIME_VAL (タイプ TIME) があります。

```
select time_val from time_test;

time_val
--------------------
20:00:00
00:00:00.5550
00:58:00
```

次の例では、TIME_VAL 列と時刻リテラル間の時間数の差を検出します。

```
select date_diff(hour, time_val, time '15:24:45') from time_test;

 date_diff
-----------
        -5
```

```
        15
        15
```

次の例では、2 つのリテラル時間値の分数の差を検出します。

```
select date_diff(minute, time '20:00:00', time '21:00:00') as nummins;

nummins
----------
60
```

TIMETZ 列の例

次のテーブルの TIMETZ_TEST の例には、3 つの値が挿入された列 TIMETZ_VAL (タイプ TIMETZ) があります。

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

次の例では、TIMETZ リテラルと timetz_val の間の時間数の差を検出します。

```
select date_diff(hours, timetz '20:00:00 PST', timetz_val) as numhours from
 timetz_test;

numhours
----------
0
-4
1
```

次の例では、2 つのリテラル TIMETZ 値間の時間数の差を検出します。

```
select date_diff(hours, timetz '20:00:00 PST', timetz '00:58:00 EST') as numhours;

numhours
----------
```

```
1
```

DATE_PART 関数

DATE_PART は式から日付部分の値を抽出します。DATE_PART は PGDATE_PART 関数のシノニムです。

構文

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

引数

field

ソースのどの部分を抽出する必要があり、サポートされている文字列値は同等の関数 EXTRACT のフィールドと同じです。

source

フィールドを抽出する DATE または INTERVAL 列。

戻り型

フィールドが「SECOND」の場合、DECIMAL(8, 6)。それ以外の場合は、INTEGER。

例

次の例では、日付値から日 (DOY) を抽出します。出力は、日付「2019-08-12」の日が であることを示しています224。つまり、2019 年 8 月 12 日は 2019 年 224 日です。

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

DATE_TRUNC 関数

DATE_TRUNC 関数は、指定した日付部分 (時、日、月など) に基づいてタイムスタンプの式またはリテラルを切り捨てます。

構文

```
date_trunc(format, datetime)
```

引数

format

切り捨てられる単位を表す形式。有効な形式は次のとおりです。

- 「YEAR」、「YYYY」、「YY」 - ts が属する年の最初の日付に切り捨て、タイムパートはゼロになります
- 「QUARTER」 - ts が属する四半期の最初の日付に切り捨て、タイムパートはゼロになります
- 「MONTH」、「MM」、「MON」 - ts が属する月の最初の日付に切り捨て、タイムパートはゼロになります
- 「WEEK」 - ts が属する週の月曜日に切り捨て、タイムパートはゼロになります
- 「DAY」、「DD」 - 時間帯をゼロにします
- 「時間」 - 分と秒を分数でゼロにします
- 「MINUTE」- 小数部分で秒をゼロにします
- 「SECOND」 - 2 番目の小数部分をゼロにします
- 「MILLISECOND」 - マイクロ秒をゼロにします
- 「MICROSECOND」 - すべてが残ります

ts

日時値

戻り型

形式モデルで指定された単位に切り捨てられたタイムスタンプ ts を返します。

例

次の例では、日付値を年初に切り捨てます。出力は、日付「2015-03-05」が 2015 年の初めである「2015-01-01」に切り捨てられたことを示しています。

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

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

DAY 関数

DAY 関数は、日付/タイムスタンプの日を返します。

日付抽出関数は、日付ベースの計算、データのフィルタリング、日付値の書式設定など、日付またはタイムスタンプの特定のコンポーネントを操作する必要がある場合に便利です。

構文

```
day(date)
```

引数

date

DATE 式または TIMESTAMP 式。

戻り値

DAY 関数は INTEGER を返します。

例

次の例では、入力日 から日付 (30) を抽出します'2009-07-30'。

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

次の例では、squirrelsテーブルの birthday列から日付を抽出し、結果を SELECT ステートメントの出力として返します。このクエリの出力は、squirrelsテーブル内の行ごとに 1 つずつ、各リスの誕生日を表す日の値のリストになります。

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

DAYOFMONTH 関数

DAYOFMONTH 関数は、日付/タイムスタンプ (月と年に応じて 1～31 の値) の日を返します。

DAYOFMONTH 関数は DAY 関数と似ていますが、名前と動作が少し異なります。DAY 関数はより一般的に使用されますが、代わりに DAYOFMONTH 関数を使用できます。このタイプのクエリは、

日付ベースの分析を実行したり、さらに処理またはレポートするために日付の特定のコンポーネントを抽出するなど、日付またはタイムスタンプデータを含むテーブルでフィルタリングしたりする必要がある場合に役立ちます。

構文

```
dayofmonth(date)
```

引数

date

DATE 式または TIMESTAMP 式。

戻り値

DAYOFMONTH 関数は INTEGER を返します。

例

次の例では、入力日 から日付 (30) を抽出します'2009-07-30'。

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

次の例では、DAYOFMONTH 関数をsquirrelsテーブルの birthday列に適用します。squirrels テーブルの各行について、birthday列の曜日が抽出され、SELECT ステートメントの出力として返されます。このクエリの出力は、squirrelsテーブル内の行ごとに 1 つずつ、各リスの誕生日を表す日の値のリストになります。

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

DAYOFWEEK 関数

DAYOFWEEK 関数は、日付またはタイムスタンプを入力として受け取り、曜日を数値 (日曜日の場合は 1、月曜日の場合は 2、土曜日の場合は 7) として返します。

この日付抽出関数は、日付ベースの計算、データのフィルタリング、日付値の書式設定など、日付またはタイムスタンプの特定のコンポーネントを操作する必要がある場合に便利です。

構文

```
dayofweek(date)
```

引数

date

DATE 式または TIMESTAMP 式。

戻り値

DAYOFWEEK 関数は INTEGER を返します。

1 = 日曜日

2 = 月曜日

3 = 火曜日

4 = 水曜日

5 = 木曜日

6 = 金曜日

7 = 土曜日

例

次の例では、この日付から曜日を抽出します。この日付は 5 (木曜日を表します) です。

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

次の例では、squirrelsテーブルの birthday列から曜日を抽出し、結果を SELECT ステートメントの出力として返します。このクエリの出力は、squirrelsテーブル内の行ごとに 1 つずつ、曜日の値のリストになり、各リスの生年月日の曜日を表します。

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

DAYOFYEAR 関数

DAYOFYEAR 関数は、日付またはタイムスタンプを入力として受け取り、その年の日 (年とうるう年かどうかに応じて 1～366 の値) を返す日付抽出関数です。

この関数は、日付ベースの計算、データのフィルタリング、日付値の書式設定など、日付またはタイムスタンプの特定のコンポーネントを操作する必要がある場合に便利です。

構文

```
dayofyear(date)
```

引数

date

DATE 式または TIMESTAMP 式。

戻り値

DAYOFYEAR 関数は INTEGER を返します (年とうるう年かどうかに応じて 1～366)。

例

次の例では、入力日 から日 (100) を抽出します'2016-04-09'。

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

次の例では、squirrelsテーブルの birthday列から曜日を抽出し、結果を SELECT ステートメントの出力として返します。

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

EXTRACT 関数

EXTRACT 関数は、TIMESTAMP、TIMESTAMPTZ、TIME、または TIMETZ 値から日付または時刻部分を返します。例としては、タイムスタンプの日、月、年、時、分、秒、ミリ秒、マイクロ秒などがあります。

構文

```
EXTRACT(datepart FROM source)
```

引数

datepart

日、月、年、時、分、秒、ミリ秒、マイクロ秒など、抽出する日付または時刻のサブフィールド。有効な値については、「日付関数またはタイムスタンプ関数の日付部分」を参照してください。

source

評価結果が TIMESTAMP、TIMESTAMPTZ、TIME、または TIMETZ のデータ型になる列または式。

戻り型

source 値が TIMESTAMP、TIME、または TIMETZ のデータ型として評価される場合は INTEGER。

source 値がデータ型 TIMESTAMPTZ として評価される場合は、DOUBLE PRECISION。

TIME の例

次のテーブルの TIME_TEST の例には、3 つの値が挿入された列 TIME_VAL (タイプ TIME) があります。

```
select time_val from time_test;

time_val
--------------------
20:00:00
00:00:00.5550
00:58:00
```

次の例は、各 time_val から分を抽出します。

```
select extract(minute from time_val) as minutes from time_test;

minutes
```

```
-----------
          0
          0
          58
```

次の例は、各 time_val から時間を抽出します。

```
select extract(hour from time_val) as hours from time_test;

hours
-----------
          20
          0
          0
```

FROM_UTC_TIMESTAMP 関数

FROM_UTC_TIMESTAMP 関数は、入力日を UTC (協定世界時) から指定されたタイムゾーンに変換します。

この関数は、日付と時刻の値を UTC から特定のタイムゾーンに変換する必要がある場合に便利です。これは、世界のさまざまな地域から発信され、適切な現地時間で提示する必要があるデータを使用する場合に重要です。

構文

```
from_utc_timestamp(timestamp, timezone
```

引数

timestamp

UTC タイムスタンプを持つ TIMESTAMP 式。

timezone

入力日付またはタイムスタンプを変換する有効なタイムゾーンである STRING 式。

戻り値

FROM_UTC_TIMESTAMP 関数は TIMESTAMP を返します。

例

次の の例では、入力日付を UTC から指定されたタイムゾーン ('Asia/Seoul') に変換します。この場合は UTC の 9 時間前です。結果の出力は、ソウルタイムゾーンの日時です2016-08-31 09:00:00。

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

HOUR 関数

HOUR 関数は、時間またはタイムスタンプを入力として受け取り、時間コンポーネント (0 ~ 23 の値) を返す時間抽出関数です。

この時間抽出関数は、時間ベースの計算、データのフィルタリング、時間値のフォーマットなど、時間またはタイムスタンプの特定のコンポーネントを操作する必要がある場合に便利です。

構文

```
hour(timestamp)
```

引数

timestamp

TIMESTAMP 式。

戻り値

HOUR 関数は INTEGER を返します。

例

次の の例では、入力タイムスタンプ から時間コンポーネント (12) を抽出します'2009-07-30 12:58:59'。

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

MINUTE 関数

MINUTE 関数は、時間またはタイムスタンプを入力として受け取り、分コンポーネント (0 ~ 60 の値) を返す時間抽出関数です。

構文

```
minute(timestamp)
```

引数

timestamp

TIMESTAMP 式または有効なタイムスタンプ形式の STRING。

戻り値

MINUTE 関数は INTEGER を返します。

例

次の の例では、入力タイムスタンプ から分コンポーネント (58) を抽出します'2009-07-30 12:58:59'。

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

MONTH 関数

MONTH 関数は、時間またはタイムスタンプを入力として受け取り、月コンポーネント (0 ~ 12 の値) を返す時間抽出関数です。

構文

```
month(date)
```

引数

date

TIMESTAMP 式または有効なタイムスタンプ形式の STRING。

戻り値

MONTH 関数は INTEGER を返します。

例

次の例では、入力タイムスタンプ から月コンポーネント (7) を抽出します'2016-07-30'。

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

SECOND 関数

SECOND 関数は、時間またはタイムスタンプを入力として受け取り、2 番目のコンポーネント (0 ~ 60 の値) を返す時間抽出関数です。

構文

```
second(timestamp)
```

引数

timestamp

TIMESTAMP 式。

戻り値

SECOND 関数は INTEGER を返します。

例

次の例では、入力タイムスタンプ から 2 番目のコンポーネント (59) を抽出します'2009-07-30 12:58:59'。

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

TIMESTAMP 関数

TIMESTAMP 関数は値 (通常は数値) を受け取り、タイムスタンプデータ型に変換します。

この関数は、時刻または日付を表す数値をタイムスタンプデータ型に変換する必要がある場合に便利です。これは、Unix タイムスタンプやエポック時間などの数値形式で保存されているデータを使用する場合に役立ちます。

構文

```
timestamp(expr)
```

引数

expr

TIMESTAMP にキャストできる任意の式。

戻り値

TIMESTAMP 関数は TIMESTAMP を返します。

例

次の例では、数値 Unix タイムスタンプ (1632416400) を対応するタイムスタンプデータ型に変換します。2021 年 9 月 22 日午後 12:00:00 UTC。

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

TO_TIMESTAMP 関数

TO_TIMESTAMP は TIMESTAMP 文字列を TIMESTAMPTZ に返します。

構文

```
to_timestamp (timestamp)
```

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

引数

timestamp

タイムスタンプ文字列、またはタイムスタンプ文字列にキャストできるデータ型。

format

Spark の日時パターンに一致する文字列リテラル。有効な日時パターンについては、「フォーマットと解析の日時パターン」を参照してください。

戻り型

TIMESTAMP

例

次の例は、TO_TIMESTAMP 関数を使用して TIMESTAMP 文字列を TIMESTAMP に変換する方法を示しています。

```
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
```

日付の TO_TIMESTAMP 部分を渡すこともできます。残りの日付部分はデフォルト値に設定されます。時刻は出力に含まれません。

```
SELECT TO_TIMESTAMP('2017','YYYY');

to_timestamp
---------------------------
2017-01-01 00:00:00+00
```

次の SQL ステートメントは、文字列「2011-12-18 24:38:15」を TIMESTAMP に変換します。結果は、時間数が 24 時間を超えているため、翌日になる TIMESTAMP になります。

```
select to_timestamp('2011-12-18 24:38:15', 'YYYY-MM-DD HH24:MI:SS');

to_timestamp
----------------------
2011-12-19 00:38:15+00
```

YEAR 関数

YEAR 関数は、日付またはタイムスタンプを入力として受け取り、年コンポーネント (4 桁の数字) を返す日付抽出関数です。

構文

```
year(date)
```

引数

date

DATE 式または TIMESTAMP 式。

戻り値

YEAR 関数は INTEGER を返します。

例

次の の例では、入力日 から年コンポーネント (2016) を抽出します'2016-07-30'。

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

次の例では、squirrelsテーブルの birthday列から年コンポーネントを抽出し、結果を SELECT ステートメントの出力として返します。このクエリの出力は、squirrelsテーブル内の行ごとに 1 つずつ、各リスの誕生日を表す年値のリストになります。

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

日付関数またはタイムスタンプ関数の日付部分

次のテーブルは、次の関数に対する引数として受け取る、日付部分および時刻部分の名前と略名を指定します。

- DATE_ADD

- DATE_DIFF
- DATE_PART
- EXTRACT

日付部分または時刻部分	省略形
millennium、millennia	mil、mils
century、centuries	c、cent、cents
decade、decades	dec、decs
epoch	epoch (EXTRACT がサポート)
year、years	y、yr、yrs
quarter、quarters	qtr、qtrs
month、months	mon、mons
week、weeks	w
day of week	dayofweek、dow、dw、weekday (DATE_PART と EXTRACT 関数 がサポート) 0～6 の整数 (0 は日曜日) を返します。 **Note** 日付部分 DOW の動作は、日時形式の文字列に使用される日付部分 day of week (D) とは異なります。D は、整数 1～7 (日曜日が 1) に基づきます。詳細については、「日時形式の文字列」を参照してください。
day of year	dayofyear、doy、dy、yearday (EXTRACT がサポート)
day、days	d

日付部分または時刻部分	省略形
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	タイムゾーン付きタイムスタンプ (TIMESTAMPTZ) の EXTRACT でのみサポートされます。

結果のバリエーション (秒、ミリ秒、マイクロ秒)

異なる日付関数が秒、ミリ秒、またはマイクロ秒を日付部分として指定する場合、クエリ結果にわずかな違いが生じます。

- EXTRACT 関数は、上位および下位の日付部分は無視し、指定された日付部分のみの整数を返します。指定された日付部分が秒の場合、ミリ秒およびマイクロ秒は結果に含まれません。指定された日付部分がミリ秒の場合、秒およびマイクロ秒は結果に含まれません。指定された日付部分がマイクロ秒の場合、秒およびミリ秒は結果に含まれません。
- DATE_PART 関数は、指定された日付部分にかかわらず、タイムスタンプの完全な秒部分を返します。必要に応じて小数値または整数を返します。

CENTURY、EPOCH、DECADE、および MIL ノート

CENTURY または CENTURIES

AWS Clean Rooms は CENTURY を「###1」で始まり、「」で終わるように解釈します###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 の AWS Clean Rooms 実装は、クラスターが存在するタイムゾーンとは無関係に 1970-01-01 00:00:00.000000 に関連しています。クラスターが設置されているタイムゾーンによって、時差による結果を補正する必要がある場合があります。

DECADE または DECADES

AWS Clean Rooms は、共通カレンダーに基づいて DECADE または DECADES DATEPART を解釈します。例えば、共通カレンダーが年 1 から始まるため、最初の 10 年 (decade 1) は 0001-01-01 から 0009-12-31 であり、2 番目の 10 年 (decade 2) は 0010-01-01 から 0019-12-31 です。例えば、decade 201 は 2000-01-01 から 2009-12-31 の期間に及びます。

```
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 または MILS

AWS Clean Rooms は MIL を解釈して、#001 年の最初の日に始まり、最後の日に終わります#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)
```

暗号化および復号関数

暗号化および復号関数は、SQL デベロッパーが読み取り可能なプレーンテキスト形式と読み取り不可能な暗号化テキスト形式の間で機密データを変換することで、不正アクセスや誤用から保護するのに役立ちます。

AWS Clean Rooms Spark SQL は、次の暗号化および復号関数をサポートしています。

トピック

- AES_ENCRYPT 関数
- AES_DECRYPT 関数

AES_ENCRYPT 関数

AES_ENCRYPT 関数は、Advanced Encryption Standard (AES) アルゴリズムを使用してデータを暗号化するために使用されます。

構文

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

引数

expr

暗号化するバイナリ値。

key

データの暗号化に使用するパスフレーズ。

16、24、32 ビットのキー長がサポートされています。

モード

メッセージの暗号化に使用するブロック暗号モードを指定します。

有効なモード: ECB (電子 CodeBook)、GCM (Galois/Counter Mode)、CBC (Cipher-Block Chaining)。

パディング

長さがブロックサイズの倍数ではないメッセージをパディングする方法を指定します。

有効な値: PKCS、NONE、DEFAULT。

DEFAULT パディングは、ECB の場合は PKCS (パブリックキー暗号化標準) 、GCM の場合は NONE、CBC の場合は PKCS を意味します。

(モード、パディング) のサポートされている組み合わせは、 ('ECB'、'PKCS')、('GCM'、'NONE')、および ('CBC'、'PKCS') です。

iv

オプションの初期化ベクトル (IV)。CBC モードと GCM モードでのみサポートされます。

有効な値: GCM の場合は 12 バイト、CBC の場合は 16 バイト。

aad

オプションの追加の認証データ (AAD)。GCM モードでのみサポートされます。これは任意の自由形式の入力にすることができ、暗号化と復号の両方に指定する必要があります。

戻り型

AES_ENCRYPT 関数は、指定されたパディングで指定されたモードで AES を使用して暗号化された expr の値を返します。

例

次の例は、Spark SQL AES_ENCRYPT 関数を使用して、指定された暗号化キーを使用してデータの文字列 (この場合は「Spark」という単語) を安全に暗号化する方法を示しています。結果として得られる暗号文は Base64-encodedされ、保存または送信が容易になります。

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

次の例は、Spark SQL AES_ENCRYPT 関数を使用して、指定された暗号化キーを使用してデータの文字列 (この場合は「Spark」という単語) を安全に暗号化する方法を示しています。結果として得られる暗号文は 16 進形式で表されます。これは、データストレージ、送信、デバッグなどのタスクに役立ちます。

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

次の例は、Spark SQL AES_ENCRYPT 関数を使用して、指定された暗号化キー、暗号化モード、パディングモードを使用して、データの文字列 (この場合は「Spark SQL」) を安全に暗号化する方法を示しています。結果として得られる暗号文は Base64-encodedされ、保存または送信が容易になります。

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

AES_DECRYPT 関数

AES_DECRYPT 関数は、Advanced Encryption Standard (AES) アルゴリズムを使用してデータを復号するために使用されます。

構文

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

引数

expr

復号するバイナリ値。

key

データの復号に使用するパスフレーズ。

パスフレーズは、暗号化された値の生成に最初に使用されたキーと一致し、16、24、または 32 バイトの長さである必要があります。

モード

メッセージの復号に使用するブロック暗号モードを指定します。

有効なモード: ECB、GCM、CBC。

パディング

長さがブロックサイズの倍数ではないメッセージをパディングする方法を指定します。

有効な値: PKCS、NONE、DEFAULT。

DEFAULT パディングは、ECB の場合は PKCS、GCM の場合は NONE、CBC の場合は PKCS を意味します。

aad

オプションの追加の認証データ (AAD)。GCM モードでのみサポートされます。これは任意の自由形式の入力にすることができ、暗号化と復号の両方に指定する必要があります。

戻り型

パディング付きのモードで AES を使用して、復号された expr の値を返します。

例

次の例は、Spark SQL AES_ENCRYPT 関数を使用して、指定された暗号化キーを使用してデータの文字列 (この場合は「Spark」という単語) を安全に暗号化する方法を示しています。結果として得られる暗号文は Base64-encodedされ、保存または送信が容易になります。

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

次の例は、Spark SQL AES_DECRYPT 関数を使用して、以前に暗号化され Base64-encodedされたデータを復号する方法を示しています。復号プロセスでは、元のプレーンテキストデータを正常に復元するために、正しい暗号化キーとパラメータ (暗号化モードとパディングモード) が必要です。

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

ハッシュ関数

ハッシュ関数は、数値入力値を別の値に変換する数学関数です。

AWS Clean Rooms Spark SQL は、次のハッシュ関数をサポートしています。

トピック

MD5 関数

MD5 暗号化ハッシュ関数を使用して、可変長文字列を 32 文字の文字列に変換します。この 32 文字の文字列は、128 ビットチェックサムの 16 進値をテキストで表記したものです。

構文

```
MD5(string)
```

引数

string

可変長文字列。

戻り型

MD5 関数は、32 文字の文字列を返します。この 32 文字の文字列は、128 ビットチェックサムの 16 進値をテキストで表記したものです。

例

以下の例では、文字列 'AWS Clean Rooms' の 128 ビット値を示します。

```
select md5('AWS Clean Rooms');
```

```
md5
----------------------------------
f7415e33f972c03abd4f3fed36748f7a
(1 row)
```

SHA 関数

SHA1 関数のシノニム。

「SHA1 関数」を参照してください。

SHA1 関数

SHA1 関数は、SHA1 暗号化ハッシュ関数を使用して、可変長文字列を 40 文字の文字列に変換します。この 40 文字の文字列は、160 ビットのチェックサムの 16 進値をテキストで表記したものです。

構文

SHA1 は SHA 関数 のシノニムです。

```
SHA1(string)
```

引数

string

可変長文字列。

戻り型

SHA1 関数は、40 文字の文字列を返します。この 40 文字の文字列は、160 ビットのチェックサムの 16 進値をテキストで表記したものです。

例

以下の例は、単語 'AWS Clean Rooms' の 160 ビット値を返します。

```
select sha1('AWS Clean Rooms');
```

SHA2 関数

SHA2 関数は、SHA2 暗号化ハッシュ関数を使用して、可変長文字列を文字列に変換します。この文字列は、指定されたビット数のチェックサムの 16 進値をテキストで表記したものです。

構文

```
SHA2(string, bits)
```

引数

string

可変長文字列。

integer

ハッシュ関数のビット数。有効な値は 0 (256 と同じ)、224、256、384、および 512 です。

戻り型

SHA2 関数は、チェックサムの 16 進値をテキストで表記した文字列を返します。また、ビット数が無効な場合は、空の文字列を返します。

例

次の例では、「AWS Clean Rooms」という語の 256 ビット値が返されます。

```
select sha2('AWS Clean Rooms', 256);
```

xxHASH64 関数

xxhash64 関数は、引数の 64 ビットハッシュ値を返します。

xxhash64() 関数は、高速かつ効率的に設計された非暗号化ハッシュ関数です。多くの場合、データ処理およびストレージアプリケーションでは、データの一意の識別子が必要ですが、データの正確な内容を秘密にする必要はありません。

SQL クエリのコンテキストでは、xxhash64() 関数を次のようなさまざまな目的で使用できます。

- テーブル内の行の一意の識別子の生成

- ハッシュ値に基づくデータのパーティション化
- カスタムインデックス作成またはデータ分散戦略の実装

特定のユースケースは、アプリケーションの要件と処理されるデータによって異なります。

構文

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

引数

expr1

任意のタイプの式。

expr2

任意のタイプの式。

戻り値

引数の 64 ビットハッシュ値 (BIGINT) を返します。ハッシュシードは 42 です。

例

次の の例では、指定された入力に基づいて 64 ビットのハッシュ値 (5602566077635097486) を生成します。最初の引数は文字列値で、この場合は「Spark」という単語です。2 番目の引数は、単一の整数値 123 を含む配列です。3 番目の引数は、ハッシュ関数のシードを表す整数値です。

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

Hyperloglog 関数

SQL の HyperLogLog (HLL) 関数は、実際の一意の要素のセットが保存されていない場合でも、大規模なデータセット内の一意の要素 (カーディナリティ) の数を効率的に推定する方法を提供します。

HLL 関数を使用する主な利点は次のとおりです。

- メモリ効率: HLL スケッチは、一意の要素の完全なセットを保存するよりもはるかに少ないメモリを必要とするため、大規模なデータセットに適しています。

- 分散コンピューティング: HLL スケッチは複数のデータソースまたは処理ノードにまたがって組み合わせることができるため、効率的な分散一意数推定が可能になります。
- おおよその結果: HLL は、精度とメモリ使用量の間の調整可能なトレードオフ (精度パラメータを使用) により、おおよその一意のカウントの推定を提供します。

これらの関数は、分析、データウェアハウス、リアルタイムストリーム処理アプリケーションなど、一意の項目の数を見積もる必要があるシナリオで特に役立ちます。

AWS Clean Rooms は、次の HLL 関数をサポートしています。

トピック

HLL_SKETCH_AGG 関数

HLL_SKETCH_AGG 集計関数は、指定された列の値から HLL スケッチを作成します。入力式の値をカプセル化する HLLSKETCH データ型を返します。

HLL_SKETCH_AGG 集計関数は任意のデータ型で動作し、NULL 値を無視します。

テーブルに行がない場合、またはすべての行が NULL の場合、結果のスケッチには `{"version":1,"logm":15,"sparse":{"indices":[],"values":[]}}` などのインデックスと値のペアがありません。

構文

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

引数

aggregate_expression

一意のカウントが発生する INT、BIGINT、STRING、または BINARY 型の式。NULL 値はすべて無視されます。

lgConfigK

デフォルトの 12 を含む、4 から 21 までのオプションの INT 定数。K の log-base-2。ここで、K はスケッチのバケットまたはスロットの数です。

戻り型

HLL_SKETCH_AGG 関数は、集計グループ内のすべての入力値を消費および集計したために計算された HyperLogLog スケッチを含む NULL 以外の BINARY バッファを返します。

例

次の例では、HyperLogLog (HLL) アルゴリズムを使用して、 col 列の値の個別の数を推定します。hll_sketch_agg(col, 12) 関数は col 列の値を集計し、精度 12 を使用して HLL スケッチを作成します。次に、 hll_sketch_estimate()関数を使用して、生成された HLL スケッチに基づいて個別の値のカウントを推定します。クエリの最終結果は 3 で、col列の値の推定個別数を表します。この場合、個別の値は 1、2、3 です。

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

次の例では、HLL アルゴリズムを使用してcol列の値の個別の数を推定しますが、HLL スケッチの精度値は指定しません。この場合、デフォルトの精度である 14 が使用されます。hll_sketch_agg(col) 関数は col列の値を取得し、HyperLogLog (HLL) スケッチを作成します。これは、要素の個別の数を推定するために使用できるコンパクトなデータ構造です。hll_sketch_estimate(hll_sketch_agg(col)) 関数は、前のステップで作成した HLL スケッチを取得し、 col 列の値の個別の数の推定値を計算します。クエリの最終結果は 3 で、col列の値の推定個別数を表します。この場合、個別の値は 1、2、3 です。

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

HLL_SKETCH_ESTIMATE 関数

HLL_SKETCH_ESTIMATE 関数は HLL スケッチを取得し、スケッチで表される一意の要素の数を推定します。HyperLogLog (HLL) アルゴリズムを使用して、特定の列の一意の値の数の確率的近似を

カウントし、HLL_SKETCH_AGG 関数によって以前に生成されたスケッチバッファと呼ばれるバイナリ表現を消費し、結果を大きな整数として返します。

HLL スケッチアルゴリズムは、大きなデータセットであっても、一意の値のセットをすべて保存することなく、一意の要素の数を効率的に推定する方法を提供します。

hll_union 関数と hll_union_agg関数は、これらのバッファを入力として消費およびマージすることで、スケッチを組み合わせることもできます。

構文

```
HLL_SKETCH_ESTIMATE (hllsketch_expression)
```

引数

hllsketch_expression

HLL_SKETCH_AGG によって生成されたスケッチを保持するBINARY式

戻り型

HLL_SKETCH_ESTIMATE 関数は、入力スケッチで表されるおおよその個別カウントである BIGINT 値を返します。

例

次の例では、HyperLogLog (HLL) スケッチアルゴリズムを使用して、col列の値の基数 (一意の数) を推定します。hll_sketch_agg(col, 12) 関数は col列を取得し、12 ビットの精度を使用して HLL スケッチを作成します。HLL スケッチは、セット内の一意の要素の数を効率的に推定できるおおよそのデータ構造です。hll_sketch_estimate() 関数は、 によって作成された HLL スケッチhll_sketch_aggを取得し、スケッチによって表される値の基数 (一意の数) を推定します。は 5 行のテストデータセットFROM VALUES (1), (1), (2), (2), (3) tab(col);を生成します。このcol列には 1、1、2、2、3 の値が含まれます。このクエリの結果は、col列の値の推定一意数 3 です。

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

次の例と前の例の違いは、精度パラメータ (12 ビット) がhll_sketch_agg関数呼び出しで指定されていないことです。この場合、デフォルトの精度である 14 ビットが使用されます。これによ

り、12 ビットの精度を使用した前の例と比較して、一意のカウントのより正確な推定が得られます。

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

HLL_UNION 関数

HLL_UNION 関数は、2 つの HLL スケッチを 1 つの統合スケッチに結合します。HyperLogLog (HLL) アルゴリズムを使用して、2 つのスケッチを 1 つのスケッチに結合します。クエリは、結果のバッファを使用して、hll_sketch_estimate関数でおおよその一意の数を長い整数として計算できます。

構文

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

引数

exprN

HLL_SKETCH_AGG によって生成されたスケッチを保持するBINARY式。

allowDifferentLgConfigK

異なる lgConfigK 値を持つ 2 つのスケッチのマージを許可するかどうかを制御するオプションの BOOLEAN 式。デフォルト値は false です。

戻り型

HLL_UNION 関数は、入力式を組み合わせた結果として計算された HyperLogLog スケッチを含む BINARY バッファを返します。allowDifferentLgConfigK パラメータが の場合true、結果スケッチは 2 つの指定されたlgConfigK値のうち小さい方を使用します。

例

次の例では、HyperLogLog (HLL) スケッチアルゴリズムを使用して、データセットcol2内の 2 つの列 col1と の値の一意の数を推定します。

hll_sketch_agg(col1) 関数は、col1列内の一意の値の HLL スケッチを作成します。

hll_sketch_agg(col2) 関数は、col2 列の一意の値の HLL スケッチを作成します。

このhll_union(...)関数は、ステップ 1 と 2 で作成した 2 つの HLL スケッチを 1 つの統合 HLL スケッチに結合します。

hll_sketch_estimate(...) 関数は、結合された HLL スケッチを取得し、 col1 と の両方の値の一意の数を推定しますcol2。

FROM VALUES 句は 5 行のテストデータセットを生成します。 col1には値 1、1、2、2、3 が含まれ、 には値 4、4、5、5、6 col2が含まれます。

このクエリの結果は、 col1と の両方の値の推定一意数でありcol2、6 です。HLL スケッチアルゴリズムは、大きなデータセットであっても、一意の値のセットをすべて保存することなく、一意の要素の数を効率的に推定する方法を提供します。この例では、 hll_union関数を使用して 2 つの列の HLL スケッチを組み合わせます。これにより、列ごとにだけでなく、データセット全体で一意の数を推定できます。

```
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
```

次の例と前の例の違いは、精度パラメータ (12 ビット) がhll_sketch_agg関数呼び出しで指定されていないことです。この場合、デフォルトの精度である 14 ビットが使用されます。これにより、12 ビットの精度を使用した前の例と比較して、一意のカウントのより正確な推定が得られます。

```
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 関数

HLL_UNION_AGG 関数は、複数の HLL スケッチを 1 つの統合スケッチに結合します。HyperLogLog (HLL) アルゴリズムを使用して、スケッチのグループを 1 つのスケッチに結合します。クエリは、結果のバッファを使用して、hll_sketch_estimate関数でおおよその一意数を計算できます。

構文

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

引数

expr

HLL_SKETCH_AGG によって生成されたスケッチを保持するBINARY式。

allowDifferentLgConfigK

異なる lgConfigK 値を持つ 2 つのスケッチのマージを許可するかどうかを制御するオプションの BOOLEAN 式。デフォルト値は false です。

戻り型

HLL_UNION_AGG 関数は、同じグループの入力式を結合した結果、計算された HyperLogLog スケッチを含む BINARY バッファを返します。allowDifferentLgConfigK パラメータが の場合true、結果スケッチは 2 つの指定されたlgConfigK値のうち小さい方を使用します。

例

次の例では、HyperLogLog (HLL) スケッチアルゴリズムを使用して、複数の HLL スケッチにわたる値の一意の数を推定します。

最初の例では、データセット内の値の一意の数を推定します。

```
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
```

内部クエリは 2 つの HLL スケッチを作成します。

- 最初の SELECT ステートメントは、1 つの値からスケッチを作成します。
- 2 番目の SELECT ステートメントは、別の 1 つの値 1 からスケッチを作成しますが、精度は 20 です。

外部クエリは HLL_UNION_AGG 関数を使用して、2 つのスケッチを 1 つのスケッチに結合します。次に、HLL_SKETCH_ESTIMATE 関数をこの組み合わせスケッチに適用して、値の一意の数を推定します。

このクエリの結果は、 col列の値の推定一意数です1。つまり、2 つの入力値 1 は、同じ値であっても一意であると見なされます。

2 番目の例には、HLL_UNION_AGG 関数の別の精度パラメータが含まれています。この場合、両方の HLL スケッチは 14 ビットの精度で作成されるため、 trueパラメータhll_union_aggで を使用して正常に組み合わせることができます。

```
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
```

クエリの最終結果は推定一意数であり、この場合も です1。つまり、2 つの入力値 1 は、同じ値であっても一意であると見なされます。

JSON 関数

相対的に小さい一連のキーと値のペアを格納する必要がある場合は、データを JSON 形式で格納すると、スペースを節約できます。JSON 文字列は単一の列に格納できるため、データを表形式で格納するより、JSON を使用する方が効率的である可能性があります。

Example

例えば、スパース表があるとします。その表では、すべての属性を完全に表すために多数の列が必要ですが、指定された行または指定された列のほとんどの列値は NULL になります。格納に JSON を使用することによって、行のデータを単一の JSON 文字列にキーと値のペアで格納できる可能性があり、格納データの少ない表の列を除去できる可能性があります。

その上、JSON 文字列を簡単に変更することによって、列を表に追加することなく、キーと値のペアをさらに格納することもできます。

JSON の使用は控えめにすることをお勧めします。JSON では、単一の列にさまざまなデータが保存され、 AWS Clean Rooms の列保存アーキテクチャが使用されません。したがってこの形式は、大きいデータセットの保存には適していません。

JSON は UTF-8 でエンコードされたテキスト文字列を使用するため、JSON 文字列を CHAR データ型または VARCHAR データ型として格納できます。文字列にマルチバイト文字が含まれている場合は、VARCHAR を使用します。

JSON 文字列は、以下のルールに従って、正しくフォーマットされた JSON である必要があります。

- ルートレベルの JSON には、JSON オブジェクトまたは JSON 配列を使用できます。JSON オブジェクトは、順序が設定されていない一連のキーと値のペアがカンマで区切られ、中括弧で囲まれたものです。

  例: `{"one":1, "two":2}`
- JSON 配列は、順序付けられた一連のカンマ区切り値が角括弧で囲まれたものです。

  例は次のとおりです: `["first", {"one":1}, "second", 3, null]`
- JSON 配列は、0 から始まるインデックスを使用します。配列内の最初の要素の位置は 0 です。JSON のキーと値のペアでは、キーは二重引用符で囲まれた文字列です。
- JSON 値には次のいずれかを指定できます。
  - JSON オブジェクト
  - JSON 配列
  - 二重引用符で囲まれた文字列
  - 数値 (整数および浮動小数点数)
  - ブール値

  - Null
- 空のオブジェクトおよび空の配列は、有効な JSON 値です。
- JSON フィールドでは、大文字と小文字が区別されます。
- JSON 構造要素 ({ }, [ ] など) は無視されます。

トピック

- GET_JSON_OBJECT 関数
- TO_JSON 関数

GET_JSON_OBJECT 関数

GET_JSON_OBJECT 関数は、 から JSON オブジェクトを抽出しますpath。

構文

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

引数

json_txt

適切に形成された JSON を含む STRING 式。

パス

適切に形成された JSON パス式を持つ STRING リテラル。

戻り値

STRING を返します。

オブジェクトが見つからない場合、NULL が返されます。

例

次の例では、JSON オブジェクトから値を抽出します。最初の引数は、単一のキーと値のペアを持つ単純なオブジェクトを表す JSON 文字列です。2 番目の引数は JSON パス式です。$ 記号は JSON オブジェクトのルートを表し、 .a部分はa「」キーに関連付けられた値を抽出することを指定しま

す。関数の出力はb「」です。これは、入力 JSON オブジェクトのa「」キーに関連付けられた値です。

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

TO_JSON 関数

TO_JSON 関数は、入力式を JSON 文字列表現に変換します。この関数は、さまざまなデータ型 (数値、文字列、ブール値など) の対応する JSON 表現への変換を処理します。

TO_JSON 関数は、構造化データ (データベース行や JSON オブジェクトなど) を JSON のようなよりポータブルで自己記述型の形式に変換する必要がある場合に役立ちます。これは、JSON 形式のデータを期待する他のシステムやサービスとやり取りする必要がある場合に特に役立ちます。

構文

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

引数

expr

JSON 文字列に変換する入力式。値、列、またはその他の有効な SQL 式を指定できます。

options:

JSON 変換プロセスをカスタマイズするために使用できるオプションの一連の設定オプション。これらのオプションには、null 値の処理、数値の表現、特殊文字の処理などが含まれます。

戻り値

指定された構造体値を持つ JSON 文字列を返します。

例

次の例では、名前付き構造体 (構造化データの一種) を JSON 文字列に変換します。最初の引数 (named_struct('a', 1, 'b', 2)) は、to_json()関数に渡される入力式です。値 1 の「a」と値 2 の「b」の 2 つのフィールドを持つ名前付き構造体を作成します。to_json() 関数は、名前付き struct を引数として受け取り、JSON 文字列表現に変換します。出力は です。これは{"a":1,"b":2}、名前付き構造体を表す有効な JSON 文字列です。

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

次の例では、タイムスタンプ値を含む名前付き構造体を、カスタマイズされたタイムスタンプ形式の JSON 文字列に変換します。最初の引数 (named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd'))) は、タイムスタンプ値を含む単一のフィールド「time」を持つ名前付き構造体を作成します。2 番目の引数 (map('timestampFormat', 'dd/MM/yyyy')) は、単一のキーと値のペアを持つマップ (キーと値のディクショナリ) を作成します。キーはtimestampFormat」で、値はdd/MM/yyyy」です。このマップは、JSON に変換するときにタイムスタンプ値に必要な形式を指定するために使用されます。to_json() 関数は、名前付き構造体を JSON 文字列に変換します。2 番目の引数であるマップは、タイムスタンプ形式をdd/MM/yyyy」にカスタマイズするために使用されます。出力は です。これは{"time":"26/08/2015"}、目的のdd/MM/yyyy」形式のタイムスタンプ値を含む単一のフィールド「time」を持つ JSON 文字列です。

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

数学関数

このセクションでは、Spark SQL AWS Clean Rooms でサポートされている数学演算子と関数について説明します。

トピック

- 数学演算子の記号
- ABS 関数
- ACOS 関数
- ASIN 関数
- ATAN 関数
- ATAN2 関数
- CBRT 関数
- CEILING (または CEIL) 関数
- COS 関数
- COT 関数

- DEGREES 関数
- DIV 関数
- EXP 関数
- FLOOR 関数
- LN 関数
- LOG 関数
- MOD 関数
- PI 関数
- POWER 関数
- RADIANS 関数
- RAND 関数
- RANDOM 関数
- ROUND 関数
- SIGN 関数
- SIN 関数
- SQRT 関数
- TRUNC 関数

数学演算子の記号

次の表に、サポートされる数学演算子の一覧を示します。

サポートされている演算子

演算子	説明	例	結果
+	加算	2 + 3	5
-	減算	2 - 3	-1
*	乗算	2 * 3	6
/	除算	4 / 2	2

演算子	説明	例	結果
%	モジュロ	5 % 4	1
^	べき算	2.0 ^ 3.0	8

例

特定の取引において支払われたコミッションに手数料 2.00 USD を加算します。

```
select commission, (commission + 2.00) as comm
from sales where salesid=10000;

commission | comm
-----------+-------
28.05      | 30.05
(1 row)
```

特定の取引において販売価格の 20% を計算します。

```
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;

pricepaid | twentypct
----------+-----------
187.00    |    37.400
(1 row)
```

継続的な成長パターンに基づいてチケット販売数を予測します。次の例では、サブクエリによって、2008 年に販売されたチケット数が返されます。その結果に、10 年にわたって継続する成長率 5% が指数関数的に乗算されます。

```
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)
```

日付 ID が 2000 以上である販売の合計支払額および合計コミッションを求め、その後、合計支払額から合計コミッションを減算します。

```
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 関数

ABS は、数値の絶対値を計算します。この数値は、リテラルでも、数値に評価される式でも構いません。

構文

```
ABS (number)
```

引数

number

数値、または数値に評価される式。SMALLINT、INTEGER、BIGINT、DECIMAL、FLOAT4、または FLOAT8 型にすることができます。

戻り型

ABS は、その引数と同じデータ型を返します。

例

-38 の絶対数を計算する

```
select abs (-38);
abs
-------
38
(1 row)
```

(14-76) の絶対数を計算する

```
select abs (14-76);
abs
-------
62
(1 row)
```

ACOS 関数

ACOS は、数値のアークコサイン (逆余弦) を返す三角関数です。戻り値はラジアンで、0 ~ PI の範囲内です。

構文

```
ACOS(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

-1 のアークコサイン (逆余弦) を返すには、次の例を使用します。

```
SELECT ACOS(-1);
```

```
+-------------------+
|       acos        |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ASIN 関数

ASIN は、数値のアークサイン (逆正弦) を返す三角関数です。戻り値はラジアンで、PI/2 ~ -PI/2 の範囲内です。

構文

```
ASIN(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

1 のアークサイン (正弦) を返すには、次の例を使用します。

```
SELECT ASIN(1) AS halfpi;

+--------------------+
|       halfpi       |
+--------------------+
| 1.5707963267948966 |
+--------------------+
```

ATAN 関数

ATAN は、数値のアークタンジェント (逆正接) を返す三角関数です。戻り値はラジアンで、-PI ~ PI の範囲内です。

構文

```
ATAN(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

1 のアークタンジェント (逆正接) を返し、その値に 4 を乗算するには、次の例を使用します。

```
SELECT ATAN(1) * 4 AS pi;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ATAN2 関数

ATAN2 は、一方の数値をもう一方の数値で除算した値のアークタンジェント (逆正接) を返す三角関数です。戻り値はラジアンで、PI/2 ~ -PI/2 の範囲内です。

構文

```
ATAN2(number1, number2)
```

引数

number1

DOUBLE PRECISION 数。

number2

DOUBLE PRECISION 数。

戻り型

DOUBLE PRECISION

例

2/2 のアークタンジェント (逆正接) を返し、その値に 4 を乗算するには、次の例を使用します。

```
SELECT ATAN2(2,2) * 4 AS PI;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

CBRT 関数

CBRT 関数は、数値の立方根を計算する数学関数です。

構文

```
CBRT (number)
```

引数

CBRT は、引数として DOUBLE PRECISION 型の数値を取ります。

戻り型

CBRT は DOUBLE PRECISION 型の数値を返します。

例

特定の取引において支払われたコミッションの立方根を計算します。

```
select cbrt(commission) from sales where salesid=10000;
```

```
cbrt
------------------
3.03839539048843
(1 row)
```

CEILING (または CEIL) 関数

CEILING 関数または CEIL 関数は、数値を最も近い整数に切り上げるために使用します。(FLOOR 関数は、数値を最も近い整数に切り下げます。)

構文

```
CEIL | CEILING(number)
```

引数

number

数値、または数値に評価される式。SMALLINT、INTEGER、BIGINT、DECIMAL、FLOAT4、または FLOAT8 型にすることができます。

戻り型

CEILING および CEIL は、引数と同じデータ型を返します。

例

特定の取引において支払われたコミッションの切り上げ値を計算します。

```
select ceiling(commission) from sales
where salesid=10000;

ceiling
---------
29
(1 row)
```

COS 関数

COS は、数値のコサイン (余弦) を返す三角関数です。戻り値はラジアンで、-1 以上 1 以下です。

構文

```
COS(double_precision)
```

引数

number

入力パラメータは倍精度の数値です。

戻り型

COS 関数は、倍精度の数値を返します。

例

次の例は、0 のコサイン (余弦) を返します。

```
select cos(0);
cos
-----
1
(1 row)
```

次の例は、PI のコサイン (余弦) を返します。

```
select cos(pi());
cos
-----
-1
(1 row)
```

COT 関数

COT は、数値のコタンジェント (余接) を返す三角関数です。入力パラメータは 0 以外である必要があります。

構文

```
COT(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

1 のコタンジェント (余接) を返すには、次の例を使用します。

```
SELECT COT(1);

+--------------------+
|        cot         |
+--------------------+
| 0.6420926159343306 |
+--------------------+
```

DEGREES 関数

ラジアンで指定された角度を、その値に相当する度数に変換します。

構文

```
DEGREES(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

0.5 ラジアンに相当する度数を返すには、次の例を使用します。

```
SELECT DEGREES(.5);

+-------------------+
|      degrees      |
+-------------------+
| 28.64788975654116 |
+-------------------+
```

PI ラジアンを度数に変換するには、次の例を使用します。

```
SELECT DEGREES(pi());

+---------+
| degrees |
+---------+
|     180 |
+---------+
```

DIV 関数

DIV 演算子は、除算除算の積分部分を返します。

構文

```
dividend div divisor
```

引数

分配

数値または間隔に評価される式。

除数

dividend が間隔の場合は一致する間隔タイプ、それ以外の場合は数値。

戻り型

BIGINT

例

次の例では、リステーブルから 2 つの列を選択します。各リスの一意の識別子を含む id 列と、年齢calculated列の整数除算を 2 でage div 2表す 列です。age div 2 計算はage列で整数除算を実行し、実質的に経過時間を最も近い偶数に切り下げます。たとえば、age列に 3、5、7、10 などの値が含まれている場合、age div 2列にはそれぞれ 1、2、3、5 の値が含まれます。

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

このクエリは、年齢範囲に基づいてデータをグループ化または分析する必要があり、最も近い偶数に切り下げて年齢値を簡素化する場合に役立ちます。結果の出力は、squirrelsテーブル内のリスごとに idと経過時間を 2 で割ったものになります。

EXP 関数

EXP 関数では、数値式、または式の累乗で累乗された自然対数 e の基数に対して指数関数を実装します。EXP 関数は、LN 関数の逆です。

構文

```
EXP (expression)
```

引数

expression

式のデータ型は INTEGER、DECIMAL、または DOUBLE PRECISION である必要があります。

戻り型

EXP は DOUBLE PRECISION 型の数値を返します。

例

EXP 関数は、継続的な成長パターンに基づいてチケット販売を予測するために使用します。次の例では、サブクエリによって、2008 年に販売されたチケット数が返されます。その結果に、10 年にわたって継続する成長率 7% を指定する EXP 関数の結果が乗算されます。

```
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 関数

FLOOR 関数は、数値を次の整数に切り捨てます。

構文

```
FLOOR (number)
```

引数

number

数値、または数値に評価される式。SMALLINT、INTEGER、BIGINT、DECIMAL、FLOAT4、または FLOAT8 型にすることができます。

戻り型

FLOOR は、その引数と同じデータ型を返します。

例

この例では、FLOOR 関数を使用する前と後に特定の販売取引に対して支払われたコミッションの値を示します。

```
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 関数

LN 関数は、入力パラメータの自然対数を返します。

構文

```
LN(expression)
```

引数

expression

関数の対象となる列または式。

> **Note**
>
> この関数は、式が AWS Clean Rooms ユーザーが作成したテーブル、または AWS Clean Rooms STL または STV システムテーブルを参照している場合、一部のデータ型にエラーを返します。

式の参照先がユーザー作成テーブルまたはシステムテーブルである場合、式のデータ型が以下のいずれかであるときに、エラーが発生します。

- BOOLEAN
- CHAR
- DATE
- DECIMAL または NUMERIC
- TIMESTAMP
- VARCHAR

以下のデータ型の式は、ユーザー作成テーブルおよび STL または STV システムテーブルで、正常に実行されます。

- BIGINT
- DOUBLE PRECISION

- INTEGER
- REAL
- SMALLINT

戻り型

LN 関数は、式と同じ型を返します。

例

次の例は、数値 2.718281828 の自然対数、または e を底とする対数を返します。

```
select ln(2.718281828);
ln
--------------------
0.9999999998311267
(1 row)
```

解は 1 の近似値になることに注意してください。

次の例は、USERS テーブル内の USERID 列の値の自然対数を返します。

```
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 関数

expr で の対数を返しますbase。

構文

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

引数

expr

この式は整数、10 進数、または浮動小数点数データ型である必要があります。

base

対数計算のベース。倍精度データ型の正の数 (1 と等しくない) である必要があります。

戻り型

LOG 関数は、倍精度の数値を返します。

例

次の例は、数値 100 の 10 を底とする対数を返します。

```
select log(10, 100);
--------
2
(1 row)
```

MOD 関数

2 つの数値の余りを返します。モジュロ演算とも呼ばれます。結果を計算するには、最初のパラメータを 2 番目のパラメータで除算します。

構文

```
MOD(number1, number2)
```

引数

number1

最初の入力パラメータは INTEGER 型、SMALLINT 型、BIGINT 型、または DECIMAL 型の数値です。一方のパラメータが DECIMAL 型である場合は、もう一方のパラメータも DECIMAL 型

である必要があります。一方のパラメータが INTEGER である場合、もう一方のパラメータは INTEGER、SMALLINT、または BIGINT のいずれかにします。両方のパラメータを SMALLINT または BIGINT にすることもできますが、一方のパラメータが BIGINT である場合に、もう一方のパラメータを SMALLINT にすることはできません。

number2

2 番目のパラメータは INTEGER 型、SMALLINT 型、BIGINT 型、または DECIMAL 型の数値です。number1 と同じデータ型ルールが number2 に適用されます。

戻り型

有効な戻り型は DECIMAL、INT、SMALLINT、および BIGINT です。両方の入力パラメータが同じ型である場合、MOD 関数の戻り型は、入力パラメータと同じ数値型になります。ただし、一方の入力パラメータが INTEGER である場合は、戻り型も INTEGER になります。

使用に関する注意事項

% をモジュロ演算子として使用できます。

例

次の例は、ある数値を別の数値で除算したときの余りを返します。

```
SELECT MOD(10, 4);

 mod
------
 2
```

次の例は、小数の結果を返します。

```
SELECT MOD(10.5, 4);

 mod
------
 2.5
```

パラメータ値をキャストできます。

```
SELECT MOD(CAST(16.4 as integer), 5);
```

```
 mod
------
 1
```

最初のパラメータを 2 で割って偶数かどうかをチェックします。

```
SELECT mod(5,2) = 0 as is_even;

 is_even
--------
 false
```

% をモジュロ演算子として使用できます。

```
SELECT 11 % 4 as remainder;

 remainder
-----------
 3
```

次の例は、CATEGORY テーブル内の奇数カテゴリーの情報を返します。

```
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 関数

pi 関数は、PI の値を小数第 14 位まで返します。

構文

```
PI()
```

戻り型

DOUBLE PRECISION

例

pi の値を返すには、次の例を使用します。

```
SELECT PI();

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

POWER 関数

POWER 関数は指数関数であり、最初の数値式がべき乗の底、2 番目の数値式がべき乗の指数です。例えば、2 の 3 乗は POWER(2,3) と表され、結果は 8 になります。

構文

```
{POWER(expression1, expression2)
```

引数

expression1

べき乗の底とする数値式。INTEGER、DECIMAL、または FLOAT データ型である必要があります。

expression2

expression1 を底とするべき乗の指数。INTEGER、DECIMAL、または FLOAT データ型である必要があります。

戻り型

DOUBLE PRECISION

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 関数

RADIANS 関数は、角度 (度数) をそれに相当するラジアンに変換します。

構文

```
RADIANS(number)
```

引数

number

入力パラメータは DOUBLE PRECISION 数です。

戻り型

DOUBLE PRECISION

例

180 度に相当するラジアンを返すには、次の例を使用します。

```
SELECT RADIANS(180);

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

```
+-------------------+
| 3.141592653589793 |
+-------------------+
```

RAND 関数

RAND 関数は、0 から 1 までのランダムな浮動小数点数を生成します。RAND 関数は、呼び出されるたびに新しい乱数を生成します。

構文

```
RAND()
```

戻り型

RANDOM は DOUBLE を返します。

例

次の例では、squirrelsテーブル内の行ごとに 0 から 1 までのランダムな浮動小数点数の列を生成します。結果の出力は、ランダムな 10 進値のリストを含む単一の列になり、リステーブルの行ごとに 1 つの値が含まれます。

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

このタイプのクエリは、ランダムなイベントをシミュレートしたり、データ分析にランダム性を導入したりするなど、乱数を生成する必要がある場合に便利です。squirrels テーブルのコンテキストでは、各リスにランダムな値を割り当てて、さらなる処理や分析に使用できる場合があります。

RANDOM 関数

RANDOM 関数は、0.0 (この値を含む) ~ 1.0 (この値は含まない) のランダム値を生成します。

構文

```
RANDOM()
```

戻り型

RANDOM は DOUBLE PRECISION 型の数値を返します。

例

1. 0 ~ 99 のランダム値を計算します。ランダムな数値が 0 ~ 1 である場合、このクエリは、0 ~ 100 のランダムな数値を生成します。

```
select cast (random() * 100 as int);

INTEGER
------
24
(1 row)
```

2. 10 個のアイテムの均一なランダムサンプルを取得します。

```
select *
from sales
order by random()
limit 10;
```

10 個のアイテムのランダムサンプルを取得しますが、料金に比例してアイテムを選択します。例えば、別の料金の 2 倍のアイテムは、クエリ結果に表示される可能性が 2 倍になります。

```
select *
from sales
order by log(1 - random()) / pricepaid
limit 10;
```

3. 次の例では、SET コマンドを使用して SEED 値を設定します。これにより RANDOM が予測可能な順序で数値を生成します。

まず、SEED 値を最初に設定せずに、3 つの整数の乱数を返します。

```
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)
```

次に、SEED 値を .25 に設定して、さらに 3 つの整数の乱数を返します。

```
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)
```

最後に、SEED 値を .25 にリセットして、RANDOM が前の 3 つの呼び出しと同じ結果を返すことを確認します。

```
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 関数

ROUND 関数は、数値を四捨五入して、最も近い整数または 10 進数にします。

ROUND 関数にはオプションで、2 番目の引数として整数を指定できます。この整数は、四捨五入後の小数点以下または小数点以上の桁数を指定します。2 番目の引数を指定しない場合、関数は最も近い整数に四捨五入されます。2 番目の引数 >n が指定されている場合、関数は小数点以下 n 桁の精度で最も近い数値に四捨五入されます。

構文

```
ROUND (number [ , integer ] )
```

引数

number

数値、または数値に評価される式。DECIMAL または FLOAT8 type. AWS Clean Rooms can は、暗黙的な変換ルールに従って他のデータ型を変換できます。

integer (オプション)

いずれかの方向で小数点以上または小数点以下の桁数を示す整数。

戻り型

ROUND は、入力引数と同じ数値データ型を返します。

例

特定の取引において支払われたコミッションを四捨五入して、最も近い整数にします。

```
select commission, round(commission)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    28
(1 row)
```

特定の取引において支払われたコミッションを四捨五入して、小数点以下第 1 位までの数値にします。

```
select commission, round(commission, 1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |  28.1
(1 row)
```

上記と同じクエリで、小数点以上 1 桁 (つまり 1 の位) までの数値にします。

```
select commission, round(commission, -1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    30
(1 row)
```

SIGN 関数

SIGN 関数は、数の符号 (正または負) を返します。SIGN 関数の結果は、引数の符号を示す 1、-1、または 0 です。

構文

```
SIGN (number)
```

引数

number

数値、または数値に評価される式。DECIMALor FLOAT8 type. AWS Clean Rooms can は、暗黙的な変換ルールに従って他のデータ型を変換できます。

戻り型

SIGN は、入力引数と同じ数値データ型を返します。入力が DECIMAL の場合、出力は DECIMAL(1,0) になります。

例

SALES テーブルから、特定の取引において支払われたコミッションの符号を判別するには、次の例を使用します。

```
SELECT commission, SIGN(commission)
FROM sales WHERE salesid=10000;

+------------+------+
| commission | sign |
+------------+------+
|      28.05 |    1 |
+------------+------+
```

SIN 関数

SIN は、数値のサイン (正弦) を返す三角関数です。戻り値は、-1 ~ 1 です。

構文

```
SIN(number)
```

引数

number

ラジアン単位の DOUBLE PRECISION 数。

戻り型

DOUBLE PRECISION

例

-PI のサイン (正弦) を返すには、次の例を使用します。

```
SELECT SIN(-PI());

+-------------------------+
|           sin           |
+-------------------------+
| -0.00000000000000012246 |
+-------------------------+
```

SQRT 関数

SQRT 関数は、数値の平方根を返します。平方根は、与えられた値を得るためにそれ自体を掛けた数値です。

構文

```
SQRT (expression)
```

引数

expression

この式は整数、10 進数、または浮動小数点数データ型である必要があります。式には関数を含めることができます。システムが暗黙的にタイプの変換を行う場合があります。

戻り型

SQRT は DOUBLE PRECISION 型の数値を返します。

例

次の例では、数値の平方根を返します。

```
select sqrt(16);
```

```
sqrt
---------------
4
```

次の例では、暗黙的なタイプの変換を実行します。

```
select sqrt('16');

sqrt
---------------
4
```

次の例では、関数をネストしてより複雑なタスクを実行します。

```
select sqrt(round(16.4));

sqrt
---------------
4
```

次の例では、円のエリアを指定したときの半径の長さが得られます。例えば、エリアを平方インチで指定すると、半径をインチで計算します。サンプルのエリアは 20 です。

```
select sqrt(20/pi());
```

これにより 5.046265044040321 という値が返されます。

次の例では、SALES テーブルから COMMISSION 値の平方根を返します。COMMISSION 列は DECIMAL 型列です。この例は、より複雑な条件ロジックを含むクエリで関数を使用する方法を示しています。

```
select sqrt(commission)
from sales where salesid < 10 order by salesid;

sqrt
------------------
10.4498803820905
3.37638860322683
7.24568837309472
```

```
5.1234753829798
...
```

次のクエリでは、上記と同じ COMMISSION 値の四捨五入された平方根を返します。

```
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
...
```

のサンプルデータの詳細については AWS Clean Rooms、「サンプルデータベース」を参照してください。

TRUNC 関数

TRUNC 関数は、数値を前の整数または小数に切り捨てます。

TRUNC 関数にはオプションで、2 番目の引数として整数を指定できます。この整数は、四捨五入後の小数点以下または小数点以上の桁数を指定します。2 番目の引数を指定しない場合、関数は最も近い整数に四捨五入されます。2 番目の引数 >n が指定されている場合、関数は小数点以下 >n 桁以上の精度で最も近い数値に四捨五入されます。この関数は、タイムスタンプも切り捨て、日付を返します。

構文

```
TRUNC (number [ , integer ] |
timestamp )
```

引数

number

数値、または数値に評価される式。DECIMAL または FLOAT8 type. AWS Clean Rooms can は、暗黙的な変換ルールに従って他のデータ型を変換できます。

integer (オプション)

小数点以上または小数点以下の桁数を示す整数。この整数が指定されなかった場合は、数値が切り捨てられて整数になります。この整数が指定された場合は、数値が切り捨てられて、指定された桁数になります。

timestamp

この関数は、タイムスタンプから取得した日付も返します。(タイムスタンプの値 00:00:00 を時刻として返すには、関数の結果をタイムスタンプにキャストします。)

戻り型

TRUNC は、最初の入力引数と同じデータ型を返します。タイムスタンプの場合、TRUNC は日付を返します。

例

特定の販売取引について支払われたコミッションを切り捨てます。

```
select commission, trunc(commission)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   111

(1 row)
```

同じコミッション値を切り捨てて、小数点以下第 1 位までの数値にします。

```
select commission, trunc(commission,1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 | 111.1

(1 row)
```

コミッションを切り捨てますが、2 番目の引数に負の値が指定されています。111.15 は切り捨てられて、110になります。

```
select commission, trunc(commission,-1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   110
(1 row)
```

SYSDATE 関数 (タイムスタンプを返す) の結果から日付部分を返します。

```
select sysdate;

timestamp
----------------------------
2011-07-21 10:32:38.248109
(1 row)

select trunc(sysdate);

trunc
------------
2011-07-21
(1 row)
```

TRUNC 関数を TIMESTAMP 列に適用します。戻り型は日付です。

```
select trunc(starttime) from event
order by eventid limit 1;

trunc
------------
2008-01-25
(1 row)
```

スカラー関数

このセクションでは、 AWS Clean Rooms Spark SQL でサポートされているスカラー関数について説明します。スカラー関数は、1 つ以上の値を入力として受け取り、1 つの値を出力として返す関数です。スカラー関数は個々の行または要素で動作し、入力ごとに 1 つの結果を生成します。

SIZE などのスカラー関数は、集計関数 (カウント、合計、平均) やテーブル生成関数 (爆発、フラット化) など、他のタイプの SQL 関数とは異なります。これらの他の関数タイプは複数の行で動作するか、複数の行を生成しますが、スカラー関数は個々の行または要素で動作します。

トピック

SIZE 関数

SIZE 関数は、既存の配列、マップ、または文字列を引数として受け取り、そのデータ構造のサイズまたは長さを表す単一の値を返します。新しいデータ構造は作成されません。これは、新しいデータ構造を作成するのではなく、既存のデータ構造のプロパティをクエリおよび分析するために使用されます。

この関数は、配列内の要素の数または文字列の長さを決定するのに役立ちます。SQL で配列やその他のデータ構造を使用する場合、データのサイズやカーディナリティに関する情報を取得できるため、特に便利です。

構文

```
size(expr)
```

引数

expr

ARRAY、MAP、または STRING 式。

戻り型

SIZE 関数は INTEGER を返します。

例

この例では、SIZE 関数が配列 に適用され`['b', 'd', 'c', 'a']`、配列内の要素の数`4`である値を返します。

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

この例では、SIZE 関数がマップ に適用され{'a': 1, 'b': 2}、マップ内のキーと値のペアの数2である値 を返します。

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

この例では、SIZE 関数が文字列 に適用され'hello world'、文字列内の文字数11である値 を返します。

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

文字列関数

文字列関数は、文字列、または文字列に評価される式を処理します。これらの関数の string 引数がリテラル値である場合は、その値を一重引用符で囲む必要があります。サポートされるデータ型には、CHAR や VARCHAR があります。

次のセクションでは、サポートされる関数の関数名と構文を示し、それらについて説明します。文字列のオフセットはすべて 1 から始まります。

トピック

- || (連結) 演算子
- BTRIM 関数
- CONCAT 関数
- FORMAT_STRING 関数
- LEFT 関数および RIGHT 関数
- LENGTH 関数
- LOWER 関数
- LPAD 関数および RPAD 関数
- LTRIM 関数
- POSITION 関数
- REGEXP_COUNT 関数
- REGEXP_INSTR 関数
- REGEXP_REPLACE 関数

- REGEXP_SUBSTR 関数
- REPEAT 関数
- REPLACE 関数
- REVERSE 関数
- RTRIM 関数
- SPLIT 関数
- SPLIT_PART 関数
- SUBSTRING 関数
- TRANSLATE 関数
- TRIM 関数
- UPPER 関数
- UUID 関数

|| (連結) 演算子

|| 記号の両側の 2 つの表現を連結し、その結果の表現を返します。

連結演算子は CONCAT 関数 に似ています。

> **Note**
>
> CONCAT 関数ならびに連結演算子のどちらにおいても、一方または両方の表現が null である場合は、連結の結果も null になります。

構文

```
expression1 || expression2
```

引数

expression1, expression2

両方の引数を、固定長または可変長の文字列または式にすることができます。

戻り型

|| 演算子は文字列を返します。文字列の型は、入力引数の型と同じです。

例

次の例では、USERS テーブルの FIRSTNAME フィールドと LASTNAME フィールドを連結します。

```
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)
```

Null を含む可能性がある列を連結するには、NVL および COALESCE 関数式を使用します。次の例は、NVL を使用して、NULL が発生するたびに 0 を返します。

```
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 関数

BTRIM 関数は、先頭および末尾の空白を削除するか、またはオプションで指定された文字列と一致する先頭および末尾の文字を削除することによって、文字列を切り捨てます。

構文

```
BTRIM(string [, trim_chars ] )
```

引数

string

切り捨てる入力 VARCHAR 文字列。

trim_chars

イッチする文字を含む VARCHAR 文字列。

戻り型

BTRIM 関数は、VARCHAR 型の文字列を返します。

例

次の例では、文字列 ' abc ' の先頭および末尾の空白を切り捨てます。

```
select '     abc    ' as untrim, btrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

次の例では、文字列 'xyzaxyzbxyzcxyz' から先頭および末尾の文字列 'xyz' を削除します。先頭および末尾にある 'xyz' は削除されますが、文字列内部にあるその文字列は削除されません。

```
select 'xyzaxyzbxyzcxyz' as untrim,
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | axyzbxyzc
```

次の例では、trim_chars リスト 'tes' のいずれかの文字と一致する文字列 'setuphistorycassettes' の先頭と末尾の部分を削除します。入力文字列の先頭または末尾にある trim_chars リストに含まれていない別の文字の前に出現する t、e または s が削除されます。

```
SELECT btrim('setuphistorycassettes', 'tes');

     btrim
-----------------
 uphistoryca
```

CONCAT 関数

CONCAT 関数は、2 つの式を連結し、結果の表現を返します。3 つ以上の式を連結するには、CONCAT 関数をネストして使用します。2 つの式の間に連結演算子 (||) を指定した場合も、CONCAT 関数と同じ結果が返されます。

> **Note**
>
> CONCAT 関数ならびに連結演算子のどちらにおいても、一方または両方の表現が null である場合は、連結の結果も null になります。

構文

```
CONCAT ( expression1, expression2 )
```

引数

expression1, expression2

どちらの引数にも、固定長文字列、可変長文字列、バイナリ式、またはこれらの入力のいずれかとして評価される式を指定できます。

戻り型

CONCAT は式を返します。式のデータ型は、入力引数の型と同じです。

入力式が異なる型である場合、 は暗黙的に型をキャスト AWS Clean Rooms しようとします。値を型キャストできない場合は、エラーが返されます。

例

次の例では、2 つの文字リテラルを連結します。

```
select concat('December 25, ', '2008');

concat
-------------------
December 25, 2008
(1 row)
```

次のクエリでは、CONCAT ではなく || 演算子を使用しており、同じ結果が返されます。

```
select 'December 25, '||'2008';

concat
-------------------
December 25, 2008
(1 row)
```

次の例では、2 つの CONCAT 関数を使用して、3 つの文字列を連結します。

```
select concat('Thursday, ', concat('December 25, ', '2008'));

concat
----------------------------
Thursday, December 25, 2008
(1 row)
```

Null を含む可能性がある列を連結するには、NVL および COALESCE 関数を使用します。次の例は、NVL を使用して、NULL が発生するたびに 0 を返します。

```
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)
```

次のクエリでは、VENUE テーブル内の CITY 値と STATE 値を連結します。

```
select concat(venuecity, venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
-------------------
DenverCO
Kansas CityMO
East RutherfordNJ
LandoverMD
(4 rows)
```

次のクエリでは、ネストされた CONCAT 関数を使用しています。このクエリは、VENUE テーブル内の CITY 値と STATE 値を連結しますが、結果の文字列をカンマおよびスペースで区切ります。

```
select concat(concat(venuecity,', '),venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
---------------------
Denver, CO
Kansas City, MO
East Rutherford, NJ
Landover, MD
(4 rows)
```

FORMAT_STRING 関数

FORMAT_STRING 関数は、テンプレート文字列のプレースホルダーを指定された引数に置き換えることで、フォーマットされた文字列を作成します。printf 形式の文字列からフォーマットされた文字列を返します。

FORMAT_STRING 関数は、テンプレート文字列のプレースホルダーを引数として渡された対応する値に置き換えることで機能します。このタイプの文字列フォーマットは、出力メッセージ、レポート、またはその他のタイプの情報テキストを生成する場合など、静的テキストと動的データの組み合わせを含む文字列を動的に構築する必要がある場合に役立ちます。FORMAT_STRING 関数は、これらのタイプのフォーマットされた文字列を作成するための簡潔で読みやすい方法を提供するため、出力を生成するコードの管理と更新が容易になります。

構文

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

引数

strfmt

STRING 式。

obj

STRING 式または数値式。

戻り型

FORMAT_STRING は STRING を返します。

例

次の例には、2 つのプレースホルダーを含むテンプレート文字列が含まれています。10 進数 (整数) 値%dの場合は 、文字列値%sの場合は です。%d プレースホルダーは小数点 (整数) 値 (100) に置き換えられ、%s プレースホルダーは文字列値 () に置き換えられます"days"。出力は、プレースホルダーが指定された引数 に置き換えられたテンプレート文字列です"Hello World 100 days"。

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

LEFT 関数および RIGHT 関数

これらの関数は、文字列の左端または右端にある指定数の文字を返します。

number はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。

構文

```
LEFT ( string,  integer )

RIGHT ( string,  integer )
```

引数

string

文字列、または文字列に評価される式。

integer

正の整数。

戻り型

LEFT および RIGHT は VARCHAR 型の文字列を返します。

例

次の例は、ID が 1000 から 1005 であるイベント名の左端 5 文字および右端 5 文字を返します。

```
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 関数

LOWER 関数

文字列を小文字に変換します。LOWER は、UTF-8 マルチバイト文字に対応しています (1 文字につき最大で 4 バイトまで)。

構文

```
LOWER(string)
```

引数

string

入力パラメータは VARCHAR 文字列 (または CHAR など、暗黙的に VARCHAR に変換できるその他のデータ型) です。

戻り型

LOWER 関数は、入力文字列のデータ型と同じデータ型の文字列を返します。

例

次の例では、CATNAME フィールドを小文字に変換します。

```
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 関数および RPAD 関数

これらの関数は、指定された長さに基づいて、文字列の前または後に文字を付加します。

構文

```
LPAD (string1, length, [ string2 ])
```

```
RPAD (string1, length, [ string2 ])
```

引数

string1

文字列、または文字列に評価される式 (文字列の列名など)。

length

関数の結果の長さを定義する整数。文字列の長さはバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。指定された長さより string1 が長い場合は、(右側が) 切り捨てられます。length が負の数値である場合は、関数の結果が空の文字列になります。

string2

string1 の前または後に付加する 1 つ以上の文字。この引数はオプションです。指定されなかった場合は、スペースが使用されます。

戻り型

これらの関数は VARCHAR データ型を返します。

例

指定された一連のイベント名を切り捨てて 20 文字にします。20 文字に満たない名前の前にはスペースを付加します。

```
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)
```

上記と同じ一連のイベント名を切り捨てて 20 文字にします。ただし、20 文字に満たない名前の後には 0123456789 を付加します。

```
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 関数

文字列の先頭から文字を切り捨てます。トリム文字リスト内の文字のみを含む最長の文字列を削除します。入力文字列にトリム文字が表示されない場合、トリミングは完了です。

構文

```
LTRIM( string [, trim_chars] )
```

引数

string

トリミングする文字列列、式、または文字列リテラル。

trim_chars

文字列の先頭からトリミングする文字を表す、文字列の列、式、または文字列リテラル。指定しなかった場合、スペースがトリム文字として使用されます。

戻り型

LTRIM 関数は、入力文字列のデータ型 (CHAR または VARCHAR) と同じデータ型の文字列を返します。

例

次の例は、listime 列から年をトリミングします。文字列リテラル '2008-' のトリム文字は、左からトリミングされる文字を示します。トリム文字 '028-' を使用した場合も、同じ結果が得られます。

```
select listid, listtime, ltrim(listtime, '2008-')
from listing
order by 1, 2, 3
limit 10;

listid |      listtime       |     ltrim
-------+---------------------+---------------
     1 | 2008-01-24 06:43:29 | 1-24 06:43:29
     2 | 2008-03-05 12:25:29 | 3-05 12:25:29
     3 | 2008-11-01 07:35:33 | 11-01 07:35:33
     4 | 2008-05-24 01:18:37 | 5-24 01:18:37
     5 | 2008-05-17 02:29:11 | 5-17 02:29:11
     6 | 2008-08-15 02:08:13 | 15 02:08:13
     7 | 2008-11-15 09:38:15 | 11-15 09:38:15
     8 | 2008-11-09 05:07:30 | 11-09 05:07:30
     9 | 2008-09-09 08:03:36 | 9-09 08:03:36
    10 | 2008-06-17 09:44:54 | 6-17 09:44:54
```

LTRIM は、文字列の先頭にあるとき、trim_chars の文字をすべて削除します。次の例は、文字 'C'、'D'、および 'G' が VENUENAME の先頭にある場合 (VARCHAR 列) 、これらの文字を切り捨てます。

```
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
```

次の例では、venueid 列から取得されたしたトリム文字 2 を使用します。

```
select ltrim('2008-01-24 06:43:29', venueid)
from venue where venueid=2;

ltrim
------------------
008-01-24 06:43:29
```

次の例では、2 が '0' トリム文字の前にあるため、文字はトリミングされません。

```
select ltrim('2008-01-24 06:43:29', '0');

ltrim
-------------------
2008-01-24 06:43:29
```

次の例では、デフォルトのスペーストリム文字を使用して、文字列の先頭から 2 つのスペースをトリミングします。

```
select ltrim('  2008-01-24 06:43:29');

ltrim
-------------------
```

```
2008-01-24 06:43:29
```

POSITION 関数

文字列内の指定されたサブ文字列の位置を返します。

構文

```
POSITION(substring IN string )
```

引数

substring

string 内を検索するサブ文字列。

string

検索する文字列または列。

戻り型

POSITION 関数は、サブ文字列の位置 (0 ではなく 1 から始まる) に対応する整数を返します。位置はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。

使用に関する注意事項

文字列内のサブ文字列がない場合、POSITION は 0 を返します。

```
select position('dog' in 'fish');

position
----------
 0
(1 row)
```

例

次の例では、fishという語の中での文字列 dogfish の位置を示します。

```
select position('fish' in 'dogfish');
```

```
position
----------
 4
(1 row)
```

次の例は、SALES テーブル内で COMMISSION が 999.00 を上回る販売取引の数を返します。

```
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 関数

文字列で正規表現パターンを検索し、このパターンが文字列内に出現する回数を示す整数を返します。一致がない場合、この関数は 0 を返します。

構文

```
REGEXP_COUNT ( source_string, pattern [, position [, parameters ] ] )
```

引数

source_string

検索する文字列式 (列名など)。

pattern

正規表現パターンを表す文字列リテラル。

position

検索を開始する source_string 内の位置を示す正の整数。position はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。デフォルトは 1 です。position が 1 より小さい場合、source_string の最初の文字から検索が開始されます。position が source_string の文字数より大きい場合、結果は 0 になります。

パラメータ

関数がパターンと一致するかを示す 1 つ以上のリテラル文字列。取り得る値には以下のものがあります。

- c – 大文字と小文字を区別する一致を実行します。デフォルトでは大文字と小文字を区別するマッチングを使用します。
- i – 大文字と小文字を区別しない一致を実行します。
- p – Perl 互換正規表現 (PCRE) 言語でパターンを解釈します。

戻り型

整数

例

次の例は、3 文字のシーケンスが出現する回数をカウントします。

```
SELECT regexp_count('abcdefghijklmnopqrstuvwxyz', '[a-z]{3}');

 regexp_count
 --------------
            8
```

次の例は、最上位ドメイン名が org または edu である回数をカウントします。

```
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
```

次の例では、大文字と小文字を区別しない一致を使用して、文字列 FOX の出現をカウントします。

```
SELECT regexp_count('the fox', 'FOX', 1, 'i');

 regexp_count
 --------------
```

```
           1
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、大文字と小文字を区別して、このような単語の出現回数をカウントします。

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'p');

 regexp_count
 --------------
            2
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。これは、PCRE で特定の意味を持つ ?= 演算子を使用します。この例では、このような単語の出現回数をカウントしますが、大文字と小文字を区別しない一致結果を使用する点で前の例とは異なります。

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'ip');

 regexp_count
 --------------
            3
```

REGEXP_INSTR 関数

文字列で正規表現パターンを検索し、一致するサブ文字列の開始位置を示す整数を返します。一致がない場合、この関数は 0 を返します。REGEXP_INSTR は 関数に似ていますが、文字列で正規表現パターンを検索することができます。

構文

```
REGEXP_INSTR ( source_string, pattern [, position [, occurrence] [, option
 [, parameters ] ] ] ] )
```

引数

source_string

検索する文字列式 (列名など)。

pattern

正規表現パターンを表す文字列リテラル。

position

検索を開始する source_string 内の位置を示す正の整数。position はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。デフォルトは 1 です。position が 1 より小さい場合、source_string の最初の文字から検索が開始されます。position が source_string の文字数より大きい場合、結果は 0 になります。

occurrence

このパターンのどの出現を使用するかを示す正の整数。REGEXP_INSTR は、最初の出現 - 1 一致をスキップします。デフォルトは 1 です。出現が 1 未満、またはソース文字列の文字数以上の場合、検索は無視され、結果は 0 となります。

option

一致の先頭文字の戻り位置 (0)、あるいは続く一致の後尾の最初の文字位置 (1) のどちらかを示す値。ゼロ以外の値は 1 と同じです。デフォルト値は 0 です。

パラメータ

関数がパターンと一致するかを示す 1 つ以上のリテラル文字列。取り得る値には以下のものがあります。

- c – 大文字と小文字を区別する一致を実行します。デフォルトでは大文字と小文字を区別するマッチングを使用します。
- i – 大文字と小文字を区別しない一致を実行します。
- e – 部分式を使用して部分文字列を抽出します。

  パターンに部分式が含まれる場合、REGEXP_INSTR は最初の部分式をパターンで使用して部分文字列を一致させます。REGEXP_INSTR は最初の部分式のみを考慮します。追加の部分式は無視されます。パターンに部分式がない場合、REGEXP_INSTR は「e」パラメータを無視します。
- p – Perl 互換正規表現 (PCRE) 言語でパターンを解釈します。

戻り型

整数

例

次の例は、ドメイン名を開始する @ 文字を検索し、最初の一致の開始位置を返します。

```
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
```

次の例は、Centerという単語のバリエーションを検索し、最初の一致の開始位置を返します。

```
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
```

次の例は、大文字と小文字を区別しない一致ロジックを使用して、文字列 FOX が最初に出現する開始位置を検索します。

```
SELECT regexp_instr('the fox', 'FOX', 1, 1, 0, 'i');

 regexp_instr
 --------------
            5
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、2 番目の単語の開始位置を見つけます。

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'p');

 regexp_instr
 --------------
           21
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、2 番目の単語の開始位置を検索しますが、大文字と小文字を区別しない一致結果を使用する点で前の例とは異なります。

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'ip');

 regexp_instr
 --------------
           15
```

REGEXP_REPLACE 関数

文字列で正規表現パターンを検索し、このパターンのすべての出現を特定の文字列に置き換えます。REGEXP_REPLACE は REPLACE 関数 関数に似ていますが、文字列で正規表現パターンを検索することができます。

REGEXP_REPLACE は、TRANSLATE 関数や REPLACE 関数 と似ています。ただし、TRANSLATE は複数の単一文字置換を行い、REPLACE は 1 つの文字列全体を別の文字列に置換しますが、REGEXP_REPLACE を使用すると正規表現パターンの文字列を検索できます。

構文

```
REGEXP_REPLACE ( source_string, pattern [, replace_string [ , position [, parameters
 ] ] ] )
```

引数

source_string

検索する文字列式 (列名など)。

pattern

正規表現パターンを表す文字列リテラル。

replace_string

パターンのすべての出現に置き換わる列名などの文字列式。デフォルトは空の文字列 ("") です。

position

検索を開始する source_string 内の位置を示す正の整数。position はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。デフォルトは 1 です。position が 1 より小さい場合、source_string の最初の文字から検索が開始されます。position が source_string の文字数より大きい場合、結果は source_string になります。

パラメータ

関数がパターンと一致するかを示す 1 つ以上のリテラル文字列。取り得る値には以下のものがあります。

- c – 大文字と小文字を区別する一致を実行します。デフォルトでは大文字と小文字を区別するマッチングを使用します。
- i – 大文字と小文字を区別しない一致を実行します。
- p – Perl 互換正規表現 (PCRE) 言語でパターンを解釈します。

戻り型

VARCHAR

pattern または replace_string のいずれかが NULL の場合、戻り値は NULL です。

例

次の例は、メールアドレスから @ とドメイン名を削除します。

```
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
```

次の例は、E メールアドレスのドメイン名を値 internal.company.com に置き換えます。

```
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
```

次の例は、大文字と小文字を区別しない一致を使用して、値 quick brown fox 内の文字列 FOX のすべての出現箇所を置き換えます。

```
SELECT regexp_replace('the fox', 'FOX', 'quick brown fox', 1, 'i');

   regexp_replace
---------------------
 the quick brown fox
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、そのような単語が出現するたびに値 [hidden] に置き換えられます。

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'p');

        regexp_replace
------------------------------
 [hidden] plain A1234 [hidden]
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例で

は、このような単語が出現するたびに値 [hidden] に置き換えられますが、大文字と小文字を区別しない一致結果を使用するという点で前の例とは異なります。

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'ip');

          regexp_replace
----------------------------------
 [hidden] plain [hidden] [hidden]
```

REGEXP_SUBSTR 関数

正規表現パターンで検索して、文字列から文字を返します。REGEXP_SUBSTR は SUBSTRING 関数 関数に似ていますが、文字列で正規表現パターンを検索することができます。この関数が正規表現を文字列内のどの文字とも一致させることができない場合、空の文字列を返します。

構文

```
REGEXP_SUBSTR ( source_string, pattern [, position [, occurrence [, parameters ] ] ] )
```

引数

source_string

検索する文字列式。

pattern

正規表現パターンを表す文字列リテラル。

position

検索を開始する source_string 内の位置を示す正の整数。位置はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。デフォルトは 1 です。position が 1 より小さい場合、source_string の最初の文字から検索が開始されます。position が source_string の文字数より大きい場合、結果は空の文字列 ("") になります。

occurrence

このパターンのどの出現を使用するかを示す正の整数。REGEXP_SUBSTR は、最初の出現 - 1 一致をスキップします。デフォルトは 1 です。出現が 1 未満、またはソース文字列の文字数以上の場合、検索は無視され、結果は NULL となります。

パラメータ

関数がパターンと一致するかを示す 1 つ以上のリテラル文字列。取り得る値には以下のものがあります。

- c – 大文字と小文字を区別する一致を実行します。デフォルトでは大文字と小文字を区別するマッチングを使用します。
- i – 大文字と小文字を区別しない一致を実行します。
- e – 部分式を使用して部分文字列を抽出します。

  パターンに部分式が含まれる場合、REGEXP_SUBSTR は最初の部分式をパターンで使用して部分文字列を一致させます。部分式は、かっこで囲まれたパターン内の式です。例えば、`'This is a (\\w+)'` というパターンでは、最初の式と `'This is a '` という文字列とそれに続く単語が一致します。e パラメータを指定した REGEXP_SUBSTR は、パターンを返す代わりに、部分式の中の文字列だけを返します。

  REGEXP_SUBSTR は最初の部分式のみを考慮します。追加の部分式は無視されます。パターンに部分式がない場合、REGEXP_SUBSTR は「e」パラメータを無視します。
- p – Perl 互換正規表現 (PCRE) 言語でパターンを解釈します。

戻り型

VARCHAR

例

次の例では、メールアドレスの @ 文字とドメイン拡張の間の分が返されます。

```
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
```

次の例は、大文字と小文字を区別しない一致を使用して、文字列 FOX の最初の出現に対応する入力の部分を返します。

```
SELECT regexp_substr('the fox', 'FOX', 1, 1, 'i');

 regexp_substr
---------------
 fox
```

次の例では、小文字で始まる入力の最初の部分を返します。これは、同じ SELECT ステートメントで c パラメータを指定しない場合と機能的には同じです。

```
SELECT regexp_substr('THE SECRET CODE IS THE LOWERCASE PART OF 1931abc0EZ.', '[a-z]+',
 1, 1, 'c');

 regexp_substr
---------------
 abc
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、2 番目の単語に対応する入力部分を返します。

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'p');

 regexp_substr
---------------
 a1234
```

次の例では、PCRE 言語で記述されたパターンを使用して、少なくとも 1 つの数字と 1 つの小文字を含む単語を検索します。PCRE で特定の先読みの意味を持つ ?= 演算子を使用します。この例では、2 番目の単語に対応する入力部分を返しますが、大文字と小文字を区別しない一致結果を使用する点で前の例とは異なります。

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'ip');

 regexp_substr
---------------
 A1234
```

次の例では、部分式を使用して、大文字と小文字を区別しないマッチングにより、パターン 'this is a (\\w+)' と一致する 2 番目の文字列を見つけます。かっこ内の部分式を返します。

```
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 関数

指定された回数だけ文字列を繰り返します。入力パラメータが数値である場合、REPEAT はそれを文字列として扱います。

構文

```
REPEAT(string, integer)
```

引数

string

最初の入力パラメータは、繰り返す文字列です。

integer

2 番目のパラメータは、文字列を繰り返す回数を示す整数です。

戻り型

REPEAT 関数は文字列を返します。

例

次の例では、CATEGORY テーブル内の CATID 列の値を 3 回繰り返します。

```
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 関数

既存の文字列内の一連の文字をすべて、指定された他の文字に置き換えます。

REPLACE は、TRANSLATE 関数や REGEXP_REPLACE 関数 と似ています。ただし、TRANSLATE は複数の単一文字置換を行い、REGEXP_REPLACE を使用すると正規表現パターンの文字列を検索できますが、REPLACE は 1 つの文字列全体を別の文字列に置換します。

構文

```
REPLACE(string1, old_chars, new_chars)
```

引数

string

検索する CHAR 型または VARCHAR 型の文字列。

old_chars

置き換える CHAR 型または VARCHAR 型の文字列。

new_chars

old_string を置き換える新しい CHAR 型または VARCHAR 型の文字列。

戻り型

VARCHAR

old_chars または new_chars のいずれかが NULL の場合、戻り値は NULL です。

例

次の例では、CATGROUP フィールド内の文字列 Shows を Theatre に変換します。

```
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 関数

REVERSE 関数は、文字列に対して機能し、文字を逆順に返します。たとえば、reverse('abcde')は edcba を返します。この関数は、数値データ型と日付データ型に加え、文字データ型に対しても機能します。ただしほとんどの場合、文字列に対して実用的な値が生成されます。

構文

```
REVERSE ( expression )
```

引数

expression

文字反転のターゲットを表す文字、日付、タイムスタンプ、または数値のデータ型を使用した式。すべての式は、可変長文字列に暗黙的に変換されます。固定幅文字列の末尾の空白は無視されます。

戻り型

REVERSE は VARCHAR を返します。

例

USERS テーブルから、5 つの異なる都市名およびそれらに対応する反転した名前を選択します。

```
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)
```

文字列として変換された 5 つの販売 ID およびそれらに対応する反転した ID を選択します。

```
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 関数

RTRIM 関数は、指定された一連の文字を文字列の末尾から切り捨てます。トリム文字リスト内の文字のみを含む最長の文字列を削除します。入力文字列にトリム文字が表示されない場合、トリミングは完了です。

構文

```
RTRIM( string, trim_chars )
```

引数

string

トリミングする文字列列、式、または文字列リテラル。

trim_chars

文字列の末尾から切り捨てる文字を表す、文字列の列、式、または文字列リテラル。指定しなかった場合、スペースがトリム文字として使用されます。

戻り型

string 引数と同じデータ型の文字列。

例

次の例では、文字列 ' abc ' の先頭および末尾の空白を切り捨てます。

```
select '     abc    ' as untrim, rtrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    |    abc
```

次の例では、文字列 'xyzaxyzbxyzcxyz' から末尾の文字列 'xyz' を削除します。末尾にある 'xyz' は削除されましたが、文字列内部にあるその文字列は削除されません。

```
select 'xyzaxyzbxyzcxyz' as untrim,
rtrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
```

```
-----------------+-----------
 xyzaxyzbxyzcxyz | xyzaxyzbxyzc
```

次の例では、trim_chars リスト 'tes' のいずれかの文字と一致する文字列 'setuphistorycassettes' の末尾の部分を削除します。入力文字列の末尾にある trim_chars リストに含まれていない別の文字の前に出現する t、e または s が削除されます。

```
SELECT rtrim('setuphistorycassettes', 'tes');

     rtrim
-----------------
 setuphistoryca
```

次の例は、文字 'Park' が存在する VENUENAME の末尾から、これらの文字を切り捨てます。

```
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
```

RTRIM は、文字 P、a、r、または k が VENUENAME の末尾にあるとき、それらをすべて削除することに注意してください。

SPLIT 関数

SPLIT 関数を使用すると、より大きな文字列から部分文字列を抽出し、配列として使用できます。SPLIT 関数は、特定の区切り文字またはパターンに基づいて文字列を個々のコンポーネントに分割する必要がある場合に便利です。

構文

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

引数

str

分割する文字列式。

[正規表現]

正規表現を表す文字列。正規表現文字列は Java 正規表現である必要があります。

制限

正規表現が適用される回数を制御する整数式。

- limit > 0: 結果の配列の長さは制限を超えず、結果の配列の最後のエントリには、最後に一致した正規表現を超えるすべての入力が含まれます。
- limit <= 0: 正規表現は可能な限り複数回適用され、結果の配列は任意のサイズにすることができます。

戻り型

SPLIT 関数は ARRAY<STRING> を返します。

の場合`limit > 0`: 結果の配列の長さは制限を超えず、結果の配列の最後のエントリには、最後に一致した正規表現を超えるすべての入力が含まれます。

の場合`limit <= 0`: 正規表現は可能な限り複数回適用され、結果の配列は任意のサイズにすることができます。

例

この例では、SPLIT 関数は、文字 'A'、、'B'または 'C' (正規表現パターン で指定) に遭遇する'oneAtwoBthreeC'場所に入力文字列を分割します'[ABC]'。結果の出力は、"one"、、"two"、"three"および空の文字列 の 4 つの要素の配列です""。

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

SPLIT_PART 関数

指定された区切り記号で文字列を分割し、指定された位置にあるパートを返します。

構文

```
SPLIT_PART(string, delimiter, position)
```

引数

string

分割する文字列の列、式、または文字列リテラル。文字列には CHAR 型または VARCHAR 型を指定できます。

delimiter

入力文字列のセクションを示す区切り文字列。

delimiter がリテラルである場合は、それを一重引用符で囲みます。

position

返す文字列の部分の位置 (1 からカウント)。1 以上の整数である必要があります。位置が文字列の部分の数より大きい場合、SPLIT_PART は空の文字列を返します。文字列で区切り文字が見つからない場合、戻り値には指定された部分の内容が含まれます。これは、文字列全体または空の値である可能性があります。

戻り型

CHAR 文字列または VARCHAR 文字列 (文字列パラメータと同じ型)。

例

次の例では、$ 区切り文字を使用して文字列リテラルを複数の部分に分割し、2 番目の部分を返します。

```
select split_part('abc$def$ghi','$',2)

split_part
----------
def
```

次の例では、$ 区切り文字を使用して文字列リテラルを複数の部分に分割します。4 の部分が見つからないため、空の文字列が返されます。

```
select split_part('abc$def$ghi','$',4)

split_part
----------
```

次の例では、# 区切り文字を使用して文字列リテラルを複数の部分に分割します。区切り文字が見つからないため、最初の部分である文字列全体が返されます。

```
select split_part('abc$def$ghi','#',1)

split_part
------------
abc$def$ghi
```

次の例は、タイムスタンプフィールド LISTTIME を年コンポーネント、月コンポーネント、および日コンポーネントに分割します。

```
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
```

次の例は、LISTTIME タイムスタンプフィールドを選択し、それを '-' 文字で分割して月 (LISTTIME 文字列の 2 番目の部分) を取得してから、各月のエントリ数をカウントします。

```
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 関数

文字列内で、指定された開始位置からの文字列のサブセットを返します。

入力が文字列の場合、抽出される文字の開始位置および文字数はバイト数ではなく文字数に基づきます。つまり、マルチバイト文字は 1 文字としてカウントされます。入力がバイナリ式の場合、開始位置と抽出される部分文字列はバイト数に基づきます。負の長さを指定することはできませんが、開始位置を負に指定することは可能です。

構文

```
SUBSTRING(charactestring FROM start_position [ FOR numbecharacters ] )
```

```
SUBSTRING(charactestring, start_position, numbecharacters )
```

```
SUBSTRING(binary_expression, start_byte, numbebytes )
```

```
SUBSTRING(binary_expression, start_byte )
```

引数

charactestring

検索する文字列。文字データ型以外のデータ型は、文字列のように扱われます。

start_position

文字列内で抽出を開始する位置 (1 から始まる)。start_position はバイト数ではなく文字数に基づくため、マルチバイト文字は 1 文字としてカウントされます。負の数を指定することもできます。

numbecharacters

抽出する文字の数 (サブ文字列の長さ)。数値文字はバイト数ではなく文字数に基づいているため、マルチバイト文字は 1 文字としてカウントされます。負の数を指定することはできません。

start_byte

バイナリ表現内の抽出を開始する (先頭を 1 とする) 位置。負の数を指定することもできます。

numbebytes

抽出するバイト数 (サブ文字列の長さ)。負の数を指定することはできません。

戻り型

VARCHAR

文字列の使用に関する注意事項

次の例では、6 番目の文字で始まる 4 文字の文字列を返します。

```
select substring('caterpillar',6,4);
substring
-----------
pill
(1 row)
```

start_position + numbecharacters が文字列の長さを超える場合、SUBSTRING は start_position から文字列の末尾まで部分文字列を返します。例:

```
select substring('caterpillar',6,8);
substring
-----------
pillar
(1 row)
```

start_position が負の数または 0 である場合、SUBSTRING 関数は、文字列の先頭文字から start_position + numbecharacters -1 文字までをサブ文字列として返します。次に例を示します。

```
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)
```

start_position + numbecharacters -1 が 0 以下である場合、SUBSTRING は空の文字列を返します。次に例を示します。

```
select substring('caterpillar',-5,4);
substring
-----------

(1 row)
```

例

次の例は、LISTING テーブル内の LISTTIME 文字列から月を返します。

```
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)
```

次の例は上記と同じですが、FROM...FOR オプションを使用します。

```
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)
```

文字列にマルチバイト文字が含まれる可能性がある場合、SUBSTRING を使用して文字列の先頭部分を期待どおりに抽出することはできません。これは、マルチバイト文字列の長さを、文字数ではなくバイト数に基づいて指定する必要があるためです。バイト数での長さに基づいて文字列の最初のセグメントを取得するためには、文字列を VARCHAR(byte_length) として CAST することで文字列を切り捨てます。このとき、byte_length は必要な長さとなります。次の例では、文字列 'Fourscore and seven' から最初の 5 バイトを抽出します。

```
select cast('Fourscore and seven' as varchar(5));

varchar
-------
Fours
```

次の例では、入力文字列 Ana の最後のスペースの後に表示される最初の名前 Silva, Ana を返します。

```
select reverse(substring(reverse('Silva, Ana'), 1, position(' ' IN reverse('Silva,
 Ana'))))
```

```
 reverse
-----------
 Ana
```

TRANSLATE 関数

任意の式において、指定された文字をすべて、指定された別の文字に置き換えます。既存の文字は、characters_to_replace 引数および characters_to_substitute 引数内の位置により置換文字にマッピングされます。characters_to_replace 引数で characters_to_substitute 引数よりも多くの文字が指定されている場合、characters_to_replace 引数からの余分な文字は戻り値で省略されます。

TRANSLATE は、REPLACE 関数や REGEXP_REPLACE 関数 と似ています。ただし、REPLACE は 1 つの文字列全体を別の文字列に置換し、REGEXP_REPLACE を使用すると正規表現パターンの文字列を検索できますが、TRANSLATE は複数の単一文字置換を行います。

いずれかの引数が null である場合、戻り値は NULL になります。

構文

```
TRANSLATE ( expression, characters_to_replace, characters_to_substitute )
```

引数

expression

変換する式。

characters_to_replace

置換する文字を含む文字列。

characters_to_substitute

代入する文字を含む文字列。

戻り型

VARCHAR

例

以下の例では、文字列内の複数の文字が置換されます。

```
select translate('mint tea', 'inea', 'osin');

translate
-----------
most tin
```

次の例では、列内のすべての値のアットマーク (@) がピリオドに置き換えられます。

```
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
```

次の例では、列内のすべての値のスペースがアンダースコアに置き換えられ、ピリオドが削除されます。

```
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 関数

先頭および末尾の空白を削除するか、またはオプションで指定された文字列と一致する先頭および末尾の文字を削除することによって、文字列を切り捨てます。

構文

```
TRIM( [ BOTH ] [ trim_chars FROM ] string
```

引数

trim_chars

(オプション) 文字列から切り捨てられる文字。このパラメータを省略すると、空白が切り捨てられます。

string

切り捨てる文字列。

戻り型

TRIM 関数は、VARCHAR 型または CHAR 型の文字列を返します。SQL コマンドで TRIM 関数を使用する場合、 は結果を VARCHAR に AWS Clean Rooms 暗黙的に変換します。SQL 関数の SELECT リストで TRIM 関数を使用する場合、 AWS Clean Rooms は結果を暗黙的に変換せず、データ型の不一致エラーを回避するために明示的な変換を実行する必要がある場合があります。明示的な変換については、 CAST 関数関数を参照してください。

例

次の例では、文字列 ' abc ' の先頭および末尾の空白を切り捨てます。

```
select '     abc    ' as untrim, trim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

次の例では、文字列 "dog" を囲む二重引用符を削除します。

```
select trim('"' FROM '"dog"');

btrim
-------
dog
```

LTRIM は、文字列の先頭にあるとき、trim_chars の文字をすべて削除します。次の例は、文字 'C'、'D'、および 'G' が VENUENAME の先頭にある場合 (VARCHAR 列) 、これらの文字を切り捨てます。

```
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 関数

文字列を大文字に変換します。UPPER は、UTF-8 マルチバイト文字に対応しています (1 文字につき最大で 4 バイトまで)。

構文

```
UPPER(string)
```

引数

string

入力パラメータは VARCHAR 文字列 (または CHAR など、暗黙的に VARCHAR に変換できるその他のデータ型) です。

戻り型

UPPER 関数は、入力文字列のデータ型と同じデータ型の文字列を返します。

例

次の例は、CATNAME フィールドを大文字に変換します。

```
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 関数

UUID 関数は、Universally Unique Identifier (UUID) を生成します。

UUIDsはグローバルに一意の識別子であり、次のようなさまざまな目的で一意の識別子を提供するために一般的に使用されます。

- データベースレコードまたはその他のデータエンティティの識別。
- ファイル、ディレクトリ、またはその他のリソースの一意の名前またはキーを生成します。
- 分散システム全体のデータを追跡し、関連付けます。
- ネットワークパケット、ソフトウェアコンポーネント、またはその他のデジタルアセットに一意の識別子を提供します。

UUID 関数は、分散システム間および長期間にわたっても、非常に高い確率で一意の UUID 値を生成します。UUIDsは通常、現在のタイムスタンプ、コンピュータのネットワークアドレス、およびその他のランダムまたは擬似ランダムデータの組み合わせを使用して生成され、生成された各 UUID が他の UUID と競合する可能性はほとんどありません。

SQL クエリのコンテキストでは、UUID 関数を使用して、データベースに挿入される新しいレコードの一意の識別子を生成したり、データのパーティショニング、インデックス作成、または一意の識別子が必要なその他の目的で一意のキーを提供したりできます。

> **Note**
>
> UUID 関数は非決定的です。

構文

```
uuid()
```

引数

UUID 関数は引数を取りません。

戻り型

UUID は、ユニバーサル一意識別子 (UUID) 文字列を返します。値は正規 UUID 36 文字の文字列として返されます。

例

次の例では、Universally Unique Identifier (UUID) を生成します。出力は、Universally Unique Identifier を表す 36 文字の文字列です。

```
SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

プライバシー関連の機能

AWS Clean Rooms には、以下の仕様のプライバシー関連のコンプライアンスに準拠するのに役立つ機能が用意されています。

- グローバルプライバシープラットフォーム (GPP) – オンラインプライバシーとデータ使用のためのグローバルで標準化されたフレームワークを確立する、Interactive Advertising Bureau (IAB) からの仕様。GPP の技術仕様の詳細については、GitHub のグローバルプライバシープラットフォームのドキュメントを参照してください。
- Transparency and Framework Framework (TCF) – 2020 年に開始された GPP の主要なコンポーネントであり、企業が EU 一般データ保護規則 (GDPR) などのプライバシー規制に準拠するための標準化された技術フレームワークを提供します。TCF を使用すると、お客様はデータ収集と処理への同意を付与または拒否できます。TCF の技術仕様の詳細については、GitHub の TCF ドキュメントを参照してください。

トピック

- consent_gpp_v1_decode 関数
- consent_tcf_v2_decode 関数

consent_gpp_v1_decode 関数

consent_gpp_v1_decode 関数は、グローバルプライバシープラットフォーム (GPP) v1 同意データをデコードするために使用されます。エンコードされた同意文字列を入力として受け取り、デコードされた同意データを返します。これには、ユーザーのプライバシー設定と同意の選択に関する情報が含まれます。この関数は、GPP v1 の同意情報を含むデータを操作する場合に便利です。これは、構造化された形式で同意データにアクセスして分析できるためです。

構文

```
consent_gpp_v1_decode(gpp_string)
```

引数

gpp_string

エンコードされた GPP v1 同意文字列。

戻り値

返されるディクショナリには、次のキーと値のペアが含まれます。

- version: 使用されている GPP 仕様のバージョン (現在は 1)。
- cmpId: 同意文字列をエンコードした同意管理プラットフォーム (CMP) の ID。
- cmpVersion: 同意文字列をエンコードした CMP のバージョン。
- consentScreen: ユーザーが同意した CMP UI の画面の ID。
- consentLanguage: 同意情報の言語コード。
- vendorListVersion: 使用されるベンダーリストのバージョン。
- publisherCountryCode: パブリッシャーの国コード。
- purposeConsent: ユーザーが同意した目的を表す整数のリスト。
- purposeLegitimateInterest: ユーザーの正当な利益が透過的に伝達された目的 IDs のリスト。
- specialFeatureOptIns: ユーザーがオプトインした特別な機能を表す整数のリスト。
- vendorConsent: ユーザーが同意したベンダー IDs のリスト。
- vendorLegitimateInterest: ユーザーの正当な利益が透過的に伝達されているベンダー IDs のリスト。

例

次の例では、エンコードされた同意文字列である 1 つの引数を取ります。ユーザーのプライバシー設定、同意の選択、その他のメタデータに関する情報など、デコードされた同意データを含むディクショナリを返します。

```
SELECT * FROM consent_gpp_v1_decode('ABCDEFGHIJK');
```

返される同意データの基本構造には、同意文字列のバージョン、CMP (Consent Management Platform) の詳細、さまざまな目的やベンダーに対するユーザーの同意と正当な利益の選択、およびその他のメタデータに関する情報が含まれます。

```
{
    "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 関数

consent_tcf_v2_decode 関数は、Transparency and Framework Framework (TCF) v2 同意データをデコードするために使用されます。エンコードされた同意文字列を入力として受け取り、デコードされた同意データを返します。これには、ユーザーのプライバシー設定と同意の選択に関する情報が含まれます。この関数は、構造化された形式で同意データにアクセスして分析できるため、TCF v2 同意情報を含むデータを操作する場合に便利です。

構文

```
consent_tcf_v2_decode(tcf_string)
```

引数

tcf_string

エンコードされた TCF v2 同意文字列。

戻り値

このconsent_tcf_v2_decode関数は、Transparency and Framework (TCF) v2 同意文字列からデコードされた同意データを含むディクショナリを返します。

返されるディクショナリには、次のキーと値のペアが含まれます。

コアセグメント

- version: 使用されている TCF 仕様のバージョン (現在は 2)。
- created: 同意文字列が作成された日時。
- lastUpdated: 同意文字列が最後に更新された日時。
- cmpId: 同意文字列をエンコードした同意管理プラットフォーム (CMP) の ID。
- cmpVersion: 同意文字列をエンコードした CMP のバージョン。
- consentScreen: ユーザーが同意した CMP UI の画面の ID。
- consentLanguage: 同意情報の言語コード。
- vendorListVersion: 使用されるベンダーリストのバージョン。
- tcfPolicyVersion: 同意文字列が基づいている TCF ポリシーのバージョン。
- isServiceSpecific: 同意が特定のサービスに固有であるか、すべてのサービスに適用されるかを示すブール値。
- useNonStandardStacks: 非標準スタックを使用するかどうかを示すブール値。
- specialFeatureOptIns: ユーザーがオプトインした特別な機能を表す整数のリスト。
- purposeConsent: ユーザーが同意した目的を表す整数のリスト。
- purposesLITransparency: ユーザーが正当な利益の透明性を付与した目的を表す整数のリスト。
- purposeOneTreatment: ユーザーが「目的 1 つの処理」をリクエストしたかどうかを示すブール値 (つまり、すべての目的が均等に扱われます) 。
- publisherCountryCode: パブリッシャーの国コード。
- vendorConsent: ユーザーが同意したベンダー IDs のリスト。
- vendorLegitimateInterest: ユーザーの正当な利益が透過的に伝達されているベンダー IDs のリスト。
- pubRestrictionEntry: パブリッシャーの制限のリスト。このフィールドには、目的 ID、制限タイプ、およびその目的制限に基づくベンダー IDs のリストが含まれます。

孤立したベンダーセグメント

- disclosedVendors: ユーザーに公開されたベンダーを表す整数のリスト。

パブリッシャー目的セグメント

- pubPurposesConsent: ユーザーが同意したパブリッシャー固有の目的を表す整数のリスト。
- pubPurposesLITransparency: ユーザーが正当な利益の透明性を付与したパブリッシャー固有の目的を表す整数のリスト。
- customPurposesConsent: ユーザーが同意したカスタム目的を表す整数のリスト。
- customPurposesLITransparency: ユーザーが正当な利益の透明性を付与したカスタム目的を表す整数のリスト。

この詳細な同意データは、個人データを使用する際のユーザーのプライバシー設定を理解し、尊重するために使用できます。

例

次の例では、エンコードされた同意文字列である 1 つの引数を取ります。ユーザーのプライバシー設定、同意の選択、その他のメタデータに関する情報など、デコードされた同意データを含むディクショナリを返します。

```
from aws_clean_rooms.functions import consent_tcf_v2_decode

consent_string = "CO1234567890abcdef"
consent_data = consent_tcf_v2_decode(consent_string)

print(consent_data)
```

返される同意データの基本構造には、同意文字列のバージョン、CMP (Consent Management Platform) の詳細、さまざまな目的やベンダーに対するユーザーの同意と正当な利益の選択、およびその他のメタデータに関する情報が含まれます。

```
    /** 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],
};
```

ウィンドウ関数

ウィンドウ関数を使用すると、分析的なビジネスクエリをより効率的に作成できます。ウィンドウ関数はパーティションまたは結果セットの「ウィンドウ」で演算し、ウィンドウのすべての行に値を返します。それに対して、ウィンドウ以外の関数は、結果セットの行ごとに計算を実行します。結果の行を集計するグループ関数とは異なり、ウィンドウ関数はテーブル式のすべての行を保持します。

戻り値はこのウィンドウの行セットの値を使用して計算されます。ウィンドウはテーブルの各行に、追加の属性を計算するために使用する行のセットを定義します。ウィンドウはウィンドウ仕様 (OVER 句) を使用して定義され、次の 3 つの主要な概念に基づいています。

- ウィンドウのパーティション、列のグループを形成 (PARTITION 句)
- ウィンドウの並び順、各パーティション内の行の順序またはシーケンスの定義 (ORDER BY 句)
- ウィンドウのフレーム、各行に関連して定義され、行のセットをさらに制限 (ROWS 仕様)

ウィンドウ関数は、最後の ORDER BY 句を除いて、クエリで実行される最後の演算のセットです。すべての結合およびすべての WHERE、GROUP BY、および HAVING 句は、ウィンドウ関数が処理

される前に完了されます。そのため、ウィンドウ関数は選択リストまたは ORDER BY 句のみに表示できます。複数のウィンドウ関数は、別のフレーム句を持つ 1 つのクエリ内で使用できます。ウィンドウ関数は、CASE などの他のスカラー式でも使用できます。

ウィンドウ関数の構文の概要

ウィンドウ関数は、次のような標準構文に従います。

```
function (expression) OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list [ frame_clause ] ] )
```

ここで、function は、このセクションで説明している関数の 1 つです。

expr_list は次のとおりです。

```
expression | column_name [, expr_list ]
```

order_list は次のとおりです。

```
expression | column_name [ ASC | DESC ]
[ NULLS FIRST | NULLS LAST ]
[, order_list ]
```

frame_clause は次のとおりです。

```
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 }}
```

引数

function

ウィンドウ関数。詳細については、個々の関数の説明を参照してください。

OVER

ウィンドウの仕様を定義する句。OVER 句はウィンドウ関数に必須であり、ウィンドウ関数を他の SQL 関数と区別します。

PARTITION BY expr_list

(オプション) PARTITION BY 句は結果セットをパーティションに再分割します。これは GROUP BY 句と似ています。パーティション句が存在する場合、関数は各パーティションの行に対して計算されます。パーティション句が指定されていない場合、1 つのパーティションにテーブル全体が含まれ、関数は完全なテーブルに対して計算されます。

ランク付け関数 DENSE_RANK、NTILE、RANK、および ROW_NUMBER では、結果セットのすべての行でグローバルな比較が必要です。PARTITION BY clauseを使用すると、クエリオプティマイザーは、パーティションに応じて複数のスライスにワークロードを分散させることにより、個々の集計を並列で実行できます。PARTITION BY 句がない場合、集計ステップを 1 つのスライスで順次実行する必要があり、特に大規模なクラスターではパフォーマンスに大きな悪影響を与えることがあります。

AWS Clean Roomsは、PARTITION BY 句の文字列リテラルをサポートしていません。

ORDER BY order_list

(オプション) ウィンドウ関数は、ORDER BY で順序仕様に従ってソートされた各パーティション内の行に適用されます。この ORDER BY 句は、frame_clause の ORDER BY 句とは異なり、両者はまったく無関係です。ORDER BY 句は、PARTITION BY 句なしで使用できます。

ランク付け関数の場合、ORDER BY 句はランク付けの値に使用する基準を特定します。集計関数の場合、パーティションで分割された行は、集計関数がフレームごとに計算される前に順序付けされる必要があります。ウィンドウ関数の種類の詳細については、「ウィンドウ関数」を参照してください。

列識別子または列識別を検証する式は、順序リストで必要とされます。定数も定数式も、列名の代用として使用することはできません。

NULL 値は独自のグループとして扱われ、NULLS FIRST または NULLS LAST オプションに従ってソートおよびランク付けされます。デフォルトでは、NULL 値は昇順ではソートされて最後にランク付けされ、降順ではソートされて最初にランク付けされます。

AWS Clean Roomsは、ORDER BY 句の文字列リテラルをサポートしていません。

ORDER BY 句を省略した場合、行の順序は不確定になります。

> **Note**
>
> ORDER BY 句がデータの一意の合計順序を生成しない場合などAWS Clean Rooms、並列システムでは、行の順序は不確定です。つまり、ORDER BY 式が重複した値を生成する場合 (部分的な順序付け) 、それらの行の戻り順序は の実行ごとに異なる場合がありますAWS Clean Rooms。そのため、ウィンドウ関数は予期しない結果または矛盾した結果を返す場合があります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

column_name

パーティション化または順序付けされる列の名前。

ASC | DESC

次のように、式のソート順を定義するオプション:

- ASC: 昇順 (数値の場合は低から高、文字列の場合は「A」から「Z」など) オプションを指定しない場合、データはデフォルトでは昇順にソートされます。
- DESC: 降順 (数値の場合は高から低、文字列の場合は「Z」から「A」)。

NULLS FIRST | NULLS LAST

NULL を NULL 以外の値より先に順序付けするか、NULL 以外の値の後に順序付けするかを指定するオプション。デフォルトでは、NULL は昇順ではソートされて最後にランク付けされ、降順ではソートされて最初にランク付けされます。

frame_clause

集計関数では、ORDER BY を使用する場合、フレーム句は関数のウィンドウで行のセットをさらに絞り込みます。これは、順序付けされた結果内の行のセットを含めるか、または除外できるようにします。フレーム句は ROWS キーワードおよび関連する指定子で構成されます。

frame 句は、ランク付け関数には適用されません。また、集計関数の OVER 句の中に ORDER BY 句がない場合は、フレーム句は必要ありません。ORDER BY 句が集計関数に使用される場合、明示的なフレーム句が必要です。

ORDER BY 句が指定されていない場合、暗黙的なフレームはバインドされません (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING と同じ)。

ROWS

この句は、現在の行からの物理オフセットを指定してウィンドウフレームを定義します。

この句は、現在のウィンドウの行、または現在の行の値を組み合わせるパーティションを指定します。また、現在の行の前後に配置される行の位置を指定する引数を使用します。すべてのウィンドウフレームの参照点は現在の行です。ウィンドウフレームがパーティションで次にスライドすると、各行は順に現在の行になります。

フレームは、次のように現在の行までと現在の行を含む行の簡易セットである場合があります。

```
{UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW}
```

また、次の 2 つの境界の間の行のセットである場合があります。

```
BETWEEN
{ UNBOUNDED PRECEDING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
AND
{ UNBOUNDED FOLLOWING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
```

UNBOUNDED PRECEDING はウィンドウがパーティションの最初の行で開始することを示し、offset PRECEDING はウィンドウが現在の行の前のオフセット値と等しい行数で開始することを示します。デフォルトは UNBOUNDED PRECEDING です。

CURRENT ROW は、ウィンドウが現在の行で開始または終了することを示します。

UNBOUNDED FOLLOWING はウィンドウがパーティションの最後の行で終了することを示し、offset FOLLOWING はウィンドウが現在の行の後のオフセット値と等しい行数で終了することを示します。

offset は、現在の行の前後にある物理的な行数を識別します。この場合、offset は、正の数値に評価される定数である必要があります。例えば、5 FOLLOWING は現在の行より 5 行後のフレームを終了します。

BETWEEN が指定されていない場合、フレームは現在の行により暗黙的に区切られます。例えば、ROWS 5 PRECEDING は ROWS BETWEEN 5 PRECEDING AND CURRENT ROW と同じです。また、ROWS UNBOUNDED FOLLOWINGは ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING と同じです。

> **Note**
>
> 開始境界が終了境界よりも大きいフレームを指定することはできません。例えば、以下のフレームはいずれも指定することができません。

```
between 5 following and 5 preceding
between current row and 2 preceding
between 3 following and current row
```

ウィンドウ関数用データの一意の並び順

ウィンドウ関数の ORDER BY 句がデータの一意および全体の並び順を生成しない場合、行の順序は不確定になります。ORDER BY 式が重複した値 (部分的な順序付け) を生成する場合、これらの行の戻り値の順序は実行時によって異なる可能性があります。この場合、ウィンドウ関数は予期しない結果または矛盾した結果を返す場合があります。

例えば、次のクエリは、複数の実行にわたって異なる結果を返します。これらの異なる結果は、`order by dateid` が SUM Window 関数でデータの一意の順序を生成しないために発生します。

```
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
...
```

この場合、2 番目の ORDER BY 列をウィンドウ関数に追加すると問題を解決できます。

```
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
...
```

サポートされている関数

AWS Clean RoomsSpark SQL は、集計とランキングの 2 種類のウィンドウ関数をサポートしています。

サポートされる集約関数は次のとおりです。

- CUME_DIST ウィンドウ関数
- DENSE_RANK ウィンドウ関数
- 最初のウィンドウ関数
- FIRST_VALUE ウィンドウ関数
- LAG ウィンドウ関数
- LAST ウィンドウ関数
- LAST_VALUE ウィンドウ関数
- LEAD ウィンドウ関数

サポートされる集計関数は次のとおりです。

- DENSE_RANK ウィンドウ関数
- PERCENT_RANK ウィンドウ関数
- RANK ウィンドウ関数
- ROW_NUMBER ウィンドウ関数

ウィンドウ関数例のサンプルテーブル

ウィンドウ関数別の例をそれぞれの説明と共に参照できます。一部の例で使用している WINSALES という名前のテーブルには、以下のテーブルのように 11 行が含まれています。

SALESID	DATEID	SELLERID	BUYERID	QTY	QTY_SHIPPED
30001	8/2/2003	3	B	10	10
10001	12/24/2003	1	C	10	10
10005	12/24/2003	1	A	30	
40001	1/9/2004	4	A	40	
10006	1/18/2004	1	C	10	
20001	2/12/2004	2	B	20	20
40005	2/12/2004	4	A	10	10
20002	2/16/2004	2	C	20	20
30003	4/18/2004	3	B	15	
30004	4/18/2004	3	B	20	
30007	9/7/2004	3	C	30	

CUME_DIST ウィンドウ関数

ウィンドウまたはパーティション内の値の累積分布を計算します。昇順の場合、累積分布は以下の式を使用して特定されます。

```
count of rows with values <= x / count of rows in the window or partition
```

ここで、x は ORDER BY 句で指定された列の現在の行の値と等しくなります。以下のデータセットは、この式の使用方法を示しています。

```
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
```

戻り値の範囲は、0 ~ 1 (0 は含みませんが 1 は含みます) です。

構文

```
CUME_DIST ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

引数

OVER

ウィンドウのパーティションを指定する句。OVER 句にウィンドウフレーム仕様を含めることはできません。

PARTITION BY partition_expression

省略可能。OVER 句の各グループのレコードの範囲を設定する式。

ORDER BY order_list

累積分布を計算する式。式は、数値データ型を含んでいるか、そのデータ型に暗黙的に変換できる必要があります。ORDER BY を省略した場合、すべての行について戻り値は 1 です。

ORDER BY で一意の並べ替えが行われない場合、行の順序は不確定になります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

戻り型

FLOAT8

例

次の例は、各販売者の数量の累積配布を計算します。

```
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
```

WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

DENSE_RANK ウィンドウ関数

DENSE_RANK ウィンドウ関数は、OVER 句の ORDER BY 式に基づいて、値のグループの値のランクを特定します。オプションの PARTITION BY 句がある場合、ランク付けは行のグループごとにリセットされます。ランク付け条件が同じ値の行は、同じランクを受け取ります。DENSE_RANK 関数はある点において RANK とは異なります。2 行以上で同点となった場合、ランク付けされた値の順位に差はありません。例えば、2 行が 1 位にランク付けされると、次のランクは 2 位になります。

同じクエリに PARTITION BY および ORDER BY 句のあるランク付け関数を使用することができます。

構文

```
DENSE_RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

引数

( )

この関数は引数を受け取りませんが、空のかっこは必要です。

OVER

DENSE_RANK 関数のウィンドウ句。

PARTITION BY expr_list

省略可能。ウィンドウを定義する 1 つ以上の式。

ORDER BY order_list

省略可能。ランク付けの値が基とする式。PARTITION BY が指定されていない場合、ORDER BY はテーブル全体を使用します。ORDER BY を省略した場合、すべての行について戻り値は 1 です。

ORDER BY で一意の並べ替えが行われない場合、行の順序は不確定になります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

戻り型

INTEGER

例

次の例では、販売数量によってテーブルを順序付けして (降順)、各行にデンス値のランクと標準のランクの両方を割り当てます。結果はウィンドウ関数の結果が提供された後にソートされます。

```
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)
```

DENSE_RANK および RANK 関数が同じクエリで並べて使用される場合、同じ行のセットに割り当てるランク付けの違いに注意してください。WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

次の例では、SELLERID によってテーブルをパーティション分割し、数量によって各パーティションを順序付けして (降順)、行ごとにデンス値のランクを割り当てます。結果はウィンドウ関数の結果が提供された後にソートされます。

```
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)
```

WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

最初のウィンドウ関数

順序付けられた行のセットがある場合、FIRST はウィンドウフレームの最初の行に関して指定された式の値を返します。

フレームの最後の行を選択する方法については、「LAST ウィンドウ関数」を参照してください。

構文

```
FIRST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

引数

expression

関数の対象となる列または式。

IGNORE NULLS

このオプションを FIRST で使用すると、関数は NULL (すべての値が NULL の場合は NULL) ではないフレームの最初の値を返します。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

関数にウィンドウ句を導入します。

PARTITION BY expr_list

1 つ以上の式で関数のウィンドウを定義します。

ORDER BY order_list

各パーティション内の行をソートします。PARTITION BY 句が指定されていない場合、ORDER BY はテーブル全体をソートします。ORDER BY 句を指定する場合、frame_clause も指定する必要があります。

FIRST 関数の結果は、データの順序によって異なります。以下の場合、結果は不確定になります。

- ORDER BY 句が指定されておらず、パーティションに式に使用する 2 つの異なる値が含まれる場合
- 式が ORDER BY リストの同じ値に対応する異なる値を検証する場合。

frame_clause

ORDER BY 句が集計関数に使用される場合、明示的なフレーム句が必要です。フレーム句は順序付けた結果の行のセットを含めるか除外して、関数のウィンドウの行のセットを絞り込みます。フレーム句は ROWS キーワードおよび関連する指定子で構成されます。「ウィンドウ関数の構文の概要」を参照してください。

戻り型

これらの関数は、プリミティブAWS Clean Roomsデータ型を使用する式をサポートします。戻り値の型は式のデータ型と同じです。

例

次の例は、収容能力によって順序付けられた結果 (高から低) で、VENUE テーブルの各会場の座席数を返します。FIRST 関数は、フレームの最初の行に対応する会場の名前を選択するために使用されます。この場合は、座席数が最も多い行です。結果は州によってパーティションで分割されるため、VENUESTATE 値が変更されると、新しい最初の値が選択されます。ウィンドウフレームはバインドされていないため、同じ最初の値が各パーティションの行ごとに選択されます。

カリフォルニアでは、Qualcomm Stadiumが最高座席数 (70561) であるため、この名前は CA パーティションのすべての行に対する最初の値です。

```
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 ウィンドウ関数

順序付けられた行のセットとすると、FIRST_VALUE はウィンドウフレームの最初の行に関して指定された式の値を返します。

フレームの最後の行を選択する方法については、「LAST_VALUE ウィンドウ関数」を参照してください。

構文

```
FIRST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

引数

expression

関数の対象となる列または式。

IGNORE NULLS

このオプションが FIRST_VALUE で使用される場合、関数は NULL ではないフレームの最初の値 (値がすべて NULL の場合は NULL) を返します。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

関数にウィンドウ句を導入します。

PARTITION BY expr_list

1 つ以上の式で関数のウィンドウを定義します。

ORDER BY order_list

各パーティション内の行をソートします。PARTITION BY 句が指定されていない場合、ORDER BY はテーブル全体をソートします。ORDER BY 句を指定する場合、frame_clause も指定する必要があります。

FIRST_VALUE 関数の結果は、データの並び順によって異なります。以下の場合、結果は不確定になります。

- ORDER BY 句が指定されておらず、パーティションに式に使用する 2 つの異なる値が含まれる場合
- 式が ORDER BY リストの同じ値に対応する異なる値を検証する場合。

frame_clause

ORDER BY 句が集計関数に使用される場合、明示的なフレーム句が必要です。フレーム句は順序付けた結果の行のセットを含めるか除外して、関数のウィンドウの行のセットを絞り込みます。フレーム句は ROWS キーワードおよび関連する指定子で構成されます。「ウィンドウ関数の構文の概要」を参照してください。

戻り型

これらの関数は、プリミティブAWS Clean Roomsデータ型を使用する式をサポートします。戻り値の型は式のデータ型と同じです。

例

次の例は、収容能力によって順序付けられた結果 (高から低) で、VENUE テーブルの各会場の座席数を返します。FIRST_VALUE 関数は、フレームの最初の行 (この場合、最高座席数の行) に対応する会場名を選択するために使用されます。結果は州によってパーティションで分割されるた

め、VENUESTATE 値が変更されると、新しい最初の値が選択されます。ウィンドウフレームはバインドされていないため、同じ最初の値が各パーティションの行ごとに選択されます。

カリフォルニアでは、Qualcomm Stadiumが最高座席数 (70561) であるため、この名前は CA パーティションのすべての行に対する最初の値です。

```
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 ウィンドウ関数

LAG ウィンドウ関数は、パーティションの現在の行より上 (前) の指定されたオフセットの行の値を返します。

構文

```
LAG (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
```

```
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

引数

value_expr

関数の対象となる列または式。

offset

値を返す現在の行より前の行数を指定するオプションのパラメータ。オフセットは整数の定数、または整数を検証する式にすることができます。オフセットを指定しない場合、 AWS Clean Roomsをデフォルト値1として使用します。0 のオフセットは現在の行を示します。

IGNORE NULLS

が使用する行を決定する際に null 値をAWS Clean Roomsスキップする必要があることを示すオプションの仕様。IGNORE NULLS がリストされていない場合、Null 値が含まれます。

> **Note**
>
> NVL または COALESCE 式を使用し、Null 値を別の値で置換できます。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

ウィンドウのパーティションおよび並び順を指定します。OVER 句にウィンドウフレーム仕様を含めることはできません。

PARTITION BY window_partition

OVER 句の各グループのレコードの範囲を設定するオプションの引数。

ORDER BY window_ordering

各パーティション内の行をソートします。

LAG ウィンドウ関数は、任意のAWS Clean Roomsデータ型を使用する式をサポートします。戻り値の型は value_expr の型と同じです。

例

次の例は、購入者 ID 3 の購入者に販売されたチケット数と購入者 3 がチケットを購入した時刻を示します。購入者 3 の以前の販売と各販売を比較するには、クエリは販売ごとに以前の販売数を返します。2008 年 1 月 16 日より前に購入されていないため、最初の以前の販売数は Null です。

```
select buyerid, saletime, qtysold,
lag(qtysold,1) over (order by buyerid, saletime) as prev_qtysold
from sales where buyerid = 3 order by buyerid, saletime;

buyerid |      saletime       | qtysold | prev_qtysold
---------+---------------------+---------+--------------
3 | 2008-01-16 01:06:09 |       1 |
3 | 2008-01-28 02:10:01 |       1 |            1
3 | 2008-03-12 10:39:53 |       1 |            1
3 | 2008-03-13 02:56:07 |       1 |            1
3 | 2008-03-29 08:21:39 |       2 |            1
3 | 2008-04-27 02:39:01 |       1 |            2
3 | 2008-08-16 07:04:37 |       2 |            1
3 | 2008-08-22 11:45:26 |       2 |            2
3 | 2008-09-12 09:11:25 |       1 |            2
3 | 2008-10-01 06:22:37 |       1 |            1
3 | 2008-10-20 01:55:51 |       2 |            1
3 | 2008-10-28 01:30:40 |       1 |            2
(12 rows)
```

LAST ウィンドウ関数

順序付けられた行のセットがある場合、LAST 関数はフレーム内の最後の行に関する式の値を返します。

フレームの最初の行を選択する方法については、「最初のウィンドウ関数」を参照してください。

構文

```
LAST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

引数

expression

関数の対象となる列または式。

IGNORE NULLS

この関数は NULL ではないフレームの最後の値 (値がすべて NULL の場合は NULL) を返します。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

関数にウィンドウ句を導入します。

PARTITION BY expr_list

1 つ以上の式で関数のウィンドウを定義します。

ORDER BY order_list

各パーティション内の行をソートします。PARTITION BY 句が指定されていない場合、ORDER BY はテーブル全体をソートします。ORDER BY 句を指定する場合、frame_clause も指定する必要があります。

結果は、データの並び順によって異なります。以下の場合、結果は不確定になります。

- ORDER BY 句が指定されておらず、パーティションに式に使用する 2 つの異なる値が含まれる場合
- 式が ORDER BY リストの同じ値に対応する異なる値を検証する場合。

frame_clause

ORDER BY 句が集計関数に使用される場合、明示的なフレーム句が必要です。フレーム句は順序付けた結果の行のセットを含めるか除外して、関数のウィンドウの行のセットを絞り込みます。フレーム句は ROWS キーワードおよび関連する指定子で構成されます。「ウィンドウ関数の構文の概要」を参照してください。

戻り型

これらの関数は、プリミティブAWS Clean Roomsデータ型を使用する式をサポートします。戻り値の型は式のデータ型と同じです。

例

次の例は、収容能力によって順序付けられた結果 (高から低) で、VENUE テーブルの各会場の座席数を返します。LAST 関数は、フレームの最後の行に対応する会場の名前を選択するために使用されます。この場合は、座席数が最も少ない行です。結果は州によってパーティションで分割されるため、VENUESTATE 値が変更されると、新しい最後の値が選択されます。ウィンドウフレームはバインドされていないため、同じ最後の値が各パーティションの行ごとに選択されます。

カリフォルニアでは、Shoreline Amphitheatreの座席数が一番低い (22000) ため、この値がパーティションのすべての行に返されます。

```
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 ウィンドウ関数

順序付けられた行のセットを考慮し、LAST_VALUE 関数は、フレームの最後の行に関する式の値を返します。

フレームの最初の行を選択する方法については、「FIRST_VALUE ウィンドウ関数」を参照してください。

構文

```
LAST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

引数

expression

関数の対象となる列または式。

IGNORE NULLS

この関数は NULL ではないフレームの最後の値 (値がすべて NULL の場合は NULL) を返します。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

関数にウィンドウ句を導入します。

PARTITION BY expr_list

1 つ以上の式で関数のウィンドウを定義します。

ORDER BY order_list

各パーティション内の行をソートします。PARTITION BY 句が指定されていない場合、ORDER BY はテーブル全体をソートします。ORDER BY 句を指定する場合、frame_clause も指定する必要があります。

結果は、データの並び順によって異なります。以下の場合、結果は不確定になります。

- ORDER BY 句が指定されておらず、パーティションに式に使用する 2 つの異なる値が含まれる場合
- 式が ORDER BY リストの同じ値に対応する異なる値を検証する場合。

frame_clause

ORDER BY 句が集計関数に使用される場合、明示的なフレーム句が必要です。フレーム句は順序付けた結果の行のセットを含めるか除外して、関数のウィンドウの行のセットを絞り込みます。フレーム句は ROWS キーワードおよび関連する指定子で構成されます。「ウィンドウ関数の構文の概要」を参照してください。

戻り型

これらの関数は、プリミティブAWS Clean Roomsデータ型を使用する式をサポートします。戻り値の型は式のデータ型と同じです。

例

次の例は、収容能力によって順序付けられた結果 (高から低) で、VENUE テーブルの各会場の座席数を返します。LAST_VALUE 関数は、フレームの最後の行 (この場合、最小座席数の行) に対応する会場名を選択するために使用されます。結果は州によってパーティションで分割されるため、VENUESTATE 値が変更されると、新しい最後の値が選択されます。ウィンドウフレームはバインドされていないため、同じ最後の値が各パーティションの行ごとに選択されます。

カリフォルニアでは、Shoreline Amphitheatreの座席数が一番低い (22000) ため、この値がパーティションのすべての行に返されます。

```
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 ウィンドウ関数

LEAD ウィンドウ関数は、パーティションの現在の行より下 (後) の指定されたオフセットの行の値を返します。

構文

```
LEAD (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

引数

value_expr

関数の対象となる列または式。

offset

値を返す現在の行より下の行数を指定するオプションのパラメータ。オフセットは整数の定数、または整数を検証する式にすることができます。オフセットを指定しない場合、 AWS Clean Roomsをデフォルト値1として使用します。0 のオフセットは現在の行を示します。

IGNORE NULLS

が使用する行を決定する際に null 値をAWS Clean Roomsスキップする必要があることを示すオプションの仕様。IGNORE NULLS がリストされていない場合、Null 値が含まれます。

> **Note**
>
> NVL または COALESCE 式を使用し、Null 値を別の値で置換できます。

RESPECT NULLS

が使用する行の決定に null 値を含めるAWS Clean Rooms必要があることを示します。IGNORE NULLS を指定しない場合、RESPECT NULLS はデフォルトでサポートされます。

OVER

ウィンドウのパーティションおよび並び順を指定します。OVER 句にウィンドウフレーム仕様を含めることはできません。

PARTITION BY window_partition

OVER 句の各グループのレコードの範囲を設定するオプションの引数。

ORDER BY window_ordering

各パーティション内の行をソートします。

LEAD ウィンドウ関数は、任意のAWS Clean Roomsデータ型を使用する式をサポートしています。戻り値の型は value_expr の型と同じです。

例

次の例は、チケットが 2008 年 1 月 1 日および 2008 年 1 月 2 日に販売された SALES テーブルのイベントの手数料、および次の販売のチケット販売に支払った手数料を示します。

```
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 ウィンドウ関数

指定の行のパーセントランクを計算します。パーセントランクは、以下の式を使用して特定されます。

```
(x - 1) / (the number of rows in the window or partition - 1)
```

ここで、x は現在の行のランクです。以下のデータセットは、この式の使用方法を示しています。

```
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
```

戻り値の範囲は、0 ~ 1 (0 と 1 を含みます) です。どのセットも、最初の行の PERCENT_RANK は 0 になります。

構文

```
PERCENT_RANK ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

引数

( )

この関数は引数を受け取りませんが、空のかっこは必要です。

OVER

ウィンドウのパーティションを指定する句。OVER 句にウィンドウフレーム仕様を含めることはできません。

PARTITION BY partition_expression

省略可能。OVER 句の各グループのレコードの範囲を設定する式。

ORDER BY order_list

省略可能。パーセントランクを計算する式。式は、数値データ型を含んでいるか、そのデータ型に暗黙的に変換できる必要があります。ORDER BY を省略した場合、すべての行について戻り値は 0 です。

ORDER BY で一意のソートが行われない場合、行の順序は不確定になります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

戻り型

FLOAT8

例

以下の例では、各販売者の販売数量のパーセントランクを計算します。

```
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
```

WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

RANK ウィンドウ関数

RANK ウィンドウ関数は、OVER 句の ORDER BY 式に基づいて、値のグループの値のランクを決定します。オプションの PARTITION BY 句がある場合、ランク付けは行のグループごとにリセットされます。ランク付け条件の値が等しい行は同じランクを受け取ります。 は、結合された行の数を結合されたランクAWS Clean Roomsに追加して次のランクを計算するため、ランクは連続した数値ではない可能性があります。例えば、2 行が 1 位にランク付けされると、次のランクは 3 位になります。

RANK と DENSE_RANK ウィンドウ関数 では異なる点があり、DENSE_RANK では、2 行以上で同点となった場合、ランク付けされた値の順位に差はありません。例えば、2 行が 1 位にランク付けされると、次のランクは 2 位になります。

同じクエリに PARTITION BY および ORDER BY 句のあるランク付け関数を使用することができます。

構文

```
RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

引数

( )

この関数は引数を受け取りませんが、空のかっこは必要です。

OVER

RANK 関数のウィンドウ句。

PARTITION BY expr_list

省略可能。ウィンドウを定義する 1 つ以上の式。

ORDER BY order_list

省略可能。ランク付けの値が基とする列を定義します。PARTITION BY が指定されていない場合、ORDER BY はテーブル全体を使用します。ORDER BY を省略した場合、すべての行について戻り値は 1 です。

ORDER BY で一意のソートが行われない場合、行の順序は不確定になります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

戻り型

INTEGER

例

次の例では、販売数量でテーブルで順序付けして (デフォルトは昇順)、行ごとにランクを割り当てます。1 のランク値は、もっとも高いランクの値です。結果はウィンドウ関数の結果が提供された後にソートされます。

```
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)
```

この例の外部 ORDER BY 句には、このクエリが実行されるたびに が一貫してソートされた結果を AWS Clean Rooms返すように、列 2 と 1 が含まれていることに注意してください。例えば、販売 ID 10001 および 10006 の行は、QTY と RNK の値が同じです。列 1 によって最終的な結果セットを順序付けると、行 10001 は常に 10006 の前になります。WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

次の例では、ウィンドウ関数 (order by qty desc) の順序付けは逆順になります。これで、最高ランク値が最大の QTY 値に適用されます。

```
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)
```

WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

次の例では、SELLERID によってテーブルをパーティション分割し、数量で各パーティションを順序付けして (降順)、行ごとにランクを割り当てます。結果はウィンドウ関数の結果が提供された後にソートされます。

```
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 ウィンドウ関数

OVER 句の ORDER BY 式に基づいて、行グループ内における (1 からカウントした) 現在の行の序数が決まります。オプションの PARTITION BY 句がある場合、序数は行グループごとにリセットされます。ORDER BY 式で同じ値を持つ行には、確定的でない方法で異なる行番号が割り当てられます。

構文

```
ROW_NUMBER () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

引数

( )

この関数は引数を受け取りませんが、空のかっこは必要です。

OVER

ROW_NUMBER 関数のウィンドウ句。

PARTITION BY expr_list

省略可能。ROW_NUMBER 関数を定義する 1 つ以上の式。

ORDER BY order_list

省略可能。行番号の基になる列を定義する式。PARTITION BY が指定されていない場合、ORDER BY はテーブル全体を使用します。

ORDER BY で一意の順序付けが行われない、または省略した場合、行の順序は不確定になります。詳細については、「ウィンドウ関数用データの一意の並び順」を参照してください。

戻り型

BIGINT

例

次の例では、SELLERID によってテーブルをパーティション化し、QTY によって各パーティションを (昇順で) 順序付けし、各行に行番号を割り当てます。結果はウィンドウ関数の結果が提供された後にソートされます。

```
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)
```

WINSALES テーブルの説明については、「ウィンドウ関数例のサンプルテーブル」を参照してください。

AWS Clean Rooms Spark SQL 条件

条件とは、評価結果として true、false、または unknown を返す、1 つまたは複数の式および論理演算子から成るステートメントです。条件は述語と呼ばれることもあります。

[Syntax] (構文)

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
| IN_condition
```

> **Note**
>
> すべての文字列比較および LIKE パターンマッチングでは、大文字と小文字が区別されます。例えば、「A」と「a」は一致しません。ただし、ILIKE 述語を使用すれば、大文字小文字を区別しないパターンマッチングを行うことができます。

Spark SQL では、次の SQL AWS Clean Rooms 条件がサポートされています。

トピック

比較演算子

比較条件では、2 つの値の間の論理的な関係を指定します。比較条件はすべて、ブール型の戻り値を返す 2 項演算子です。

AWS Clean Rooms Spark SQL は、次の表で説明する比較演算子をサポートしています。

演算子	構文	説明
!	`!expression`	論理NOT演算子。ブール式を無効にするために使用されます。つまり、式の値の逆を返します。 ! 演算子を AND や OR などの他の論理演算子と組み合わせて、より複雑なブール式を作成することもできます。
<	`a < b`	less than 比較演算子。2 つの値を比較し、左側の値が右側の値よりも小さいかどうかを判断するために使用されます。
>	`a > b`	比較演算子より大きい。2 つの値を比較し、左側の値が右側の値より大きいかどうかを判断するために使用されます。
<=	`a <= b`	比較演算子以下。2 つの値を比較し、左側の値が右側の値以下の`true`場合は を返します。`false`それ以外の場合は を返します。
>=	`a >= b`	比較演算子以上の 。2 つの値を比較し、左側の値が右側の値以上かどうかを判断するために使用されます。

演算子	構文	説明
=	a = b	等価比較演算子。2 つの値を比較し、等しいtrue場合は を返します。falseそれ以外の場合は を返します。
<> または !=	a <> b、または a != b	比較演算子と等しくないは、2 つの値を比較し、等しくない場合は を返します。falseそれ以外の場合はtrue を返します。
==	a == b	標準等価比較演算子。2 つの値を比較し、等しいtrue場合は を返します。falseそれ以外の場合は を返します。 **Note** == 演算子は、文字列値を比較するときに大文字と小文字が区別されます。大文字と小文字を区別しない比較を実行する必要がある場合は、UPPER() や LOWER() などの関数を使用して、比較前に値を同じケースに変換できます。

例

ここで、比較条件の簡単な例をいくつか示します。

```
a = 5
```

```
a < b
min(x) >= 5
qtysold = any (select qtysold from sales where dateid = 1882
```

次のクエリは、現在フォージングされていないすべてのリスの ID 値を返します。

```
SELECT id FROM squirrels
WHERE !is_foraging
```

次のクエリは、VENUE テーブルから席数が 10,000 席を超える会場を返します。

```
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)
```

この例では、USERS テーブルからロックミュージックが好きなユーザー (USERID) を選択します。

```
select userid from users where likerock = 't' order by 1 limit 5;

userid
--------
3
5
6
13
16
(5 rows)
```

この例では、USERS テーブルから、ロックミュージックを好きかどうか不明であるユーザー (USERID) を選択します。

```
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
```

TIME 列の例

次のテーブルの TIME_TEST の例には、3 つの値が挿入された列 TIME_VAL (タイプ TIME) があります。

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

次の例では、各 timetz_val から時間を抽出します。

```
select time_val from time_test where time_val < '3:00';
   time_val
---------------
 00:00:00.5550
 00:58:00
```

次の例では、2 つの時刻リテラルを比較します。

```
select time '18:25:33.123456' = time '18:25:33.123456';
 ?column?
----------
 t
```

TIMETZ 列の例

次のテーブルの TIMETZ_TEST の例には、3 つの値が挿入された列 TIMETZ_VAL (タイプ TIMETZ) があります。

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

次の例では、3:00:00 UTC 未満の TIMETZ 値のみを選択します。値を UTC に変換した後に比較が行われます。

```
select timetz_val from timetz_test where timetz_val < '3:00:00 UTC';

   timetz_val
---------------
 00:00:00.5550+00
```

次の例では、2 つの TIMETZ リテラルを比較します。タイムゾーンは、比較のために無視されます。

```
select time '18:25:33.123456 PST' < time '19:25:33.123456 EST';

 ?column?
----------
 t
```

論理条件

論理条件は、2 つの条件の結果を結合して 1 つの結果を作成します。論理条件はすべて、ブール型の戻り値を返す 2 項演算子です。

構文

```
expression
{ AND | OR }
expression
NOT expression
```

論理条件では 3 値ブール論理を使用します。この場合、Null 値は unknown 関係を表現します。次の表で論理条件の結果について説明します。ここで、E1 と E2 は式を表します。

E1	E2	E1 AND E2	E1 OR E2	NOT E2
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
FALSE	TRUE	FALSE	TRUE	
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	
UNKNOWN	TRUE	UNKNOWN	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	

NOT 演算子は AND 演算子より先に評価され、AND 演算子は OR 演算子より先に評価されます。括弧が使用されている場合、評価のデフォルトの順序より優先されます。

例

次の例では、USERS テーブルから、ラスベガスもスポーツも好きであるユーザーの USERID および USERNAME を返します。

```
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)
```

次の例では、USERS テーブルから、ラスベガスが好き、スポーツが好き、または両方が好きのいずれかに該当するユーザーの USERID と USERNAME を返します。このクエリでは、前の例の出力と、ラスベガスのみ好き、またはスポーツのみ好きなユーザーとをすべて返します。

```
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)
```

次のクエリでは、OR 条件を囲む括弧を使用して、マクベスが上演されたニューヨークあるいはカリフォルニアの劇場を探します。

```
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
...
```

この例の括弧を削除すると、クエリの論理および結果が変更されます。

次の例では、NOT 演算子を使用します。

```
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
...
```

次の例では、NOT 条件の後に AND 条件を使用しています。

```
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)
```

パターンマッチング条件

パターンマッチング演算子は、条件式で指定されたパターンの文字列を検索し、一致が見つかったかどうかに応じて true または false を返します。 AWS Clean Rooms Spark SQL は、パターンマッチングに次のメソッドを使用します。

- LIKE 式

  LIKE 演算子は、列名などの文字列式を、ワイルドカード文字 % (パーセント) および _ (アンダースコア) を使用したパターンと比較します。LIKE パターンマッチングは常に文字列全体を網羅します。LIKE は大文字と小文字を区別する一致を実行します。

トピック

- LIKE
- RLIKE

LIKE

LIKE 演算子は、列名などの文字列式を、ワイルドカード文字 % (パーセント) および _ (アンダースコア) を使用したパターンと比較します。LIKE パターンマッチングは常に文字列全体を網羅します。文字列内の任意の場所にあるシーケンスをマッチングするには、パターンがパーセント符号で始まりパーセント符号で終了する必要があります。

LIKE では大文字と小文字が区別されます。

構文

```
expression [ NOT ] LIKE | pattern [ ESCAPE 'escape_char' ]
```

引数

expression

列名など、有効な UTF-8 文字式。

LIKE

LIKE は大文字小文字を区別するパターンマッチングを実行します。マルチバイト文字に対して大文字と小文字を区別しないパターンの一致を実行するには、LIKE 条件の pattern と pattern で LOWER 関数を使用します。

= や <> などの比較述語とは対照的に、LIKE 述語は末尾のスペースを暗黙的に無視しません。末尾のスペースを無視するには、RTRIM を使用するか、または CHAR 列を VARCHAR に明示的にキャストします。

~~ 演算子は LIKE と同等です。また、!~~演算子は NOT LIKE と同等です。

pattern

マッチングするパターンが含まれる有効な UTF-8 文字式。

escape_char

パターン内でメタ文字をエスケープする文字式。デフォルトは 2 個のバックスラッシュ (「\\」) です。

pattern にメタ文字が含まれていない場合、pattern は文字列そのものを表現するにすぎません。その場合、LIKE は等号演算子と同じ働きをします。

どちらの文字式も CHAR または VARCHAR のデータ型になることができます。文字式の型が異なる場合、 AWS Clean Rooms は pattern のデータ型を expression のデータ型に変換します。

LIKE では、次のパターンマッチングメタ文字をサポートしています。

演算子	説明
%	ゼロ個以上の任意の文字シーケンスをマッチングします。
_	任意の 1 文字をマッチングします。

例

次の例では、LIKE を使用したパターンマッチングの例を示します。

式	戻り値
`'abc' LIKE 'abc'`	True
`'abc' LIKE 'a%'`	True
`'abc' LIKE '_B_'`	False
`'abc' LIKE 'c%'`	False

次の例では、名前が「E」で始まるすべての市を見つけます。

```
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
...
```

次の例では、姓に「ten」が含まれるユーザーを見つけます。

```
select distinct lastname from users
where lastname like '%ten%' order by lastname;
lastname
-------------
Christensen
Wooten
...
```

次の例では、3 番目と 4 番目の文字が「ea」である都市を検索します。

```
select distinct city from users where city like '__EA%' order by city;
city
-------------
Brea
Clearwater
Great Falls
Ocean City
Olean
Wheaton
(6 rows)
```

次の例では、デフォルトのエスケープ文字列 (\\) を使用して「start_」を含む文字列を検索します (テキスト start、それに続いてアンダースコア _)。

```
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)
```

次の例では、エスケープ文字として '^' を指定し、そのエスケープ文字を使用して「start_」を含む文字列を検索します (テキスト start、それに続いてアンダースコア _)。

```
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 演算子を使用すると、文字列が指定された正規表現パターンと一致するかどうかを確認できます。

str `true`が と一致する場合は `regexp`を返します。`false`それ以外の場合は を返します。

構文

```
rlike(str, regexp)
```

引数

str

文字列式

regexp

文字列式。正規表現文字列は Java 正規表現である必要があります。

文字列リテラル (正規表現パターンを含む) は SQL パーサーでエスケープされません。たとえば、「\abc」と一致させるには、regexp の正規表現を「^\abc$」にすることができます。

例

次の例では、`spark.sql.parser.escapedStringLiterals`設定パラメータの値を
に設定します`true`。このパラメータは Spark SQL エンジンに固有です。Spark SQL の
`spark.sql.parser.escapedStringLiterals`パラメータは、SQL パーサーがエスケープされた文字列リテラルを処理する方法を制御します。に設定すると`true`、パーサーは文字列リテラル内のバックスラッシュ文字 (\) をエスケープ文字として解釈し、文字列値に改行、タブ、引用符などの特殊文字を含めることができます。

```
SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
```

たとえば、 でspark.sql.parser.escapedStringLiterals=true、SQL クエリで次の文字列リテラルを使用できます。

```
SELECT 'Hello, world!\n'
```

改行文字は\n、出力でリテラル改行文字として解釈されます。

次の の例では、正規表現パターンの一致を実行します。最初の引数は RLIKE 演算子に渡されます。これはファイルパスを表す文字列で、実際のユーザー名はパターン「****」に置き換えられます。2 番目の引数は、マッチングに使用される正規表現パターンです。出力 (true) は、最初の文字列 () が正規表現パターン ('%SystemDrive%\Users****') と一致することを示します'%SystemDrive%\\Users.*'。

```
SELECT rlike('%SystemDrive%\Users\John', '%SystemDrive%\Users.*');
true
```

BETWEEN 範囲条件

BETWEEN 条件では、キーワード BETWEEN および AND を使用して、値が範囲内に入っているかどうか式をテストします。

構文

```
expression [ NOT ] BETWEEN expression AND expression
```

式は、数値データ型、文字データ型、または日時データ型とすることができますが、互換性を持つ必要があります。範囲は両端を含みます。

例

最初の例では、2、3、または 4 のいずれかのチケットの販売を登録したトランザクション数をカウントします。

```
select count(*) from sales
where qtysold between 2 and 4;

count
--------
104021
```

```
(1 row)
```

範囲条件は開始値と終了値を含みます。

```
select min(dateid), max(dateid) from sales
where dateid between 1900 and 1910;

min  | max
-----+-----
1900 | 1910
```

範囲条件内の最初の式は最小値、2 番目の式は最大値である必要があります。次の例は、式の値のせいで、常にゼロ行を返します。

```
select count(*) from sales
where qtysold between 4 and 2;

count
-------
0
(1 row)
```

しかし、NOT 修飾子を加えると、論理が反転し、すべての行がカウントされます。

```
select count(*) from sales
where qtysold not between 4 and 2;

count
--------
172456
(1 row)
```

次のクエリは、20000 ~ 50000 席を備えた会場のリストを返します。

```
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)
```

次の例は、日付値に BETWEEN を使用する方法を示しています。

```
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
```

BETWEEN の範囲は包括的ですが、日付はデフォルトで 00:00:00 の時刻値であることに注意してください。サンプルクエリで有効な 1 月 3 日の行は、販売時間が 1/3/2008 00:00:00 の行だけです。

Null 条件

NULL 条件は、値が見つからないか、値が不明であるときに、Null かどうかテストします。

構文

```
expression IS [ NOT ] NULL
```

引数

expression

列のような任意の式。

IS NULL

式の値が Null の場合は true で、式が値を持つ場合は false です。

IS NOT NULL

式の値が Null の場合は false で、式が値を持つ場合は true です。

例

この例では、SALES テーブルの QTYSOLD フィールドで Null が何回検出されるかを示します。

```
select count(*) from sales
where qtysold is null;
count
-------
0
(1 row)
```

EXISTS 条件

EXISTS 条件は、サブクエリ内に行が存在するかどうかをテストし、サブクエリが少なくとも 1 つの行を返した場合に true を返します。NOT が指定されると、条件はサブクエリが行を返さなかった場合に true を返します。

構文

```
[ NOT ] EXISTS (table_subquery)
```

引数

EXISTS

table_subquery が少なくとも 1 つの行を返した場合に true となります。

NOT EXISTS

table_subquery が行を返さない場合に true になります。

table_subquery

評価結果として 1 つまたは複数の列と 1 つまたは複数の行を持つテーブルを返します。

例

この例では、任意の種類の販売があった日付ごとに、1 回ずつ、すべての日付識別子を返します。

```
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 条件は、一連の値の中に、またはサブクエリ内にあるメンバーシップの値をテストします。

構文

```
expression [ NOT ] IN (expr_list | table_subquery)
```

引数

expression

expr_list または table_subquery に対して評価される数値、文字、または日時であり、当該リストまたはサブクエリのデータ型との互換性が必要です。

expr_list

1 つまたは複数のカンマ区切り式、あるいは括弧で囲まれたカンマ区切り式の 1 つまたは複数のセット。

table_subquery

評価結果として 1 つまたは複数の行を持つテーブルを返すサブクエリですが、その選択リスト内の列数は 1 個に制限されています。

IN | NOT IN

IN は、式が式リストまたはクエリのメンバーである場合に true を返します。NOT IN は、式がメンバーでない場合に true を返します。expression の結果が Null である場合、または、一致する expr_list 値または table_subquery 値がなく、これらの比較行の 1 つ以上の結果が Null である場合、IN と NOT IN は NULL を返し、行は返されません。

例

次の条件は、リストされた値の場合にのみ true を返します。

```
qtysold in (2, 4, 5)
date.day in ('Mon', 'Tues')
date.month not in ('Oct', 'Nov', 'Dec')
```

大規模 IN リストの最適化

クエリのパフォーマンスを最適化するために、10 個を超える値が含まれる IN リストは内部的にスカラー配列として評価されます。10 個未満の値が含まれる IN リストは一連の OR 述語として評価されます。SMALLINT、INTEGER、BIGINT、REAL、DOUBLE PRECISION、BOOLEAN、CHAR、VARCHAR、DATE、TIMESTAMP、および TIMESTAMPTZ データ型では最適化がサポートされています。

この最適化の効果を確認するには、クエリの EXPLAIN 出力を調べてください。次に例を示します。

```
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)
```

ネストされたデータのクエリ

AWS Clean Rooms は、リレーショナルデータとネストされたデータへの SQL 互換アクセスを提供します。

AWS Clean Rooms は、ネストされたデータにアクセスするときに、パスナビゲーションにドット表記と配列サブスクリプトを使用します。また、FROM 句の項目で配列を反復処理し、ネスト解除の操作に使用することもできます。以下のトピックでは、パスおよび配列のナビゲーション、ネスト解除、または結合を、配列/構造体/マップデータ型で行う場合の、さまざまなクエリパターンについて説明します。

トピック

ナビゲーション

AWS Clean Rooms では、それぞれ[...]角括弧表記とドット表記を使用して、配列と構造へのナビゲーションが可能になります。さらに、ドット表記を使用して構造体に、角括弧表記を使用して配列にナビゲーションを混在させることができます。

Example

例えば、次のクエリ例は、c_orders 配列データ列が構造体を持つ配列であり、属性の名前が o_orderkey であると仮定しています。

```
SELECT cust.c_orders[0].o_orderkey FROM customer_orders_lineitem AS cust;
```

フィルタリング、結合、集約など、すべてのタイプのクエリでドットと角括弧の表記を使用できます。この表記は、通常の場合に列参照を含んでいるクエリで使用します。

Example

次の例では、結果をフィルタリングする SELECT ステートメントを使用します。

```
SELECT count(*) FROM customer_orders_lineitem WHERE c_orders[0].o_orderkey IS NOT NULL;
```

Example

次の例では、GROUP BY 句と ORDER BY 句の両方で角括弧とドットのナビゲーションを使用します。

```
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;
```

ネストされていないクエリ

クエリのネストを解除するために、 は配列の反復 AWS Clean Rooms を有効にします。このために、クエリの FROM 句を使用して配列上をナビゲートします。

Example

前の例を使用して、次の例では、属性値 `c_orders` を繰り返し処理しています。

```
SELECT o FROM customer_orders_lineitem c, c.c_orders o;
```

ネスト解除構文は、FROM 句の拡張です。標準 SQL では、FROM 句 x（AS）y は x と関連する各タプルを y が反復処理することを意味します。この場合、x は関連を指し、y はその関連 x のための エイリアスを指します。同様に、FROM 句の項目 x（AS）y を使用してネスト解除する構文では、y が配列式 x 内の各値を反復処理することを意味します。この場合、x は配列式であり、y は x のエイリアスです。

左のオペランドは、通常のナビゲーションのためにドットと角括弧の表記を使用することもできます。

Example

前の例で見てみましょう。

- customer_orders_lineitem c は customer_order_lineitem ベーステーブルの反復処理
- c.c_orders o は c.c_orders array の反復処理

配列内の配列である o_lineitems 属性を反復処理するには、複数の句を追加する必要があります。

```
SELECT o, l FROM customer_orders_lineitem c, c.c_orders o, o.o_lineitems l;
```

AWS Clean Rooms は、 ATキーワードを使用して配列を反復処理するときに配列インデックスもサポートします。句 x AS y AT z は、配列 x を反復処理し、配列インデックスとしてフィールド z を生成します。

Example

次の例は、配列インデックスがどのように機能するかを示しています。

```
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

次の例では、スカラー配列の繰り返し処理を行います。

```
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

次の例では、複数のレベルの配列を繰り返し処理します。この例では、複数の UNNEST 句を使用して、最も内側の配列を反復処理します。f.multi_level_array AS array は multi_level_array を反復処理します。array AS element は、multi_level_array 内の配列に対する反復処理を表します。

```
CREATE TABLE foo AS SELECT json_parse('[[1.1, 1.2], [2.1, 2.2], [3.1, 3.2]]') AS
 multi_level_array;

SELECT array, element FROM foo AS f, f.multi_level_array AS array, array AS element;

   array   | element
-----------+---------
 [1.1,1.2] | 1.1
 [1.1,1.2] | 1.2
 [2.1,2.2] | 2.1
 [2.1,2.2] | 2.2
 [3.1,3.2] | 3.1
 [3.1,3.2] | 3.2
(6 rows)
```

Lax のセマンティクス

デフォルトでは、ネストされたデータ値に対するナビゲーションオペレーションでナビゲーションが無効である場合、エラーを返す代わりに null を返します。ネストされたデータ値がオブジェクトでない場合、またはネストされたデータ値がオブジェクトであるが、クエリで使用される属性名が含まれていない場合、オブジェクトのナビゲーションは無効です。

Example

例えば、次のクエリは、ネストされたデータ列の c_orders で無効な属性名にアクセスします。

```
SELECT c.c_orders.something FROM customer_orders_lineitem c;
```

ネストされたデータ値が配列でない場合、または配列インデックスが範囲外の場合、配列ナビゲーションは null を返します。

Example

次のクエリは、c_orders[1][1] が範囲外であるため、null を返します。

```
SELECT c.c_orders[1][1] FROM customer_orders_lineitem c;
```

内観の種類

ネストされたデータ型の列は、値に関する型およびその他の型情報を返す検査関数をサポートします。AWS Clean Rooms は、ネストされたデータ列に対して次のブール関数をサポートしています。

- 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

入力値が null の場合、これらの関数はすべて false を返します。IS_SCALAR、IS_OBJECT、および IS_ARRAY は相互に排他的であり、null を除くすべての可能な値をカバーします。データに対応する型を推測するために、 は、次の例に示すように、ネストされたデータ値の型 (最上位レベル) を返す JSON_TYPEOF 関数 AWS Clean Rooms を使用します。

```
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 リファレンスのドキュメント履歴

次の表に、 AWS Clean Rooms SQL リファレンスのドキュメントリリースを示します。

このドキュメントの更新に関する通知については、RSS フィードにサブスクライブできます。RSS の更新をサブスクリプションするには、使用しているブラウザで RSS プラグインを有効にする必要があります。

変更	説明	日付
Spark SQL がヒントをサポート	AWS Clean Rooms Spark SQL は、クエリのパフォーマンスを最適化し、コンピューティングコストを削減するためのクエリヒントをサポートします。	2026 年 1 月 20 日
Spark SQL が CACHE TABLE をサポート	AWS Clean Rooms Spark SQL は CACHE TABLE コマンドをサポートしています。これにより、既存のテーブルをキャッシュしたり、クエリ結果から新しいテーブルを作成およびキャッシュして、クエリのパフォーマンスを向上させることができます。	2025 年 10 月 22 日
Spark SQL が FIRST および LAST ウィンドウ関数をサポート	AWS Clean Rooms Spark SQL は、FIRST 関数と LAST 関数をサポートしています。	2025 年 6 月 12 日
Spark SQL 関数のドキュメントの更新	サポートされている Spark SQL 関数を正確に反映するためのドキュメントのみの更新。<=> 演算子、SIMILAR TO、LISTAGG、ARRAY_INSERT など、サポートされ	2025 年 5 月 20 日

	ていない 25 個の関数に関するドキュメントを削除しました。関数名を DATEADD から DATE_ADD、DATEDIFF から DATE_DIFF、ISNULL から IS_NULL、ISNOTNULL から IS_NOT_NULL に変更しました。DATE_PART の例の誤字を修正しました。	
AWS Clean Rooms Spark SQL	お客様は、Spark SQL 分析エンジンでサポートされているいくつかの SQL 条件、関数、コマンド、および規則を使用してクエリを実行できるようになりました。	2024 年 10 月 29 日
SQL コマンドと SQL 関数 – 更新	JOIN 句、EXCEPT 集合演算子、CASE 条件式、および ANY_VALUE、NVL と COALESCE、NULLIF、CAST、CONVERT、CONVERT_TIMEZONE、EXTRACT、MOD、SIGN、CONCAT、FIRST_VALUE、および LAST_VALUE の各関数の例が追加されました。	2024 年 2 月 1 日

SQL 関数 - 更新	AWS Clean Rooms は、配列、SUPER、VARBYTE の SQL 関数をサポートするようになりました。ACOS、ASIN、ATAN、ATAN2、COT、DEXP、PI、POW、RADIANS、SIN の算術関数がサポートされるようになりました。CAN_JSON_PARSE、JSON_PARSE、および JSON_SERIALIZE の JSON 関数がサポートされるようになりました。	2023 年 10 月 6 日
ネストされたデータ型のサポート	AWS Clean Rooms がネストされたデータ型をサポートするようになりました。	2023 年 8 月 30 日
SQL の命名規則 - 更新	予約済みの列名を明確にするための、ドキュメントのみの変更です。	2023 年 8 月 16 日
一般提供	AWS Clean Rooms SQL リファレンスが一般公開されました。	2023 年 7 月 31 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。