SQL 참조

AWS Clean Rooms

AWS Clean Rooms: SQL 참조

Table of Contents

개요 1
규칙 1
이름 지정 규칙 2
구성된 테이블 연결 이름 및 열 2
예약어 3
SQL 엔진의 데이터 유형 지원 5
숫자 데이터 유형 6
부울 데이터 형식 8
날짜 및 시간 데이터 유형 8
문자 데이터 유형 9
구조화된 데이터 형식 10
AWS Clean Rooms Spark SQL 13
리터럴 13
+(연결) 연산자 14
데이터 타입 15
멀티바이트 문자 17
숫자형 17
문자 형식 24
날짜/시간 형식 26
부울 유형 43
바이너리 유형 46
중첩된 유형 46
형식 호환성 및 변환 48
SQL 명령 53
캐시 테이블 53
힌트 56
SELECT 63
SQL 함수 107
집계 함수 108
배열 함수 130
조건식 140
생성자 함수 153
데이터 형식 지정 함수 156
날짜 및 시간 함수 183

암호화 및 복호화 함수 211
해시 함수 215
하이퍼로그 함수 218
JSON 함수 225
수학 함수 229
스칼라 함수 260
문자열 함수 261
개인 정보 보호 관련 함수 305
윈도우 함수 310
SQL 조건 341
비교 연산자 342
논리 조건 347
패턴 일치 조건 351
BETWEEN 범위 조건 356
NULL 조건 358
EXISTS 조건 359
IN 조건 360
중첩된 데이터 쿼리 362
탐색 362
쿼리 중첩 해제 363
Lax 의미 체계 365
내부 검사 유형 365
문서 이력 367
..... ccclxx

의 SQL 개요AWS Clean Rooms

AWS Clean Rooms SQL 참조에 오신 것을 환영합니다.

AWS Clean Rooms는 데이터베이스 및 데이터베이스 객체 작업에 사용하는 명령과 함수로 구성된 쿼리 언어인 업계 표준 구조화 쿼리 언어(SQL)를 기반으로 구축되었습니다. SQL은 또한 데이터 형식, 표현식 및 리터럴의 사용에 대한 규칙을 적용하기도 합니다.

다음 주제에서는이 SQL 참조에 사용되는 규칙 및 이름 지정 규칙에 대한 일반적인 정보를 제공합니다.

주제

- SQL 참조 규칙
- SQL 이름 지정 규칙
- SQL 엔진의 데이터 유형 지원

다음 섹션에서는 리터럴, 데이터 유형, SQL 명령, SQL 함수 유형 및 사용할 수 있는 SQL 조건에 대한 정보를 제공합니다AWS Clean Rooms.

- AWS Clean Rooms Spark SQL

에 대한 자세한 내용은 AWS Clean Rooms사용 설명서 및 AWS Clean RoomsAPI 참조를 AWS Clean Rooms참조하세요.

SQL 참조 규칙

이 섹션에서는 SQL 표현식, 명령어 및 함수의 구문 작성에 사용되는 규칙에 대해 설명합니다.

문자	설명
CAPS	대문자로 된 단어는 키워드입니다.
[ ]	대괄호는 선택적 인수를 나타냅니다. 다수의 인수가 대괄호로 묶이면 인수를 얼마든지 선택할 수 있다는 것을 의미합니다. 또한 별도의 라인에서 대괄호로 묶이는 인수는 구문 분석기에서 인수가 구문에 나열된 순서를 그대로 따른다는 것을 의미합니다.

문자	설명
{ }	중괄호는 중괄호 안의 인수 중 하나를 선택해야 함을 나타냅니다.
|	파이프는 인수 중에서 하나를 선택할 수 있다는 것을 의미합니다.
기울임꼴	기울임꼴 단어는 자리 표시자를 나타냅니다. 기울임꼴 단어 자리에 적절한 값을 넣어야 합니다.
...	줄임표는 앞의 요소를 반복할 수 있음을 나타냅니다.
'	작은따옴표로 묶인 단어는 따옴표를 입력해야 함을 나타냅니다.

SQL 이름 지정 규칙

다음 섹션에서는 AWS Clean Rooms의 SQL 명명 규칙을 설명합니다.

주제

- 구성된 테이블 연결 이름 및 열
- 예약어

구성된 테이블 연결 이름 및 열

쿼리가 가능한 구성원은 구성된 테이블 연결 이름을 쿼리의 테이블 이름으로 사용합니다. 구성된 테이블 연결 이름과 구성된 테이블 열을 쿼리에서 별칭으로 지정할 수 있습니다.

구성된 테이블 연결 이름, 구성된 테이블 열 이름 및 별칭에는 다음과 같은 이름 지정 규칙이 적용됩니다.

- 영숫자, 밑줄(_) 또는 하이픈(-) 문자만 사용해야 하지만 하이픈으로 시작하거나 끝날 수 없습니다.
  - (사용자 지정 분석 규칙만 해당) 달러 기호($)를 사용할 수 있지만 달러 따옴표 문자열 상수 뒤에 오는 패턴은 사용할 수 없습니다.

    달러 따옴표로 묶인 문자열 상수는 다음과 같이 구성됩니다.

    - 달러 기호($)
    - 0개 이상의 문자로 구성된 선택적 “태그”
    - 또 다른 달러 기호
    - 문자열 내용을 구성하는 임의의 문자 시퀀스
    - 달러 기호($)
    - 달러 인용의 시작과 동일한 태그
    - 달러 기호

    예: `$$invalid$$`
- 연속 하이픈(-) 문자는 포함할 수 없습니다.
- 다음 접두사로 시작할 수 없습니다.

  `padb_`, `pg_`, `stcs_`, `stl_`, `stll_`, `stv_`, `svcs_`, `svl_`, `svv_`, `sys_`, `systable_`
- 백슬래시 문자(\) , 따옴표(') 또는 큰따옴표가 없는 공백은 포함할 수 없습니다.
- 알파벳이 아닌 문자로 시작하는 경우 큰따옴표(” “) 안에 넣어야 합니다.
- 하이픈(-) 문자가 포함된 경우 큰따옴표(” “) 안에 넣어야 합니다.
- 길이는 1~127자여야 합니다.
- 예약어는 큰따옴표(” “) 안에 넣어야 합니다.
- 다음 열 이름은에서 사용할 수 없습니다 AWS Clean Rooms (따옴표 포함).
    - oid
    - 테이블 ID
    - xmin
    - cmin
    - xmax
    - cmax
    - ctid

예약어

다음은의 예약된 단어 목록입니다 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 엔진의 데이터 유형 지원

AWS Clean Rooms 는 여러 SQL 엔진과 언어를 지원합니다. 이러한 구현 전반의 데이터 유형 시스템을 이해하는 것은 성공적인 데이터 협업 및 분석에 매우 중요합니다. 다음 표에는 AWS Clean Rooms SQL, Snowflake SQL 및 Spark SQL에서 동일한 데이터 형식이 나와 있습니다.

숫자 데이터 유형

숫자 유형은 정확한 정수부터 대략적인 부동 소수점 값에 이르기까지 다양한 종류의 숫자를 나타냅니다. 숫자 유형 선택은 스토리지 요구 사항과 계산 정밀도에 모두 영향을 미칩니다. 정수 유형은 바이트 크기에 따라 다르지만 십진수 및 부동 소수점 유형은 정밀도 및 스케일 옵션이 다릅니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
8바이트 정수	BIGINT	지원되지 않음	BIGINT, LONG	-9,223,372,036,854,775,808에서 9,223,372,036,854,775,807까지의 부호 있는 정수입니다.
4바이트 정수	INT	지원되지 않음	INT, INTEGER	-2,147,483,648에서 2,147,483,647까지의 부호 있는 정수
2바이트 정수	SMALLINT	지원되지 않음	SMALLINT, SHORT	-32,768~32,767 범위의 부호 있는 정수
1바이트 정수	지원되지 않음	지원되지 않음	TINYINT, BYTE	-128~127 범위의 부호 있는 정수
Double Precision 부동 소수점	DOUBLE, DOUBLE PRECISION	FLOAT, FLOAT4, FLOAT8, DOUBLE, DOUBLE 정밀도, REAL	DOUBLE	8바이트 배정밀도 부동 소수점 숫자

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
단일 정밀도 부동 소수점	REAL, FLOAT	지원되지 않음	FLOAT	4바이트 단일 정밀도 부동 소수점 숫자
십진수(고정 정밀도)	DECIMAL	소수, 숫자, 숫자 **Note** Snowflake는 더 작은 너비의 정확한 숫자 유형(INT, BIGINT, SMALLINT 등)을 NUMBER로 자동으로 별칭을 지정합니다.	소수, 숫자,	임의 정밀도 부호 있는 십진수
십진수(정밀도 포함)	DECIMAL(p)	DECIMAL(p), NUMBER(p)	DECIMAL(p)	고정밀도 십진수
십진수(배율 포함)	DECIMAL (p,s)	DECIMAL(p,s), NUMBER(p,s)	DECIMAL (p,s)	눈금이 있는 고정밀도 십진수

부울 데이터 형식

부울 유형은 단순 true/false 논리 값을 나타냅니다. 이러한 유형은 SQL 엔진에서 일관되며 플래그, 조건 및 논리적 작업에 일반적으로 사용됩니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
불	BOOLEAN	BOOLEAN	BOOLEAN	true/false 값을 나타냅니다.

날짜 및 시간 데이터 유형

날짜 및 시간 유형은 다양한 수준의 정밀도와 시간대 인식으로 시간 데이터를 처리합니다. 이러한 유형은 날짜, 시간 및 타임스탬프를 저장하는 다양한 형식을 지원하며 시간대 정보를 포함하거나 제외하는 옵션을 제공합니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Date	날짜	날짜	날짜	시간대가 없는 날짜 값(년, 월, 일)
Time	TIME	지원되지 않음	지원되지 않음	시간대가 없는 UTC 기준 시간대
TZ 사용 시간	TIMETZ	지원되지 않음	지원되지 않음	시간대가 있는 UTC 기준 시간대
타임스탬프	TIMESTAMP	TIMESTAMP, TIMESTAMP_NTZ	TIMESTAMP_NTZ	표준 시간대가 없는 타임스탬프

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
				Note NTZ는 "시간대 없음"을 나타냅니다.
TZ를 사용하는 타임스탬프	TIMESTAMPTZ	TIMESTAMP_LTZ	TIMESTAMP, TIMESTAMP_LTZ	현지 시간대가 있는 타임스탬프 **Note** LTZ는 "로컬 시간대"를 나타냅니다.

문자 데이터 유형

문자 유형은 텍스트 데이터를 저장하여 고정 길이 옵션과 가변 길이 옵션을 모두 제공합니다. 이러한 유형은 텍스트 문자열과 이진 데이터를 처리하며, 스토리지 할당을 제어하는 선택적 길이 사양을 제공합니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
고정 길이 문자	CHAR	CHAR, CHARACTER	CHAR, CHARACTER	고정 길이 문자열

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
길이가 인 고정 길이 문자	CHAR(n)	CHAR(n), CHARACTER(n)	CHAR(n), CHARACTER(n)	길이가 지정된 고정 길이 문자열
가변 길이 문자	VARCHAR	VARCHAR, STRING, TEXT	VARCHAR, 문자열	가변 길이 문자열
길이가 인 가변 길이 문자	VARCHAR(n)	VARCHAR(n), STRING(n), TEXT(n)	VARCHAR(n)	길이 제한이 있는 가변 길이 문자열
바이너리	VARBYTE	BINARY, VARBINARY	BINARY	이진 바이트 시퀀스
길이가 인 이진수	VARBYTE(n)	지원되지 않음	지원되지 않음	길이 제한이 있는 이진 바이트 시퀀스

구조화된 데이터 형식

구조화된 유형을 사용하면 여러 값을 단일 필드로 결합하여 복잡한 데이터 구성을 수행할 수 있습니다. 여기에는 정렬된 컬렉션에 대한 배열, 키-값 페어에 대한 맵, 명명된 필드를 사용하여 사용자 지정 데이터 구조를 생성하기 위한 구조가 포함됩니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Array	ARRAY<type>	ARRAY(유형)	ARRAY<type>	동일한 유형의 정렬된 요소 시퀀스

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
				Note 배열 유형에는 동일한 유형의 요소가 포함되어야 합니다.
맵	MAP<key,value>	MAP(키,값)	MAP<key,value>	키-값 페어 모음 **Note** 맵 유형에는 동일한 유형의 요소가 포함되어야 합니다.

데이터 유형	AWS Clean Rooms SQL	Snowflake SQL	Spark SQL	Description
Struct	STRUCT< field1: type1, field2: type2>	OBJECT(field1 type1, field2 type2)	STRUCT< field1: type1, field2: type2 >	지정된 유형의 명명된 필드가 있는 구조 **Note** 구조화된 유형 구문은 구현마다 약간 다를 수 있습니다.
슈퍼	SUPER	지원되지 않음	지원되지 않음	복잡한 유형을 포함한 모든 데이터 유형을 지원하는 유연한 유형

AWS Clean Rooms Spark SQL

AWS Clean Rooms Spark SQL은 데이터 유형, 표현식 및 리터럴 사용에 관한 규칙을 적용합니다.

AWS Clean Rooms Spark SQL에 대한 자세한 내용은 AWS Clean Rooms 사용 설명서 및 AWS Clean Rooms API 참조를 참조하세요.

다음 주제에서는 AWS Clean Rooms Spark SQL에서 지원되는 리터럴, 데이터 유형, 명령, 함수 및 조건에 대한 정보를 제공합니다.

주제

- 리터럴
- 데이터 타입
- AWS Clean Rooms Spark SQL 명령
- AWS Clean Rooms Spark SQL 함수
- AWS Clean Rooms Spark SQL 조건

리터럴

리터럴 또는 상수는 연속된 문자 또는 숫자 상수로 구성된 데이터 고정 값입니다.

AWS Clean Rooms Spark SQL은 다음과 같은 여러 유형의 리터럴을 지원합니다.

- 숫자 리터럴(정수, 소수, 부동 소수점 수)
- 문자열, 문자열 또는 문자 상수라고도 하는 문자 리터럴은 문자열 값을 지정하는 데 사용됩니다.
- 날짜, 시간, 타임스탬프 리터럴(날짜/시간 데이터 형식과 함께 사용됨). 자세한 내용은 날짜, 시간 및 타임스탬프 리터럴 단원을 참조하십시오.
- 간격 리터럴. 자세한 내용은 간격 리터럴 단원을 참조하십시오.
- 부울 리터럴입니다. 자세한 내용은 부울 리터럴 단원을 참조하십시오.
- Null 리터럴 - null 값을 지정하는 데 사용됩니다.
- 유니코드 일반 범주(Cc)의 TAB, CARRIAGE RETURN(CR), LINE FEED(LF) 유니코드 제어 문자만 지원됩니다.

AWS Clean Rooms Spark SQL은 SELECT 절의 문자열 리터럴에 대한 직접 참조를 지원하지 않지만 CAST와 같은 함수 내에서 사용할 수 있습니다.

+(연결) 연산자

숫자 리터럴, 문자열 리터럴 및/또는 날짜/시간 및 간격 리터럴을 연결합니다. + 기호의 양쪽에 있으며 + 기호 양쪽의 입력값에 따라 다른 유형을 반환합니다.

구문

```
numeric + string
```

```
date + time
```

```
date + timetz
```

인수의 순서는 반대로 할 수 있습니다.

인수

numeric literals

숫자를 나타내는 리터럴이나 상수는 정수 또는 부동 소수점이 될 수 있습니다.

string literals

문자열, 문자열 또는 문자 상수

date

DATE 열 또는 묵시적으로 DATE로 변환되는 표현식입니다.

time

TIME 열 또는 묵시적으로 TIME으로 변환되는 표현식입니다.

timetz

TIMETZ 열 또는 묵시적으로 TIMETZ로 변환되는 표현식입니다.

예제

다음 예제 테이블 TIME_TEST에는 3개의 값이 삽입된 TIME_VAL(TIME 형식) 열이 있습니다.

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

데이터 타입

AWS Clean RoomsSpark SQL이 저장하거나 검색하는 각 값에는 고정된 연결 속성 집합이 있는 데이터 형식이 있습니다. 데이터 형식은 테이블을 생성할 때 선언됩니다. 열이나 인수에 포함될 수 있는 값도 이 데이터 형식에 따라 결정됩니다.

다음 표에는 AWS Clean RoomsSpark SQL에서 사용할 수 있는 데이터 유형이 나열되어 있습니다.

데이터 형식 이름	데이터 유형	에일리어스	설명
ARRAY	the section called “중첩된 유형”	해당 사항 없음	배열 중첩된 데이터 유형
BIGINT	the section called “숫자형”	해당 사항 없음	8바이트 부호화 정수
BINARY	the section called “바이너리 유형”	해당 사항 없음	바이트 시퀀스 값
BOOLEAN	the section called “부울 유형”	BOOL	논리적 부울(true/false)
BYTE	the section called “숫자형”	해당 사항 없음	1바이트 부호 있는 정수, -128~127
CHAR	the section called “문자 형식”	CHARACTER	고정 길이 문자열
DATE	the section called “날짜/시간 형식”	해당 사항 없음	날짜(년, 월, 일)
DECIMAL	the section called “숫자형”	NUMERIC	정밀도를 선택할 수 있는 정확한 숫자
FLOAT	the section called “숫자형”	FLOAT8, DOUBLE PRECISION	배정밀도 부동 소수점 수

데이터 형식 이름	데이터 유형	에일리어스	설명
INTEGER	the section called “숫자형”	INT	4바이트 부호화 정수
INTERVAL	the section called “날짜/시간 형식”	해당 사항 없음	일별 주문 또는 연간/월별 주문의 기간
LONG	the section called “숫자형”	해당 사항 없음	8바이트 부호 있는 정수
MAP	the section called “중첩된 유형”	해당 사항 없음	맵 중첩된 데이터 유형
REAL	the section called “숫자형”	FLOAT4	단정밀도 부동 소수점 수
SHORT	the section called “숫자형”	해당 사항 없음	2바이트 부호 있는 정수.
SMALLINT	the section called “숫자형”	해당 사항 없음	2바이트 부호화 정수
STRUCT	the section called “중첩된 유형”	해당 사항 없음	구조체 중첩된 데이터 유형
TIMESTAMP_LTZ	the section called “날짜/시간 형식”	해당 사항 없음	현지 시간대가 있는 시간대
TIMESTAMP_NTZ	the section called “날짜/시간 형식”	해당 사항 없음	시간대가 없는 시간대
TINYINT	the section called “숫자형”	해당 사항 없음	1바이트 부호 있는 정수, -128~127
VARCHAR	the section called “문자 형식”	CHARACTER VARYING	사용자 정의 제한이 포함된 가변 길이 문자열

> **Note**
>
> ARRAY, STRUCT 및 MAP 중첩 데이터 유형은 현재 사용자 지정 분석 규칙에만 사용할 수 있습니다. 자세한 내용은 중첩된 유형 단원을 참조하십시오.

멀티바이트 문자

VARCHAR 데이터 형식은 최대 4바이트의 UTF-8 멀티바이트 문자를 지원합니다. 5바이트 이상의 문자는 지원되지 않습니다. 멀티바이트 문자가 포함된 VARCHAR 열의 크기를 계산하려면 문자 수를 문자당 바이트 수와 곱셈합니다. 예를 들어 문자열에 한자가 4개 포함되어 있고, 각 문자의 길이가 3바이트라면 문자열을 저장하는 데 VARCHAR(12) 열이 필요합니다.

VARCHAR 데이터 유형은 다음과 같이 잘못된 UTF-8 코드포인트를 지원하지 않습니다.

`0xD800 – 0xDFFF`(바이트 시퀀스: `ED A0 80~ED BF BF`)

CHAR 데이터 유형은 멀티바이트 문자를 지원하지 않습니다.

숫자형

숫자 데이터 형식으로는 정수, 소수 및 부동 소수점 수가 있습니다.

주제

- 정수 형식
- DECIMAL 또는 NUMERIC 형식
- 부동 소수점 형식
- 숫자 값 계산

정수 형식

다음 데이터 형식을 사용하여 다양한 범위의 정수를 저장합니다. 각 형식마다 허용 범위를 벗어나는 값은 저장할 수 없습니다.

이름	스토리지	Range
SMALLINT	2 bytes	2 bytes

이름	스토리지	Range
SHORT	2 bytes	2 bytes
INTEGER 또는 INT	4 bytes	4 bytes
BIGINT	8 bytes	8 bytes
LONG	8 bytes	8 bytes

DECIMAL 또는 NUMERIC 형식

소수 또는 숫자 데이터 형식을 사용하여 사용자 정의 정밀도가 포함된 값을 저장합니다. 여기에서 소수와 숫자 키워드는 동일한 의미로 통용됩니다. 하지만 본 문서에서는 소수가 이 데이터 형식에서 우선적으로 사용되는 용어입니다. 실제로 숫자는 일반적으로 정수, 소수 및 부동 소수점 데이터 형식을 일컬을 때 사용됩니다.

스토리지	Range
가변적, 비압축 소수 형식인 경우 최대 128비트	최대 38자리 정밀도의 128비트 부호화 정수

테이블에서 DECIMAL 열은 *precision*과 *scale*을 지정하여 다음과 같이 정의합니다.

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

precision

정수에서 전체 유효 자릿수, 즉 소수점 양변의 자릿수를 말합니다. 예를 들어 숫자 48.2891의 정밀도는 6이고, 소수점 자릿수는 4입니다. 정밀도를 따로 지정하지 않을 경우 기본 정밀도는 18입니다. 최대 정밀도는 38입니다.

입력 값에서 소수점 왼쪽의 자릿수가 열 정밀도에서 소수점 자릿수를 뺀 값보다 큰 경우에는 값을 열에 복사하거나, 삽입하거나 혹은 업데이트할 수 없습니다. 이 규칙은 열 정의의 범위를 벗어나는 모든 값에 적용됩니다. 예를 들어 numeric(5,2) 열에서는 허용되는 값의 범위가 -999.99~999.99입니다.

##

값의 소수부, 즉 소수점 오른쪽의 소수 자릿수를 말합니다. 정수는 소수 자릿수가 0입니다. 열 명세에서 소수점 자릿수 값은 정밀도 값보다 작거나 같아야 합니다. 소수점 자릿수를 따로 지정하지 않을 경우 기본 소수점 자릿수는 18입니다. 최대 소수점 자릿수는 37입니다.

테이블에 로드되는 입력 값의 소수점 자릿수가 열의 소수점 자릿수보다 큰 경우에는 값이 지정한 자릿수로 반올림됩니다. 예를 들어 SALES 테이블의 PRICEPAID 열이 DECIMAL(8,2) 열이라고 가정하겠습니다. 이때 DECIMAL(8,4) 값이 PRICEPAID 열에 삽입되면 값의 소수점 자릿수가 2로 반올림됩니다.

```
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)
```

하지만 테이블에서 선택한 값의 명시적인 변환 결과는 반올림되지 않습니다.

> **Note**
>
> DECIMAL(19,0) 열에 삽입할 수 있는 최대 양의 값은 9223372036854775807(2^{63} -1)입니다. 음의 최댓값은 -9223372036854775807입니다. 예를 들어 9999999999999999999(919개) 값을 삽입하려고 하면 오버플로우 오류가 발생합니다. 소수점 위치에 상관없이 AWS Clean Rooms에서 소수로 표현할 수 있는 가장 큰 문자열은 9223372036854775807입니다. 예를 들어 DECIMAL(19,18) 열에 로드할 수 있는 가장 큰 값은 9.223372036854775807입니다.
> 이러한 규칙은 다음과 같은 이유로 적용됩니다.
>
> - 유효 자릿수가 19자리 이하인 DECIMAL 값은 8바이트 정수로 내부적으로 저장됩니다.
> - 유효 자릿수가 20~38자리인 DECIMAL 값은 16바이트 정수로 저장됩니다.

128비트 DECIMAL 또는 NUMERIC 열 사용에 대한 주의 사항

애플리케이션에 해당 전체 자릿수가 필요한지 확실하지 않은 경우 DECIMAL 열에 최대 전체 자릿수를 임의로 지정하지 않도록 합니다. 128비트 값은 64비트 값보다 두 배 많은 디스크 공간을 사용하므로 쿼리 실행 시간이 느려질 수 있습니다.

부동 소수점 형식

REAL 및 DOUBLE PRECISION 데이터 형식을 사용하여 가변 정밀도의 숫자 값을 저장합니다. 부동 소수점 형식은 부정확합니다. 이 말은 일부 값이 근사치로 저장되어 특정 값을 저장하거나 반환할 때 약간 불일치가 발생할 수 있다는 것을 의미합니다. 따라서 정확한 저장 및 계산이 필요하다면(금전적 액수 등) DECIMAL 데이터 형식을 사용하세요.

REAL은 부동 소수점 산술에 대한 IEEE 표준 754에 따라 단정밀도 부동 소수점 형식을 나타냅니다. 정밀도는 약 6자리이며 범위는 약 1E-37~1E+37입니다. 이 데이터 유형을 FLOAT4로 지정할 수도 있습니다.

DOUBLE PRECISION은 이진 부동 소수점 산술에 대한 IEEE 표준 754에 따른 배정밀도 부동 소수점 형식을 나타냅니다. 정밀도는 약 15자리이며 범위는 약 1E-307~1E+308입니다. 이 데이터 유형을 FLOAT 또는 FLOAT8로 지정할 수도 있습니다.

숫자 값 계산

에서 계산은 더하기AWS Clean Rooms, 빼기, 곱하기, 나누기 등의 이진 수학 연산을 나타냅니다. 이번 섹션에서는 이러한 연산에 따라 예상되는 반환 형식을 비롯해 DECIMAL 데이터 형식이 포함되어 있을 경우 정밀도와 소수점 자릿수의 계산 공식에 대해서 설명합니다.

쿼리 처리 시 숫자 값을 계산할 때는 계산이 불가능하거나 쿼리가 숫자 오버플로우 오류를 반환하는 상황이 발생할 수 있습니다. 그 밖에도 계산된 값의 소수점 자릿수가 바뀌거나 예상과 다를 수도 있습니다. 일부 연산에서는 명시적 변환(형식 승격) 또는 AWS Clean Rooms 구성 파라미터를 사용하여 이러한 문제를 피할 수 있습니다.

SQL 함수를 사용하는 비슷한 계산 결과에 대한 자세한 내용은 AWS Clean Rooms Spark SQL 함수 섹션을 참조하세요.

계산 반환 형식

에서 지원되는 숫자 데이터 형식 집합을 고려할 때 다음 표에는 추가AWS Clean Rooms, 빼기, 곱하기 및 나누기 작업에 대해 예상되는 반환 유형이 나와 있습니다. 표 왼쪽에서 첫 번째 열은 계산에 포함되는 첫 번째 피연산자이고, 맨 위의 행은 두 번째 피연산자를 의미합니다.

피연산자 1	피연산자 1	반환 타입
SMALLINT 또는 SHORT	SMALLINT 또는 SHORT	SMALLINT 또는 SHORT
SMALLINT 또는 SHORT	INTEGER	INTEGER
SMALLINT 또는 SHORT	BIGINT	BIGINT
SMALLINT 또는 SHORT	DECIMAL	DECIMAL
SMALLINT 또는 SHORT	FLOAT4	FLOAT8
SMALLINT 또는 SHORT	FLOAT8	FLOAT8
INTEGER	INTEGER	INTEGER
INTEGER	BIGINT 또는 LONG	BIGINT 또는 LONG
INTEGER	DECIMAL	DECIMAL
INTEGER	FLOAT4	FLOAT8
INTEGER	FLOAT8	FLOAT8
BIGINT 또는 LONG	BIGINT 또는 LONG	BIGINT 또는 LONG
BIGINT 또는 LONG	DECIMAL	DECIMAL
BIGINT 또는 LONG	FLOAT4	FLOAT8
BIGINT 또는 LONG	FLOAT8	FLOAT8
DECIMAL	DECIMAL	DECIMAL
DECIMAL	FLOAT4	FLOAT8
DECIMAL	FLOAT8	FLOAT8
FLOAT4	FLOAT8	FLOAT8
FLOAT8	FLOAT8	FLOAT8

계산된 DECIMAL 결과의 정밀도와 소수점 자릿수

다음 표는 수학 연산이 DECIMAL 결과를 반환할 때 정밀도와 소수점 자릿수를 계산하는 규칙을 요약한 것입니다. 이 표에서 p1 및 s1는 계산에서 첫 번째 피연산자의 정밀도와 규모를 나타냅니다. p2 및 는 두 번째 피연산자의 정밀도와 규모를 s2 나타냅니다. 이 계산과 상관없이 최대 결과 정밀도는 38이고, 최대 결과 소수점 자릿수도 38입니다.

연산	결과 정밀도 및 소수점 자릿수
+ 또는 -	Scale = `max(s1,s2)` precision = `max(p1-s1,p2-s2)+1+scale`
*	Scale = `s1+s2` precision = `p1+p2+1`
/	Scale = `max(4,s1+p2-s2+1)` precision = `p1-s1+ s2+scale`

예를 들어 SALES 테이블의 PRICEPAID 열과 COMMISSION 열이 모두 DECIMAL(8,2) 열이라고 가정하겠습니다. 이때 PRICEPAID를 COMMISSION으로(혹은 그 반대로) 나누면 다음과 같은 공식이 적용됩니다.

```
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)
```

다음 계산은 UNION, INTERSECT, EXCEPT 같은 집합 연산자나 COALESCE, DECODE 같은 함수를 사용해 DECIMAL 값에 대한 연산 결과 정밀도와 소수점 자릿수를 계산하기 위한 일반 규칙입니다.

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

예를 들어 DECIMAL(7,2) 열 1개가 포함된 DEC1 테이블이 DECIMAL(15,3) 열 1개가 포함된 DEC2 테이블과 조인되어 DEC3 테이블을 생성한다고 가정할 때, DEC3의 스키마를 보면 NUMERIC(15,3) 열이 되는 것을 알 수 있습니다.

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

위 예에서 적용되는 공식은 다음과 같습니다.

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

Scale = max(2,3) = 3

Result = DECIMAL(15,3)
```

나누기 연산에 대한 주의 사항

나누기 연산에서 0으로 나누는(divide-by-zero) 조건은 오류를 반환합니다.

정밀도와 소수점 자릿수를 계산한 후에는 소수점 자릿수 한계로 100이 적용됩니다. 계산된 결과 소수점 자릿수가 100보다 크면 나누기 결과가 다음과 같이 조정됩니다.

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

계산된 정밀도가 최대 정밀도(38)보다 크면 정밀도가 38로 줄어들고, 소수점 자릿수는 다음 공식의 결과 값이 됩니다. `max(38 + scale - precision), min(4, 100))`

오버플로우 조건

모든 수치 계산에서는 오버플로우 여부를 확인합니다. 정밀도가 19 이하인 DECIMAL 데이터는 64비트 정수로 저장됩니다. 정밀도가 19보다 큰 DECIMAL 데이터는 128비트 정수로 저장됩니다. DECIMAL 값은 모두 최대 정밀도가 38이고, 최대 소수점 자릿수가 37입니다. 오버플로우 오류는 값이 이러한 제한을 초과할 때 발생하며, 중간 결과 집합과 최종 결과 집합 모두에 적용됩니다.

- 명시적 캐스팅은 특정 데이터 값이 캐스트 함수에서 지정한 요청된 정밀도 또는 규모에 맞지 않을 때 런타임 오버플로 오류를 발생시킵니다. 예를 들어 SALES 테이블의 PRICEPAID 열(DECIMAL(8,2) 열)에 있는 값을 모두 변환할 수는 없기 때문에 다음과 같이 DECIMAL(7,3) 결과를 반환합니다.

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

이러한 오류가 발생하는 이유는 PRICEPAID 열에서 일부 더 큰 값은 변환할 수 없기 때문입니다.

- 곱하기 연산은 결과 소수점 자릿수가 각 피연산자의 소수점 자릿수 합인 결과를 산출합니다. 예를 들어 두 피연산자의 소수점 자릿수가 4라고 한다면 결과 소수점 자릿수는 8이 되고 소수점 왼쪽에는 10자리만 남게 됩니다. 따라서 둘 다 유효 소수점 자릿수를 가지고 있는 큰 수 2개를 곱할 경우에는 비교적 오버플로우 조건이 발생하기 쉽습니다.

INTEGER 및 DECIMAL 형식을 사용한 숫자 계산

계산 시 피연산자 하나가 INTEGER 데이터 형식이고, 나머지 피연산자가 DECIMAL 데이터 형식인 경우에는 INTEGER 피연산자가 묵시적으로 DECIMAL로 변환됩니다.

- SMALLINT 또는 SHORT는 DECIMAL(5,0)로 캐스팅됩니다.
- INTEGER는 DECIMAL(10,0)로 변환됨
- BIGINT 또는 LONG는 DECIMAL(19,0)로 캐스팅됩니다.

예를 들어 SALES.COMMISSION(DECIMAL(8,2) 열)과 SALES.QTYSOLD(SMALLINT 열)를 곱하면 다음과 같이 변환됩니다.

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

문자 형식

문자 데이터 형식에는 CHAR(문자)와 VARCHAR(가변 문자)가 포함됩니다.

주제

- CHAR 또는 CHARACTER
- VARCHAR 또는 CHARACTER VARYING
- 후행 공백의 중요성

CHAR 또는 CHARACTER

CHAR 또는 CHARACTER 열은 고정 길이 문자열을 저장하는 데 사용됩니다. 이 문자열은 공백으로 채워지므로 CHAR(10) 열은 항상 10바이트의 스토리지를 차지합니다.

```
char(10)
```

길이 명세가 없는 CHAR 열은 CHAR(1) 열이 됩니다.

CHAR 및 VARCHAR 데이터 형식은 문자가 아닌 바이트로 정의됩니다. CHAR 열에는 단일 바이트 문자만 포함되기 때문에 예를 들어 CHAR(10) 열이라고 하면 최대 10바이트의 문자열이 포함될 수 있습니다.

이름	스토리지	범위(열의 너비)
CHAR 또는 CHARACTER	후행 공백(있는 경우)을 포함한 문자열 길이	4096 bytes

VARCHAR 또는 CHARACTER VARYING

VARCHAR 또는 CHARACTER VARYING 열은 제한이 고정되어 있는 가변 길이 문자열을 저장하는 데 사용됩니다. 이 문자열은 공백으로 채워지지 않으므로 VARCHAR(120) 열은 단일 바이트 문자 120개, 2바이트 문자 60개, 3바이트 문자 40개 또는 4바이트 문자 30개까지 구성됩니다.

```
varchar(120)
```

VARCHAR 데이터 형식은 문자가 아닌 바이트 단위로 정의됩니다. VARCHAR에는 멀티바이트 문자가 포함되어 문자당 최대 4바이트까지 가능합니다. 예를 들어 VARCHAR(12)라고 하면 단일 바이트 문자 12개, 2바이트 문자 6개, 3바이트 문자 4개, 또는 4바이트 문자 3개가 포함될 수 있습니다.

이름	스토리지	범위(열의 너비)
VARCHAR 또는 CHARACTER VARYING	4바이트 + 전체 문자 바이트, 여기에서 각 문자의 길이는 1~4바이	65535바이트(64K -1)

이름	스토리지	범위(열의 너비)
	트가 될 수 있습니다.	

후행 공백의 중요성

CHAR 및 VARCHAR 데이터 형식은 모두 n 바이트 길이까지 문자열을 저장합니다. 더 긴 문자열을 이러한 유형의 열에 저장하려고 하면 오류가 발생합니다. 그러나 추가 문자가 모두 스페이스(공백)인 경우 문자열은 최대 길이까지 잘립니다. 문자열이 최대 길이보다 짧을 경우 CHAR 값은 공백으로 채워지지만 VARCHAR 값은 공백 없이 문자열을 저장합니다.

CHAR 값에서 후행 공백은 언제나 의미상 유의적이지 않습니다. CHAR 값 2개를 비교할 때는 무시되고, LENGTH 계산에 포함되지 않으며, 그리고 CHAR 값을 다른 문자열 형식으로 변환할 때는 제거됩니다.

값을 서로 비교할 경우 VARCHAR 값과 CHAR 값의 후행 공백은 의미상 유의적이지 않습니다.

LENGTH 계산을 실행하면 길이에 포함된 후행 공백까지 합쳐서 VARCHAR 문자열의 길이를 반환합니다. 하지만 고정 길이 문자열에서는 후행 공백을 길이에 포함하여 계산하지 않습니다.

날짜/시간 형식

날짜/시간 데이터 유형에는 DATE, TIME, TIMESTAMP_LTZ 및 TIMESTAMP_NTZ가 포함됩니다.

주제

- DATE
- TIMESTAMP_LTZ
- TIMESTAMP_NTZ
- 날짜/시간 형식의 예제
- 날짜, 시간 및 타임스탬프 리터럴
- 간격 리터럴
- 간격 데이터 유형 및 리터럴

DATE

DATE 데이터 형식은 타임스탬프 없이 날짜만 저장하는 데 사용됩니다.

이름	스토리지	Range	해결 방법
DATE	4 bytes	4713 BC~294276 AD	1일

TIMESTAMP_LTZ

TIMESTAMP_LTZ 데이터 형식을 사용하여 날짜, 시간 및 현지 시간대가 포함된 전체 타임스탬프 값을 저장합니다.

TIMESTAMP는 세션 현지 시간대와 함께 필드 `year`, `second`, `monthday`, `hour`, `minute`,의 값으로 구성된 값을 나타냅니다. `timestamp` 값은 절대 시점을 나타냅니다.

Spark의 TIMESTAMP는 TIMESTAMP_LTZ 및 TIMESTAMP_NTZ 변형 중 하나와 연결된 사용자 지정 별칭입니다. 구성을 통해 기본 타임스탬프 유형을 TIMESTAMP_LTZ(기본값) 또는 TIMESTAMP_NTZ로 설정할 수 있습니다`spark.sql.timestampType`.

TIMESTAMP_NTZ

TIMESTAMP_NTZ 데이터 형식을 사용하여 현지 시간대 없이 날짜, 시간 등의 전체 타임스탬프 값을 저장합니다.

TIMESTAMP는 필드 `year`, , `month`, `day`, `minute`, `hour`의 값으로 구성된 값을 나타냅니다`second`. 모든 작업은 시간대를 고려하지 않고 수행됩니다.

Spark의 TIMESTAMP는 TIMESTAMP_LTZ 및 TIMESTAMP_NTZ 변형 중 하나와 연결된 사용자 지정 별칭입니다. 구성을 통해 기본 타임스탬프 유형을 TIMESTAMP_LTZ(기본값) 또는 TIMESTAMP_NTZ로 설정할 수 있습니다`spark.sql.timestampType`.

날짜/시간 형식의 예제

다음 예시에서는 에서 지원하는 AWS Clean Rooms 날짜/시간 유형을 사용하는 방법을 보여줍니다.

날짜 예제

다음은 다른 형식의 날짜를 삽입한 후 출력을 표시하는 예입니다.

```
select * from datetable order by 1;

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

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

타임스탬프 값을 DATE 열에 삽입하면 시간 구간은 무시하고 날짜만 로드됩니다.

시간 예제

다음은 다른 형식의 TIME 및 TIMETZ 값을 삽입한 후 출력을 표시하는 예입니다.

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

날짜, 시간 및 타임스탬프 리터럴

다음은 AWS Clean RoomsSpark SQL에서 지원하는 날짜, 시간 및 타임스탬프 리터럴 작업 규칙입니다.

날짜

다음 표에는 AWS Clean Rooms테이블에 로드할 수 있는 리터럴 날짜 값의 유효한 예인 입력 날짜가 나와 있습니다. 기본 `MDY DateStyle` 모드가 유효한 것으로 간주됩니다. 이 모드는 `1999-01-08`, `01/02/00`과 같은 문자열에서 월 값이 일 값에 선행함을 의미합니다.

> **Note**
>
> 날짜 또는 타임스탬프 리터럴은 테이블에 로드할 때 인용 부호로 묶어야 합니다.

입력 날짜	전체 날짜
1999년 1월 8일	1999년 1월 8일
1999년 1월 8일	1999년 1월 8일
1999년 1월 8일	1999년 1월 8일
1999년 1월 8일	1999년 1월 8일
1999년 1월 8일	2000년 1월 31일

입력 날짜	전체 날짜
2000년 1월 31일	2000년 1월 31일
2000년 1월 31일	2000년 1월 31일
20080215	2008년 2월 15일
080215	2008년 2월 15일
2008.366	2008년 12월 31일(날짜에서 3자리 구간에 입력할 수 있는 숫자는 001부터 366까지임)

Times

다음 표에는 AWS Clean Rooms테이블에 로드할 수 있는 리터럴 시간 값의 유효한 예인 입력 시간이 나와 있습니다.

입력 시간	설명(시간 구간)
04:05:06.789	오전 4시 5분 6.789초
04:05:06	오전 4시 5분 6초
04:05	오전 4시 5분 정각
040506	오전 4시 5분 6초
오전 4시 5분	오전 4시 5분 정각(오전은 옵션)
04:05 PM	오후 4시 5분 정각(시간 값은 12보다 작아야 함)
16:05	오후 4시 5분 정각

특수한 날짜/시간 값

다음 테이블에는 날짜/시간 리터럴 및 날짜 함수의 인수로 사용할 수 있는 특수 값을 보여줍니다. 이 특수 값을 사용하려면 작은따옴표가 필요하며, 쿼리 처리 시 타임스탬프 정규 값으로 변환됩니다.

특수 값	설명
now	현재 트랜잭션의 시작 시간으로 평가되며, 마이크로초 정밀도로 타임스탬프를 반환합니다.
today	해당하는 날짜로 평가되며, 시간 부분을 0으로 타임스탬프를 반환합니다.
tomorrow	해당하는 날짜로 평가되며, 시간 부분을 0으로 타임스탬프를 반환합니다.
yesterday	해당하는 날짜로 평가되며, 시간 부분을 0으로 타임스탬프를 반환합니다.

다음 예제에서는 now 및가 DATE_ADD 함수와 today 작동하는 방법을 보여줍니다.

```
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)
```

간격 리터럴

다음은 Spark SQL에서AWS Clean Rooms 지원하는 간격 리터럴 작업 규칙입니다.

간격 리티럴은 12 hours 또는 6 weeks 같이 일정한 기간을 구분할 때 사용됩니다. 또한 날짜/시간 표현식이 포함된 조건이나 계산에서도 간격 리터럴을 사용할 수 있습니다.

> **Note**
>
> AWS Clean Rooms테이블의 열에는 INTERVAL 데이터 형식을 사용할 수 없습니다.

간격은 INTERVAL 키워드와 숫자 기간, 그리고 지원되는 날짜 부분으로 표현됩니다. 예를 들면 `INTERVAL '7 days'` 또는 `INTERVAL '59 minutes'`와 같습니다. 더욱 정확한 간격을 표현하기 위해 여러 수량과 단위를 연결할 수도 있습니다(예: `INTERVAL '7 days, 3 hours, 59 minutes'`). 각 단위의 약어와 복수 표현도 지원됩니다. 예를 들어 `5 s`, `5 second` 및 `5 seconds` 모두 동일한 간격입니다.

날짜 부분을 지정하지 않을 경우 간격 값은 초를 의미합니다. 기간 값은 소수로 지정할 수도 있습니다(예: `0.5 days`).

예제

다음은 모두 간격 값을 다르게 하여 계산하는 예들입니다.

다음 예는 지정된 날짜에 1초를 더합니다.

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

다음 예는 지정된 날짜에 1분을 더합니다.

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

다음 예는 지정한 날짜에 3시간 35분을 더합니다.

```
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)
```

다음 예는 지정된 날짜에 52주를 더합니다.

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

다음 예는 지정한 날짜에 1주, 1시간, 1분, 1초를 더합니다.

```
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)
```

다음 예는 지정한 날짜에 12시간(반일)을 더합니다.

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

다음 예는 2023년 3월 31일에서 4개월을 뺀 값이며 결과는 2022년 11월 30일입니다. 계산에서는 한 달의 일수를 고려합니다.

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

?column?
---------------------
```

```
2022-11-30 00:00:00
```

간격 데이터 유형 및 리터럴

간격 데이터 유형을 사용하여 기간을 seconds, minutes, hours, days, months 및 years 단위로 저장할 수 있습니다. 날짜 및 타임스탬프에 간격 추가, 간격 합산, 날짜 또는 타임스탬프에서 간격 빼기 등 날짜/시간 계산에 간격 데이터 유형 및 리터럴을 사용할 수 있습니다. 간격 리터럴은 테이블의 간격 데이터 유형 열에 대한 입력 값으로 사용할 수 있습니다.

간격 데이터 유형 구문

기간을 년 및 월 단위로 저장하도록 간격 데이터 유형을 지정하려면:

```
INTERVAL year_to_month_qualifier
```

기간을 일, 시간, 분, 초 단위로 저장하도록 간격 데이터 유형을 지정하려면:

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

간격 리터럴 구문

기간을 년 및 월 단위로 정의하도록 간격 리터럴을 지정하려면:

```
INTERVAL quoted-string year_to_month_qualifier
```

기간을 일, 시간, 분, 초 단위로 정의하도록 간격 리터럴을 지정하려면:

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

인수

quoted-string

수량과 날짜/시간 단위를 입력 문자열로 지정하여 양수 또는 음수 값을 지정합니다. 따옴표로 묶인 문자열에 숫자만 포함된 경우 AWS Clean Rooms는 year_to_month_qualifier 또는 day_to_second_qualifier의 단위를 결정합니다. 예를 들어 '23' MONTH는 1 year 11 months를 나타내고, '-2' DAY는 -2 days 0 hours 0 minutes 0.0 seconds를 나타내며, '1-2'

MONTH는 1 year 2 months를 나타내고, '13 day 1 hour 1 minute 1.123 seconds' SECOND는 13 days 1 hour 1 minute 1.123 seconds를 나타냅니다. 간격 출력 형식에 대한 자세한 내용은 간격 스타일 섹션을 참조하세요.

year_to_month_qualifier

간격 범위를 지정합니다. 한정자를 사용하고 시간 단위가 한정자보다 작은 간격을 생성하면는 간격의 작은 부분을 AWS Clean Rooms잘라내고 폐기합니다. year_to_month_qualifier의 유효한 값은 다음과 같습니다.

- YEAR
- MONTH
- YEAR TO MONTH

day_to_second_qualifier

간격 범위를 지정합니다. 한정자를 사용하고 시간 단위가 한정자보다 작은 간격을 생성하면는 간격의 작은 부분을 AWS Clean Rooms잘라내고 폐기합니다. day_to_second_qualifier의 유효한 값은 다음과 같습니다.

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

INTERVAL 리터럴의 출력은 지정된 가장 작은 INTERVAL 구성 요소까지 잘립니다. 예를 들어 MINUTE 한정자를AWS Clean Rooms 사용하는 경우는 MINUTE보다 작은 시간 단위를 폐기합니다.

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

결과 값은 '1 day 01:01:00'까지 잘립니다.

fractional_precision

간격에 허용되는 소수 자릿수를 지정하는 파라미터 옵션입니다. fractional_precision 인수는 간격에 SECOND가 포함된 경우에만 지정해야 합니다. 예를 들어 `SECOND(3)`는 세 자리 소수만 허용하는 간격을 생성합니다(예: 1.234초). 최대 소수 자릿수는 6자리입니다.

세션 구성 `interval_forbid_composite_literals`는 YEAR TO MONTH 및 DAY TO SECOND 부분을 모두 사용하여 간격을 지정하는 경우 오류가 반환되는지 여부를 결정합니다.

간격 산술

간격 값을 다른 날짜/시간 값과 함께 사용하여 산술 연산을 수행할 수 있습니다. 다음 테이블에는 사용 가능한 작업과 각 작업에서 얻을 수 있는 데이터 유형이 설명되어 있습니다.

> **Note**
>
> `date` 및 `timestamp` 결과를 모두 생성할 수 있는 작업은 방정식과 관련된 최소 시간 단위를 기반으로 합니다. 예를 들어, `interval`을 `date`에 추가하는 경우 YEAR TO MONTH 간격이면 `date`이 반환되고, DAY TO SECOND 간격이면 timestamp가 반환됩니다.

첫 번째 피연산자가 `interval`인 작업은 지정된 두 번째 피연산자에 대해 다음과 같은 결과를 생성합니다.

연산자	Date	타임스탬프	간격	Numeric
-	해당 사항 없음	해당 사항 없음	간격	해당 사항 없음
+	Date	Date/Timestamp	간격	해당 사항 없음
*	해당 사항 없음	해당 사항 없음	해당 사항 없음	간격
/	해당 사항 없음	해당 사항 없음	해당 사항 없음	간격

첫 번째 피연산자가 `date`인 작업은 지정된 두 번째 피연산자에 대해 다음과 같은 결과를 생성합니다.

연산자	Date	타임스탬프	간격	Numeric
-	Numeric	간격	Date/Timestamp	Date
+	해당 사항 없음	해당 사항 없음	해당 사항 없음	해당 사항 없음

첫 번째 피연산자가 timestamp인 작업은 지정된 두 번째 피연산자에 대해 다음과 같은 결과를 생성합니다.

연산자	Date	타임스탬프	간격	Numeric
-	Numeric	간격	타임스탬프	타임스탬프
+	해당 사항 없음	해당 사항 없음	해당 사항 없음	해당 사항 없음

간격 스타일

- postgres - PostgreSQL 스타일을 따릅니다. 기본값입니다.
- postgres_verbose - PostgreSQL 상세 정보 표시 스타일을 따릅니다.
- sql_standard - SQL 표준 간격 리터럴 스타일을 따릅니다.

다음 명령은 간격 스타일을 sql_standard로 설정합니다.

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

postgres 출력 형식

다음은 postgres 간격 스타일의 출력 형식입니다. 각 숫자 값은 음수일 수 있습니다.

```
'<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 출력 형식

postgres_verbose 구문은 postgres와 비슷하지만 postgres_verbose 출력에는 시간 단위도 포함됩니다.

```
'[@] <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 출력 형식

간격 연도-월 값의 형식은 다음과 같습니다. 간격 앞에 음수 부호를 지정하면 간격이 음수임을 나타내며 전체 간격에 적용됩니다.

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

간격 일-초 값의 형식은 다음과 같습니다.

```
'[-]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
```

간격 데이터 유형의 예

다음 예제에서는 테이블을 사용해 INTERVAL 데이터 유형을 사용하는 방법을 보여 줍니다.

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

간격 리터럴의 예

다음 예제는 간격 스타일을 postgres로 설정한 상태에서 실행됩니다.

다음 예제에서는 1년의 INTERVAL 리터럴을 생성하는 방법을 보여 줍니다.

```
select INTERVAL '1' YEAR

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

한정자를 초과하는 quoted-string을 지정하면 간격에서 남은 시간 단위가 잘립니다. 다음 예제에서는 13개월의 간격이 1년 1개월이 되지만 나머지 1개월은 YEAR 한정자로 인해 제외됩니다.

```
select INTERVAL '13 months' YEAR

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

간격 문자열보다 낮은 한정어를 사용하는 경우 남은 단위가 포함됩니다.

```
select INTERVAL '13 months' MONTH

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

간격에 정밀도를 지정하면 소수 자릿수가 잘려 지정된 정밀도까지 줄어듭니다.

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

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

정밀도를 지정하지 않으면는 최대 정밀도 6을 AWS Clean Rooms사용합니다.

```
select INTERVAL '1.23456789' SECOND

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

다음 예제에서는 범위가 지정된 간격을 생성하는 방법을 보여 줍니다.

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

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

한정자에 따라 지정하는 단위가 결정됩니다. 예를 들어 다음 예제에서는 이전 예제와 동일한 '2:2'의 따옴표로 묶인 문자열을 사용하더라도는 한정자 때문에 다른 시간 단위를 사용한다는 것을 AWS Clean Rooms인식합니다.

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

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

각 단위의 약어 및 복수형도 지원됩니다. 예를 들어 5s, 5 second, 5 seconds는 동일한 간격입니다. 지원되는 단위는 연, 월, 시간, 분, 초입니다.

```
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
```

한정자 구문이 없는 간격 리터럴의 예

> **Note**
>
> 다음 예에서는 `YEAR TO MONTH` 또는 `DAY TO SECOND` 한정자 없이 간격 리터럴을 사용하는 방법을 보여줍니다. 한정자와 함께 권장 간격 리터럴을 사용하는 방법에 대한 자세한 내용은 간격 데이터 유형 및 리터럴 섹션을 참조하세요.

간격 리티럴은 `12 hours` 또는 `6 months` 같이 일정한 기간을 구분할 때 사용됩니다. 또한 날짜/시간 표현식이 포함된 조건이나 계산에서도 간격 리터럴을 사용할 수 있습니다.

간격 리터럴은 INTERVAL 키워드와 숫자 기간, 그리고 지원되는 날짜 부분으로 표현됩니다. 예를 들면 `INTERVAL '7 days'` 또는 `INTERVAL '59 minutes'`와 같습니다. 더욱 정확한 간격을 표현하기 위해 여러 수량과 단위를 연결할 수도 있습니다(예: `INTERVAL '7 days, 3 hours, 59 minutes'`). 각 단위의 약어와 복수 표현도 지원됩니다. 예를 들어 `5 s`, `5 second` 및 `5 seconds` 모두 동일한 간격입니다.

날짜 부분을 지정하지 않을 경우 간격 값은 초를 의미합니다. 기간 값은 소수로 지정할 수도 있습니다(예: `0.5 days`).

다음은 모두 간격 값을 다르게 하여 계산하는 예들입니다.

다음은 지정된 날짜에 1초를 더합니다.

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

다음은 지정된 날짜에 1분을 더합니다.

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

다음은 지정한 날짜에 3시간 35분을 더합니다.

```
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)
```

다음은 지정된 날짜에 52주를 더합니다.

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

다음은 지정한 날짜에 1주, 1시간, 1분, 1초를 더합니다.

```
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)
```

다음은 지정한 날짜에 12시간(반일)을 더합니다.

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

다음은 2023년 2월 15일에서 4개월을 뺀 값이며 결과는 2022년 10월 15일입니다.

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

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

다음은 2023년 3월 31일에서 4개월을 뺀 값이며 결과는 2022년 11월 30일입니다. 계산에서는 한 달의 일수를 고려합니다.

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

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

부울 유형

부울 데이터 형식은 단일 바이트 열에 true 또는 false 값을 저장하는 데 사용됩니다. 다음 표는 부울 값에서 가능한 세 가지 상태와 이러한 상태를 나타내는 리터럴 값에 대해 설명한 것입니다. 입력 문자열에 상관없이 Boolean 열은 true일 때는 "t"를, 그리고 false일 때는 "f"를 저장 및 출력합니다.

State	유효한 리터럴 값	스토리지
True	`TRUE 't' 'true' 'y' 'yes' '1'`	1바이트
False	`FALSE 'f' 'false' 'n' 'no' '0'`	1바이트
알 수 없음	`NULL`	1바이트

IS 비교를 사용해 WHERE 절의 조건자로 부울 값만 확인할 수 있습니다. IS 비교는 SELECT 목록의 부울 값에는 사용할 수 없습니다.

예제

BOOLEAN 열을 사용하여 각 고객의 "Active/Inactive" 상태를 CUSTOMER 테이블에 저장할 수 있습니다.

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

이 예에서 다음 쿼리는 USERS 테이블에서 스포츠를 좋아하지만 영화를 좋아하지 않는 사용자를 선택합니다.

```
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)
```

다음은 USERS 테이블에서 록 음악을 좋아하는지 알 수 없는 사용자를 선택하는 예입니다.

```
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)
```

다음 예에서는 SELECT 목록에 IS 비교를 사용했기 때문에 오류를 반환합니다.

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

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

다음 예는 SELECT 목록에서 IS 비교 대신에 같음 비교(=)를 사용했기 때문에 성공합니다.

```
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    |
```

부울 리터럴

다음 규칙은 Spark SQL에서AWS Clean Rooms 지원하는 부울 리터럴 작업을 위한 것입니다.

부울 리터럴을 사용하여 TRUE 또는와 같은 부울 값을 지정합니다FALSE.

구문

```
TRUE | FALSE
```

예제

다음 예제에서는 지정된 값이 TRUE 인 열을 보여줍니다.

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

바이너리 유형

BINARY 데이터 형식을 사용하여 해석되지 않은 고정 길이의 바이너리 데이터를 저장하고 관리하여 특정 사용 사례에 효율적인 저장 및 비교 기능을 제공합니다.

BINARY 데이터 형식은 저장되는 데이터의 실제 길이에 관계없이 고정된 바이트 수를 저장합니다. 최대 길이는 일반적으로 255바이트입니다.

BINARY는 이미지, 문서 또는 기타 유형의 파일과 같이 해석되지 않은 원시 바이너리 데이터를 저장하는 데 사용됩니다. 데이터는 문자 인코딩이나 해석 없이 제공된 그대로 저장됩니다. BINARY 열에 저장된 이진 데이터는 문자 인코딩 또는 데이터 정렬 규칙이 아닌 실제 이진 값을 기반으로 byte-by-byte 비교 및 정렬됩니다.

다음 예제 쿼리는 문자열의 이진 표현을 보여줍니다"abc". 문자열의 각 문자는 16진수 형식으로 ASCII 코드로 표시됩니다. "a"는 0x61, "b"는 0x62, "c"는 0x63입니다. 결합된 경우 이러한 16진수 값은 이진 표현를 형성합니다"616263".

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

중첩된 유형

AWS Clean Rooms는 중첩된 데이터 유형, 특히 STRUCT, ARRAY 및 MAP 열 유형이 있는 데이터와 관련된 쿼리를AWS Glue 지원합니다. 사용자 지정 분석 규칙만 중첩 데이터 유형을 지원합니다.

특히 중첩된 데이터 유형은 SQL 데이터베이스의 관계형 데이터 모델의 엄격한 테이블형 구조를 따르지 않습니다.

중첩된 데이터 유형에는 데이터 내의 고유한 엔티티를 참조하는 태그가 포함됩니다. 여기에는 배열, 중첩 구조 및 JSON과 같은 직렬화 형식과 연결된 기타 복잡한 구조와 같은 복소수 값이 포함될 수 있습니다. 중첩된 데이터 유형은 개별 중첩된 데이터 유형 필드 또는 객체에 대해 최대 1MB의 데이터를 지원합니다.

주제

- ARRAY 유형
- MAP 유형
- STRUCT 유형
- 중첩된 데이터 유형의 예

ARRAY 유형

ARRAY 유형을 사용하여 유형이 인 요소의 시퀀스로 구성된 값을 나타냅니다`elementType`.

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

ARRAY 유형의 요소가 `null` 값을 가질 수 있는지 여부를 나타내는 `containsNull` 데 사용합니다.

MAP 유형

MAP 유형을 사용하여 키-값 페어 집합으로 구성된 값을 나타냅니다.

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

`keyType`: 키의 데이터 유형

`valueType`: 값의 데이터 유형

키는 `null` 값을 가질 수 없습니다. MAP 유형 값의 값에 `null` 값이 있을 수 있는지 여부를 나타내는 `valueContainsNull` 데 사용합니다.

STRUCT 유형

STRUCT 유형을 사용하여 StructFields(필드) 시퀀스에 설명된 구조의 값을 나타냅니다.

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

StructField(name, dataType, nullable): StructType의 필드를 나타냅니다.

`dataType`: 필드의 데이터 형식

`name`: 필드의 이름

`nullable`를 사용하여 이러한 필드의 값에 `null` 값이 있을 수 있는지 여부를 나타냅니다.

중첩된 데이터 유형의 예

`struct<given:varchar, family:varchar>` 유형에는 두 개의 속성 이름, 즉 `given`, 및 `family`가 있으며, 각 이름은 `varchar` 값에 해당합니다.

`array<varchar>` 유형의 경우 배열은 `varchar`의 목록으로 지정됩니다.

`array<struct<shipdate:timestamp, price:double>>` 유형은 `struct<shipdate:timestamp, price:double>` 유형이 있는 요소 목록을 나타냅니다.

`map` 데이터 유형은 `structs`의 `array`처럼 동작합니다. 여기서 배열의 각 요소에 대한 속성 이름은 `key`로 표시되고 `value`에 매핑됩니다.

Example

예를 들어, `map<varchar(20), varchar(20)>` 유형은 `array<struct<key:varchar(20), value:varchar(20)>>`로 취급되며, `key`와 `value`는 기초 데이터에서 맵의 속성을 나타냅니다.

가 배열 및 구조 탐색을 AWS Clean Rooms활성화하는 방법에 대한 자세한 내용은 섹션을 참조하세요탐색.

쿼리의 FROM 절을 사용하여 배열을 탐색하여가 배열에 대한 반복을 AWS Clean Rooms활성화하는 방법에 대한 자세한 내용은 섹션을 참조하세요쿼리 중첩 해제.

형식 호환성 및 변환

다음 주제에서는 AWS Clean RoomsSpark SQL에서 유형 변환 규칙 및 데이터 유형 호환성이 작동하는 방식을 설명합니다.

주제

- 호환성
- 일번적인 호환성 및 변환 규칙
- 묵시적인 변환 형식

호환성

데이터 형식 일치, 즉 리터럴 값 및 상수를 데이터 형식과 일치시키는 것은 아래 작업을 포함해 다양한 데이터베이스 작업에서 발생합니다.

- 테이블에 대한 데이터 조작 언어(DML) 작업
- UNION, INTERSECT 및 EXCEPT 쿼리
- CASE 표현식
- LIKE, IN 등 조건자 평가
- 데이터를 비교하거나 추출하는 SQL 함수에 대한 평가
- 수학 연산자를 사용한 비교

위의 작업 결과는 형식 변환 규칙과 데이터 형식 호환성에 따라 달라집니다. 호환성에는 특정 값과 특정 데이터 형식의 1대 1 일치가 항상 필요한 것은 아니라는 의미가 내포되어 있습니다. 일부 데이터 형식은 호환이 가능하기 때문에 묵시적 변환, 즉 강제 변환이 가능합니다. 자세한 내용은 묵시적인 변환 형식 단원을 참조하십시오. 데이터 형식이 호환되지 않을 때는 명시적인 변환 함수를 사용하여 다른 데이터 형식으로 값을 변환할 수 있는 경우도 있습니다.

일번적인 호환성 및 변환 규칙

호환성 및 변환 규칙은 다음과 같습니다.

- 일반적으로 동일한 형식 카테고리에 해당하는 데이터 형식(여러 가지 숫자 데이터 형식 등)은 서로 호환이 가능하기 때문에 묵시적으로 변환할 수 있습니다.

  예를 들어 묵시적인 변환을 통해 소수 값을 정수 열에 삽입할 수 있습니다. 이때 소수는 정수로 반올림됩니다. 또는 날짜에서 2008 같은 숫자 값을 추출하여 정수 열에 삽입하는 것도 가능합니다.
- 숫자 데이터 형식은 범위 외 값을 삽입하려고 할 때 오버플로우 조건이 발생할 가능성이 높습니다. 예를 들어 정밀도가 5인 소수 값은 정밀도가 4로 정의된 DECIMAL 열에 맞지 않습니다. 정수 또는 십진수 전체 부분이 잘리지 않습니다. 그러나 십진수의 소수 부분은 필요에 따라 위 또는 아래로 반올림할 수 있습니다. 하지만 테이블에서 선택한 값의 명시적인 변환 결과는 반올림되지 않습니다.
- 서로 다른 유형의 문자열이 호환됩니다. 예를 들어 단일 바이트 데이터가 포함된 VARCHAR 열과 CHAR 열 문자열은 서로 호환이 되어 묵시적으로 변환할 수 있습니다. 멀티바이트 데이터가 포함되는 VARCHAR 문자열은 호환되지 않습니다. 그 밖에 문자열이 적합한 리터럴 값인 경우에는 문자열을 날짜, 시간, 타임스탬프 또는 숫자 값으로 변환할 수도 있습니다. 선행 또는 후행 공백은 무시됩니다. 반대로 날짜, 시간, 타임스탬프 및 숫자 값을 고정 길이 또는 가변 길이 문자열로 변환하는 것도 가능합니다.

> **Note**
>
> 문자열을 숫자 형식으로 변환하려면 문자열에 숫자를 표현한 문자가 있어야 합니다. 예를 들어 `'1.0'`이나 `'5.9'` 같은 문자열은 소수 값으로 변환할 수 있지만 문자열 `'ABC'`는 어떤 숫자 형식으로도 변환할 수 없습니다.

- DECIMAL 값을 문자열과 비교하는 경우는 문자열을 DECIMAL 값으로 변환하려고 AWS Clean Rooms시도합니다. 모든 다른 숫자 값을 문자열과 비교하는 경우 숫자 값이 문자열로 변환됩니다. 반대 변환(예: 문자열을 정수로 변환하거나 DECIMAL 값을 문자열로 변환)을 적용하려면 CAST 함수과 같은 명시적 함수를 사용하세요.
- 64비트 DECIMAL 또는 NUMERIC 값의 정밀도를 높여서 변환하려면 CAST 또는 CONVERT 같은 명시적인 변환 함수를 사용해야 합니다.

묵시적인 변환 형식

묵시적인 변환 유형은 다음과 같이 두 가지입니다.

- 인수의 묵시적 변환(INSERT 또는 UPDATE 명령의 값 설정 등)
- 표현식의 묵시적 변환(WHERE 절의 비교 등)

다음 표는 인수 또는 표현식에서 묵시적으로 변환할 수 있는 데이터 형식을 나열한 것입니다. 그 밖에 명시적인 변환 함수를 통한 변환도 가능합니다.

입력 형식	출력 형식
BIGINT	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	INTEGER
	REAL (FLOAT4)

입력 형식	출력 형식
	SMALLINT 또는 SHORT
	VARCHAR
CHAR	VARCHAR
DATE	CHAR
	VARCHAR
	TIMESTAMP
	TIMESTAMPTZ
DECIMAL (NUMERIC)	BIGINT 또는 LONG
	CHAR
	DOUBLE PRECISION (FLOAT8)
	INTEGER INT)
	REAL (FLOAT4)
	SMALLINT 또는 SHORT
	VARCHAR
DOUBLE PRECISION (FLOAT8)	BIGINT 또는 LONG
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	REAL (FLOAT4)
	SMALLINT 또는 SHORT

입력 형식	출력 형식
	VARCHAR
INTEGER (INT)	BIGINT 또는 LONG
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)
	REAL (FLOAT4)
	SMALLINT 또는 SHORT
	VARCHAR
REAL (FLOAT4)	BIGINT 또는 LONG
	CHAR
	DECIMAL (NUMERIC)
	INTEGER (INT)
	SMALLINT 또는 SHORT
	VARCHAR
SMALLINT	BIGINT 또는 LONG
	BOOLEAN
	CHAR
	DECIMAL (NUMERIC)
	DOUBLE PRECISION (FLOAT8)

입력 형식	출력 형식
	INTEGER (INT)
	REAL (FLOAT4)
	VARCHAR
TIME	VARCHAR
	TIMETZ

> **Note**
>
> DATE, TIME, TIMESTAMP_LTZ, TIMESTAMP_NTZ 또는 문자열 간의 암시적 변환은 현재 세션 시간대를 사용합니다.
> VARBYTE 데이터 유형은 암시적으로 다른 데이터 유형으로 변환될 수 없습니다. 자세한 내용은 CAST 함수 단원을 참조하십시오.

AWS Clean Rooms Spark SQL 명령

Spark SQL에서는 AWS Clean Rooms 다음 SQL 명령이 지원됩니다.

주제

- 캐시 테이블
- 힌트
- SELECT

캐시 테이블

CACHE TABLE 명령은 기존 테이블의 데이터를 캐싱하거나 쿼리 결과가 포함된 새 테이블을 생성하고 캐싱합니다.

> **Note**
>
> 캐시된 데이터는 전체 쿼리에 대해 유지됩니다.

구문, 인수 및 일부 예제는 Apache Spark SQL 참조에서 가져옵니다.

구문

CACHE TABLE 명령은 세 가지 구문 패턴을 지원합니다.

AS 사용(괄호 제외): 쿼리 결과를 기반으로 새 테이블을 생성하고 캐싱합니다.

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

AS 및 괄호 사용: 첫 번째 구문과 유사하게 작동하지만 괄호를 사용하여 쿼리를 명시적으로 그룹화합니다.

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

AS 사용 안 함: SELECT 문을 사용하여 캐싱할 행을 필터링하여 기존 테이블을 캐싱합니다.

```
CACHE TABLE cache_table_identifier query;
```

위치:

- 모든 문은 세미콜론(;)으로 끝나야 합니다.
- query는 일반적으로 SELECT 문입니다.
- 쿼리 주위의 괄호는 AS에서 선택 사항입니다.
- AS 키워드는 선택 사항입니다.

파라미터

cache_table_identifier

캐시된 테이블의 이름입니다. 선택적 데이터베이스 이름 한정자를 포함할 수 있습니다.

AS

쿼리 결과에서 새 테이블을 생성하고 캐싱할 때 사용되는 키워드입니다.

query

캐시할 데이터를 정의하는 SELECT 문 또는 기타 쿼리입니다.

예시

다음 예제에서는 캐시된 테이블이 전체 쿼리에 대해 유지됩니다. 캐싱 후 *cache_table_identifier*를 참조하는 후속 쿼리는 *sourceTable*을 다시 계산하거나 읽는 대신 캐시된 버전에서 읽습니다. 이렇게 하면 자주 액세스하는 데이터의 쿼리 성능이 향상될 수 있습니다.

쿼리 결과에서 필터링된 테이블 생성 및 캐시

첫 번째 예제에서는 쿼리 결과에서 새 테이블을 생성하고 캐싱하는 방법을 보여줍니다. 이 명령은 SELECT 문 주위에 괄호 없이 AS 키워드를 사용합니다. 상태가 '인 'cache_table_identifier'의 행만 포함하는 'sourceTable'라는 새 테이블을 생성합니다active'. 쿼리를 실행하고, 결과를 새 테이블에 저장하고, 새 테이블의 콘텐츠를 캐싱합니다. 원래 'sourceTable'는 변경되지 않으며 캐시된 데이터를 사용하려면 후속 쿼리가 'cache_table_identifier'를 참조해야 합니다.

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

괄호로 묶인 SELECT 문을 사용하여 쿼리 결과 캐시

두 번째 예제에서는 SELECT 문 주위에 괄호를 사용하여 쿼리 결과를 지정된 이름(cache_table_identifier)의 새 테이블로 캐싱하는 방법을 보여줍니다. 이 명령은 상태가 '인 'cache_table_identifier'의 행만 포함하는 'sourceTable'이라는 새 테이블을 생성합니다active'. 쿼리를 실행하고, 결과를 새 테이블에 저장하고, 새 테이블의 콘텐츠를 캐싱합니다. 원래 'sourceTable'는 변경되지 않습니다. 후속 쿼리는 캐시된 데이터를 사용하려면 'cache_table_identifier'를 참조해야 합니다.

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

필터 조건으로 기존 테이블 캐시

세 번째 예제에서는 다른 구문을 사용하여 기존 테이블을 캐싱하는 방법을 보여줍니다. 'AS' 키워드와 괄호를 생략하는이 구문은 일반적으로 새 테이블을 생성하는 대신 'cache_table_identifier'이라는 기존 테이블에서 지정된 행을 캐싱합니다. SELECT 문은 캐시할 행을 결정하는 필터 역할을 합니다.

> **Note**
>
> 이 구문의 정확한 동작은 데이터베이스 시스템에 따라 다릅니다. 항상 특정 AWS 서비스에 대해 올바른 구문을 확인합니다.

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

힌트

SQL 분석을 위한 힌트는 쿼리 실행 전략을 안내하는 최적화 지침을 AWS Clean Rooms제공하므로 쿼리 성능을 개선하고 컴퓨팅 비용을 절감할 수 있습니다. 힌트는 Spark 분석 엔진이 실행 계획을 생성하는 방법을 제안합니다.

구문

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

힌트는 주석 스타일 구문을 사용하여 SQL 쿼리에 포함되며 SELECT 키워드 바로 뒤에 배치해야 합니다.

지원되는 힌트 유형

AWS Clean Rooms 는 조인 힌트와 파티셔닝 힌트라는 두 가지 범주의 힌트를 지원합니다.

주제

- 조인 힌트
- 파티셔닝 힌트

조인 힌트

조인 힌트는 쿼리 실행을 위한 조인 전략을 제안합니다. 구문, 인수 및 일부 예제는 자세한 내용은 Apache Spark SQL 참조에서 가져옵니다.

브로드캐스트

브로드캐스트 조인을 AWS Clean Rooms 사용하는를 제안합니다. 힌트가 있는 조인 측은 autoBroadcastJoinThreshold와 관계없이 브로드캐스트됩니다. 조인의 양쪽에 브로드캐스트 힌트가 있는 경우 크기가 더 작은 조인(통계 기준)이 브로드캐스트됩니다.

별칭: BROADCASTJOIN, MAPJOIN

파라미터: 테이블 식별자(선택 사항)

예:

```
-- 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

셔플 정렬 병합 조인을 AWS Clean Rooms 사용하는 제안을 합니다.

별칭: SHUFFLE_MERGE, MERGEJOIN

파라미터: 테이블 식별자(선택 사항)

예:

```
-- 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

셔플 해시 조인을 AWS Clean Rooms 사용하는 방법을 제안합니다. 양쪽에 셔플 해시 힌트가 있는 경우 쿼리 옵티마이저는 더 작은 쪽(통계 기반)을 빌드 쪽으로 선택합니다.

파라미터: 테이블 식별자(선택 사항)

예:

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

SHUFFLE_REPLICATE_NL

shuffle-and-replicate 중첩 루프 조인을 AWS Clean Rooms 사용하는 방법을 제안합니다.

파라미터: 테이블 식별자(선택 사항)

예:

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

Spark SQL의 힌트 문제 해결

다음 표에는 SparkSQL에서 힌트가 적용되지 않는 일반적인 시나리오가 나와 있습니다. 자세한 내용은 the section called “고려 사항 및 제한 사항” 섹션을 참조하세요.

사용 사례	쿼리 예제
테이블 참조를 찾을 수 없음	`SELECT /*+ BROADCAST(fake_table) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

사용 사례	쿼리 예제
조인 작업에 참여하지 않는 테이블	`SELECT /*+ BROADCAST(s) */ * FROM students s WHERE s.age > 25;`
중첩 하위 쿼리의 테이블 참조	`SELECT /*+ BROADCAST(s) */ * FROM employees e INNER JOIN (SELECT * FROM students s WHERE s.age > 20) sub ON e.eid = sub.sid;`
테이블 참조 대신 열 이름	`SELECT /*+ BROADCAST(e.eid) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
필수 파라미터가 없는 힌트	`SELECT /*+ BROADCAST */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`
테이블 별칭 대신 기본 테이블 이름	`SELECT /*+ BROADCAST(employees) */ * FROM employees e INNER JOIN students s ON e.eid = s.sid;`

파티셔닝 힌트

파티셔닝 힌트는 실행기 노드 간의 데이터 분산을 제어합니다. 여러 파티셔닝 힌트가 지정되면 여러 노드가 논리적 계획에 삽입되지만 최적화 프로그램이 가장 왼쪽에 있는 힌트를 선택합니다.

COALESCE

파티션 수를 지정된 파티션 수로 줄입니다.

파라미터: 숫자 값(필수) - 1~2147483647 사이의 양의 정수여야 합니다.

예:

```
-- Reduce to 5 partitions
```

```
SELECT /*+ COALESCE(5) */ employee_id, salary
FROM employees;
```

REPARTITION

지정된 파티셔닝 표현식을 사용하여 지정된 수의 파티션으로 데이터를 재파티션합니다. 라운드 로빈 배포를 사용합니다.

파라미터:

- 숫자 값(선택 사항) - 파티션 수. 1~2147483647 사이의 양의 정수여야 합니다.
- 열 식별자(선택 사항) - 분할 기준 열입니다. 이러한 열은 입력 스키마에 있어야 합니다.
- 둘 다 지정하면 숫자 값이 먼저 와야 합니다.

예:

```
-- 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

지정된 열에서 범위 파티셔닝을 사용하여 지정된 수의 파티션으로 데이터를 재파티션합니다.

파라미터:

- 숫자 값(선택 사항) - 파티션 수. 1~2147483647 사이의 양의 정수여야 합니다.
- 열 식별자(선택 사항) - 분할 기준 열입니다. 이러한 열은 입력 스키마에 있어야 합니다.
- 둘 다 지정하면 숫자 값이 먼저 와야 합니다.

예:

```
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;
```

재분배

쿼리 결과 출력 파티션을 리밸런싱하여 모든 파티션의 크기가 적절한지(너무 작지도 크지도 않음) 확인합니다. 이는 최선의 작업입니다. 스큐가 있는 경우 AWS Clean Rooms 는 스큐된 파티션을 분할하여 너무 크지 않도록 합니다. 이 힌트는 너무 작거나 큰 파일을 방지하기 위해 쿼리 결과를 테이블에 작성해야 하는 경우에 유용합니다.

파라미터:

- 숫자 값(선택 사항) - 파티션 수. 1~2147483647 사이의 양의 정수여야 합니다.
- 열 식별자(선택 사항) - SELECT 출력 목록에 열이 나타나야 합니다.
- 둘 다 지정하면 숫자 값이 먼저 와야 합니다.

예:

```
-- 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;
```

여러 힌트 결합

쉼표로 구분하여 단일 쿼리에 여러 힌트를 지정할 수 있습니다.

```
-- 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;
```

고려 사항 및 제한 사항

- 힌트는 명령이 아닌 최적화 제안입니다. 쿼리 최적화 프로그램은 리소스 제약 조건 또는 실행 조건에 따라 힌트를 무시할 수 있습니다.
- 힌트는 CreateAnalysisTemplate 및 StartProtectedQuery APIs 모두에 대한 SQL 쿼리 문자열에 직접 포함됩니다.
- 힌트는 SELECT 키워드 바로 뒤에 배치해야 합니다.
- 명명된 파라미터는 힌트에서 지원되지 않으며 예외가 발생합니다.
- REPARTITION amd REPARTITION_BY_RANGE 힌트의 열 이름은 입력 스키마에 있어야 합니다.
- REBALANCE 힌트의 열 이름은 SELECT 출력 목록에 나타나야 합니다.
- 숫자 파라미터는 1에서 2147483647 사이의 양의 정수여야 합니다. 1e1과 같은 과학 표기법은 지원되지 않습니다.
- 힌트는 차등 프라이버시 SQL 쿼리에서 지원되지 않습니다.
- SQL 쿼리에 대한 힌트는 PySpark 작업에서 지원되지 않습니다. PySpark 작업에서 실행 계획에 대한 지시문을 제공하려면 데이터 프레임 API를 사용합니다. 자세한 내용은 Apache Spark DataFrame API 문서를 참조하세요.

SELECT

SELECT 명령은 테이블 및 사용자 정의 함수의 행을 반환합니다

AWS Clean RoomsSpark SQL에서는 다음 SELECT SQL 명령, 절 및 집합 연산자가 지원됩니다.

주제

- SELECT list
- WITH 절
- FROM 절
- JOIN 절
- WHERE 절
- VALUES 절
- GROUP BY 절
- HAVING 절
- 집합 연산자
- ORDER BY 절
- 하위 쿼리 예
- 상관관계가 있는 하위 쿼리

구문, 인수 및 일부 예제는 Apache Spark SQL 참조에서 가져옵니다.

SELECT list

SELECT list는 쿼리에서 반환하려는 열, 함수 및 표현식의 이름을 지정합니다. 목록은 쿼리의 출력을 나타냅니다.

구문

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

파라미터

DISTINCT

하나 이상의 열에서 일치하는 값을 바탕으로 결과 집합에서 중복된 행을 제거하는 옵션입니다.

expression

쿼리에서 참조하는 테이블에 존재하는 하나 이상의 열에서 형성되는 표현식입니다. 표현식은 SQL 함수를 포함할 수 있습니다. 예제:

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

AS column_alias

최종 결과 집합에 사용될 열의 임시 이름입니다. AS 키워드는 옵션입니다. 예제:

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

표현식에 대해 단순한 열 이름이 아닌 별칭을 지정하지 않을 경우 결과 집합은 그 열에 기본 이름을 적용합니다.

> **Note**
>
> 별칭은 대상 목록에 정의되는 즉시 인식됩니다. 동일한 대상 목록에 정의된 다른 표현식에는 별칭을 사용할 수 없습니다.

WITH 절

WITH 절은 쿼리에 있는 SELECT 목록에 선행하는 선택적 절입니다. WITH 절은 하나 이상의 common_table_expressions를 정의합니다. 각 공통 테이블 표현식(CTE) 은 뷰 정의와 유사한 임시 테이블을 정의합니다. FROM 절에서 이러한 임시 테이블을 참조할 수 있습니다. 임시 테이블은 자신이 속한 쿼리가 실행되는 동안에만 사용됩니다. WITH 절에 있는 각각의 CTE는 테이블 이름, 열 이름의 선택적 목록, 테이블로 평가되는 쿼리 표현식(SELECT 문)을 지정합니다.

WITH 절 하위 쿼리는 단일 쿼리를 실행하는 내내 사용 가능한 테이블을 정의하는 효율적인 방법입니다. 모든 경우에 있어 SELECT 문의 본문에 하위 쿼리를 사용하여 같은 결과를 얻을 수 있지만, WITH 절 하위 쿼리는 더 간단하게 쓰고 읽을 수 있습니다. 가능한 경우 여러 번 참조되는 WITH 절 하위 쿼리는 공통 하위 표현식으로 최적화됩니다. 즉, WITH 하위 쿼리를 한 번 평가하고 그 결과를 재사용할 수 있습니다. (공통 하위 표현식은 WITH 절에 정의되는 하위 표현식으로 제한되지 않습니다.)

구문

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

여기서 common_table_expression은 비재귀적일 수 있습니다. 다음은 비재귀 형식입니다.

```
CTE_table_name AS ( query )
```

Parameters

common_table_expression

FROM 절에서 참조할 수 있는 임시 테이블을 정의하고 해당 테이블이 속한 쿼리 실행 중에만 사용됩니다.

CTE_table_name

WITH 절 하위 쿼리의 결과를 정의하는 임시 테이블의 고유한 이름입니다. 단일 WITH 절 내에서 중복되는 이름을 사용할 수 없습니다. 각각의 하위 쿼리에는 FROM 절에서 참조될 수 있는 테이블 이름이 주어져야 합니다.

query

에서 AWS Clean Rooms지원하는 모든 SELECT 쿼리입니다. SELECT을(를) 참조하세요.

사용 노트

다음 SQL 문에 WITH 절을 사용할 수 있습니다.

- SELECT, WITH, UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT 또는 EXCEPT ALL

WITH 절을 포함한 쿼리의 FROM 절이 WITH 절로 정의되는 테이블 중 참조하지 않는 테이블이 있을 경우 WITH 절이 무시되고 쿼리가 정상적으로 실행됩니다.

WITH 절 하위 쿼리로 정의되는 테이블은 WITH 절이 시작하는 SELECT 쿼리의 범위에서만 참조될 수 있습니다. 예를 들어, SELECT 목록, WHERE 절 또는 HAVING 절에 있는 하위 쿼리의 FROM 절에서 그와 같은 테이블을 참조할 수 있습니다. 하위 쿼리에 WITH 절을 사용할 수 없고 기본 쿼리 또는 다른 하위 쿼리의 FROM 절에서 WITH 절의 테이블을 참조할 수 없습니다. 이 쿼리 패턴으로 인해 WITH 절 테이블에 대해 `relation table_name doesn't exist` 형식의 오류 메시지가 발생합니다.

WITH 절 하위 쿼리 내에서 다른 WITH 절을 지정할 수 없습니다.

WITH 절 하위 쿼리에 의해 정의되는 테이블에 대한 전방 참조를 할 수 없습니다. 예를 들어, 다음 쿼리는 테이블 W1의 정의에서 테이블 W2에 대한 전방 참조 때문에 오류를 반환합니다.

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

예제

다음 예에서는 WITH 절을 포함하는 쿼리로서 가능한 가장 간단한 사례를 보여줍니다. VENUECOPY라는 이름의 WITH 쿼리는 VENUE 테이블에서 모든 행을 선택합니다. 다음에는 기본 쿼리가 VENUECOPY에서 모든 행을 선택합니다. VENUECOPY 테이블은 이 쿼리의 지속 시간 동안에만 존재합니다.

```
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)
```

다음 예에서는 VENUE_SALES와 TOP_VENUES라는 두 테이블을 생성하는 WITH 절을 보여줍니다. 두 번째 WITH 절 테이블은 첫 번째 WITH 절 테이블에서 선택합니다. 다음에는, 기본 쿼리 블록의 WHERE 절이 TOP_VENUES 테이블을 포함하는 하위 쿼리를 포함합니다.

```
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)
```

다음 두 예에서는 WITH 절 하위 쿼리를 기반으로 테이블 참조의 범위에 대한 규칙을 보여줍니다. 첫 번째 쿼리가 실행되지만 두 번째 쿼리는 예상된 오류가 발생하며 실패합니다. 첫 번째 쿼리는 기본 쿼리의 SELECT 목록 내에 WITH 절 하위 쿼리가 있습니다. WITH 절(HOLIDAYS)에 의해 정의되는 테이블은 SELECT 목록에 있는 하위 쿼리의 FROM 절에 참조됩니다.

```
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)
```

기본 쿼리뿐 아니라 SELECT 목록 하위 쿼리에서 HOLIDAYS 테이블 참조를 시도하므로 두 번째 쿼리는 실패합니다. 기본 쿼리 참조는 범위를 벗어나는 주제입니다.

```
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 절

쿼리의 FROM 절은 데이터가 선택되는 테이블 참조(테이블, 뷰, 하위 쿼리)를 나열합니다. 여러 개의 테이블 참조가 목록에 표시되는 경우 FROM 절 또는 WHERE 절에서 알맞은 구문을 사용하여 테이블을 조인해야 합니다. 조인 기준이 지정되지 않은 경우 시스템에서는 쿼리를 크로스 조인(데카르트 곱)으로 처리합니다.

주제

- 구문
- Parameters
- 사용 노트

구문

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

여기서 table_reference는 다음 중 하나입니다.

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

Parameters

with_subquery_table_name

WITH 절에서 하위 쿼리에 의해 정의되는 테이블입니다.

table_name

테이블 또는 뷰의 이름입니다.

별칭

테이블 또는 뷰의 임시 대체 이름입니다. 하위 쿼리에서 파생되는 테이블에 대해 별칭을 입력해야 합니다. 다른 테이블 참조에서 별칭은 옵션입니다. AS 키워드는 항상 옵션입니다. 테이블 별칭은 WHERE 절과 같이 쿼리의 다른 부분에 있는 테이블을 편리하게 식별하는 바로 가기의 역할을 합니다.

예제:

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

정의된 테이블 별칭을 정의한 경우 쿼리에서 해당 테이블을 참조할 때 별칭을 사용해야 합니다.

예를 들어 쿼리가 `SELECT "tbl"."col" FROM "tbl" AS "t"`와 같으면 이제 테이블 이름이 기본적으로 재정의되므로 쿼리가 실패합니다. 이 경우 유효한 쿼리는 `SELECT "t"."col" FROM "tbl" AS "t"`와 같습니다.

column_alias

테이블 또는 뷰에 있는 열의 임시 대체 이름입니다.

subquery

테이블로 평가되는 쿼리 표현식입니다. 테이블은 쿼리의 지속 시간 동안만 존재하며 일반적으로 이름 또는 별칭이 주어집니다. 그러나 별칭이 필수는 아닙니다. 하위 쿼리에서 파생되는 테이블의 열

이름을 정의할 수도 있습니다. 하위 쿼리의 결과를 다른 테이블에 조인하고 쿼리의 다른 곳에서 열을 선택하거나 제한하려는 경우 열 별칭의 이름 지정이 중요합니다.

하위 쿼리는 ORDER BY 절을 포함할 수 있지만, LIMIT 또는 OFFSET 절도 지정하지 않으면 ORDER BY 절이 아무런 효과도 없을 수 있습니다.

NATURAL

두 테이블에서 조인 열로서 똑같이 명명된 열의 쌍을 전부 자동으로 사용하는 조인을 정의합니다. 명시적 조인 조건은 필요하지 않습니다. 예를 들어, CATEGORY 및 EVENT 테이블에 모두 CATID로 명명된 열이 있는 경우 이러한 테이블의 자연 조인은 CATID 열에 적용되는 조인입니다.

> **Note**
>
> NATURAL 조인이 지정되어 있지만 조인되는 테이블에 똑같은 이름의 열 쌍이 존재하지 않는 경우 쿼리는 기본적으로 크로스 조인이 됩니다.

join_type

다음과 같은 조인 유형 중 하나를 지정합니다.

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

크로스 조인은 정규화되지 않은 조인으로, 두 테이블의 데카르트 곱을 반환합니다.

내부 및 외부 조인은 정규화된 조인입니다. 이런 조인은 FROM 절에서 ON 또는 USING 구문으로 암시적으로(자연 조인으로) 정규화되거나 WHERE 절 조건으로 암시적으로 정규화됩니다.

내부 조인은 조인 조건이나 조인 열의 목록을 기반으로 일치하는 행만 반환합니다. 외부 조인은 동등한 내부 조인이 반환하는 모든 행과 "왼쪽" 테이블, "오른쪽" 테이블 또는 두 테이블 모두에서 일치하지 않는 행을 반환합니다. 왼쪽 테이블은 처음에 목록으로 표시되는 테이블이고, 오른쪽 테이블은 두 번째로 목록으로 표시되는 테이블입니다. 일치하지 않는 행은 출력 열의 간격을 채우기 위해 NULL 값을 포함합니다.

ON join_condition

조인 열이 ON 키워드 뒤에 나오는 조건으로 규정되는 조인 사양의 유형입니다. 예:

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

USING (join_column [, ...])

조인 열이 괄호 안에 묶여 표시되는 조인 사양의 유형입니다. 여러 개의 조인 열이 지정되어 있는 경우 이런 열은 쉼표로 구분됩니다. USING 키워드는 목록에 선행해야 합니다. 예:

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

사용 노트

조인 열은 비교 가능한 데이터 형식이 있어야 합니다.

NATURAL 또는 USING 조인은 중간 결과 집합에 조인 열의 각 쌍 중 하나만 유지합니다.

ON 구문이 있는 조인은 중간 결과 집합에 두 조인 열을 모두 유지합니다.

또한 WITH 절 섹션도 참조하세요.

JOIN 절

SQL JOIN 절은 공통 필드를 기반으로 두 개 이상의 테이블에서 데이터를 결합하는 데 사용됩니다. 지정된 조인 메서드에 따라 결과가 변경될 수도 있고 변경되지 않을 수도 있습니다. 왼쪽 및 오른쪽 외부 조인은 다른 테이블에서 일치 항목이 발견되지 않을 때 조인된 테이블 중 하나에서 값을 유지합니다.

JOIN 유형과 조인 조건의 조합에 따라 최종 결과 집합에 포함되는 행이 결정됩니다. 그런 다음 SELECT 및 WHERE 절은 반환되는 열과 행을 필터링하는 방법을 제어합니다. 다양한 JOIN 유형과 이를 효과적으로 사용하는 방법을 이해하는 것은 SQL에서 매우 중요한 기술입니다. 유연하고 강력한 방식으로 여러 테이블의 데이터를 결합할 수 있기 때문입니다.

구문

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

파라미터

SELECT column1, column2, ..., columnN

결과 집합에 포함할 열입니다. JOIN과 관련된 테이블 중 하나 또는 둘 다에서 열을 선택할 수 있습니다.

FROM 테이블1

JOIN 작업의 첫 번째(왼쪽) 테이블입니다.

[조인 | 내부 조인 | 왼쪽 [외부] 조인 | 오른쪽 [외부] 조인 | 전체 [외부] 조인] 테이블2:

수행할 JOIN의 유형입니다. JOIN 또는 INNER JOIN은 두 테이블에서 일치하는 값이 있는 행만 반환합니다.

왼쪽 [OUTER] JOIN은 왼쪽 테이블의 모든 행을 오른쪽 테이블의 일치하는 행과 함께 반환합니다.

RIGHT [OUTER] JOIN은 왼쪽 테이블의 일치하는 행과 함께 오른쪽 테이블의 모든 행을 반환합니다.

FULL [OUTER] JOIN은 일치하는 항목이 있는지 여부에 관계없이 두 테이블의 모든 행을 반환합니다.

CROSS JOIN은 두 테이블에서 행의 데카르트 곱을 생성합니다.

ON table1.column = table2.column

두 테이블의 행이 일치하는 방식을 지정하는 조인 조건입니다. 조인 조건은 하나 이상의 열을 기반으로 할 수 있습니다.

WHERE 조건:

지정된 조건에 따라 결과 집합을 추가로 필터링하는 데 사용할 수 있는 선택적 절입니다.

예제

다음 예는 USING 절을 사용한 두 테이블 간의 조인입니다. 이 경우 listid 및 eventid 열이 조인 열로 사용됩니다. 결과는 5개 행으로 제한됩니다.

```
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
```

조인 유형

INNER

기본 조인 유형입니다. 두 테이블 참조에서 일치하는 값이 있는 행을 반환합니다.

INNER JOIN은 SQL에서 사용되는 가장 일반적인 유형의 조인입니다. 공통 열 또는 열 집합을 기반으로 여러 테이블의 데이터를 결합하는 강력한 방법입니다.

구문:

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

다음 쿼리는 고객과 주문 테이블 간에 일치하는 customer_id 값이 있는 모든 행을 반환합니다. 결과 세트에는 customer_id, name, order_id 및 order_date 열이 포함됩니다.

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

다음 쿼리는 LISTING 테이블과 SALES 테이블 간의 내부 조인(JOIN 키워드 사용 안 함)이며, 여기서 LISTING 테이블의 LISTID는 1에서 5 사이입니다. 이 쿼리는 LISTING 테이블(왼쪽 테이블) 및 SALES 테이블(오른쪽 테이블)에 있는 LISTID 열 값과 일치합니다. 결과는 LISTID 1, 4, 5가 조건과 일치한다는 것을 보여줍니다.

```
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
```

다음 예는 ON 절과의 내부 조인입니다. 이 경우 NULL 행은 반환되지 않습니다.

```
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
```

다음 쿼리는 FROM 절에 있는 두 하위 쿼리의 내부 조인입니다. 다음 쿼리는 다양한 범주의 이벤트(콘서트 및 쇼)에 대해 판매된 티켓과 판매되지 않은 티켓의 수를 찾습니다. 이러한 FROM 절 하위 쿼리는 table 하위 쿼리로서, 여러 개의 열과 행을 반환할 수 있습니다.

```
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
```

왼쪽 [외부]

왼쪽 테이블 참조의 모든 값과 오른쪽 테이블 참조의 일치하는 값을 반환하거나 일치 항목이 없는 경우 NULL을 추가합니다. 이를 왼쪽 외부 조인이라고도 합니다.

왼쪽(첫 번째) 테이블의 모든 행과 오른쪽(두 번째) 테이블의 일치하는 행을 반환합니다. 오른쪽 테이블에 일치하는 항목이 없는 경우 결과 집합에는 오른쪽 테이블의 열에 대한 NULL 값이 포함됩니다. OUTER 키워드는 생략할 수 있으며 조인은 단순히 LEFT JOIN으로 작성할 수 있습니다. 왼쪽 외부 조인의 반대는 오른쪽 테이블의 모든 행과 왼쪽 테이블의 일치하는 행을 반환하는 오른쪽 외부 조인입니다.

구문:

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

다음 쿼리는 주문 테이블의 일치하는 행과 함께 고객 테이블의 모든 행을 반환합니다. 고객에게 주문이 없는 경우 결과 세트에는 order_id 및 order_date 열에 대한 NULL 값과 함께 해당 고객의 정보가 계속 포함됩니다.

```
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;
```

다음 쿼리는 왼쪽 외부 조인입니다. 왼쪽 및 오른쪽 외부 조인은 다른 테이블에서 일치 항목이 발견되지 않을 때 조인된 테이블 중 하나에서 값을 유지합니다. 왼쪽 및 오른쪽 테이블은 구문에 나열되는 첫 번째 및 두 번째 테이블입니다. NULL 값은 결과 집합의 "간격"을 채우는 데 사용됩니다. 이 쿼리는 LISTING 테이블(왼쪽 테이블) 및 SALES 테이블(오른쪽 테이블)에 있는 LISTID 열 값과 일치합니다. 결과는 LISTIDs 2와 3이 판매를 초래하지 않았음을 보여줍니다.

```
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
```

오른쪽 [외부]

오른쪽 테이블 참조의 모든 값과 왼쪽 테이블 참조의 일치하는 값을 반환하거나 일치 항목이 없는 경우 NULL을 추가합니다. 이를 오른쪽 외부 조인이라고도 합니다.

오른쪽(초) 테이블의 모든 행과 왼쪽(첫 번째) 테이블의 일치하는 행을 반환합니다. 왼쪽 테이블에 일치하는 항목이 없는 경우 결과 세트에는 왼쪽 테이블의 열에 대한 NULL 값이 포함됩니다. OUTER 키워드는 생략할 수 있으며 조인은 단순히 오른쪽 조인으로 작성할 수 있습니다. 오른쪽 외부 조인의 반대편은 왼쪽 테이블의 모든 행과 오른쪽 테이블의 일치하는 행을 반환하는 왼쪽 외부 조인입니다.

구문:

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

다음 쿼리는 주문 테이블의 일치하는 행과 함께 고객 테이블의 모든 행을 반환합니다. 고객에게 주문이 없는 경우 결과 세트에는 order_id 및 order_date 열에 대한 NULL 값과 함께 해당 고객의 정보가 계속 포함됩니다.

```
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;
```

다음 쿼리는 오른쪽 외부 조인입니다. 이 쿼리는 LISTING 테이블(왼쪽 테이블) 및 SALES 테이블(오른쪽 테이블)에 있는 LISTID 열 값과 일치합니다. 결과는 LISTID 1, 4, 5가 조건과 일치한다는 것을 보여줍니다.

```
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
```

전체 [외부]

두 관계의 모든 값을 반환하고 일치하는 값이 없는 측에 NULL 값을 추가합니다. 이를 전체 외부 조인이라고도 합니다.

일치 항목이 있는지 여부에 관계없이 왼쪽 테이블과 오른쪽 테이블 모두에서 모든 행을 반환합니다. 일치하는 항목이 없는 경우 결과 집합에는 일치하는 행이 없는 테이블의 열에 대한 NULL 값이 포함됩니다. OUTER 키워드는 생략할 수 있으며 조인은 단순히 FULL JOIN으로 작성할 수 있습니다. FULL OUTER JOIN은 LEFT OUTER JOIN 또는 RIGHT OUTER JOIN보다 덜 일반적으로 사용되지만 일치하는 항목이 없더라도 두 테이블의 모든 데이터를 확인해야 하는 특정 시나리오에서 유용할 수 있습니다.

구문:

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

다음 쿼리는 고객 및 주문 테이블의 모든 행을 반환합니다. 고객에게 주문이 없는 경우 결과 세트에는 order_id 및 order_date 열에 대한 NULL 값과 함께 해당 고객의 정보가 계속 포함됩니다. 주문에 연결된 고객이 없는 경우 결과 세트에는 customer_id 및 이름 열에 대한 NULL 값과 함께 해당 주문이 포함됩니다.

```
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;
```

다음 쿼리는 전체 조인입니다. 전체 조인은 다른 테이블에서 일치 항목이 발견되지 않을 때 조인된 테이블의 값을 유지합니다. 왼쪽 및 오른쪽 테이블은 구문에 나열되는 첫 번째 및 두 번째 테이블입니다. NULL 값은 결과 집합의 "간격"을 채우는 데 사용됩니다. 이 쿼리는 LISTING 테이블(왼쪽 테이블) 및 SALES 테이블(오른쪽 테이블)에 있는 LISTID 열 값과 일치합니다. 결과는 LISTIDs 2와 3이 판매를 초래하지 않았음을 보여줍니다.

```
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
```

다음 쿼리는 전체 조인입니다. 이 쿼리는 LISTING 테이블(왼쪽 테이블) 및 SALES 테이블(오른쪽 테이블)에 있는 LISTID 열 값과 일치합니다. 판매로 이어지지 않는 행(LISTID 2 및 3)만 결과에 있습니다.

```
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
```

[왼쪽] 세미

오른쪽과 일치하는 테이블 참조의 왼쪽에서 값을 반환합니다. 이를 왼쪽 세미 조인이라고도 합니다.

왼쪽(첫 번째) 테이블에서 오른쪽(두 번째) 테이블에 일치하는 행이 있는 행만 반환합니다. 오른쪽 테이블의 열은 반환하지 않고 왼쪽 테이블의 열만 반환합니다. 왼쪽 SEMI JOIN은 두 번째 테이블에서 데

이터를 반환할 필요 없이 한 테이블에서 일치하는 행을 다른 테이블에서 찾으려는 경우에 유용합니다. LEFT SEMI JOIN은 IN 또는 EXISTS 절과 함께 하위 쿼리를 사용하는 것보다 더 효율적인 대안입니다.

구문:

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

다음 쿼리는 주문 테이블에 주문이 하나 이상 있는 고객에 대해 고객 테이블의 customer_id 및 이름 열만 반환합니다. 결과 세트에는 주문 테이블의 열이 포함되지 않습니다.

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

CROSS JOIN

두 관계의 데카르트 곱을 반환합니다. 즉, 결과 집합에는 조건이나 필터가 적용되지 않고 두 테이블에서 가능한 모든 행 조합이 포함됩니다.

CROSS JOIN은 고객 및 제품 정보의 가능한 모든 조합을 표시하는 보고서를 생성하는 경우와 같이 두 테이블에서 가능한 모든 데이터 조합을 생성해야 하는 경우에 유용합니다. 교차 조인은 ON 절에서 조인 조건이 없기 때문에 다른 조인 유형(내부 조인, 왼쪽 조인 등)과 다릅니다. CROSS JOIN에는 조인 조건이 필요하지 않습니다.

구문:

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

다음 쿼리는 고객 및 제품 테이블에서 customer_id, customer_name, product_id 및 product_name의 가능한 모든 조합을 포함하는 결과 세트를 반환합니다. 고객 테이블에 10개의 행이 있고 제품 테이블에 20개의 행이 있는 경우 CROSS JOIN의 결과 집합에는 10 x 20 = 200개의 행이 포함됩니다.

```
SELECT customers.customer_id, customers.name, products.product_id,
 products.product_name
FROM customers
```

```
CROSS JOIN products;
```

다음 쿼리는 결과를 제한하는 술어가 있는, LISTING 테이블과 SALES 테이블의 교차 조인 또는 데카르트 조인입니다. 이 쿼리는 SALES 테이블과 LISTING 테이블의 LISTID(두 테이블 모두 LISTID 1, 2, 3, 4, 5) 열 값과 일치합니다. 결과는 20개의 행이 조건과 일치한다는 것을 보여줍니다.

```
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
```

앤티 조인

오른쪽 테이블 참조와 일치하지 않는 왼쪽 테이블 참조의 값을 반환합니다. 이를 왼쪽 안티 조인이라고도 합니다.

ANTI JOIN은 한 테이블에서 일치하는 행이 없는 행을 다른 테이블에서 찾으려는 경우에 유용한 작업입니다.

구문:

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

다음 쿼리는 주문하지 않은 모든 고객을 반환합니다.

```
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

두 관계의 행이 이름이 일치하는 모든 열에 대해 동등하게 암시적으로 일치되도록 지정합니다.

두 테이블 간에 동일한 이름 및 데이터 형식의 열을 자동으로 일치시킵니다. ON 절에서 조인 조건을 명시적으로 지정할 필요는 없습니다. 두 테이블 간의 일치하는 모든 열을 결과 집합에 결합합니다.

NATURAL JOIN은 조인하려는 테이블에 동일한 이름과 데이터 형식의 열이 있는 경우 편리한 간편성입니다. 그러나 일반적으로 보다 명시적인 INNER JOIN을 사용하는 것이 좋습니다. 조인 조건을 보다 명시적이고 이해하기 쉽게 만드는 ON 구문입니다.

구문:

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

다음 예제는 다음 열과 `departments`함께 두 테이블, `employees` 및 간의 자연 조인입니다.

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

다음 쿼리는 `department_id` 열을 기반으로 두 테이블 간에 일치하는 모든 행의 이름, 성 및 부서 이름을 포함하는 결과 세트를 반환합니다.

```
SELECT e.first_name, e.last_name, d.department_name
```

```
FROM employees e
NATURAL JOIN departments d;
```

다음 예는 두 테이블 간의 자연 조인입니다. 이 경우 listid, sellerid, eventid, dateid 열은 두 테이블 모두에서 동일한 이름과 데이터 형식을 가지므로 조인 열로 사용됩니다. 결과는 5개 행으로 제한됩니다.

```
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 절

WHERE 절은 테이블을 조인하거나 테이블의 열에 조건자를 적용하는 조건을 포함합니다. WHERE 절 또는 FROM 절에서 알맞은 구문을 사용하여 테이블을 내부 조인할 수 있습니다. FROM 절에는 외부 조인 기준을 지정해야 합니다.

구문

```
[ WHERE condition ]
```

condition

테이블 열에서 조인 조건 또는 조건자 같이, 부울 결과를 포함한 임의의 검색 조건입니다. 다음 예는 유효한 조인 조건입니다.

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

다음 예는 테이블의 열에 유효한 조건입니다.

```
catgroup like 'S%'
venueseats between 20000 and 50000
```

```
eventname in('Jersey Boys','Spamalot')
year=2008
length(catdesc)>25
date_part(month, caldate)=6
```

조건은 단순하거나 복잡할 수 있는데, 복잡한 조건의 경우 괄호를 사용하여 논리 단위를 분리할 수 있습니다. 다음 예에서는 조인 조건이 괄호로 묶여 있습니다.

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

사용 노트

select list 표현식을 참조하기 위해 WHERE 절의 별칭을 사용할 수 있습니다.

WHERE 절에서 집계 함수의 결과를 제한할 수 없습니다. 결과를 제한하려면 HAVING 절을 사용하십시오.

WHERE 절에서 제한되는 열은 FROM 절의 테이블 참조로부터 파생해야 합니다.

예제

다음 쿼리는 SALES 및 EVENT 테이블에 대한 조인 조건, EVENTNAME 열의 조건자, STARTTIME 열의 두 조건자를 비롯한 다양한 WHERE 절 제한 사항의 조합을 사용합니다.

```
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 절

VALUES 절은 테이블을 참조할 필요 없이 쿼리에 직접 행 값 집합을 제공하는 데 사용됩니다.

VALUES 절은 다음 시나리오에서 사용할 수 있습니다.

- INSERT INTO 문에서 VALUES 절을 사용하여 테이블에 삽입되는 새 행의 값을 지정할 수 있습니다.
- 테이블을 참조할 필요 없이 VALUES 절을 단독으로 사용하여 임시 결과 집합 또는 인라인 테이블을 생성할 수 있습니다.
- VALUES 절을 WHERE, ORDER BY 또는 LIMIT과 같은 다른 SQL 절과 결합하여 결과 집합의 행을 필터링, 정렬 또는 제한할 수 있습니다.

이 절은 영구 테이블을 생성하거나 참조할 필요 없이 SQL 문에 직접 작은 데이터 세트를 삽입, 쿼리 또는 조작해야 할 때 특히 유용합니다. 이를 통해 열 이름과 각 행에 해당하는 값을 정의할 수 있으므로 별도의 테이블을 관리하는 오버헤드 없이 임시 결과 세트를 생성하거나 데이터를 즉시 삽입할 수 있습니다.

구문

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

파라미터

expression

값을 생성하는 하나 이상의 값, 연산자 및 SQL 함수의 조합을 지정하는 표현식입니다.

테이블_별칭

선택적 열 이름 목록을 사용하여 임시 이름을 지정하는 별칭입니다.

예제

다음 예제에서는 두 개의 열, col1 및가 있는 인라인 테이블, 임시 테이블과 유사한 결과 세트를 생성합니다col2. 결과 집합의 단일 행에는 1각각 "one" 및 값이 포함됩니다. 쿼리 SELECT * FROM 부분은이 임시 결과 집합에서 모든 열과 행을 검색하기만 하면 됩니다. VALUES 절이 열 이름을 명시적으로 지정하지 않기 때문에 열 이름(col1 및 col2)은 데이터베이스 시스템에서 자동으로 생성됩니다.

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

사용자 지정 열 이름을 정의하려면 다음과 같이 VALUES 절 뒤에 AS 절을 사용하여 정의할 수 있습니다.

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

이렇게 하면 기본 `name` 및 `id`대신 열 이름이 `col1` 및 인 임시 결과 집합이 생성됩니다`col2`.

GROUP BY 절

GROUP BY 절은 쿼리에 대한 그룹화 열을 식별합니다. 쿼리가 SUM, AVG 및 COUNT와 같은 표준 함수로 집계를 계산할 때 그룹화 열을 선언해야 합니다. 집계 함수가 SELECT 표현식에 있는 경우 집계 함수에는 없는 SELECT 표현식의 모든 열이 GROUP BY 절에 있어야 합니다.

자세한 내용은 AWS Clean Rooms Spark SQL 함수 단원을 참조하십시오.

구문

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

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

파라미터

expr

열 또는 표현식의 목록은 쿼리의 선택 목록에 있는 비집계 표현식의 목록과 일치해야 합니다. 예를 들어, 다음과 같이 간단한 쿼리를 생각해 보세요.

```
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)
```

이 쿼리에서 선택 목록은 2개의 집계 표현식으로 구성됩니다. 첫 번째 표현식은 SUM 함수를 사용하고 두 번째 표현식은 COUNT 함수를 사용합니다. 나머지 두 개의 열 LISTID 및 EVENTID를 그룹화 열로 선언해야 합니다.

GROUP BY 절에서 표현식은 서수를 사용하여 선택 목록을 참조할 수도 있습니다. 예를 들어, 이전 예는 다음과 같이 줄일 수도 있습니다.

```
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

집계 확장 ROLLUP을 사용하여 단일 문에서 여러 GROUP BY 작업을 수행할 수 있습니다. 집계 확장 및 관련 기능에 대한 자세한 내용은 집계 확장을 참조하세요.

집계 확장

AWS Clean Rooms는 집계 확장을 지원하여 단일 문에서 여러 GROUP BY 작업을 수행합니다.

GROUPING SETS

단일 명령문에서 하나 이상의 그룹화 집합을 계산합니다. 그룹화 집합은 쿼리의 결과 집합을 그룹화할 수 있는 0개 이상의 열 집합인 단일 GROUP BY 절의 집합입니다. 집합을 그룹화하여 그룹화하는 것은 서로 다른 열로 그룹화된 하나의 결과 집합에서 UNION ALL 쿼리를 실행하는 것과 같습니다. 예를 들어, GROUP BY GROUPING SETS((a), (b))는 GROUP BY a UNION ALL GROUP BY b와 동일합니다.

다음 예에서는 제품 카테고리와 판매된 제품 종류에 따라 그룹화된 주문 테이블 제품의 비용을 반환합니다.

```
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

이전 열이 후속 열의 부모로 간주되는 계층 구조를 가정합니다. ROLLUP은 제공된 열을 기준으로 데이터를 그룹화하여 그룹화된 행 외에 그룹화 열의 모든 수준에서 총계를 나타내는 추가 소계 행을 반환합니다. 예를 들어 GROUP BY ROLLUP ((a), (b) 를 사용하여 b가 a의 하위 섹션이라고 가정하면서 먼저 a로 그룹화된 다음 b로 그룹화된 결과 집합을 반환할 수 있습니다. ROLLUP은 또한 열을 그룹화하지 않고 전체 결과 집합이 있는 행을 반환합니다.

GROUP BY ROLLUP((a), (b))는 GROUP BY GROUPING SETS((a,b), (a), ())와 같습니다.

다음 예는 먼저 범주별로 그룹화된 주문 테이블의 제품 비용을 반환한 다음 제품을 범주의 하위 부문으로 사용하여 반환합니다.

```
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

제공된 열을 기준으로 데이터를 그룹화하여 그룹화된 행 외에 그룹화 열의 모든 수준에서 합계를 나타내는 추가 소계 행을 반환합니다. CUBE는 ROLLUP과 동일한 행을 반환하는 동시에 ROLLUP에서 다루지 않는 그룹화 열의 모든 조합에 대해 소계 행을 추가합니다. 예를 들어 GROUP BY CUBE ((a), (b)를 사용하여 b가 a의 하위 섹션이고 그 다음 b만으로 그룹화된 결과 집합을 반환할 수 있습니다. CUBE는 또한 열을 그룹화하지 않고 전체 결과 집합이 있는 행을 반환합니다.

GROUP BY CUBE((a), (b))는 GROUP BY GROUPING SETS((a, b), (a), (b), ())와 같습니다.

다음 예는 먼저 범주별로 그룹화된 주문 테이블의 제품 비용을 반환한 다음 제품을 범주의 하위 부문으로 사용하여 반환합니다. ROLLUP에 대한 앞의 예와 달리 문은 그룹화 열의 모든 조합에 대한 결과를 반환합니다.

```
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 절

HAVING 절은 쿼리가 반환하는 중간 그룹화 결과 집합에 조건을 적용합니다.

구문

```
[ HAVING condition ]
```

예를 들어, SUM 함수의 결과를 제한할 수 있습니다.

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

모든 WHERE 절 조건이 적용되고 GROUP BY 작업이 완료된 후 HAVING 조건이 적용됩니다.

조건 자체는 WHERE 절 조건과 같은 형식을 취합니다.

사용 노트

- HAVING 절 조건에서 참조되는 열은 그룹화 열이거나 집계 함수의 결과를 참조하는 열이어야 합니다.
- HAVING 절에서 다음을 지정할 수는 없습니다.
  - 선택 목록 항목을 참조하는 서수. GROUP BY 및 ORDER BY 절만이 서수를 허용합니다.

예제

다음 쿼리는 이름을 기준으로 모든 이벤트에 대한 총 티켓 판매액을 계산한 다음, 총 판매액이 $800,000 미만인 이벤트를 제거합니다. HAVING 조건은 선택 목록에서 집계 함수의 결과에 적용됩니다. `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)
```

다음 쿼리는 비슷한 결과 집합을 계산합니다. 하지만 이 경우에는 HAVING 조건이 선택 목록에 지정되지 않은 집계에 적용됩니다(sum(qtysold)). 티켓이 2,000장보다 많이 팔리지 않은 이벤트가 최종 결과에서 제거됩니다.

```
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)
```

집합 연산자

집합 연산자는 두 개의 개별 쿼리 표현식의 결과를 비교하고 병합하는 데 사용됩니다.

AWS Clean RoomsSpark SQL은 다음 표에 나열된 다음 세트 연산자를 지원합니다.

연산자 설정
INTERSECT
모두 인터스펙트
EXCEPT
모두 제외
UNION
UNION ALL

예를 들어, 웹사이트의 어떤 사용자가 구매자인 동시에 판매자인지 알고 싶지만 이런 사용자들의 사용자 이름이 별개의 열이나 테이블에 저장되어 있는 경우 이러한 두 가지 사용자 유형의 교집합을 찾을 수 있습니다. 어떤 웹사이트 사용자가 구매자이고 판매자는 아닌지 알고 싶으면 EXCEPT 연산자를 사용하여 두 사용자 목록 사이의 차이를 찾을 수 있습니다. 역할과는 상관없이 모든 사용자의 목록을 빌드하려면 UNION 연산자를 사용할 수 있습니다.

> **Note**
>
> ORDER BY, LIMIT, SELECT TOP 및 OFFSET 절은 UNION, UNION ALL, INTERSECT 및 EXCEPT 집합 연산자로 병합된 쿼리 표현식에 사용할 수 없습니다.

주제

- 구문
- 파라미터
- 설정 연산자에 대한 평가 순서
- 사용 노트
- UNION 쿼리 예
- UNION ALL 쿼리 예
- INTERSECT 쿼리 예
- EXCEPT 쿼리 예

구문

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

파라미터

subquery1, subquery2

선택 목록의 형태로 UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT 또는 EXCEPT ALL 연산자를 따르는 두 번째 쿼리 표현식에 해당하는 쿼리 표현식입니다. 이 두 표현식에는 호환 데이터 형식을 가진 같은 개수의 출력 열이 있어야 합니다. 그렇지 않으면 두 결과 집합을 비교 및 병합할 수 없습니다. 설정 작업은 다양한 범주의 데이터 형식 간에 암시적 변환을 허용하지 않습니다. 자세한 내용은 형식 호환성 및 변환 단원을 참조하십시오.

무제한 개수의 쿼리 표현식을 포함하는 쿼리를 빌드하고 임의의 조합으로 UNION, INTERSECT 및 EXCEPT 연산자와 연결할 수 있습니다. 예를 들어, 테이블 T1, T2 및 T3에 호환되는 열 집합이 포함되어 있다고 가정하면 다음 쿼리 구조가 유효합니다.

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

UNION [모두 | DISTINCT]

행이 한 표현식이나 두 표현식 모두에서 파생하는지에 상관없이, 두 쿼리 표현식에서 행을 반환하는 작업을 설정합니다.

교차 [모두 | 고유]

두 쿼리 표현식에서 파생하는 행을 반환하는 작업을 설정합니다. 두 표현식에서 모두 반환되지 않는 행은 삭제됩니다.

예외 [모두 | 구분]

두 쿼리 표현식 중 하나에서 파생하는 행을 반환하는 작업을 설정합니다. 첫 번째 결과 테이블에는 있지만 두 번째 결과 테이블에는 없는 행에 대한 결과가 반환될 수 있다.

EXCEPT ALL은 결과 행에서 중복 항목을 제거하지 않습니다.

MINUS 및 EXCEPT는 정확히 동의어입니다.

설정 연산자에 대한 평가 순서

UNION 및 EXCEPT 설정 연산자는 좌우선 결합 연산자입니다. 우선순위에 영향을 주기 위해 괄호가 지정되어 있지 않은 경우 이러한 설정 연산자의 조합은 왼쪽에서 오른쪽으로 계산됩니다. 예를 들어 다음 쿼리에서, T1 및 T2의 UNION이 먼저 계산된 다음 UNION 결과에 대해 EXCEPT 작업이 수행됩니다.

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

동일한 쿼리에 연산자 조합이 사용될 때 INTERSECT 연산자가 UNION 및 EXCEPT 연산자보다 우선합니다. 예를 들어 다음 쿼리는 T2 및 T3의 교집합을 계산한 다음 그 결과와 T1의 합집합을 구합니다.

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

괄호를 추가하면 다른 계산 순서를 적용할 수 있습니다. 다음 경우에는 T1 및 T2의 합집합 결과가 T3와 교집합을 이루고, 쿼리가 다른 결과를 낳을 가능성이 있습니다.

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

사용 노트

- 설정 작업 쿼리의 결과에 반환되는 열 이름은 첫 번째 쿼리 표현식의 테이블에서 가져온 열 이름(또는 별칭)입니다. 열의 값이 설정 연산자의 어느 한쪽에 있는 테이블에서 파생한다는 점에서 이런 열 이름은 오해를 불러일으킬 가능성이 있으므로, 결과 집합에 대해 의미 있는 별칭을 부여하고 싶을 수도 있습니다.

- 설정 연산자 쿼리가 10진수 결과를 반환할 때 그에 상응하는 결과 열은 같은 정밀도와 규모를 반환하도록 승격됩니다. 예를 들어, 다음 쿼리에서 T1.REVENUE가 DECIMAL(10,2) 열이고 T2.REVENUE가 DECIMAL(8,4) 열인 경우 10진수 결과는 DECIMAL(12,4)로 승격됩니다.

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

  규모는 두 열의 최대 규모인 4입니다. T1.REVENUE는 소수점 왼쪽에 8자리가 필요하므로(12 - 4 = 8) 정밀도는 12입니다. 이러한 유형 승격은 UNION 양쪽 모두의 값이 전부 결과에 부합하도록 합니다. 64비트 값의 경우, 최대 결과 정밀도는 19이고 최대 결과 규모는 18입니다. 128비트 값의 경우, 최대 결과 정밀도는 38이고 최대 결과 규모는 37입니다.

  결과 데이터 유형이 AWS Clean Rooms정밀도 및 규모 제한을 초과하는 경우 쿼리는 오류를 반환합니다.

- 설정 작업의 경우, 각각 상응하는 열 쌍에 대해 두 데이터 값이 equal 또는 both NULL인 경우 두 행이 동일한 것으로 처리됩니다. 예를 들어, 테이블 T1과 T2에 모두 한 열과 한 행이 있고 그 행이 두 테이블에서 모두 NULL인 경우 두 테이블에 대해 INTERSECT 연산을 수행하면 바로 그 행이 반환됩니다.

UNION 쿼리 예

다음 UNION 쿼리에서 SALES 테이블의 행은 LISTING 테이블의 행과 병합됩니다. 각각의 테이블에서 호환되는 3개의 열이 선택되며, 이 경우에는 해당하는 열들의 이름과 데이터 형식이 동일합니다.

```
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
```

다음 예는 결과 집합에서 어떤 쿼리 표현식이 각각의 행을 생성했는지 볼 수 있도록 UNION 쿼리의 출력에 리터럴 값을 추가할 수 있는 방법을 보여줍니다. 이 쿼리는 첫 번째 쿼리 표현식의 행을 "B"(buyer)로 식별하고 두 번째 쿼리 표현식의 행을 "S"(seller)로 식별합니다.

이 쿼리는 $10,000 이상의 티켓 거래에 대해 구매자와 판매자를 식별합니다. UNION 연산자의 어느 한 쪽에서 두 쿼리 표현식의 유일한 차이점은 SALES 테이블에 대한 조인 열입니다.

```
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
```

중복된 행이 발견되는 경우 결과에 이런 행을 유지해야 하므로, 다음 예에서는 UNION ALL 연산자를 사용합니다. 이벤트 ID의 특정 시리즈에 대해, 쿼리는 각 이벤트와 관련된 각각의 판매에 대해 0개 이상의 행을 반환하고 그 이벤트의 각 목록에 대해 0개 또는 1개의 행을 반환합니다. 이벤트 ID는 LISTING 및 EVENT 테이블에서 각각의 행에 고유하지만, SALES 테이블에서 이벤트 및 목록 ID의 동일한 조합에 대해 여러 개의 판매 건이 있을 수 있습니다.

결과 집합의 세 번째 열은 행의 원본을 식별합니다. 행의 출처가 SALES 테이블인 경우 SALESROW 열에 "YES"로 표시됩니다. (SALESROW는 SALES.LISTID의 별칭입니다.) 행의 출처가 LISTING 테이블인 경우 SALESROW 열에 "No"로 표시됩니다.

이 경우, 결과 집합은 목록 500, 이벤트 7787에 대해 3개의 판매 행으로 구성됩니다. 즉, 이 목록 및 이벤트 조합에 대해 3가지 다른 트랜잭션이 발생했습니다. 다른 두 목록 501 및 502에서는 어떤 판매도 생성되지 않았으므로, 쿼리가 이들 목록 ID에 대해 생성하는 유일한 행의 출처는 LISTING 테이블입니다(SALESROW = 'No').

```
select eventid, listid, 'Yes' as salesrow
from sales
```

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

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

ALL 키워드 없이 같은 쿼리를 실행하는 경우 결과에는 판매 거래 중 하나만 유지됩니다.

```
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 쿼리 예

중복된 행이 발견되는 경우 결과에 이런 행을 유지해야 하므로, 다음 예에서는 UNION ALL 연산자를 사용합니다. 이벤트 ID의 특정 시리즈에 대해, 쿼리는 각 이벤트와 관련된 각각의 판매에 대해 0개 이상의 행을 반환하고 그 이벤트의 각 목록에 대해 0개 또는 1개의 행을 반환합니다. 이벤트 ID는 LISTING 및 EVENT 테이블에서 각각의 행에 고유하지만, SALES 테이블에서 이벤트 및 목록 ID의 동일한 조합에 대해 여러 개의 판매 건이 있을 수 있습니다.

결과 집합의 세 번째 열은 행의 원본을 식별합니다. 행의 출처가 SALES 테이블인 경우 SALESROW 열에 "YES"로 표시됩니다. (SALESROW는 SALES.LISTID의 별칭입니다.) 행의 출처가 LISTING 테이블인 경우 SALESROW 열에 "No"로 표시됩니다.

이 경우, 결과 집합은 목록 500, 이벤트 7787에 대해 3개의 판매 행으로 구성됩니다. 즉, 이 목록 및 이벤트 조합에 대해 3가지 다른 트랜잭션이 발생했습니다. 다른 두 목록 501 및 502에서는 어떤 판매도 생성되지 않았으므로, 쿼리가 이들 목록 ID에 대해 생성하는 유일한 행의 출처는 LISTING 테이블입니다(SALESROW = 'No').

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

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

ALL 키워드 없이 같은 쿼리를 실행하는 경우 결과에는 판매 거래 중 하나만 유지됩니다.

```
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 쿼리 예

다음 예를 첫 번째 UNION 예와 비교해 보십시오. 두 예에서는 사용되는 설정 연산자만 다를 뿐이지만, 그 결과는 매우 상이합니다. 다음과 같이 행들 중 하나만 같습니다.

```
235494 |    23875 |    8771
```

이 행이 양쪽 테이블에서 발견된 5개 행의 제한된 결과에 있는 유일한 행입니다.

```
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
```

다음 쿼리는 3월에 뉴욕과 로스앤젤레스의 두 도시에서 모두 현장에서 이루어진 (티켓이 판매된) 이벤트를 찾습니다. 두 쿼리 표현식의 차이점은 VENUECITY 열에 대한 제약 조건입니다.

```
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 쿼리 예

데이터베이스의 CATEGORY 테이블에는 다음 11개의 행이 포함되어 있습니다.

```
 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 테이블(스테이징 테이블)에 추가적인 행이 한 개 있다고 가정합니다.

```
 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)
```

두 테이블 사이의 차이점을 반환합니다. 다시 말해, CATEGORY_STAGE 테이블에는 있지만 CATEGORY 테이블에는 없는 행을 반환합니다.

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

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

다음과 같은 동등한 쿼리는 동의어 MINUS를 사용합니다.

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

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

SELECT 표현식의 순서를 반대로 하면 쿼리가 아무런 행도 반환하지 않습니다.

ORDER BY 절

ORDER BY 절은 쿼리의 결과 집합을 정렬합니다.

> **Note**
>
> 가장 바깥쪽 ORDER BY 표현식에는 선택 목록에 있는 열만 있어야 합니다.

주제

- 구문
- 파라미터

- 사용 노트
- ORDER BY 사용 예

구문

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

파라미터

expression

쿼리 결과의 정렬 순서를 지정하는 표현식입니다. 선택 목록에 하나 이상의 열이 구성되어 있습니다. 결과는 이진 UTF-8 순서를 기준으로 반환됩니다. 다음을 지정할 수도 있습니다.

- 선택 목록 항목의 위치(또는 선택 목록이 없는 경우 테이블에서 열의 위치)를 나타내는 서수
- 선택 목록 항목을 정의하는 별칭

ORDER BY 절에 여러 개의 표현식이 포함되어 있을 때는 결과 집합이 첫 번째 표현식에 따라 정렬된 다음, 두 번째 표현식이 첫 번째 표현식의 일치하는 값을 가진 행에 적용되는 등의 방식이 적용됩니다.

ASC | DESC

표현식의 정렬 순서를 정의하는 옵션으로서 각각 다음과 같은 의미를 갖습니다.

- ASC: 오름차순(예: 숫자 값의 경우 낮은 값에서 높은 값 순, 문자열의 경우 'A'에서 'Z'의 순. 지정된 옵션이 없는 경우에는 데이터가 기본적으로 오름차순으로 정렬됩니다.
- DESC: 내림차순(숫자 값의 경우 높은 값에서 낮은 값 순, 문자열의 경우 'Z'에서 'A'의 순).

NULLS FIRST | NULLS LAST

NULL 값의 순서를 NULL 값 이외의 값 이전에 결정할지, 혹은 이후에 결정할지 지정하는 옵션입니다. 기본적으로 NULL 값은 ASC 순서에서는 마지막에 정렬 후 순위가 결정되며, DESC 순서에서는 처음에 정렬 후 순위가 결정됩니다.

LIMIT number | ALL

쿼리가 반환하는 정렬된 행의 수를 제어하는 옵션입니다. LIMIT 수는 양의 정수여야 합니다. 최댓값은 2147483647입니다.

LIMIT 0은 아무런 행도 반환하지 않습니다. 이 구문을 테스트 목적으로 사용할 수 있습니다. 즉, 쿼리가 실행되는지 확인하거나(어떤 행도 표시하지 않음) 테이블에서 열 목록을 반환합니다. LIMIT 0을 사용하여 열 목록을 반환하는 경우 ORDER BY 절은 중복입니다. 기본값은 LIMIT ALL입니다.

OFFSET start

행 반환을 위해 시작하기 전에 start 앞에 있는 행의 개수를 건너뛰도록 지정하는 옵션입니다. OFFSET 수는 양의 정수여야 합니다. 최댓값은 2147483647입니다. LIMIT 옵션과 함께 사용 시, OFFSET개의 행을 건너뛴 후 반환되는 LIMIT 행 수를 카운트하기 시작합니다. LIMIT 옵션이 사용되지 않는 경우 결과 집합의 행 개수는 건너뛰는 행 개수만큼 감소됩니다. OFFSET 절에 의해 건너뛰는 행을 계속 스캔해야 하므로, 큰 OFFSET 값을 사용하기에 부족할 수 있습니다.

사용 노트

ORDER BY 절을 사용할 때 다음과 같이 예상되는 동작에 유의하세요.

- NULL 값은 다른 모든 값보다 "높은 값"으로 간주됩니다. 기본 오름차순 정렬 순서에 따라 NULL 값은 끝에 정렬됩니다. 이 동작을 변경하려면 NULLS FIRST 옵션을 사용하세요.
- 쿼리에 ORDER BY 절이 포함되어 있지 않을 때, 시스템에서는 행 순서를 예측할 수 없는 결과 집합을 반환합니다. 같은 쿼리를 두 번 실행할 경우 결과 집합을 다른 순서로 반환할 수도 있습니다.
- ORDER BY 절 없이 LIMIT 및 OFFSET 옵션을 사용할 수 있지만, 일관성 있는 행 집합을 반환하려면 ORDER BY와 함께 이러한 옵션을 사용하세요.
- 와 같은 병렬 시스템에서 AWS Clean RoomsORDER BY가 고유한 순서를 생성하지 않으면 행의 순서는 비결정적입니다. 즉, ORDER BY 표현식이 중복 값을 생성하는 경우 해당 행의 반환 순서는 다른 시스템 또는의 한 실행에서 다음 실행AWS Clean Rooms으로 다를 수 있습니다.
- AWS Clean Rooms는 ORDER BY 절에서 문자열 리터럴을 지원하지 않습니다.

ORDER BY 사용 예

두 번째 열인 CATGROUP 열을 기준으로 정렬된 CATEGORY 테이블에서 11개의 행을 전부 반환합니다. 같은 CATGROUP 값을 가진 결과에 대해서는 문자열의 길이를 기준으로 CATDESC 열 값의 순서를 지정합니다. 그런 다음 열 CATID 및 CATNAME을 기준으로 정렬합니다.

```
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)
```

가장 높은 QTYSOLD 값을 기준으로 정렬된 SALES 테이블에서 선택한 열을 반환합니다. 결과를 맨 위의 10개 행으로 제한합니다.

```
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 구문을 사용하여 열 목록은 반환하고 행은 반환하지 않습니다.

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

하위 쿼리 예

다음 예에서는 하위 쿼리가 SELECT 쿼리에 적합한 다른 방법을 보여줍니다. 하위 쿼리의 다른 사용 예는 예제 섹션을 참조하세요.

SELECT 목록 하위 쿼리

다음 예에서는 SELECT 목록에 하위 쿼리를 포함합니다. 이 하위 쿼리는 스칼라이므로 한 개의 열과 한 개의 값만 반환하며, 이는 외부 쿼리에서 반환되는 각 행에 대한 결과에서 반복됩니다. 이 쿼리는 외부 쿼리에 의해 정의된 바와 같이 2008년의 다른 두 분기(2분기 및 3분기)에 대한 판매액 값과 하위 쿼리가 계산하는 Q1SALES 값을 비교합니다.

```
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 절 하위 쿼리

다음 예에서는 WHERE 절에 테이블 하위 쿼리를 포함합니다. 이 하위 쿼리는 여러 개의 행을 만들어냅니다. 이 경우에는 행에 한 개의 열만 포함되지만, 테이블 하위 쿼리는 다른 테이블과 마찬가지로 여러 개의 열과 행을 포함할 수 있습니다.

이 쿼리는 최대 판매 티켓 수를 기준으로 상위 10개의 판매사를 찾습니다. 톱 10 목록은 티켓 판매소가 있는 도시에 사는 사용자를 제거하는 하위 쿼리에 의해 한정됩니다. 이 쿼리는 다양한 방법으로 작성할 수 있습니다. 예를 들어, 하위 쿼리를 기본 쿼리 내의 조인으로 다시 작성할 수 있습니다.

```
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 절 하위 쿼리

WITH 절을(를) 참조하세요.

상관관계가 있는 하위 쿼리

다음 예에서는 WHERE 절에 상관관계가 있는 하위 쿼리가 포함됩니다. 이런 종류의 하위 쿼리는 자신의 열과 외부 쿼리에 의해 생성되는 열 사이에 하나 이상의 상관관계를 포함합니다. 이 경우 상관관계는 `where s.listid=l.listid`입니다. 외부 쿼리가 생성하는 각각의 행에 자격을 주거나 자격을 취소하는 하위 쿼리가 실행됩니다.

```
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)
```

지원되지 않는 상관관계를 가진 하위 쿼리 패턴

쿼리 플래너는 하위 쿼리 상관관계 제거라는 쿼리 재작성 방법을 사용하여 MPP 환경에서 실행하기 위해 상관관계가 있는 하위 쿼리의 여러 패턴을 최적화합니다. 상관관계가 있는 몇 가지 유형의 하위 쿼리는 데코레이션할 AWS Clean Rooms수 없고 지원하지 않는 패턴을 따릅니다. 다음 상관관계 참조를 포함하는 쿼리는 오류를 반환합니다.

- "건너뛰기 수준의 상관관계 참조"라고도 하는, 쿼리 블록을 건너뛰는 상관관계 참조. 예를 들어, 다음 쿼리에서 상관관계 참조를 포함하는 블록과 건너뛰는 블록은 NOT EXISTS 조건자에 의해 연결됩니다.

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

  이 경우에 건너뛰는 블록은 LISTING 테이블에 대한 하위 쿼리입니다. 상관관계 참조는 EVENT 테이블과 SALES 테이블의 상관관계를 지정합니다.
- 외부 조인에서 ON 절의 일부인 하위 쿼리에서의 상관관계 참조:

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

  ON 절은 외부 쿼리의 EVENT에 대한 하위 쿼리에 있는 SALES에서의 상관관계 참조를 포함합니다.
- AWS Clean Rooms시스템 테이블에 대한 Null 구분 상관 관계 참조입니다. 예제:

  ```
  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);
  ```
- 창 함수를 포함하는 하위 쿼리 내에서의 상관관계 참조.

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

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

- GROUP BY 열에서 상관관계를 가진 하위 쿼리의 결과에 대한 참조. 예:

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

- IN 조건자에 의해 외부 쿼리에 연결된 집계 함수와 GROUP BY 절이 있는 하위 쿼리에서의 상관관계 참조. (이 제한 사항은 MIN 및 MAX 집계 함수에는 적용되지 않습니다.) 예제:

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

AWS Clean Rooms Spark SQL 함수

AWS Clean Rooms Spark SQL은 다음 SQL 함수를 지원합니다.

주제

- 집계 함수
- 배열 함수
- 조건식
- 생성자 함수
- 데이터 형식 지정 함수
- 날짜 및 시간 함수
- 암호화 및 복호화 함수
- 해시 함수
- 하이퍼로그 함수
- JSON 함수
- 수학 함수
- 스칼라 함수

- 문자열 함수
- 개인 정보 보호 관련 함수
- 윈도우 함수

집계 함수

AWS Clean Rooms Spark SQL의 집계 함수는 행 그룹에 대한 계산 또는 작업을 수행하고 단일 값을 반환하는 데 사용됩니다. 데이터 분석 및 요약 작업에 필수적입니다.

AWS Clean Rooms Spark SQL은 다음과 같은 집계 함수를 지원합니다.

주제

- ANY_VALUE 함수
- APPROX COUNT_DISTINCT 함수
- APPROX PERCENTILE 함수
- AVG 함수
- BOOL_AND 함수
- BOOL_OR 함수
- CardINALITY 함수
- COLLECT_LIST 함수
- COLLECT_SET 함수
- COUNT 및 COUNT DISTINCT 함수
- COUNT 함수
- MAX 함수
- MEDIAN 함수
- MIN 함수
- PERCENTILE 함수
- SKEWNESS 함수
- STDDEV_SAMP 및 STDDEV_POP 함수
- SUM 및 SUM DISTINCT 함수
- VAR_SAMP 및 VAR_POP 함수

ANY_VALUE 함수

ANY_VALUE 함수는 입력 표현식 값에서 비결정적으로 값을 반환합니다. 이 함수는 입력 식으로 반환되는 행이 없는 경우 NULL을 반환할 수 있습니다.

구문

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. 표현식은 다음 데이터 유형 중 하나입니다.

isIgnoreNull

함수가 null이 아닌 값만 반환해야 하는지 여부를 결정하는 부울입니다.

반환

expression과 동일한 데이터 형식을 반환합니다.

사용 노트

열에 대한 ANY_VALUE 함수를 지정하는 문이 두 번째 열 참조도 포함하는 경우 두 번째 열은 GROUP BY 절에 나타나거나 집계 함수에 포함되어야 합니다.

예시

다음 예시는 `eventname`이 `Eagles`인 모든 `dateid`의 인스턴스를 반환합니다.

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

다음은 결과입니다.

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

다음 예시는 `eventname`이 `Eagles` 또는 `Cold War Kids`인 모든 `dateid`의 인스턴스를 반환합니다.

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

다음은 결과입니다.

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

APPROX COUNT_DISTINCT 함수

APPROX COUNT_DISTINCT는 열 또는 데이터 세트의 고유 값 수를 추정하는 효율적인 방법을 제공합니다.

구문

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

인수

expr

고유 값 수를 추정하려는 표현식 또는 열입니다.

단일 열, 복잡한 표현식 또는 열 조합일 수 있습니다.

relativeSD

추정의 원하는 상대 표준 편차를 지정하는 선택적 파라미터입니다.

0에서 1 사이의 값으로, 추정치의 최대 허용 상대 오차를 나타냅니다. relativeSD 값이 작을수록 더 정확하지만 추정 속도가 느려집니다.

이 파라미터가 제공되지 않으면 기본값(일반적으로 약 0.05 또는 5%)이 사용됩니다.

반환

HyperLogLog++로 추정 카디널리티를 반환합니다. relativeSD는 허용되는 최대 상대 표준 편차를 정의합니다.

예제

다음 쿼리는 col1 열의 고유 값 수를 추정하며 상대 표준 편차는 1%(0.01)입니다.

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

다음 쿼리는 col1 열에 3개의 고유 값(값 1, 2, 3)이 있다고 추정합니다.

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

APPROX PERCENTILE 함수

APPROX PERCENTILE은 전체 데이터 세트를 정렬할 필요 없이 지정된 표현식 또는 열의 백분위수 값을 추정하는 데 사용됩니다. 이 함수는 정확한 백분위수 계산을 수행하는 계산 오버헤드 없이 대규모 데이터 세트의 분포를 빠르게 이해하거나 백분위수 기반 지표를 추적해야 하는 시나리오에서 유용합니다. 그러나 속도와 정확도의 장단점을 이해하고 사용 사례의 특정 요구 사항에 따라 적절한 내결함성을 선택하는 것이 중요합니다.

구문

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

인수

expr

백분위수 값을 추정하려는 표현식 또는 열입니다.

단일 열, 복잡한 표현식 또는 열 조합일 수 있습니다.

Percentile

추정하려는 백분위수 값으로, 0에서 1 사이의 값으로 표시됩니다.

예를 들어 0.5는 50번째 백분위수(중앙값)에 해당합니다.

정확도

백분위수 추정치의 원하는 정확도를 지정하는 선택적 파라미터입니다. 0에서 1 사이의 값으로, 추정치의 최대 허용 상대 오차를 나타냅니다. accuracy 값이 작을수록 더 정확하지만 추정 속도가 느려집니다. 이 파라미터가 제공되지 않으면 기본값(일반적으로 약 0.05 또는 5%)이 사용됩니다.

반환

정렬된 열 값에서 가장 작은 값인 숫자 또는 ANSI 간격 열 열의 대략적인 백분위수(최소에서 최대로 정렬됨)를 반환하여 열 값의 백분율 이하가 값보다 작거나 해당 값과 같지 않도록 합니다.

백분율 값은 0.0에서 1.0 사이여야 합니다. 정확도 파라미터(기본값: 10000)는 메모리 비용으로 근사 정확도를 제어하는 양수 리터럴입니다.

정확도 값이 높을수록 정확도가 향상되며, 이는 근사치의 상대 오차`1.0/accuracy`입니다.

백분율이 배열인 경우 백분율 배열의 각 값은 0.0에서 1.0 사이여야 합니다. 이 경우는 지정된 백분율 배열에서 열 열의 대략적인 백분위수 배열을 반환합니다.

예시

다음 쿼리는 `response_time` 열의 95번째 백분위수를 추정하며 최대 상대 오차는 1%(0.01)입니다.

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

다음 쿼리는 `tab` 테이블에서 `col` 열의 50번째, 40번째 및 10번째 백분위수 값을 추정합니다.

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

다음 쿼리는 열에 있는 값의 50번째 백분위수(중앙값)를 추정합니다.

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

AVG 함수

AVG 함수는 입력 표현식 값의 평균(산술 평균)을 반환합니다. AVG 함수는 숫자 값을 사용하고 NULL 값을 무시합니다.

구문

```
AVG (column)
```

인수

column

함수가 실행되는 대상 열입니다. 열은 다음 데이터 유형 중 하나입니다.

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

데이터 타입

AVG 함수에서 지원하는 인수 유형은SMALLINT, INTEGER, BIGINT, DECIMAL, 및 DOUBLE입니다.

AVG 함수에서 지원되는 반환 형식은 다음과 같습니다.

- 모든 정수형 인자에 대한 BIGINT
- 부동 소수점 인수에 대한 DOUBLE
- 다른 인수 형식의 표현과 동일한 데이터 형식을 반환합니다

DECIMAL 인수가 있는 AVG 함수 결과의 기본 정밀도는 38입니다. 함수 결과의 비율은 인수 비율과 동일합니다. 예를 들어, DEC(5,2) 열의 AVG는 DEC(38,2) 데이터 유형을 반환합니다.

예제

SALES 테이블에서 트랜잭션 1회당 판매된 평균 수량을 구합니다.

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

BOOL_AND 함수

BOOL_AND 함수는 단일 부울 또는 정수 열이나 표현식에서 실행됩니다. 이 함수는 BIT_AND 및 BIT_OR 함수와 비슷한 로직을 적용합니다. 이 함수의 반환 형식은 부울 값(`true` 또는 `false`)입니다.

집합의 모든 값이 true이면 BOOL_AND 함수가 `true`(`t`)를 반환합니다. 하나라도 값이 false이면 함수가 `false`(`f`)를 반환합니다.

구문

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. 이 표현식의 데이터 형식은 부울 또는 정수가 되어야 합니다. 함수의 반환 형식은 부울입니다.

DISTINCT | ALL

인수가 DISTINCT일 때는 함수가 결과를 계산하기 전에 지정한 표현식의 중복 값을 모두 제거합니다. 인수가 ALL일 때는 함수가 모든 중복 값을 그대로 유지합니다. ALL이 기본값입니다.

예시

부울 함수는 부울 표현식이나 정수 표현식에 대해 사용할 수 있습니다.

예를 들어 다음 쿼리는 TICKIT 데이터베이스에서 부울 열이 일부 포함되어 있는 표준 USERS 테이블을 통해 결과를 반환합니다.

BOOL_AND 함수는 5개 행 모두에서 `false`를 반환합니다. 해당 주마다 모든 사용자가 스포츠를 좋아하는 것은 아닙니다.

```
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 함수

BOOL_OR 함수는 단일 부울 또는 정수 열이나 표현식에서 실행됩니다. 이 함수는 BIT_AND 및 BIT_OR 함수와 비슷한 로직을 적용합니다. 이 함수의 반환 형식은 부울 값(true, false 또는 NULL) 입니다.

세트에서 값이 true이면 BOOL_OR 함수가 true(t)를 반환합니다. 세트에서 값이 false이면 함수가 false(f)를 반환합니다. 값을 알 수 없는 경우 NULL을 반환할 수 있습니다.

구문

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. 이 표현식의 데이터 형식은 부울 또는 정수가 되어야 합니다. 함수의 반환 형식은 부울입니다.

DISTINCT | ALL

인수가 DISTINCT일 때는 함수가 결과를 계산하기 전에 지정한 표현식의 중복 값을 모두 제거합니다. 인수가 ALL일 때는 함수가 모든 중복 값을 그대로 유지합니다. ALL이 기본값입니다.

예시

부울 함수는 부울 표현식이나 정수 표현식에 사용할 수 있습니다. 예를 들어 다음 쿼리는 TICKIT 데이터베이스에서 부울 열이 일부 포함되어 있는 표준 USERS 테이블을 통해 결과를 반환합니다.

BOOL_OR 함수는 5개 행 모두에서 true를 반환합니다. 즉, 해당 주마다 1명 이상의 사용자가 스포츠를 좋아합니다.

```
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)
```

다음 예는 NULL을 반환합니다.

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

CardINALITY 함수

CardINALITY 함수는 ARRAY 또는 MAP 표현식(expr)의 크기를 반환합니다.

이 함수는 배열의 크기 또는 길이를 찾는 데 유용합니다.

구문

```
cardinality(expr)
```

인수

expr

ARRAY 또는 MAP 표현식입니다.

반환

배열 또는 맵(INTEGER)의 크기를 반환합니다.

이 로 설정`false`되거나 `sizeOfNull`가 로 설정된 경우 함수`enabled`는 null 입력에 NULL 대해를 반환합니다`true`.

그렇지 않으면 함수는 null 입력에 -1 대해를 반환합니다. 기본 설정을 사용하면 함수가 null 입력에 -1 대해를 반환합니다.

예제

다음 쿼리는 지정된 배열의 카디널리티 또는 요소 수를 계산합니다. 배열(`'b', 'd', 'c', 'a'`)에는 4개의 요소가 있으므로이 쿼리의 출력은 입니다`4`.

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

COLLECT_LIST 함수

COLLECT_LIST 함수는 고유하지 않은 요소 목록을 수집하고 반환합니다.

이 함수 유형은 행 집합에서 여러 값을 단일 배열로 수집하거나 데이터 구조를 나열하려는 경우에 유용합니다.

> **Note**
>
> 수집된 결과의 순서는 셔플 작업이 수행된 후 비결정적일 수 있는 행의 순서에 따라 달라지므로 함수는 비결정적입니다.

구문

```
collect_list(expr)
```

인수

expr

모든 유형의 표현식입니다.

반환

인수 유형의 ARRAY를 반환합니다. 배열의 요소 순서는 비결정적입니다.

NULL 값은 제외됩니다.

DISTINCT를 지정하면 함수는 고유한 값만 수집하며 `collect_set` 집계 함수의 동의어입니다.

예제

다음 쿼리는 열의 모든 값을 목록으로 수집합니다. VALUES 절은 행이 3개인 인라인 테이블을 생성하는 데 사용되며, 각 행에는 각각 값이 1, 2, 1인 단일 열 열 열이 있습니다. 그런 다음 `collect_list()` 함수를 사용하여 열의 모든 값을 단일 배열로 집계합니다. 이 SQL 문의 출력은 입력 데이터에 표시된 순서대로 열의 모든 값을 `[1,2,1]`포함하는 배열입니다.

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

COLLECT_SET 함수

COLLECT_SET 함수는 일련의 고유한 요소를 수집하고 반환합니다.

이 함수는 중복을 포함하지 않고 행 집합의 모든 고유 값을 단일 데이터 구조로 수집하려는 경우에 유용합니다.

> **Note**
>
> 수집된 결과의 순서는 셔플 작업이 수행된 후 비결정적일 수 있는 행의 순서에 따라 달라지므로 함수는 비결정적입니다.

구문

```
collect_set(expr)
```

인수

expr

MAP를 제외한 모든 유형의 표현식입니다.

반환

인수 유형의 ARRAY를 반환합니다. 배열의 요소 순서는 비결정적입니다.

NULL 값은 제외됩니다.

예제

다음 쿼리는 열의 모든 고유 값을 집합으로 수집합니다. VALUES 절은 행이 3개인 인라인 테이블을 생성하는 데 사용되며, 각 행에는 각각 값이 1, 2, 1인 단일 열 열 열이 있습니다. 그런 다음 `collect_set()` 함수를 사용하여 열의 모든 고유 값을 단일 세트로 집계합니다. 이 SQL 문의 출력은 열의 고유 값을 `[1,2]`포함하는 집합입니다. 중복 값 1은 결과에 한 번만 포함됩니다.

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

COUNT 및 COUNT DISTINCT 함수

COUNT 함수는 표현식에서 정의하는 행 수를 계산합니다. COUNT DISTINCT 함수는 열 또는 표현식에서 NULL이 아닌 고유 값의 수를 계산합니다. 계산을 수행하기 전에 지정된 표현식에서 중복된 값을 모두 제거합니다.

구문

```
COUNT (DISTINCT column)
```

인수

column

함수가 실행되는 대상 열입니다.

데이터 타입

COUNT 함수와 COUNT DISTINCT 함수는 모든 인수 데이터 형식을 지원합니다.

COUNT DISTINCT 함수는 BIGINT를 반환합니다.

예시

Florida 주의 모든 사용자 수를 계산합니다.

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

EVENT 테이블에서 고유한 장소 ID의 수를 모두 계산합니다.

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

COUNT 함수

COUNT 함수는 표현식에서 정의하는 행의 수를 계산합니다.

COUNT 함수는 다음과 같은 변형이 있습니다.

- COUNT (*)는 NULL 값의 유무에 상관없이 대상 테이블에서 모든 행의 수를 계산합니다.
- COUNT (expression)는 특정 열 또는 표현식에서 NULL을 제외한 값이 포함된 행의 수를 계산합니다.
- COUNT (DISTINCT expression)는 임의의 열 또는 표현식에서 NULL을 제외한 고유 값의 수를 계산합니다.

구문

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

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. COUNT 함수는 모든 인수 데이터 형식을 지원합니다.

DISTINCT | ALL

인수가 DISTINCT일 때는 행의 수를 계산하기 전에 지정한 표현식에서 중복 값을 모두 제거합니다. 인수가 ALL일 때는 함수가 표현식의 모든 중복 값을 그대로 유지한 채 행의 수를 계산합니다. ALL이 기본값입니다.

반환 타입

COUNT 함수는 BIGINT를 반환합니다.

예시

Florida 주의 모든 사용자 수를 계산합니다.

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

count
-------
```

```
510
```

EVENT 테이블에서 모든 이벤트 이름의 수를 계산합니다.

```
select count(eventname) from event;

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

EVENT 테이블에서 모든 이벤트 이름의 수를 계산합니다.

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

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

EVENT 테이블에서 고유한 장소 ID의 수를 모두 계산합니다.

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

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

개별 판매자가 4장 이상의 티켓을 한 묶음으로 판매한 횟수를 계산합니다. 결과는 판매자 ID로 구분합니다.

```
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 함수

MAX 함수는 행 집합에서 최댓값을 반환합니다. DISTINCT 또는 ALL은 사용할 수 있지만 결과에 아무런 영향도 끼치지 않습니다.

구문

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. 표현식은 모든 숫자 데이터 형식입니다.

DISTINCT | ALL

인수가 DISTINCT일 때는 함수가 최댓값을 계산하기 전에 지정한 표현식에서 중복 값을 모두 제거합니다. 인수가 ALL일 때는 함수가 표현식의 모든 중복 값을 그대로 유지한 채 최댓값을 계산합니다. ALL이 기본값입니다.

데이터 타입

expression과 동일한 데이터 형식을 반환합니다.

예시

모든 판매에서 지불된 최고 가격을 구합니다.

```
select max(pricepaid) from sales;

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

모든 판매에서 티켓 1장당 지불된 최고 가격을 구합니다.

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

max_ticket_price
```

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

MEDIAN 함수

구문

```
MEDIAN ( median_expression )
```

인수

median_expression

함수가 실행되는 대상 열 또는 표현식입니다.

MIN 함수

MIN 함수는 행 집합에서 최솟값을 반환합니다. DISTINCT 또는 ALL은 사용할 수 있지만 결과에 아무런 영향도 끼치지 않습니다.

구문

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

인수

표현식

함수가 실행되는 대상 열 또는 표현식입니다. 표현식은 모든 숫자 데이터 형식입니다.

DISTINCT | ALL

인수가 DISTINCT일 때는 함수가 최솟값을 계산하기 전에 지정한 표현식에서 중복 값을 모두 제거합니다. 인수가 ALL일 때는 함수가 표현식의 모든 중복 값을 그대로 유지한 채 최솟값을 계산합니다. ALL이 기본값입니다.

데이터 타입

expression과 동일한 데이터 형식을 반환합니다.

예시

모든 판매에서 지불된 최저 가격을 구합니다.

```
select min(pricepaid) from sales;

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

모든 판매에서 티켓 1장당 지불된 최저 가격을 구합니다.

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

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

PERCENTILE 함수

PERCENTILE 함수는 먼저 col 열의 값을 정렬한 다음 지정된에서 값을 찾아 정확한 백분위수 값을 계산하는 데 사용됩니다percentage.

PERCENTILE 함수는 정확한 백분위수 값을 계산해야 하고 사용 사례에 대한 계산 비용이 허용 가능한 경우에 유용합니다. APPROX_PERCENTILE 함수보다 더 정확한 결과를 제공하지만 특히 대규모 데이터 세트의 경우 속도가 느릴 수 있습니다.

반면 APPROX_PERCENTILE 함수는 지정된 오류 허용 오차로 백분위수 값의 추정치를 제공할 수 있는 보다 효율적인 대안이므로 절대 정밀도보다 속도가 우선 순위가 높은 시나리오에 더 적합합니다.

구문

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

인수

열

백분위수 값을 계산하려는 표현식 또는 열입니다.

백분율

계산하려는 백분위수 값으로, 0에서 1 사이의 값으로 표시됩니다.

예를 들어 0.5는 50번째 백분위수(중앙값)에 해당합니다.

빈도

col 열에 있는 각 값의 빈도 또는 가중치를 지정하는 선택적 파라미터입니다. 제공된 경우 함수는 각 값의 빈도를 기반으로 백분위수를 계산합니다.

반환

지정된 백분율에서 숫자 또는 ANSI 간격 열 열 열의 정확한 백분위 값을 반환합니다.

백분율 값은 0.0에서 1.0 사이여야 합니다.

빈도 값은 양의 정수여야 합니다.

예제

다음 쿼리는 col 열 값의 30% 이상인 값을 찾습니다. 값이 0과 10이므로 30번째 백분위수는 데이터의 30% 이상인 값이기 때문에 3.0입니다.

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

SKEWNESS 함수

SKEWNESS 함수는 그룹의 값에서 계산된 왜도 값을 반환합니다.

왜도는 데이터 세트의 비대칭 또는 대칭 부족을 설명하는 통계적 척도입니다. 데이터 배포의 형태에 대한 정보를 제공합니다.

이 함수는 데이터 세트의 통계적 속성을 이해하고 추가 분석 또는 의사 결정을 알리는 데 유용할 수 있습니다.

구문

```
skewness(expr)
```

인수

expr

숫자로 평가되는 표현식입니다.

반환

DOUBLE을 반환합니다.

DISTINCT를 지정하면 함수는 고유한 expr 값 집합에서만 작동합니다.

예시

다음 쿼리는 col 열에 있는 값의 왜도를 계산합니다. 이 예제에서 VALUES 절은 행이 4개인 인라인 테이블을 생성하는 데 사용됩니다. 여기서 각 행에는 값이 -10, -20, 100 및 1000col인 단일 열이 있습니다. 그런 다음 skewness() 함수를 사용하여 col 열에 있는 값의 왜도를 계산합니다. 결과 1.1135657469022011은 데이터의 왜도 정도와 방향을 나타냅니다. 양수 왜도 값은 데이터가 오른쪽으로 기울어져 있고 값의 대부분은 배포의 왼쪽에 집중되어 있음을 나타냅니다. 음의 왜도 값은 데이터가 왼쪽으로 기울어져 있고 값의 대부분은 배포의 오른쪽에 집중되어 있음을 나타냅니다.

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

다음 쿼리는 열에 있는 값의 왜도를 계산합니다. 이전 예제와 마찬가지로 VALUES 절은 행이 4개인 인라인 테이블을 생성하는 데 사용됩니다. 여기서 각 행에는 값이 -1000, -100, 10 및 20col인 단일 열이 있습니다. 그런 다음 skewness() 함수를 사용하여 col 열에 있는 값의 왜도를 계산합니다. 결과 -1.1135657469022011은 데이터의 왜도 정도와 방향을 나타냅니다. 이 경우 음의 왜도 값은 데이터가 왼쪽으로 기울어져 있고 값의 대부분은 배포의 오른쪽에 집중되어 있음을 나타냅니다.

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

STDDEV_SAMP 및 STDDEV_POP 함수

STDDEV_SAMP 및 STDDEV_POP 함수는 숫자 값(정수, 소수 또는 부동 소수점) 집합의 표본 표준 편차와 모 표준 편차를 반환합니다. STDDEV_SAMP 함수의 결과는 동일한 값 집합의 표본 분산 제곱근과 동일합니다.

STDDEV_SAMP와 STDDEV는 동일한 함수이기 때문에 동의어나 마찬가지입니다.

구문

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

표현식에는 숫자 데이터 형식이 있어야 합니다. 표현식의 데이터 형식과 상관없이 이 함수의 반환 형식은 배정밀도 숫자입니다.

> **Note**
>
> 표준 편차는 부동 소수점 연산을 통해 계산하지만 약간 부정확할 수 있습니다.

사용 노트

단일 값으로 구성된 표현식에 대해 표본 표준 편차(STDDEV 또는 STDDEV_SAMP)를 계산할 경우 함수 결과는 0이 아닌 NULL이 됩니다.

예시

다음 쿼리는 VENUE 테이블에서 VENUESEATS 열의 값 평균과 그 뒤를 이어 동일한 값 집합의 표본 표준 편차 및 모 표준 편차를 반환합니다. VENUESEATS는 INTEGER 열입니다. 결과의 크기는 2자리로 줄어듭니다.

```
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)
```

다음 쿼리는 SALES 테이블에서 COMMISSION 열의 표본 표준 편차를 반환합니다. COMMISSION은 DECIMAL 열입니다. 결과의 크기는 10자리로 줄어듭니다.

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

```
from sales;

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

다음 쿼리는 COMMISSION 열의 표본 표준 편차를 정수로 변환합니다.

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

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

다음 쿼리는 COMMISSION 열의 표본 표준 편차와 표본 분산 제곱근을 모두 반환합니다. 이 두 가지의 계산 결과는 동일합니다.

```
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 및 SUM DISTINCT 함수

SUM 함수는 입력 열 또는 표현식 값의 합을 반환합니다. SUM 함수는 숫자 값을 사용하고 NULL 값을 무시합니다.

SUM DISTINCT 함수는 합계를 계산하기 전에 지정된 표현식에서 중복된 값을 모두 제거합니다.

구문

```
SUM (DISTINCT column )
```

인수

column

함수가 실행되는 대상 열입니다. 열은 모든 숫자 데이터 형식입니다.

예시

SALES 테이블에서 지불된 모든 수수료의 합을 구합니다.

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

SALES 테이블에서 지급된 모든 개별 수수료의 합계를 구합니다.

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

VAR_SAMP 및 VAR_POP 함수

VAR_SAMP 및 VAR_POP 함수는 숫자 값(정수, 소수 또는 부동 소수점) 집합의 표본 분산과 모 분산을 반환합니다. VAR_SAMP 함수의 결과는 동일한 값 집합의 표본 표준 편차를 제곱한 것과 동일합니다.

VAR_SAMP 및 VARIANCE는 동일한 함수이기 때문에 동의어나 마찬가지입니다.

구문

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

표현식의 데이터 형식은 정수, 소수 또는 부동 소수점이 되어야 합니다. 표현식의 데이터 형식과 상관없이 이 함수의 반환 형식은 배정밀도 숫자입니다.

> **Note**
>
> 두 함수의 결과는 데이터 웨어하우스 클러스터에서 각각 클러스터 구성에 따라 다를 수 있습니다.

사용 노트

단일 값으로 구성된 표현식에 대해 표본 분산(VARIANCE 또는 VAR_SAMP)을 계산할 경우 함수 결과는 0이 아닌 NULL이 됩니다.

예시

다음 쿼리는 LISTING 테이블에서 NUMTICKETS 열의 표본 및 모 분산을 반올림하여 반환합니다.

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

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

다음 쿼리는 동일한 계산을 실행하지만 결과를 소수 값으로 변환합니다.

```
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)
```

배열 함수

이 섹션에서는 AWS Clean Rooms에서 지원되는 SQL의 배열 함수를 설명합니다.

주제

- ARRAY 함수
- ARRAY_CONTAINS 함수
- ARRAY_DISTINCT 함수
- ARRAY_EXCEPT 함수
- ARRAY_INTERSECT 함수
- ARRAY_JOIN 함수
- ARRAY_REMOVE 함수

- ARRAY_UNION 함수
- EXPLODE 함수
- FLATTEN 함수

ARRAY 함수

지정된 요소로 배열을 생성합니다.

구문

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

인수

expr1, expr2

날짜 및 시간 유형을 제외한 모든 데이터 유형의 표현식. 인수는 동일한 데이터 형식일 필요는 없습니다.

반환 타입

배열 함수는 표현식의 요소가 있는 ARRAY를 반환합니다.

예제

다음 예에서는 숫자 값의 배열과 다양한 데이터 형식의 배열을 보여줍니다.

```
--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 함수

ARRAY_CONTAINS 함수를 사용하여 배열 데이터 구조에 대한 기본 멤버십 검사를 수행할 수 있습니다. ARRAY_CONTAINS 함수는 배열 내에 특정 값이 있는지 확인해야 할 때 유용합니다.

구문

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

인수

array

검색할 ARRAY입니다.

USD 상당

배열 요소와 가장 덜 일반적인 유형을 공유하는 유형의 표현식입니다.

반환 타입

ARRAY_CONTAINS 함수는 BOOLEAN을 반환합니다.

값이 NULL인 경우 결과는 NULL입니다.

배열의 요소가 NULL인 경우 값이 다른 요소와 일치하지 않으면 결과는 NULL입니다.

예시

다음 예제에서는 배열에 값이 [1, 2, 3] 포함되어 있는지 확인합니다4. 배열 [1, 2, 3]에는 값이 포함되어 있지 않으므로 array_contains 함수4는를 반환합니다false.

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

다음 예제에서는 배열에 값이 [1, 2, 3] 포함되어 있는지 확인합니다2. 배열에는 값이 [1, 2, 3] 포함되어 있으므로 array_contains 함수2는를 반환합니다true.

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

ARRAY_DISTINCT 함수

ARRAY_DISTINCT 함수를 사용하여 배열에서 중복 값을 제거할 수 있습니다. ARRAY_DISTINCT 함수는 배열에서 중복 항목을 제거하고 고유한 요소만 사용해야 하는 경우에 유용합니다. 이는 반복되는 값의 간섭 없이 데이터 세트에 대한 작업 또는 분석을 수행하려는 시나리오에서 유용할 수 있습니다.

구문

```
array_distinct(array)
```

인수

array

ARRAY 표현식입니다.

반환 타입

ARRAY_DISTINCT 함수는 입력 배열의 고유한 요소만 포함하는 ARRAY를 반환합니다.

예시

이 예제에서 입력 배열에는의 중복 값이 `[1, 2, 3, null, 3]` 포함되어 있습니다3. `array_distinct` 함수는이 중복 값을 제거하고 고유한 요소가 있는 새 배열을 3 반환합니다`[1, 2, 3, null]`.

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

이 예제에서 입력 배열에는 2 및의 중복 값이 `[1, 2, 2, 3, 3, 3]` 포함되어 있습니다3. `array_distinct` 함수는 이러한 중복을 제거하고 고유한 요소가 있는 새 배열을 반환합니다`[1, 2, 3]`.

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

ARRAY_EXCEPT 함수

ARRAY_EXCEPT 함수는 두 개의 배열을 인수로 받아 첫 번째 배열에는 있지만 두 번째 배열에는 없는 요소만 포함하는 새 배열을 반환합니다.

ARRAY_EXCEPT는 배열마다 고유한 요소를 찾아야 할 때 유용합니다. 이는 두 데이터 세트 간의 차이를 찾는 등 배열에서 세트와 유사한 작업을 수행해야 하는 시나리오에서 유용할 수 있습니다.

구문

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

인수

배열1

유사한 요소가 있는 모든 유형의 ARRAY입니다.

array2

가장 덜 일반적인 유형을 array1의 요소와 공유하는 요소의 ARRAY입니다.

반환 타입

ARRAY_EXCEPT 함수는 중복 없이 일치하는 유형의 ARRAY를 array1에 반환합니다.

예시

이 예제에서 첫 번째 배열에는 요소 1, 2 및 3이 [1, 2, 3] 포함됩니다. 두 번째 배열에는 요소 2, 3, 4가 [2, 3, 4] 포함됩니다. 이 array_except 함수는 두 번째 배열에도 존재하므로 첫 번째 배열에서 요소 2와 3을 제거합니다. 결과 출력은 배열 입니다[1].

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

이 예제에서 첫 번째 배열에는 요소 1, 2 및 3이 [1, 2, 3] 포함됩니다. 두 번째 배열에는 요소 1, 3 및 5가 [1, 3, 5] 포함됩니다. 이 array_except 함수는 두 번째 배열에도 존재하므로 첫 번째 배열에서 요소 1과 3을 제거합니다. 결과 출력은 배열 입니다[2].

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

```
 [2]
```

ARRAY_INTERSECT 함수

ARRAY_INTERSECT 함수는 두 개의 배열을 인수로 받아 두 입력 배열에 있는 요소가 포함된 새 배열을 반환합니다. 이 함수는 두 배열 간의 공통 요소를 찾아야 할 때 유용합니다. 이는 두 데이터 세트 간의 교차점을 찾는 등 배열에서 세트와 유사한 작업을 수행해야 하는 시나리오에서 유용할 수 있습니다.

구문

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

인수

array1

유사한 요소가 있는 모든 유형의 ARRAY입니다.

array2

가장 덜 일반적인 유형을 array1의 요소와 공유하는 요소의 ARRAY입니다.

반환 타입

ARRAY_INTERSECT 함수는 array1과 array2 모두에 중복 및 요소가 포함되지 않은 array1에 일치하는 유형의 ARRAY를 반환합니다.

예시

이 예제에서 첫 번째 배열에는 요소 1, 2 및 3이 `[1, 2, 3]` 포함됩니다. 두 번째 배열에는 요소 1, 3 및 5가 `[1, 3, 5]` 포함됩니다. ARRAY_INTERSECT 함수는 두 배열 사이의 공통 요소인 1과 3을 식별합니다. 결과 출력 배열은 입니다`[1, 3]`.

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

ARRAY_JOIN 함수

ARRAY_JOIN 함수는 두 개의 인수를 사용합니다. 첫 번째 인수는 조인할 입력 배열입니다. 두 번째 인수는 배열 요소를 연결하는 데 사용할 구분자 문자열입니다. 이 함수는 문자열 배열(또는 기타 데이터

유형)을 하나의 연결된 문자열로 변환해야 할 때 유용합니다. 이는 표시 목적이나 추가 처리에 사용할 때와 같이 값 배열을 형식이 지정된 단일 문자열로 표시하려는 시나리오에서 유용할 수 있습니다.

구문

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

인수

array

모든 ARRAY 유형이지만 해당 요소는 문자열로 해석됩니다.

delimiter

연결된 배열 요소를 분리하는 데 사용되는 STRING입니다.

nullReplacement

결과에서 NULL 값을 표현하는 데 사용되는 STRING입니다.

반환 타입

ARRAY_JOIN 함수는 배열 요소가 구분 기호로 구분되고 null 요소가 로 대체되는 STRING을 반환합니다nullReplacement. nullReplacement를 생략하면 null 요소가 필터링됩니다. 인수가 인 경우 NULL결과는 입니다NULL.

예시

이 예제에서 ARRAY_JOIN 함수는 배열을 가져['hello', 'world']와 구분자' '(스페이스 문자)를 사용하여 요소를 조인합니다. 결과 출력은 문자열 입니다'hello world'.

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

이 예제에서 ARRAY_JOIN 함수는 배열을 가져['hello', null, 'world']와 구분자' '(스페이스 문자)를 사용하여 요소를 조인합니다. null 값은 제공된 대체 문자열','(쉼표)로 대체됩니다. 결과 출력은 문자열 입니다'hello , world'.

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

ARRAY_REMOVE 함수

ARRAY_REMOVE 함수는 두 개의 인수를 사용합니다. 첫 번째 인수는 요소가 제거될 입력 배열입니다. 두 번째 인수는 배열에서 제거될 값입니다. 이 함수는 배열에서 특정 요소를 제거해야 하는 경우에 유용합니다. 이는 값 배열에서 데이터 정리 또는 사전 처리를 수행해야 하는 시나리오에서 유용할 수 있습니다.

구문

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

인수

array

어레이.

element

배열 요소와 가장 흔하지 않은 유형을 공유하는 유형의 표현식입니다.

반환 타입

ARRAY_REMOVE 함수는 배열 유형과 일치하는 결과 유형을 반환합니다. 제거할 요소가 인 경우 NULL결과는 입니다NULL.

예시

이 예제에서 ARRAY_REMOVE 함수는 배열을 가져[1, 2, 3, null, 3]와 값 3의 모든 발생을 제거합니다. 결과 출력은 배열 입니다[1, 2, null].

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

ARRAY_UNION 함수

ARRAY_UNION 함수는 두 개의 배열을 인수로 받아 두 입력 배열의 고유한 요소가 포함된 새 배열을 반환합니다. 이 함수는 두 배열을 결합하고 중복 요소를 제거해야 할 때 유용합니다. 이는 두 데이터 세트 간의 결합 찾기와 같이 배열에서 세트와 유사한 작업을 수행해야 하는 시나리오에서 유용할 수 있습니다.

구문

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

인수

array1

어레이.

array2

array1과 동일한 유형의 ARRAY입니다.

반환 타입

ARRAY_UNION 함수는 배열과 동일한 유형의 ARRAY를 반환합니다.

예제

이 예제에서 첫 번째 배열에는 요소 1, 2 및 3이 [1, 2, 3] 포함됩니다. 두 번째 배열에는 요소 1, 3 및 5가 [1, 3, 5] 포함됩니다. ARRAY_UNION 함수는 두 배열의 고유한 요소를 결합하여 출력 배열을 생성합니다[1, 2, 3, 5]. T

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

EXPLODE 함수

EXPLODE 함수는 배열 또는 맵 열이 있는 단일 행을 여러 행으로 변환하는 데 사용되며, 여기서 각 행은 배열 또는 맵의 단일 요소에 해당합니다.

구문

```
explode(expr)
```

인수

expr

배열 표현식 또는 맵 표현식.

반환 타입

EXPLODE 함수는 행 세트를 반환합니다. 여기서 각 행은 입력 배열 또는 맵의 단일 요소를 나타냅니다.

출력 행의 데이터 형식은 입력 배열 또는 맵에 있는 요소의 데이터 형식에 따라 달라집니다.

예시

다음 예제에서는 단일 행 배열[10, 20]을 가져와 각각 배열 요소(10 및 20) 중 하나를 포함하는 두 개의 개별 행으로 변환합니다.

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

첫 번째 예제에서는 입력 배열이 인수로에 직접 전달되었습니다`explode()`. 이 예제에서는 입력 배열이 => 구문을 사용하여 지정됩니다. 여기서 열 이름(`collection`)이 명시적으로 제공됩니다.

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

두 접근 방식 모두 유효하며 동일한 결과를 달성하지만 단순한 배열 리터럴이 아닌 더 큰 데이터 세트에서 열을 탐색해야 하는 경우 두 번째 구문이 더 유용할 수 있습니다.

FLATTEN 함수

FLATTEN 함수는 중첩된 배열 구조를 단일 플랫 배열로 "고정"하는 데 사용됩니다.

구문

```
flatten(arrayOfArrays)
```

인수

arrayOfArrays

배열의 배열입니다.

반환 타입

FLATTEN 함수는 배열을 반환합니다.

예제

이 예제에서 입력은 두 개의 내부 배열이 있는 중첩 배열이고 출력은 내부 배열의 모든 요소를 포함하는 단일 플랫 배열입니다. FLATTEN 함수는 중첩 배열을 가져`[[1, 2], [3, 4]]`와 모든 요소를 단일 배열로 결합합니다`[1, 2, 3, 4]`.

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

조건식

SQL에서는 조건식이 특정 조건에 따라 결정을 내리는 데 사용됩니다. 이를 통해 SQL 문의 흐름을 제어하고 다른 값을 반환하거나 하나 이상의 조건 평가에 따라 다른 작업을 수행할 수 있습니다.

AWS Clean Rooms 는 다음과 같은 조건 표현식을 지원합니다.

주제

- CASE 조건식
- COALESCE expression
- GREATEST 및 LEAST 표현식
- IF 표현식
- IS_NULL 표현식
- IS_NOT_NULL 표현식
- NVL 및 COALESCE 함수
- NVL2 함수
- NULLIF 함수

CASE 조건식

CASE 표현식은 다른 언어에서 발견되는 if/then/else 문과 비슷한 조건 표현식입니다. CASE는 다수의 조건이 있을 때 결과를 지정하는 데 사용됩니다. SELECT 명령과 같이 SQL 표현식이 유효한 경우 CASE를 사용합니다.

CASE 표현식은 단순(simple)과 검색(searched), 두 가지 유형이 있습니다.

- 단순 CASE 표현식에서는 표현식과 값을 비교합니다. 이때 일치하는 부분이 발견되면 THEN 절에서 지정된 작업이 적용됩니다. 일치하는 부분이 발견되지 않으면 ELSE 절에서 지정된 작업이 적용됩니다.
- 검색 CASE 표현식에서는 각 CASE가 부울 표현식에 따라 평가되고, CASE 문이 처음 일치하는 CASE를 반환합니다. WHEN 절 사이에서 일치하는 부분이 발견되지 않으면 ELSE 절의 작업이 반환됩니다.

구문

다음은 조건을 일치시키는 데 사용되는 단순 CASE 문입니다.

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

다음은 각 조건을 평가하는 데 사용되는 검색 CASE 문입니다.

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

인수

expression

열 이름 또는 유효한 표현식입니다.

USD 상당

숫자 상수나 문자열 같이 표현식과 함께 비교하는 값입니다.

result

표현식 또는 부울 조건을 평가할 때 반환되는 대상 값 또는 표현식입니다. 모든 결과 표현식의 데이터 형식은 단일 출력 형식으로 변환할 수 있어야 합니다.

condition

true 또는 false로 평가되는 부울 표현식 condition이 true이면 CASE 표현식의 값은 조건 다음에 오는 결과이며 나머지 CASE 표현식은 처리되지 않습니다. condition이 false이면 이후의 모든 WHEN 절이 평가됩니다. WHEN condition 결과가 true가 아닌 경우 CASE 표현식의 값은 ELSE 절의 결과입니다. ELSE 절을 생략한 상태에서 condition이 true가 아닌 경우 NULL 값이 결과로 반환됩니다.

예시

VENUE 테이블에 대한 쿼리에서 단순 CASE 표현식을 사용하여 `New York City`를 `Big Apple`로 변경합니다. 그 밖의 도시 이름은 모두 `other`로 변경합니다.

```
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
...
```

검색 CASE 표현식을 사용하여 개별 티켓 판매에 대한 PRICEPAID 값을 기준으로 그룹 번호를 할당합니다.

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

pricepaid |  case
----------+---------
```

```
12624     | group 2
10000     | group 3
10000     | group 3
9996      | group 1
9988      | group 1
...
```

COALESCE expression

COALESCE 표현식은 목록에서 NULL이 아닌 첫 번째 표현식의 값을 반환합니다. 모든 표현식이 NULL이면 결과는 NULL입니다. NULL이 아닌 값이 있으면 목록의 나머지 표현식은 평가되지 않습니다.

이러한 표현식은 원하는 값이 없거나 NULL일 때 대체 값을 반환하는 데 유용합니다. 예를 들어 쿼리가 전화번호 3개(휴대 전화, 자택 전화 또는 직장 전화의 순) 중에서 테이블에서 처음 발견되는 값(NULL 아님) 하나를 반환합니다.

구문

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

예시

COALESCE 표현식을 두 열에 적용합니다.

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

NVL 표현식의 기본 열 이름이 COALESCE입니다. 다음 쿼리에서도 동일한 결과가 반환됩니다.

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

GREATEST 및 LEAST 표현식

다수의 표현식 목록에서 가장 크거나 가장 작은 값을 반환합니다.

구문

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

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

파라미터

expression_list

열 이름과 같이 쉼표로 구분된 표현식 목록입니다. 표현식은 모두 공통 데이터 형식으로 변환 가능해야 합니다. 목록에서 NULL 값은 무시됩니다. 표현식이 모두 NULL로 평가되면 결과로 NULL이 반환됩니다.

반환

제공된 표현식 목록에서 가장 큰 값(GREATEST) 또는 가장 작은 값(LEAST)을 반환합니다.

예제

다음은 `firstname` 또는 `lastname`에서 알파벳 순으로 가장 높은 값을 반환하는 예입니다.

```
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 표현식

IF 조건 함수는 조건에 따라 두 값 중 하나를 반환합니다.

이 함수는 SQL에서 조건 평가를 기반으로 결정을 내리고 다른 값을 반환하는 데 사용되는 일반적인 제어 흐름 문입니다. 쿼리 내에서 간단한 if-else 로직을 구현하는 데 유용합니다.

구문

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

인수

expr1

평가되는 조건 또는 표현식입니다. 인 경우 `true`함수는 expr2 값을 반환합니다. expr1이 인 경우 `false`함수는 expr3 값을 반환합니다.

expr2

expr1이 인 경우 평가되고 반환되는 표현식입니다`true`.

expr3

expr1이 인 경우 평가되고 반환되는 표현식입니다`false`.

반환

가 로 `expr1` 평가되면를 반환하고`true`, `expr2`그렇지 않으면를 반환합니다`expr3`.

예제

다음 예제에서는 `if()` 함수를 사용하여 조건에 따라 두 값 중 하나를 반환합니다. 평가 중인 조건은 이며`1 < 2`, 이므로 `true`첫 번째 값이 반환`'a'`됩니다.

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

IS_NULL 표현식

IS_NULL 조건식은 값이 null인지 확인하는 데 사용됩니다.

이 표현식은의 동의어입니다IS NULL.

구문

```
is_null(expr)
```

인수

expr

모든 유형의 표현식입니다.

반환

IS_NULL 조건식은 부울을 반환합니다. expr1가 NULL이면를 반환하고true, 그렇지 않으면를 반환합니다false.

예시

다음 예제에서는 값이 null인지 확인하고 11이 null이 아닌 유효한 값true이므로 부울 결과를 반환합니다.

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

다음 예제에서는 squirrels 테이블에서 id 열을 선택하지만 연령 열이 인 행에 대해서만 선택합니다null.

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

IS_NOT_NULL 표현식

IS_NOT_NULL 조건식은 값이 null이 아닌지 확인하는 데 사용됩니다.

이 표현식은의 동의어입니다IS NOT NULL.

구문

```
is_not_null(expr)
```

인수

expr

모든 유형의 표현식입니다.

반환

IS_NOT_NULL 조건식은 부울을 반환합니다. expr1가 NULL이 아니면를 반환하고true, 그렇지 않으면를 반환합니다false.

예시

다음 예제에서는 값이 null1이 아닌지 확인하고 1이 유효한 null이 아닌 값true이므로 부울 결과를 반환합니다.

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

다음 예제에서는 squirrels 테이블에서 id 열을 선택하지만 연령 열이가 아닌 행에 대해서만 선택합니다null.

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

NVL 및 COALESCE 함수

일련의 표현식에서 NULL이 아닌 첫 번째 표현식의 값을 반환합니다. NULL이 아닌 값이 발견되면 목록의 나머지 표현식은 평가되지 않습니다.

NVL은 COALESCE와 동일합니다. 이 둘은 동의어입니다. 이 항목에서는 구문을 설명하고 두 가지에 대한 예제를 모두 제공합니다.

구문

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

COALESCE의 구문은 동일합니다.

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

모든 표현식이 NULL이면 결과는 NULL입니다.

이러한 함수는 기본 값이 없거나 NULL일 때 보조 값을 반환하려는 경우에 유용합니다. 예를 들어 쿼리는 사용 가능한 세 가지 전화번호(휴대폰, 집 또는 회사) 중 첫 번째 전화번호를 반환할 수 있습니다. 함수의 표현식 순서에 따라 평가 순서가 결정됩니다.

인수

expression

NULL 상태로 평가되는 표현식(열 이름 등)입니다.

반환 타입

AWS Clean Rooms 는 입력 표현식을 기반으로 반환된 값의 데이터 유형을 결정합니다. 입력 표현식의 데이터 유형에 공통 유형이 없는 경우 오류가 반환됩니다.

예시

목록에 정수 표현식이 포함된 경우 함수는 정수를 반환합니다.

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

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

이 예는 NVL을 사용한다는 점을 제외하면 이전 예와 동일하며 동일한 결과를 반환합니다.

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

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

다음은 문자열 유형을 반환하는 예입니다.

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

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

다음 예에서는 표현식 목록에서 데이터 유형이 다양하기 때문에 오류가 발생합니다. 이 경우 목록에 문자열 유형과 숫자 유형이 모두 있습니다.

```
SELECT COALESCE(NULL, 'AWS Clean Rooms', 12);
```

```
ERROR: invalid input syntax for integer: "AWS Clean Rooms"
```

NVL2 함수

지정하는 표현식의 평가 결과가 NULL 또는 NOT NULL인지 여부에 따라 두 값 중 하나를 반환합니다.

구문

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

인수

expression

NULL 상태로 평가되는 표현식(열 이름 등)입니다.

not_null_return_value

expression이 NOT NULL로 평가되면 반환되는 값입니다. not_null_return_value 값은 expression과 동일한 데이터 형식이거나, 혹은 묵시적으로 이 데이터 형식으로 변환 가능해야 합니다.

null_return_value

expression이 NULL로 평가되면 반환되는 값입니다. null_return_value 값은 expression과 동일한 데이터 형식이거나, 혹은 묵시적으로 이 데이터 형식으로 변환 가능해야 합니다.

반환 타입

NVL2 반환 형식은 다음과 같이 결정됩니다.

- not_null_return_value 또는 null_return_value이 NULL이면 not null 표현식의 데이터 형식이 반환됩니다.

not_null_return_value와 null_return_value 모두 NULL이 아닌 경우에는 다음과 같습니다.

- not_null_return_value와 null_return_value의 데이터 형식이 동일하면 해당 데이터 형식이 반환됩니다.
- not_null_return_value와 null_return_value의 숫자 데이터 형식이 다르면 가장 작으면서 호환도 가능한 숫자 데이터 형식이 반환됩니다.
- not_null_return_value와 null_return_value의 날짜/시간 데이터 형식이 다르면 타임스탬프 데이터 형식이 반환됩니다.

- not_null_return_value와 null_return_value의 문자 데이터 형식이 다르면 not_null_return_value의 데이터 형식이 반환됩니다.
- not_null_return_value와 null_return_value의 데이터 형식이 숫자와 비숫자로 섞여 있으면 not_null_return_value의 데이터 형식이 반환됩니다.

> **Important**
>
> not_null_return_value의 데이터 형식이 반환되는 마지막 두 경우에서는 null_return_value가 묵시적으로 해당 데이터 형식으로 변환됩니다. 이때 데이터 형식이 서로 호환되지 않으면 함수가 중단됩니다.

사용 노트

NVL2일 때는 not_null_return_value 또는 null_return_value 파라미터의 값 중에서 함수에서 선택하는 값과 함께 not_null_return_value의 데이터 형식이 반환됩니다.

예를 들어 column1이 NULL이라고 가정하면 다음 두 쿼리에서 동일한 값이 반환됩니다. 하지만 DECODE의 반환 값 데이터 형식은 INTEGER인 반면 NVL2의 반환 값 데이터 형식은 VARCHAR입니다.

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

예제

다음은 일부 샘플 데이터를 수정한 후 두 필드를 평가하여 적합한 사용자 연락처를 제공하는 예입니다.

```
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 함수

두 인수를 비교하여 동일한 경우에는 NULL을 반환합니다. 같지 않으면 첫 번째 인수가 반환됩니다.

구문

NULLIF 표현식은 두 인수를 비교하여 동일한 경우에는 NULL을 반환합니다. 같지 않으면 첫 번째 인수가 반환됩니다. 이 표현식은 NVL 또는 COALESCE 표현식의 정반대입니다.

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

인수

expression1, expression2

비교 대상인 열 또는 표현식입니다. 반환 형식은 첫 번째 표현식의 형식과 동일합니다.

예시

다음 예에서는 인수가 같지 않기 때문에 쿼리가 문자열 `first`를 반환합니다.

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

case
-------
```

```
first
```

다음 예에서는 리터럴 인수가 같기 때문에 쿼리가 NULL을 반환합니다.

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

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

다음 예에서는 정수 인수가 같지 않기 때문에 쿼리가 1을 반환합니다.

```
SELECT NULLIF(1, 2);

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

다음 예에서는 정수 인수가 같기 때문에 쿼리가 NULL을 반환합니다.

```
SELECT NULLIF(1, 1);

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

다음은 LISTID와 SALESID 값이 일치할 때 쿼리가 NULL을 반환하는 예입니다.

```
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)
```

생성자 함수

SQL 생성자 함수는 배열 또는 맵과 같은 새 데이터 구조를 생성하는 데 사용되는 함수입니다.

일부 입력 값을 가져와서 새 데이터 구조 객체를 반환합니다. 생성자 함수는 일반적으로 ARRAY 또는 MAP와 같이 생성한 데이터 형식의 이름을 따서 명명됩니다.

생성자 함수는 기존 데이터에서 작동하고 단일 값을 반환하는 스칼라 함수 또는 집계 함수와 다릅니다. 생성자 함수는 추가 데이터 처리 또는 분석에 사용할 수 있는 새 데이터 구조를 생성하는 데 사용됩니다.

AWS Clean Rooms 는 다음 생성자 함수를 지원합니다.

주제

- MAP 생성자 함수
- NAMED_STRUCT 생성자 함수
- STRUCT 생성자 함수

MAP 생성자 함수

MAP 생성자 함수는 지정된 키/값 페어로 맵을 생성합니다.

MAP과 같은 구성자 함수는 SQL 쿼리 내에서 프로그래밍 방식으로 새 데이터 구조를 생성해야 할 때 유용합니다. 이를 통해 추가 데이터 처리 또는 분석에 사용할 수 있는 복잡한 데이터 구조를 구축할 수 있습니다.

구문

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

인수

키0

유사한 유형의 표현식입니다. 모든 key0는 가장 덜 일반적인 유형을 공유해야 합니다.

값 0

모든 유형의 표현식입니다. 모든 valueN은 가장 덜 일반적인 유형을 공유해야 합니다.

반환

MAP 함수는 키가 key0의 최소 공통 유형으로 입력되고 값이 value0의 최소 공통 유형으로 입력되는 MAP을 반환합니다.

예시

다음 예제에서는 두 개의 키-값 페어로 새 맵을 생성합니다. 키`1.0`는 값과 연결됩니다`'2'`. 키는 값과 `3.0` 연결됩니다`'4'`. 그러면 결과 맵이 SQL 문의 출력으로 반환됩니다.

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

NAMED_STRUCT 생성자 함수

NAMED_STRUCT 생성자 함수는 지정된 필드 이름과 값을 사용하여 구조체를 생성합니다.

NAMED_STRUCT와 같은 구성자 함수는 SQL 쿼리 내에서 프로그래밍 방식으로 새 데이터 구조를 생성해야 할 때 유용합니다. 이를 통해 추가 데이터 처리 또는 분석에 사용할 수 있는 구조 또는 레코드와 같은 복잡한 데이터 구조를 구축할 수 있습니다.

구문

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

인수

이름1

STRING 리터럴 이름 지정 필드 1.

val1

필드 1의 값을 지정하는 모든 유형의 표현식입니다.

반환

NAMED_STRUCT 함수는 val1 유형과 일치하는 필드 1이 있는 구조체를 반환합니다.

예시

다음 예제에서는 이름이 지정된 세 개의 필드가 있는 새 구조를 생성합니다. 필드에 값이 할당"a"됩니다1. 필드에"b"는 값이 할당됩니다2.. 필드에"c"는 값이 할당됩니다3. 그러면 결과 구조가 SQL 문의 출력으로 반환됩니다.

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

STRUCT 생성자 함수

STRUCT 생성자 함수는 지정된 필드 값으로 구조체를 생성합니다.

STRUCT와 같은 구성자 함수는 SQL 쿼리 내에서 프로그래밍 방식으로 새 데이터 구조를 생성해야 할 때 유용합니다. 이를 통해 추가 데이터 처리 또는 분석에 사용할 수 있는 구조 또는 레코드와 같은 복잡한 데이터 구조를 구축할 수 있습니다.

구문

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

인수

col1

열 이름 또는 유효한 표현식입니다.

반환

STRUCT 함수는 expr1 유형과 일치하는 field1이 있는 구조체를 반환합니다.

인수의 이름이 참조인 경우 이름을 사용하여 필드의 이름을 지정합니다. 그렇지 않으면 필드의 이름이 colN으로 지정되며, 여기서 N은 구조체에서 필드의 위치입니다.

예시

다음 예제에서는 세 개의 필드가 있는 새 구조체를 생성합니다. 첫 번째 필드에는 값 1이 할당됩니다. 두 번째 필드에는 값 2가 할당됩니다. 세 번째 필드에는 값 3이 할당됩니다. 기본적으로 결과 구조체의 필드는 인수 목록의 위치에 따라 col3, col1 col2및 로 이름이 지정됩니다. 그러면 결과 구조가 SQL 문의 출력으로 반환됩니다.

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

데이터 형식 지정 함수

데이터 형식 지정 함수를 사용하면 한 가지 데이터 형식에서 다른 데이터 형식으로 값을 변환할 수 있습니다. 이러한 함수에서는 모두 첫 번째 인수가 형식을 지정할 대상이 되는 값이고, 두 번째 인수에는 새로운 형식을 위한 템플릿이 포함되어 있습니다.

AWS Clean Rooms Spark SQL은 여러 데이터 형식 지정 함수를 지원합니다.

주제

- BASE64 함수
- CAST 함수
- DECODE 함수
- ENCODE 함수
- HEX 함수
- STR_TO_MAP 함수
- TO_CHAR
- TO_DATE 함수
- TO_NUMBER
- UNBASE64 함수
- UNHEX 함수
- 날짜/시간 형식 문자열
- 숫자 형식 문자열

BASE64 함수

BASE64 함수는 MIME용 RFC2045 Base64 전송 인코딩을 사용하여 표현식을 기본 64 문자열로 변환합니다.

구문

```
base64(expr)
```

인수

expr

함수가 BINARY로 해석할 BINARY 표현식 또는 STRING입니다.

반환 타입

STRING

예시

지정된 문자열 입력을 Base64 인코딩 표현으로 변환하려면 다음 예제를 사용합니다. 결과는 'U3BhcmsgU1FM'인 입력 문자열 'Spark SQL'의 Base64 인코딩 표현입니다.

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

CAST 함수

CAST 함수는 한 데이터 유형을 다른 호환 가능한 데이터 유형으로 변환합니다. 예를 들어 문자열을 날짜로 변환하거나 숫자 형식을 문자열로 변환할 수 있습니다. CAST는 런타임 변환을 수행합니다. 즉, 변환을 수행해도 원본 테이블의 값 데이터 형식은 변경되지 않습니다. 쿼리의 컨텍스트에서만 변경됩니다.

특정 데이터 유형은 CAST 함수를 사용하여 다른 데이터 유형으로 명시적으로 변환해야 합니다. 다른 데이터 형식은 CAST를 사용하지 않고 다른 명령의 일부로 암시적으로 변환할 수 있습니다. 형식 호환성 및 변환을(를) 참조하세요.

구문

한 가지 데이터 형식에서 다른 데이터 형식으로 표현식을 변환할 때는 다음과 같이 두 가지 동등한 구문 형식을 사용할 수 있습니다.

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

인수

expression

열 이름이나 리터럴 같이 하나 이상의 값으로 평가되는 표현식입니다. null 값을 변환하면 마찬가지로 null이 반환됩니다. 표현식에는 공백이나 빈 문자열이 포함되면 안 됩니다.

type

BINARY 및 BINARY VARYING 데이터 형식을 데이터 타입 제외하고 지원되는 중 하나입니다.

반환 타입

CAST는 type 인수에서 지정하는 데이터 형식을 반환합니다.

> **Note**
>
> AWS Clean Rooms 다음과 같이 정밀도를 잃는 DECIMAL 변환과 같은 문제가 있는 변환을 수행하려고 하면가 오류를 반환합니다.
>
> ```
> select 123.456::decimal(2,1);
> ```
>
> 둘째, 오버플로우를 야기하는 INTEGER 변환입니다.
>
> ```
> select 12345678::smallint;
> ```

예시

다음에 동등한 2개의 쿼리입니다. 두 쿼리 모두 소수 값을 정수로 변환합니다.

```
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)
```

다음과 같은 결과가 나옵니다. 실행하는 데 샘플 데이터가 필요하지 않습니다.

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

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

이 예에서는 각 결과에서 시간이 제거되고 타임스탬프 열의 값이 날짜로 변환됩니다.

```
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)
```

이전 예에서 설명한 대로 CAST를 사용하지 않은 경우 결과에 2008-02-18 02:36:48처럼 시간이 포함됩니다.

다음 쿼리는 가변 문자 데이터를 날짜로 캐스팅합니다. 실행하는 데 샘플 데이터가 필요하지 않습니다.

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

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

다음은 날짜 열의 값이 타임스탬프로 변환된 예입니다.

```
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)
```

이전 예와 같은 경우에는 TO_CHAR를 사용하여 출력 형식을 추가로 제어할 수 있습니다.

다음은 정수가 문자열로 변환된 예입니다.

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

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

다음은 DECIMAL(6,3) 값이 DECIMAL(4,1) 값으로 변환된 예입니다.

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

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

이 예에서는 좀 더 복잡한 표현식을 보여줍니다. 다음은 SALES 테이블의 PRICEPAID 열(DECIMAL(8,2) 열)을 DECIMAL(38,2) 열로 변환하고, 값을 10000000000000000000과 곱합니다.

```
select salesid, pricepaid::decimal(38,2)*10000000000000000000
as value from sales where salesid<10 order by salesid;
```

```
 salesid |           value
---------+----------------------------
       1 | 72800000000000000000000.00
       2 |  76000000000000000000000.00
       3 | 35000000000000000000000.00
       4 | 17500000000000000000000.00
       5 | 15400000000000000000000.00
       6 | 39400000000000000000000.00
       7 | 78800000000000000000000.00
       8 | 19700000000000000000000.00
       9 | 59100000000000000000000.00

(9 rows)
```

DECODE 함수

DECODE 함수는 특정 문자 인코딩을 사용하여 문자열을 이진 형식으로 변환하는 데 사용되는 ENCODE 함수의 대응 함수입니다. DECODE 함수는 이진 데이터를 가져와서 지정된 문자 인코딩을 사용하여 읽을 수 있는 문자열 형식으로 다시 변환합니다.

이 함수는 데이터베이스에 저장된 이진 데이터로 작업해야 하고 사람이 읽을 수 있는 형식으로 표시해야 하거나 서로 다른 문자 인코딩 간에 데이터를 변환해야 하는 경우에 유용합니다.

구문

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

인수

expr

문자 집합으로 인코딩된 BINARY 표현식입니다.

문자 집합

STRING 표현식입니다.

지원되는 문자 집합 인코딩(대/소문자 비구분): 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE'및 'UTF-16'.

반환 타입

DECODE 함수는 STRING을 반환합니다.

예시

다음 예제에는 UTF-8 문자 인코딩을 사용하여 메시지 데이터를 이진 형식으로 message_text 저장하는 라는 열이 messages 있는 라는 테이블이 있습니다. DECODE 함수는 이진 데이터를 다시 읽기 가능한 문자열 형식으로 변환합니다. 이 쿼리의 출력은 메시지 테이블에 저장된 메시지의 읽을 수 있는 텍스트이며, ID는 'utf-8' 인코딩을 사용하여 이진 형식에서 문자열로 123변환됩니다.

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

ENCODE 함수

ENCODE 함수는 지정된 문자 인코딩을 사용하여 문자열을 이진 표현으로 변환하는 데 사용됩니다.

이 함수는 이진 데이터로 작업해야 하거나 다른 문자 인코딩 간에 변환해야 하는 경우에 유용합니다. 예를 들어 바이너리 스토리지가 필요한 데이터베이스에 데이터를 저장할 때 또는 다른 문자 인코딩을 사용하는 시스템 간에 데이터를 전송해야 할 때 ENCODE 함수를 사용할 수 있습니다.

구문

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

인수

문자열

인코딩할 STRING 표현식입니다.

문자 집합

인코딩을 지정하는 STRING 표현식입니다.

지원되는 문자 집합 인코딩(대/소문자 비구분): 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE'및 'UTF-16'.

반환 타입

ENCODE 함수는 BINARY를 반환합니다.

예시

다음 예제에서는 'utf-8' 인코딩을 사용하여 문자열을 이진 표현'abc'으로 변환합니다.이 경우 원래 문자열이 반환됩니다. 이는 'utf-8' 인코딩이 문자당 단일 바이트를 사용하여 전체 ASCII 문자 세트('a', 'b'및 문자 포함'c')를 나타낼 수 있는 가변 너비 문자 인코딩이기 때문입니다. 따라서를 'abc' 사용하는의 이진 표현'utf-8'은 원래 문자열과 동일합니다.

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

HEX 함수

HEX 함수는 숫자 값(정수 또는 부동 소수점 숫자)을 해당 16진수 문자열 표현으로 변환합니다.

16진수는 16개의 고유 기호(0~9 및 A~F)를 사용하여 숫자 값을 나타내는 숫자 시스템입니다. 컴퓨터 과학 및 프로그래밍에서 일반적으로 이진 데이터를 보다 간결하고 사람이 읽을 수 있는 형식으로 표현하는 데 사용됩니다.

구문

```
hex(expr)
```

인수

expr

BIGINT, BINARY 또는 STRING 표현식입니다.

반환 타입

HEX는 STRING을 반환합니다. 함수는 인수의 16진수 표현을 반환합니다.

예시

다음 예제에서는 정수 값 17을 입력으로 받아 HEX() 함수를 적용합니다. 출력은 입력 값의 16진수 표현11인 입니다17.

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

다음 예제에서는 문자열을 16진수 표현'Spark_SQL'으로 변환합니다. 출력은 입력 문자열의 16진수 표현537061726B2053514C인 입니다'Spark_SQL'.

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

이 예제에서는 문자열 'Spark_SQL'이 다음과 같이 변환됩니다.

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

이러한 16진수 값을 연결하면 최종 출력이 "됩니다537061726B2053514C".

STR_TO_MAP 함수

STR_TO_MAP 함수는 string-to-map 변환 함수입니다. 맵(또는 사전)의 문자열 표현을 실제 맵 데이터 구조로 변환합니다.

이 함수는 SQL에서 맵 데이터 구조를 사용해야 하지만 처음에는 데이터가 문자열로 저장되는 경우에 유용합니다. 문자열 표현을 실제 맵으로 변환하면 맵 데이터에 대한 작업 및 조작을 수행할 수 있습니다.

구문

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

인수

텍스트

맵을 나타내는 STRING 표현식입니다.

pairDelim

항목을 구분하는 방법을 지정하는 선택적 STRING 리터럴입니다. 기본적으로 쉼표()로 설정됩니다','.

keyValueDelim

각 키-값 페어를 구분하는 방법을 지정하는 선택적 STRING 리터럴입니다. 기본적으로 콜론()으로 설정됩니다':'.

반환 타입

STR_TO_MAP 함수는 키와 값 모두에 대해 STRING의 MAP을 반환합니다. pairDelim과 keyValueDelim은 모두 정규식으로 처리됩니다.

예시

다음 예제에서는 입력 문자열과 두 구분 기호 인수를 가져와서 문자열 표현을 실제 맵 데이터 구조로 변환합니다. 이 특정 예제에서 입력 문자열은 다음 키-값 페어가 있는 맵을 'a:1,b:2,c:3' 나타냅니다. 'a'는 키이고, '1'는 값입니다. 'b'는 키이고, '2'는 값입니다. 'c'는 키이고, '3'는 값입니다. ',' 구분 기호는 키-값 페어를 구분하는 데 사용되며 ':' 구분 기호는 각 페어 내에서 키와 값을 구분하는 데 사용됩니다. 이 쿼리의 출력은 입니다{"a":"1","b":"2","c":"3"}. 이는 결과 맵 데이터 구조입니다. 여기서 키는 'a', 'b'및 'c'이고 해당 값은 '1', '2'및 입니다'3'.

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

다음 예제는 STR_TO_MAP 함수가 입력 문자열이 특정 형식이고 키-값 페어가 올바르게 구분될 것으로 예상한다는 것을 보여줍니다. 입력 문자열이 예상 형식과 일치하지 않는 경우 함수는 여전히 맵 생성을 시도하지만 결과 값이 예상과 다를 수 있습니다.

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

TO_CHAR

TO_CHAR는 타임스탬프 또는 숫자 표현식을 문자열 데이터 형식으로 변환합니다.

구문

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

인수

timestamp_expression

TIMESTAMP 또는 TIMESTAMPTZ 형식 값이나, 혹은 묵시적으로 타임스탬프로 강제 변환할 수 있는 값으로 평가되는 표현식입니다.

numeric_expression

숫자 데이터 형식 값이나, 혹은 묵시적으로 숫자 형식으로 강제 변환할 수 있는 값으로 평가되는 표현식입니다. 자세한 내용은 숫자형 단원을 참조하십시오. TO_CHAR는 숫자 문자열의 왼쪽에 공백을 삽입합니다.

> **Note**
>
> TO_CHAR는 128비트 DECIMAL 값을 지원하지 않습니다.

format

새로운 값의 형식입니다. 유효한 형식은 날짜/시간 형식 문자열 및 숫자 형식 문자열 섹션을 참조하세요.

반환 타입

VARCHAR

예시

다음 예는 타임스탬프를 9자에 덧붙인 월 이름, 요일 이름, 날짜를 포함한 형식의 날짜 및 시간 값으로 변환합니다.

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

다음 예는 타임스탬프를 연중 일 번호 값으로 변환합니다.

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

to_char
```

```
-------------------------
365
```

다음 예는 타임스탬프를 ISO 요일 번호로 변환합니다.

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

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

다음 예는 날짜에서 월 이름을 추출합니다.

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

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

다음은 EVENT 테이블의 STARTTIME 값을 각각 시간, 분 및 초로 구성된 문자열로 변환하는 예입니다.

```
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)
```

다음은 전체 타임스탬프 값을 다른 형식으로 변환하는 예입니다.

```
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)
```

다음은 타임스탬프 리터럴을 문자열로 변환하는 예입니다.

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

다음은 숫자를 끝에 음의 부호가 있는 문자열로 변환하는 예입니다.

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

다음은 숫자를 통화 기호가 있는 문자열로 변환하는 예입니다.

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

다음은 음수에 꺾쇠 괄호를 사용하여 숫자를 문자열로 변환하는 예입니다.

```
select to_char(-125.88, '$999D99PR');
to_char
---------
$<125.88>
(1 row)
```

다음은 숫자를 로마 숫자 문자열로 변환하는 예입니다.

```
select to_char(125, 'RN');
to_char
---------
CXXV
```

```
(1 row)
```

다음 예에서는 요일을 표시합니다.

```
SELECT to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS');
               to_char
-----------------------
Wednesday, 31 09:34:26
```

다음 예에서는 숫자의 서수 접미사를 표시합니다.

```
SELECT to_char(482, '999th');
               to_char
-----------------------
 482nd
```

다음은 SALES 테이블의 지불 가격에서 수수료를 빼는 예입니다. 그런 다음 차액을 반올림하여 to_char 열과 같이 로마 숫자로 변환합니다.

```
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)
```

다음은 to_char 열과 같이 통화 기호를 차액 값에 추가하는 예입니다.

```
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)
```

다음은 각 판매가 이루어진 세기를 나열하는 예입니다.

```
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)
```

다음은 EVENT 테이블의 STARTTIME 값을 각각 시간, 분, 초 및 시간대로 구성된 문자열로 변환하는 예입니다.

```
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)
```

다음은 초, 밀리초, 마이크로초에 따라 형식을 지정하는 예입니다.

```
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 함수

TO_DATE는 문자열로 표현된 날짜를 DATE 데이터 형식으로 변환합니다.

구문

```
TO_DATE (date_str)
```

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

인수

date_str

날짜 문자열 또는 날짜 문자열로 캐스팅할 수 있는 데이터 유형입니다.

format

Spark의 날짜/시간 패턴과 일치하는 문자열 리터럴입니다. 유효한 날짜/시간 패턴은 서식 지정 및 구문 분석을 위한 날짜/시간 패턴을 참조하세요.

반환 타입

TO_DATE는 format 값에 따라 DATE를 반환합니다.

format으로의 변환이 실패하면 오류가 반환됩니다.

예시

다음 SQL 문은 날짜 `02 Oct 2001`을 날짜 데이터 형식으로 변환합니다.

```
select to_date('02 Oct 2001', 'dd MMM yyyy');

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

다음 SQL 문은 문자열 `20010631`을 날짜로 변환합니다.

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

다음 SQL 문은 문자열 `20010631`을 날짜로 변환합니다.

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

6월에 30일만 있기 때문에 결과는 null 값입니다.

```

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

TO_NUMBER

TO_NUMBER는 문자열을 숫자(소수) 값으로 변환합니다.

구문

```
to_number(string, format)
```

인수

string

실행할 문자열입니다. 형식은 리터럴 값이 되어야 합니다.

format

두 번째 인수는 숫자 값 생성을 위한 문자열의 구문 분석 방식을 나타내는 형식 문자열입니다. 예를 들어 format이 '99D999'이면 변환 대상인 문자열이 5자리로 구성되어 있으며 세 번째 자리에 소수점이 있는 것을 의미합니다. 따라서 to_number('12.345','99D999')는 숫자 값으로 12.345를 반환합니다. 유효한 형식 목록은 숫자 형식 문자열 섹션을 참조하세요.

반환 타입

TO_NUMBER는 DECIMAL 숫자를 반환합니다.

format으로의 변환이 실패하면 오류가 반환됩니다.

예시

다음은 문자열 12,454.8-을 숫자로 변환하는 예입니다.

```
select to_number('12,454.8-', '99G999D9S');

to_number
-----------
-12454.8
```

다음은 문자열 $ 12,454.88을 숫자로 변환하는 예입니다.

```
select to_number('$ 12,454.88', 'L 99G999D99');

to_number
-----------
12454.88
```

다음은 문자열 $ 2,012,454.88을 숫자로 변환하는 예입니다.

```
select to_number('$ 2,012,454.88', 'L 9,999,999.99');

to_number
-----------
```

```
2012454.88
```

UNBASE64 함수

UNBASE64 함수는 인수를 기본 64 문자열에서 바이너리로 변환합니다.

Base64 인코딩은 일반적으로 다양한 통신 채널(예: 이메일, URL 파라미터 또는 데이터베이스 스토리지)을 통해 전송하기에 안전한 텍스트 형식으로 이진 데이터(예: 이미지, 파일 또는 암호화된 정보)를 나타내는 데 사용됩니다.

UNBASE64 함수를 사용하면이 프로세스를 되돌리고 원래 바이너리 데이터를 복구할 수 있습니다. 이러한 유형의 기능은 Base64를 데이터 전송 메커니즘으로 사용하는 외부 시스템 또는 APIs 하는 시나리오에서 유용할 수 있습니다. Base64

구문

```
unbase64(expr)
```

인수

expr

base64 형식의 STRING 표현식입니다.

반환 타입

BINARY

예시

다음 예제에서는 Base64-encoded 문자열'U3BhcmsgU1FM'이 원래 문자열 로 다시 변환됩니다'Spark SQL'.

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

UNHEX 함수

UNHEX 함수는 16진수 문자열을 원래 문자열 표현으로 다시 변환합니다.

이 함수는 16진수 형식으로 저장되거나 전송된 데이터로 작업해야 하고 추가 처리 또는 표시를 위해 원래 문자열 표현을 복원해야 하는 시나리오에서 유용할 수 있습니다.

UNHEX 함수는 HEX 함수와 일치합니다.

구문

```
unhex(expr)
```

인수

expr

16진수 문자의 STRING 표현식입니다.

반환 타입

UNHEX는 BINARY를 반환합니다.

expr의 길이가 홀수인 경우 첫 번째 문자는 삭제되고 결과는 null 바이트로 채워집니다. expr에 16진수가 아닌 문자가 포함된 경우 결과는 NULL입니다.

예시

다음 예제에서는 UNHEX() 및 DECODE() 함수를 함께 사용하여 16진수 문자열을 원래 문자열 표현으로 다시 변환합니다. 쿼리의 첫 번째 부분에서는 UNHEX() 함수를 사용하여 16진수 문자열 '537061726B2053514C'를 이진 표현으로 변환합니다. 쿼리의 두 번째 부분에서는 DECODE() 함수를 사용하여 UNHEX() 함수에서 가져온 이진 데이터를 'UTF-8' 문자 인코딩을 사용하여 문자열로 다시 변환합니다. 쿼리의 출력은 16진수로 변환된 다음 다시 문자열로 변환된 원래 문자열 'Spark_SQL'입니다.

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

날짜/시간 형식 문자열

다음과 같은 일반적인 시나리오에서 날짜/시간 패턴을 사용할 수 있습니다.

- CSV 및 JSON 데이터 소스를 사용하여 날짜/시간 콘텐츠를 구문 분석하고 형식을 지정하는 경우
- 다음과 같은 함수를 사용하여 문자열 유형과 날짜 또는 타임스탬프 유형 간에 변환하는 경우:
  - unix_timestamp
  - date_format

- to_unix_timestamp
- from_unixtime
- to_date
- to_timestamp
- from_utc_timestamp
- to_utc_timestamp

날짜 및 타임스탬프 구문 분석과 형식 지정에는 다음 표의 패턴 문자를 사용합니다.

날짜 부분 또는 시간 부분	의미	예시
a	오전 또는 오후로 표시되는 오늘의 오전 또는 오후	PM
D	3자리 숫자로 표시된 요일	189
d	2자리 숫자로 표시된 월의 일	28
E	텍스트로 표시되는 요일	화요일 화요일
F	1자리 숫자로 표시된 월의 정렬된 요일	3
G	텍스트로 표시되는 Era 표시기	AD Anno Domini
h	오전 또는 오후 시간, 2자리 숫자로 표시	12
H	시간대, 0~23 사이의 2자리 숫자로 표시	0
k	1~24 사이의 2자리 숫자로 표시되는 시간대	1

날짜 부분 또는 시간 부분	의미	예시
K	오전 또는 오후 시간, 0~11 사이의 2자리 숫자로 표시	0
m	2자리 숫자로 표시되는 분	30
M/L	해당 연도의 월, 한 달로 표시	7 07 7월 7월
O	UTC의 현지화된 영역 오프셋	GMT+8 GMT+8:00 UTC-08:00
Q/q	해당 연도의 분기, 숫자(1~4) 또는 텍스트로 표시	3 03 Q3 3분기
s	2자리 숫자로 표시된 1분초	55
S	분수로 표시되는 1초의 분수	978
V	zone-id로 표시되는 시간대 식별자	America/Los_Angeles Z 08:30

날짜 부분 또는 시간 부분	의미	예시
x	UTC에서의 영역 오프셋(offset-X)	+0000 -08 -0830 -08:30 -083015 -08:30:15
X	UTC에서의 영역 오프셋. 여기서 Z는 0을 나타냅니다.	Z -08 -0830 -08:30 -083015 -08:30:15
y	연도, 연도로 표시	2020 20
z	텍스트로 표시되는 시간대 이름	태평양 표준시 PST
Z	UTC에서의 영역 오프셋(오프셋-Z)	+0000 -0800 -08:00
'	구분 기호로 표시되는 텍스트 이스케이프	N/A

날짜 부분 또는 시간 부분	의미	예시
"	리터럴로 표시되는 작은따옴표	'
[	선택적 섹션 시작	N/A
]	선택적 섹션 끝	N/A

패턴 문자 수에 따라 형식 유형이 결정됩니다.

텍스트 형식

- 축약된 양식에 1~3자 사용(예: 월요일의 경우 "월")
- 전체 양식에 정확히 4개의 문자를 사용합니다(예: "월요일").
- 5개 이상의 문자를 사용하지 마세요. 그러면 오류가 발생합니다.

숫자 형식(n)

- 값 n은 허용되는 최대 문자 수를 나타냅니다.
- 단일 문자 패턴의 경우:
  - 출력은 패딩 없이 최소 자릿수를 사용합니다.
- 여러 문자 패턴의 경우:
  - 출력은 문자 수 너비와 일치하도록 0으로 채워집니다.
- 구문 분석 시 입력에는 정확한 자릿수가 포함되어야 합니다.

숫자/텍스트 형식

- 문자가 3개 이상인 경우 텍스트 형식 규칙을 따릅니다.
- 문자 수를 줄이려면 숫자 형식 규칙을 따르세요.

분수 형식

- 1~9개의 'S' 문자 사용(예: SSSSSS)
- 구문 분석의 경우:
  - 1과 S 문자 수 사이의 분수 허용

- 형식 지정의 경우:
  - S 문자 수와 일치하는 0이 있는 패드
- 마이크로초 정밀도에 대해 최대 6자리 지원
- 나노초를 구문 분석할 수 있지만 추가 숫자를 잘라냅니다.

연도 형식

- 문자 수는 패딩의 최소 필드 너비를 설정합니다.
- 두 글자의 경우:
  - 마지막 두 숫자를 인쇄합니다.
  - 2000-2099 사이의 연도 구문 분석
- 4자 미만(2자 제외):
  - 음수 연도에 대해서만 기호를 표시합니다.
- 7개 이상의 문자를 사용하지 마세요. 그러면 오류가 발생합니다.

월 형식

- 표준 양식의 경우 'M'을 사용하고 독립 실행형 양식의 경우 'L'을 사용합니다.
- 단일 'M' 또는 'L':
  - 패딩 없이 월 번호 1~12 표시
- 'MM' 또는 'LL':
  - 패딩이 있는 월 번호 01~12를 표시합니다.
- 'MMM':
  - 표준 형식으로 축약된 월 이름을 표시합니다.
  - 전체 날짜 패턴의 일부여야 합니다.
- 'LLL':
  - 독립 실행형 형식으로 축약된 월 이름을 표시합니다.
  - 월 전용 형식 지정에 사용
- 'MMMM':
  - 표준 형식으로 전체 월 이름을 표시합니다.
  - 날짜 및 타임스탬프에 사용

- 'LLLL':
  - 전체 월 이름을 독립 실행형 형식으로 표시합니다.
  - 월 전용 형식 지정에 사용

시간대 형식

- am-pm: 문자 1개만 사용
- 영역 ID(V): 문자 2개만 사용
- 영역 이름(z):
  - 1~3자: 짧은 이름을 표시합니다.
  - 4자: 전체 이름을 표시합니다.
  - 5개 이상의 문자를 사용하지 마세요.

오프셋 형식

- X 및 x:
  - 문자 1개: 시간(+01) 또는 시간-분(+0130)을 표시합니다.
  - 문자 2개: 콜론 없이 시간-분 표시(+0130)
  - 문자 3개: 콜론(+01:30)이 있는 시간-분을 표시합니다.
  - 4자: 콜론 없이 hour-minute-second 표시(+013015)
  - 5자: 콜론(+01:30:15)이 있는 hour-minute-second를 표시합니다.
  - X는 제로 오프셋에 'Z'를 사용합니다.
  - x는 제로 오프셋에 '+00', '+0000' 또는 '+00:00'을 사용합니다.
- O:
  - 문자 1개: 짧은 형식 표시(GMT+8)
  - 4자: 전체 형식 표시(GMT+08:00)
- Z:
  - 1~3자: 콜론 없이 시간-분 표시(+0130)
  - 4자: 현지화된 전체 양식을 표시합니다.
  - 5자: 콜론이 있는 hour-minute-second를 표시합니다.

선택적 예제

- 대괄호[]를 사용하여 선택적 콘텐츠 표시
- 선택적 섹션을 중첩할 수 있습니다.
- 모든 유효한 데이터가 출력에 표시됩니다.
- 입력에서 전체 선택적 섹션을 생략할 수 있습니다.

> **Note**
>
> 기호 'E', 'F', 'q' 및 'Q'는 날짜/시간 형식(예: date_format)에서만 작동합니다. 날짜/시간 구문 분석(예: to_timestamp)에 사용하지 마세요.

숫자 형식 문자열

다음 숫자 형식 문자열은 TO_NUMBER 및 TO_CHAR와 같은 함수에 적용됩니다.

- 문자열을 숫자 형식으로 지정하는 예는 TO_NUMBER 섹션을 참조하세요.
- 숫자를 문자열 형식으로 지정하는 예는 TO_CHAR 섹션을 참조하세요.

형식	Description
9	지정한 자리 수를 포함한 숫자 값
0	선행 제로를 포함한 숫자 값
.(마침표), D	소수점
, (comma)	천 단위 구분자
CC	세기 코드. 예를 들어 21세기는 2001-01-01부터 시작됩니다(TO_CHAR에서만 지원됨).
FM	채우기 모드. 공백 및 제로 채움은 제한됩니다.
PR	꺾쇠 괄호 안의 음수 값
S	숫자에 첨부되는 부호
L	지정한 위치의 통화 기호

형식	Description
G	그룹 구분자
MI	0보다 작은 숫자일 때 지정한 위치의 마이너스 부호
PL	0보다 큰 숫자일 때 지정한 위치의 플러스 부호
SG	지정한 위치의 플러스 또는 마이너스 부호
RN	1부터 3999 사이의 로마 숫자(TO_CHAR에 한해 지원됨)
TH 또는 th	서수 접미사. 분수 또는 0보다 작은 값은 변환하지 않습니다.

날짜 및 시간 함수

날짜 및 시간 함수를 사용하면 날짜의 일부 추출, 날짜 계산 수행, 날짜 및 시간 형식 지정, 현재 날짜 및 시간 작업 등 날짜 및 시간 데이터에 대한 광범위한 작업을 수행할 수 있습니다. 이러한 함수는 데이터 분석, 보고 및 임시 데이터와 관련된 데이터 조작과 같은 작업에 필수적입니다.

AWS Clean Rooms 는 다음 날짜 및 시간 함수를 지원합니다.

주제

- ADD_MONTHS 함수
- CONVERT_TIMEZONE 함수
- CURRENT_DATE 함수
- CURRENT_TIMESTAMP 함수
- DATE_ADD 함수
- DATE_DIFF 함수
- DATE_PART 함수
- DATE_TRUNC 함수
- DAY 함수
- DAYOFMONTH 함수

- DAYOFWEEK 함수
- DAYOFYEAR 함수
- EXTRACT 함수
- FROM_UTC_TIMESTAMP 함수
- 시간 함수
- MINUTE 함수
- MONTH 함수
- 두 번째 함수
- TIMESTAMP 함수
- TO_TIMESTAMP 함수
- YEAR 함수
- 날짜 또는 타임스탬프 함수의 날짜 부분

ADD_MONTHS 함수

ADD_MONTHS는 지정한 월 수를 날짜 또는 타임스탬프 값이나 표현식에 더합니다. DATE_ADD 함수가 이와 유사한 기능을 제공합니다.

구문

```
ADD_MONTHS( {date | timestamp}, integer)
```

인수

date | timestamp

날짜 또는 타임스탬프 열이거나, 혹은 묵시적으로 날짜 또는 타임스탬프로 변환되는 표현식입니다. 날짜가 월의 마지막 날이거나, 혹은 결과에 따른 월이 더 짧은 경우에는 함수가 월의 마지막 날을 결과로 반환합니다. 그 밖에 다른 날짜일 때는 날짜 표현식과 동일한 날의 숫자가 결과로 반환됩니다.

integer

양의 또는 음의 정수입니다. 날짜에서 월을 뺄 때는 음의 정수를 사용합니다.

반환 타입

TIMESTAMP

예제

다음은 TRUNC 함수 내에서 ADD_MONTHS 함수를 사용하는 쿼리입니다. TRUNC 함수는 ADD_MONTHS 결과에서 시간 부분을 제거합니다. ADD_MONTHS 함수는 CALDATE 열의 각 값에 12개월을 합산합니다.

```
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)
```

다음은 일 수가 서로 다른 월의 날짜에 대해 ADD_MONTHS 함수를 실행할 경우 동작에 대해서 설명하는 예입니다.

```
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 함수

CONVERT_TIMEZONE은 시간대끼리 타임스탬프를 변환합니다. 이 함수는 일광 절약 시간에 맞춰 자동으로 조정됩니다.

구문

```
CONVERT_TIMEZONE ( ['source_timezone',] 'target_timezone', 'timestamp')
```

인수

source_timezone

(옵션) 현재 타임스탬프의 시간대입니다. 기본값은 UTC입니다.

target_timezone

새로운 타임스탬프의 시간대입니다.

timestamp

타임스탬프 열 또는 묵시적으로 타임스탬프로 변환되는 표현식입니다.

반환 타입

TIMESTAMP

예제

다음은 타임스탬프 값을 기본 UTC 시간대에서 PST로 변환하는 예입니다.

```
select convert_timezone('PST', '2008-08-21 07:23:54');

 convert_timezone
------------------------
2008-08-20 23:23:54
```

다음은 LISTTIME 열의 타임스탬프 값을 기본 UTC 시간대에서 PST로 변환하는 예입니다. 타임스탬프가 일광 절약 시간(PST)에 해당하더라도 대상 시간대가 약어로 지정되어 있기 때문에 스탠다드 타임으로 변환됩니다.

```
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
```

다음은 LISTTIME 열의 타임스탬프 값을 기본 UTC 시간대에서 US/Pacific 시간대로 변환하는 예입니다. 대상 시간대가 시간대 이름을 사용하고 있고, 타임스탬프가 일광 절약 시간에 해당하기 때문에 함수가 일광 절약 시간을 반환합니다.

```
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
```

다음은 타임스탬프 문자열을 EST에서 PST로 변환하는 예입니다.

```
select convert_timezone('EST', 'PST', '20080305 12:25:29');

 convert_timezone
-------------------
2008-03-05 09:25:29
```

다음은 대상 시간대가 시간대 이름(America/New_York)을 사용하고 있고, 타임스탬프가 스탠다드 타임에 해당하기 때문에 타임스탬프를 미국 동부 스탠다드 타임으로 변환하는 예입니다.

```
select convert_timezone('America/New_York', '2013-02-01 08:00:00');

 convert_timezone
---------------------
2013-02-01 03:00:00
(1 row)
```

다음은 대상 시간대가 시간대 이름(America/New_York)을 사용하고 있고, 타임스탬프가 일광 절약 시간에 해당하기 때문에 타임스탬프를 미국 동부 일광 절약 시간으로 변환하는 예입니다.

```
select convert_timezone('America/New_York', '2013-06-01 08:00:00');

 convert_timezone
---------------------
2013-06-01 04:00:00
```

```
(1 row)
```

다음은 오프셋의 사용을 설명하는 예입니다.

```
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 함수

CURRENT_DATE는 현재 세션 시간대(기본 UTC)의 날짜를 기본 형식(YYYY-MM-DD)으로 반환합니다.

> **Note**
>
> CURRENT_DATE는 현재 문이 아닌 현재 트랜잭션의 시작 날짜를 반환합니다. 10/01/08 23:59에 여러 문이 포함된 트랜잭션을 시작하고, CURRENT_DATE가 포함된 문이 10/02/08 00:00에 실행되는 시나리오를 생각해 보세요. CURRENT_DATE는 10/02/08이 아닌 10/01/08을 반환합니다.

구문

```
CURRENT_DATE
```

반환 타입

날짜

예제

다음 예시에서는 현재 날짜(함수가 실행되는 AWS 리전 에서)를 반환합니다.

```
select current_date;
```

```
   date
------------
2008-10-01
```

CURRENT_TIMESTAMP 함수

CURRENT_TIMESTAMP는 날짜, 시간 및 (선택 사항) 밀리초 또는 마이크로초를 포함하여 현재 날짜 및 시간을 반환합니다.

이 함수는 이벤트의 타임스탬프를 기록하거나, 시간 기반 계산을 수행하거나, 날짜/시간 열을 채우는 등 현재 날짜 및 시간을 가져와야 하는 경우에 유용합니다.

구문

```
current_timestamp()
```

반환 타입

CURRENT_TIMESTAMP 함수는 DATE를 반환합니다.

예제

다음 예제에서는 쿼리가 실행되는 시점의 현재 날짜 및 시간, 즉 2020년 4월 25일 오후 15:49:11.914(3:49:11.914)를 반환합니다.

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

다음 예시에서는 `squirrels` 테이블의 각 행에 대한 현재 날짜 및 시간을 검색합니다.

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

DATE_ADD 함수

start_date 이후 num_days인 날짜를 반환합니다.

구문

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

인수

시작_날짜

시작 날짜 값입니다.

num_일

추가할 일 수(정수). 양수는 일을 더하고 음수는 일을 뺍니다.

반환 타입

날짜

예제

다음 예제에서는 날짜에 하루를 추가합니다.

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

Result:
2016-07-31
```

다음 예제에서는 며칠을 추가합니다.

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

Result:
2016-08-04
```

사용 노트

이 설명서는 Spark SQL의 DATE_ADD 함수에 대한 것으로, 일부 다른 SQL 변형에 비해 날짜에 일을 추가할 수 있는 더 간단한 인터페이스를 제공합니다. 월 또는 년과 같은 다른 간격을 추가하려면 다른 함수가 필요할 수 있습니다.

DATE_DIFF 함수

DATE_DIFF는 두 날짜 또는 시간 표현식의 날짜 부분 간의 차이를 반환합니다.

구문

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

인수

endDate

DATE 표현식입니다.

startDate

DATE 표현식입니다.

반환 타입

BIGINT

DATE 열이 있는 예

다음은 두 리터럴 날짜 값 사이의 차이 값(주 수)을 구하는 예입니다.

```
select date_diff(week,'2009-01-01','2009-12-31') as numweeks;

numweeks
----------
52
(1 row)
```

다음은 두 리터럴 날짜 값 사이의 차이 값(시간)을 구하는 예입니다. 날짜의 시간 값을 제공하지 않는 경우 기본값은 00:00:00입니다.

```
select date_diff(hour, '2023-01-01', '2023-01-03 05:04:03');

date_diff
----------
53
(1 row)
```

다음은 두 리터럴 TIMESTAMETZ 값 사이의 차이(일수)를 구하는 예시입니다.

```
Select date_diff(days, 'Jun 1,2008  09:59:59 EST', 'Jul 4,2008  09:59:59 EST')

date_diff
----------
33
```

다음은 테이블의 동일한 행에 있는 두 날짜 사이의 차이 값(일)을 구하는 예입니다.

```
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)
```

다음은 이전 날짜와 오늘 날짜의 리터럴 값 사이에서 차이 값(분기 수)을 구하는 예입니다. 이번 예는 오늘 날짜가 2008년 6월 5일이라는 가정을 전제로 합니다. 날짜 부분은 전체 이름으로 또는 약어로 지정할 수 있습니다. DATE_DIFF 함수의 기본 열 이름은 DATE_DIFF입니다.

```
select date_diff(qtr, '1998-07-01', current_date);

date_diff
-----------
40
(1 row)
```

다음은 SALES 테이블과 LISTING 테이블을 조인하여 두 테이블의 나열 이후 목록 1000부터 1005까지 티켓이 판매된 일수를 계산하는 예입니다. 두 목록에서 가장 길게 기다린 판매 일수는 15일이었고, 가장 짧은 기다린 판매 일수는 1일 미만이었습니다(0일).

```
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)
```

다음은 판매자들이 모든 티켓이 판매될 때까지 기다린 평균 시간을 계산하는 예입니다.

```
select avg(date_diff(hours, listtime, saletime)) as avgwait
from sales, listing
where sales.listid = listing.listid;

avgwait
---------
465
(1 row)
```

TIME 열이 있는 예

다음 예제 테이블 TIME_TEST에는 3개의 값이 삽입된 TIME_VAL(TIME 형식) 열이 있습니다.

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

다음 예에서는 TIME_VAL 열과 시간 리터럴 간의 시간 차이를 찾습니다.

```
select date_diff(hour, time_val, time '15:24:45') from time_test;

 date_diff
-----------
        -5
        15
        15
```

다음 예에서는 두 리터럴 시간 값 간의 분 수 차이를 찾습니다.

```
select date_diff(minute, time '20:00:00', time '21:00:00') as nummins;

nummins
----------
60
```

TIMETZ 열이 있는 예

다음 예제 테이블 TIMETZ_TEST에는 3개의 값이 삽입된 TIMETZ_VAL(TIMETZ 형식) 열이 있습니다.

```
select timetz_val from timetz_test;

timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

다음 예에서는 TIMETZ 리터럴과 timetz_val 간의 시간 차이를 찾습니다.

```
select date_diff(hours, timetz '20:00:00 PST', timetz_val) as numhours from
 timetz_test;

numhours
----------
0
-4
1
```

다음 예에서는 두 리터럴 TIMETZ 값 간의 시간 차이를 찾습니다.

```
select date_diff(hours, timetz '20:00:00 PST', timetz '00:58:00 EST') as numhours;

numhours
----------
1
```

DATE_PART 함수

DATE_PART는 표현식에서 날짜 부분 값을 추출합니다. DATE_PART는 PGDATE_PART 함수의 동의어입니다.

구문

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

인수

필드

소스의 어떤 부분을 추출해야 하며 지원되는 문자열 값은 동등한 함수 EXTRACT의 필드와 동일합니다.

source

필드를 추출해야 하는 DATE 또는 INTERVAL 열입니다.

반환 타입

필드가 'SECOND'인 경우 DECIMAL(8, 6). 다른 모든 경우에는 INTEGER입니다.

예제

다음 예시에서는 날짜 값에서 연도의 일(DOY)을 추출합니다. 출력은 "2019-08-12" 날짜의 연도 요일이임을 보여줍니다224. 즉, 2019년 8월 12일이 2019년 224일입니다.

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

DATE_TRUNC 함수

DATE_TRUNC 함수는 시간, 일 또는 월 등 지정하는 날짜 부분을 기준으로 타임스탬프 표현식 또는 리터럴을 자릅니다.

구문

```
date_trunc(format, datetime)
```

인수

format

잘라낼 단위를 나타내는 형식입니다. 유효한 형식은 다음과 같습니다.

- "YEAR", "YYYY", "YY" - ts가 속하는 연도의 첫 번째 날짜로 자릅니다. 시간 부분은 0이 됩니다.

- "쿼터" - ts가 속하는 분기의 첫 번째 날짜로 자릅니다. 시간 부분은 0이 됩니다.
- "MONTH", "MM", "MON" - ts가 속하는 달의 첫 번째 날짜로 자릅니다. 시간 부분은 0이 됩니다.
- "WEEK" - ts가 속하는 주의 월요일로 자릅니다. 시간 부분은 0이 됩니다.
- "DAY", "DD" - 시간 부분 제로 아웃
- "시간" - 분수 부분으로 분과 초를 0으로 바꿉니다.
- "MINUTE" - 분수 부분으로 두 번째 부분을 제로 아웃합니다.
- "SECOND" - 두 번째 분수 부분 제로 아웃
- "MILLISECOND" - 마이크로초 제로 아웃
- "MICROSECOND" - 모든 것이 남아 있음

ts

날짜/시간 값

반환 타입

형식 모델에 의해 지정된 단위로 잘린 타임스탬프 ts를 반환합니다.

예제

다음 예제에서는 날짜 값을 연초로 잘라냅니다. 출력은 "2015-03-05" 날짜가 2015년 초인 "2015-01-01"로 잘렸음을 보여줍니다.

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

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

DAY 함수

DAY 함수는 날짜/타임스탬프의 요일을 반환합니다.

날짜 추출 함수는 날짜 기반 계산 수행, 데이터 필터링 또는 날짜 값 형식 지정과 같이 날짜 또는 타임스탬프의 특정 구성 요소를 사용해야 할 때 유용합니다.

구문

```
day(date)
```

인수

date

DATE 또는 TIMESTAMP 표현식입니다.

반환

DAY 함수는 INTEGER를 반환합니다.

예제

다음 예제에서는 입력 날짜에서 월의 날짜(30)를 추출합니다'2009-07-30'.

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

다음 예제에서는 squirrels 테이블의 birthday 열에서 월의 날짜를 추출하고 SELECT 문의 출력으로 결과를 반환합니다. 이 쿼리의 출력은 squirrels 테이블의 각 행에 대해 하나씩 일 값 목록이 되며, 이는 각 스쿼럴의 생일에 대한 월의 일을 나타냅니다.

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

DAYOFMONTH 함수

DAYOFMONTH 함수는 날짜/타임스탬프의 요일(월과 연도에 따라 1~31 사이의 값)을 반환합니다.

DAYOFMONTH 함수는 DAY 함수와 비슷하지만 이름이 약간 다르고 동작이 약간 다릅니다. DAY 함수가 더 일반적으로 사용되지만 DAYOFMONTH 함수를 대안으로 사용할 수 있습니다. 이 유형의 쿼리는 추가 처리 또는 보고를 위해 날짜의 특정 구성 요소를 추출하는 등 날짜 또는 타임스탬프 데이터가 포함된 테이블에서 날짜 기반 분석 또는 필터링을 수행해야 하는 경우에 유용할 수 있습니다.

구문

```
dayofmonth(date)
```

인수

date

DATE 또는 TIMESTAMP 표현식입니다.

반환

DAYOFMONTH 함수는 INTEGER를 반환합니다.

예제

다음 예제에서는 입력 날짜에서 월의 날짜(30)를 추출합니다'2009-07-30'.

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

다음 예제에서는 squirrels 테이블의 birthday 열에 DAYOFMONTH 함수를 적용합니다. squirrels 테이블의 각 행에 대해 birthday 열에서 월의 날짜가 추출되어 SELECT 문의 출력으로 반환됩니다. 이 쿼리의 출력은 squirrels 테이블의 각 행에 대해 하나씩 일 값 목록이 되며, 이는 각 스쿼럴의 생일에 대한 월의 일을 나타냅니다.

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

DAYOFWEEK 함수

DAYOFWEEK 함수는 날짜 또는 타임스탬프를 입력으로 사용하고 요일을 숫자로 반환합니다(일요일은 1, 월요일은 2, 토요일은 7).

이 날짜 추출 함수는 날짜 기반 계산 수행, 데이터 필터링 또는 날짜 값 형식 지정과 같이 날짜 또는 타임스탬프의 특정 구성 요소를 사용해야 하는 경우에 유용합니다.

구문

```
dayofweek(date)
```

인수

date

DATE 또는 TIMESTAMP 표현식입니다.

반환

DAYOFWEEK 함수는 INTEGER를 반환합니다.

1 = 일요일

2 = 월요일

3 = 화요일

4 = 수요일

5 = 목요일

6 = 금요일

7 = 토요일

예제

다음 예제에서는이 날짜에서 5(목요일을 나타냄)인 요일을 추출합니다.

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

다음 예제에서는 `squirrels` 테이블의 `birthday` 열에서 요일을 추출하고 SELECT 문의 출력으로 결과를 반환합니다. 이 쿼리의 출력은 `squirrels` 테이블의 각 행에 대해 하나씩 요일 값 목록이 되며, 이는 각 스쿼럴의 생일에 대한 요일을 나타냅니다.

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

DAYOFYEAR 함수

DAYOFYEAR 함수는 날짜 또는 타임스탬프를 입력으로 받아 연도의 날짜(연도 및 윤년 여부에 따라 1~366 사이의 값)를 반환하는 날짜 추출 함수입니다.

이 함수는 날짜 기반 계산 수행, 데이터 필터링 또는 날짜 값 형식 지정과 같이 날짜 또는 타임스탬프의 특정 구성 요소를 사용해야 하는 경우에 유용합니다.

구문

```
dayofyear(date)
```

인수

date

DATE 또는 TIMESTAMP 표현식입니다.

반환

DAYOFYEAR 함수는 INTEGER(연도 및 윤년 여부에 따라 1~366)를 반환합니다.

예제

다음 예제에서는 입력 날짜에서 연도의 날짜(100)를 추출합니다'2016-04-09'.

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

다음 예제에서는 squirrels 테이블의 birthday 열에서 연도의 날짜를 추출하고 SELECT 문의 출력으로 결과를 반환합니다.

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

EXTRACT 함수

EXTRACT 함수는 TIMESTAMP, TIMESTAMPTZ, TIME 또는 TIMETZ 값에서 날짜 또는 시간 부분을 반환합니다. 예에는 타임스탬프의 일, 월, 년, 시, 분, 초, 밀리초 또는 마이크로초가 포함됩니다.

구문

```
EXTRACT(datepart FROM source)
```

인수

datepart

일, 월, 년, 시, 분, 초, 밀리초 또는 마이크로초 등 추출할 날짜 또는 시간의 하위 필드입니다. 에 대해 가능한 값은 날짜 또는 타임스탬프 함수의 날짜 부분 섹션을 참조하세요.

source

TIMESTAMP, TIMESTAMPTZ, TIME 또는 TIMETZ의 데이터 유형으로 평가되는 열 또는 표현식입니다.

반환 타입

source 값이 TIMESTAMP, TIME 또는 TIMETZ의 데이터 유형으로 평가되는 경우 INTEGER입니다.

source 값이 TIMESTAMPTZ의 데이터 유형으로 평가되는 경우 DOUBLE PRECISION입니다.

TIME을 사용한 예제

다음 예제 테이블 TIME_TEST에는 3개의 값이 삽입된 TIME_VAL(TIME 형식) 열이 있습니다.

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

다음 예에서는 각 time_val에서 분을 추출합니다.

```
select extract(minute from time_val) as minutes from time_test;

minutes
-----------
         0
         0
         58
```

다음 예에서는 각 time_val에서 시간을 추출합니다.

```
select extract(hour from time_val) as hours from time_test;

hours
-----------
         20
         0
         0
```

FROM_UTC_TIMESTAMP 함수

FROM_UTC_TIMESTAMP 함수는 입력 날짜를 UTC(협정 세계시)에서 지정된 시간대로 변환합니다.

이 함수는 날짜 및 시간 값을 UTC에서 특정 시간대로 변환해야 하는 경우에 유용합니다. 이는 전 세계 여러 지역에서 시작되어 적절한 현지 시간으로 표시해야 하는 데이터로 작업할 때 중요할 수 있습니다.

구문

```
from_utc_timestamp(timestamp, timezone
```

인수

timestamp

UTC 타임스탬프가 있는 TIMESTAMP 표현식입니다.

시간대

입력 날짜 또는 타임스탬프를 변환해야 하는 유효한 시간대인 STRING 표현식입니다.

반환

FROM_UTC_TIMESTAMP 함수는 TIMESTAMP를 반환합니다.

예제

다음 예제에서는 입력 날짜를 UTC에서 지정된 시간대('Asia/Seoul')로 변환합니다.이 경우 UTC보다 9시간 앞서야 합니다. 결과 출력은 서울 시간대의 날짜 및 시간, 즉 입니다2016-08-31 09:00:00.

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

시간 함수

시간 함수는 시간 또는 타임스탬프를 입력으로 사용하고 시간 구성 요소(0~23 사이의 값)를 반환하는 시간 추출 함수입니다.

이 시간 추출 함수는 시간 기반 계산 수행, 데이터 필터링 또는 시간 값 형식 지정과 같이 시간 또는 타임스탬프의 특정 구성 요소를 사용해야 하는 경우에 유용합니다.

구문

```
hour(timestamp)
```

인수

timestamp

TIMESTAMP 표현식입니다.

반환

HOUR 함수는 INTEGER를 반환합니다.

예제

다음 예시에서는 입력 타임스탬프에서 시간 구성 요소(12)를 추출합니다'2009-07-30 12:58:59'.

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

MINUTE 함수

MINUTE 함수는 시간 또는 타임스탬프를 입력으로 사용하고 분 구성 요소(0~60 사이의 값)를 반환하는 시간 추출 함수입니다.

구문

```
minute(timestamp)
```

인수

timestamp

TIMESTAMP 표현식 또는 유효한 타임스탬프 형식의 STRING입니다.

반환

MINUTE 함수는 INTEGER를 반환합니다.

예제

다음 예제에서는 입력 타임스탬프에서 분 구성 요소(58)를 추출합니다'2009-07-30 12:58:59'.

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

MONTH 함수

MONTH 함수는 시간 또는 타임스탬프를 입력으로 사용하고 월 구성 요소(0~12 사이의 값)를 반환하는 시간 추출 함수입니다.

구문

```
month(date)
```

인수

date

TIMESTAMP 표현식 또는 유효한 타임스탬프 형식의 STRING입니다.

반환

MONTH 함수는 INTEGER를 반환합니다.

예제

다음 예제에서는 입력 타임스탬프에서 월 구성 요소(7)를 추출합니다`'2016-07-30'`.

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

두 번째 함수

SECOND 함수는 시간 또는 타임스탬프를 입력으로 받아 두 번째 구성 요소(0~60 사이의 값)를 반환하는 시간 추출 함수입니다.

구문

```
second(timestamp)
```

인수

timestamp

TIMESTAMP 표현식입니다.

반환

SECOND 함수는 INTEGER를 반환합니다.

예제

다음 예제에서는 입력 타임스탬프에서 두 번째 구성 요소(59)를 추출합니다'2009-07-30 12:58:59'.

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

TIMESTAMP 함수

TIMESTAMP 함수는 값(일반적으로 숫자)을 가져와 타임스탬프 데이터 형식으로 변환합니다.

이 함수는 시간 또는 날짜를 나타내는 숫자 값을 타임스탬프 데이터 형식으로 변환해야 하는 경우에 유용합니다. 이는 Unix 타임스탬프 또는 에포크 시간과 같은 숫자 형식으로 저장된 데이터로 작업할 때 유용할 수 있습니다.

구문

```
timestamp(expr)
```

인수

expr

TIMESTAMP로 캐스팅할 수 있는 모든 표현식입니다.

반환

TIMESTAMP 함수는 TIMESTAMP를 반환합니다.

예제

다음 예제에서는 UTC 기준 2021년 9월 22일 오후 12:00:00에 숫자 Unix 타임스탬프(1632416400)를 해당 타임스탬프 데이터 형식으로 변환합니다.

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

TO_TIMESTAMP 함수

TO_TIMESTAMP는 TIMESTAMP 문자열을 TIMESTAMPTZ로 변환합니다.

구문

```
to_timestamp (timestamp)
```

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

인수

timestamp

타임스탬프 문자열 또는 타임스탬프 문자열로 캐스팅할 수 있는 데이터 유형입니다.

format

Spark의 날짜/시간 패턴과 일치하는 문자열 리터럴입니다. 유효한 날짜/시간 패턴은 서식 지정 및 구문 분석을 위한 날짜/시간 패턴을 참조하세요.

반환 타입

TIMESTAMP

예제

다음 예제에서는 TO_TIMESTAMP 함수를 사용하여 TIMESTAMP 문자열을 TIMESTAMP로 변환하는 방법을 보여줍니다.

```
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
```

날짜의 TO_TIMESTAMP 부분을 전달할 수 있습니다. 나머지 날짜 부분은 기본값으로 설정됩니다. 출력에 시간이 포함됩니다.

```
SELECT TO_TIMESTAMP('2017','YYYY');

to_timestamp
---------------------------
2017-01-01 00:00:00+00
```

다음 SQL 문은 문자열 '2011-12-18 24:38:15'를 TIMESTAMP로 변환합니다. 결과는 시간 수가 24시간을 초과하기 때문에 다음 날에 해당하는 TIMESTAMP입니다.

```
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 함수

YEAR 함수는 날짜 또는 타임스탬프를 입력으로 받아 연도 구성 요소(4자리 숫자)를 반환하는 날짜 추출 함수입니다.

구문

```
year(date)
```

인수

date

DATE 또는 TIMESTAMP 표현식입니다.

반환

YEAR 함수는 INTEGER를 반환합니다.

예제

다음 예시에서는 입력 날짜에서 연도 구성 요소(2016)를 추출합니다`'2016-07-30'`.

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

다음 예제에서는 `squirrels` 테이블의 `birthday` 열에서 연도 구성 요소를 추출하고 SELECT 문의 출력으로 결과를 반환합니다. 이 쿼리의 출력은 `squirrels` 테이블의 각 행에 대해 하나씩 연도 값 목록이 되며, 각 스쿼럴의 생일 연도를 나타냅니다.

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

날짜 또는 타임스탬프 함수의 날짜 부분

다음 표는 아래 함수의 인수로 허용되는 날짜 부분과 시간 부분의 이름 및 약어를 구분한 것입니다.

- DATE_ADD
- DATE_DIFF
- DATE_PART
- EXTRACT

날짜 부분 또는 시간 부분	약어
millennium, millennia	mil, mils
century, centuries	c, cent, cents
decade, decades	dec, decs
Epoch	epoch(EXTRACT에서 지원됨)
year, years	y, yr, yrs
quarter, quarters	qtr, qtrs
month, months	mon, mons
week, weeks	w
요일	dayofweek, dow, dw, weekday(DATE_PART 및 EXTRACT 함수에서 지원됨) 일요일부터 시작하여 0~6의 정수를 반환합니다. **Note** DOW 날짜 부분은 날짜/시간 형식 문자열에 사용되는 요일(D) 날짜 부분과 동작이 다릅니다. D는 정수 1~7을 기반으로합니다. 여기서 일요일은 1입니다. 자세한 내용은 날짜/시간 형식 문자열 단원을 참조하십시오.

날짜 부분 또는 시간 부분	약어
day_of_year	dayofyear, doy, dy, yearday(EXTRACT에서 지원됨)
day, days	d
hour, hours	h, hr, hrs
minute, minutes	m, min, mins
second, seconds	s, sec, secs
millisecond, milliseconds	ms, msec, msecs, msecond, mseconds, millisec, millisecs, millisecon
microsecond, microseconds	microsec, microsecs, microsecond, usecond, useconds, us, usec, usecs
timezone, timezone_hour, timezone_minute	시간대를 포함한 타임스탬프(TIMESTAMPTZ)인 경우에만 EXTRACT에서 지원됩니다.

초, 밀리초 및 마이크로초에서 결과의 차이

다른 날짜 함수에서 초, 밀리초 또는 마이크로초를 날짜 부분으로 지정하더라도 쿼리 결과에서는 최소한의 차이만 발생합니다.

- EXTRACT 함수는 지정한 날짜 부분에 한해 정수를 반환하고 더 높거나 낮은 단위의 날짜 부분은 무시합니다. 예를 들어 지정한 날짜 부분이 초라면 밀리초와 마이크로초는 결과에 포함되지 않습니다. 지정한 날짜 부분이 밀리초라면 초와 마이크로초가 포함되지 않습니다. 지정한 날짜 부분이 마이크로초라면 초와 밀리초가 포함되지 않습니다.
- DATE_PART 함수는 지정하는 날짜 부분에 상관없이 타임스탬프에서 전체 초 부분을 반환하며, 이때 반환되는 값은 필요에 따라 소수 값이 될 수도 있고, 정수가 될 수도 있습니다.

CENTURY, EPOCH, DECADE 및 MIL 참고 사항

CENTURY 또는 CENTURIES

AWS Clean Rooms 는 CENTURY를 연도 ###1로 시작하고 연도 로 끝나도록 해석합니다. ###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 AWS Clean Rooms 구현은 클러스터가 상주하는 시간대와 관계없이 1970-01-01 00:00:00.000000을 기준으로 합니다. 따라서 클러스터가 상주하는 시간대에 따라 시간차를 기준으로 결과를 오프셋 처리해야 할 수도 있습니다.

DECADE 또는 DECADES

AWS Clean Rooms 는 공통 달력을 기반으로 DECADE 또는 DECADES DATEPART를 해석합니다. 예를 들어 일반 역법은 0001년부터 시작하기 때문에 첫 10년(decade 1)은 0001-01-01부터 0009-12-31까지이며, 두 번째 10년(decade 2)은 0010-01-01부터 0019-12-31까지입니다. 이러한 식으로 decade 201은 2000-01-01부터 2009-12-31까지입니다.

```
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 또는 MILS

AWS Clean Rooms 는 MIL을 #001 연도의 첫 번째 날로 시작하고 연도의 마지막 날로 끝나도록 해석합니다. #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)
```

암호화 및 복호화 함수

암호화 및 복호화 함수는 SQL 개발자가 읽기 가능하고 일반 텍스트 형식과 읽기 불가능하며 암호 텍스트 형식 간에 변환하여 민감한 데이터를 무단 액세스 또는 오용으로부터 보호하는 데 도움이 됩니다.

AWS Clean Rooms Spark SQL은 다음과 같은 암호화 및 복호화 함수를 지원합니다.

주제

- AES_ENCRYPT 함수
- AES_DECRYPT 함수

AES_ENCRYPT 함수

AES_ENCRYPT 함수는 고급 암호화 표준(AES) 알고리즘을 사용하여 데이터를 암호화하는 데 사용됩니다.

구문

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

인수

expr

암호화할 이진 값입니다.

키

데이터를 암호화하는 데 사용할 암호입니다.

16, 24 및 32비트의 키 길이가 지원됩니다.

mode

메시지를 암호화하는 데 사용할 블록 암호 모드를 지정합니다.

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

패딩

길이가 블록 크기의 배수가 아닌 메시지를 패딩하는 방법을 지정합니다.

유효한 값: PKCS, NONE, DEFAULT.

DEFAULT 패딩은 ECB용 PKCS(퍼블릭 키 암호화 표준), GCM용 NONE 및 CBC용 PKCS를 의미합니다.

(모드, 패딩)의 지원되는 조합은 ('ECB', 'PKCS'), ('GCM', 'NONE') 및 ('CBC', 'PKCS')입니다.

IV

선택적 초기화 벡터(IV). CBC 및 GCM 모드에서만 지원됩니다.

유효한 값: GCM의 경우 12바이트, CBC의 경우 16바이트.

aad

선택적 추가 인증 데이터(AAD). GCM 모드에서만 지원됩니다. 이는 자유 형식 입력일 수 있으며 암호화 및 복호화 모두에 제공되어야 합니다.

반환 타입

AES_ENCRYPT 함수는 지정된 패딩과 함께 지정된 모드에서 AES를 사용하여 암호화된 expr 값을 반환합니다.

예시

다음 예제에서는 Spark SQL AES_ENCRYPT 함수를 사용하여 지정된 암호화 키를 사용하여 데이터 문자열(이 경우 "Spark")을 안전하게 암호화하는 방법을 보여줍니다. 그러면 결과 사이퍼텍스트가 Base64-encoded되어 저장하거나 전송하기가 더 쉬워집니다.

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

다음 예제에서는 Spark SQL AES_ENCRYPT 함수를 사용하여 지정된 암호화 키를 사용하여 데이터 문자열(이 경우 "Spark")을 안전하게 암호화하는 방법을 보여줍니다. 그런 다음 결과 사이퍼텍스트는 16진수 형식으로 표시되며, 이는 데이터 스토리지, 전송 또는 디버깅과 같은 작업에 유용할 수 있습니다.

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

다음 예제에서는 Spark SQL AES_ENCRYPT 함수를 사용하여 지정된 암호화 키, 암호화 모드 및 패딩 모드를 사용하여 데이터 문자열(이 경우 "Spark SQL")을 안전하게 암호화하는 방법을 보여줍니다. 그러면 결과 사이퍼텍스트가 Base64-encoded되어 저장하거나 전송하기가 더 쉬워집니다.

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

AES_DECRYPT 함수

AES_DECRYPT 함수는 고급 암호화 표준(AES) 알고리즘을 사용하여 데이터를 복호화하는 데 사용됩니다.

구문

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

인수

expr

복호화할 이진 값입니다.

키

데이터를 복호화하는 데 사용할 암호입니다.

암호는 원래 암호화된 값을 생성하는 데 사용된 키와 일치해야 하며 16, 24 또는 32바이트 길이여야 합니다.

mode

메시지를 복호화하는 데 사용할 블록 암호 모드를 지정합니다.

유효한 모드: ECB, GCM, CBC.

패딩

길이가 블록 크기의 배수가 아닌 메시지를 패딩하는 방법을 지정합니다.

유효한 값: PKCS, NONE, DEFAULT.

DEFAULT 패딩은 ECB용 PKCS, GCM용 NONE 및 CBC용 PKCS를 의미합니다.

aad

선택적 추가 인증 데이터(AAD). GCM 모드에서만 지원됩니다. 이는 자유 형식 입력일 수 있으며 암호화 및 복호화 모두에 제공되어야 합니다.

반환 타입

패딩이 있는 모드에서 AES를 사용하여 expr의 복호화된 값을 반환합니다.

예시

다음 예제에서는 Spark SQL AES_ENCRYPT 함수를 사용하여 지정된 암호화 키를 사용하여 데이터 문자열(이 경우 "Spark")을 안전하게 암호화하는 방법을 보여줍니다. 그러면 결과 사이퍼텍스트가 Base64-encoded되어 저장하거나 전송하기가 더 쉬워집니다.

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

다음 예제에서는 Spark SQL AES_DECRYPT 함수를 사용하여 이전에 암호화되고 Base64-encoded된 데이터를 복호화하는 방법을 보여줍니다. 복호화 프로세스를 수행하려면 원래 일반 텍스트 데이터를 성공적으로 복구하려면 올바른 암호화 키와 파라미터(암호화 모드 및 패딩 모드)가 필요합니다.

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

```
Spark SQL
```

해시 함수

해시 함수는 숫자 입력 값을 다른 값으로 변환하는 수학 함수입니다.

AWS Clean Rooms Spark SQL은 다음 해시 함수를 지원합니다.

주제

- MD5 함수
- SHA 함수
- SHA1 함수
- SHA2 함수
- xxHASH64 함수

MD5 함수

MD5 암호화 해시 함수를 사용하여 가변 길이 문자열을 128비트 체크섬의 16진수 값을 텍스트로 표현한 32자 문자열로 변환합니다.

구문

```
MD5(string)
```

인수

string

가변 길이 문자열입니다.

반환 타입

MD5 함수는 128비트 체크섬의 16진수 값을 텍스트로 표현한 32자 문자열을 반환합니다.

예시

다음은 문자열 'AWS Clean Rooms'를 128비트 값으로 표현한 예입니다.

```
select md5('AWS Clean Rooms');
```

```
md5
----------------------------------
f7415e33f972c03abd4f3fed36748f7a
(1 row)
```

SHA 함수

SHA1 함수의 동의어입니다.

SHA1 함수을(를) 참조하세요.

SHA1 함수

SHA1 함수는 SHA1 암호화 해시 함수를 사용하여 가변 길이 문자열을 160비트 체크섬의 16진수 값을 텍스트로 표현한 40자 문자열로 변환합니다.

구문

SHA1은 SHA 함수의 동의어입니다.

```
SHA1(string)
```

인수

string

가변 길이 문자열입니다.

반환 타입

SHA1 함수는 160비트 체크섬의 16진수 값을 텍스트로 표현한 40자 문자열을 반환합니다.

예제

다음은 단어 'AWS Clean Rooms'를 160비트 값으로 반환하는 예입니다.

```
select sha1('AWS Clean Rooms');
```

SHA2 함수

SHA2 함수는 SHA2 암호화 해시 함수를 사용하여 가변 길이 문자열을 문자열로 변환합니다. 문자열은 지정된 비트 수가 있는 체크섬의 16진수 값을 텍스트로 표현한 것입니다.

구문

```
SHA2(string, bits)
```

인수

string

가변 길이 문자열입니다.

integer

해시 함수의 비트 수입니다. 유효한 값은 0(256과 동일), 224, 256, 384 및 512입니다.

반환 타입

SHA2 함수는 체크섬의 16진수 값을 텍스트로 표현한 문자열을 반환하거나 비트 수가 유효하지 않은 경우 빈 문자열을 반환합니다.

예제

다음 예제에서는 단어 'AWS Clean Rooms를 256비트 값으로 반환합니다.

```
select sha2('AWS Clean Rooms', 256);
```

xxHASH64 함수

xxhash64 함수는 인수의 64비트 해시 값을 반환합니다.

xxhash64() 함수는 빠르고 효율적으로 설계된 비암호화 해시 함수입니다. 데이터 처리 및 스토리지 애플리케이션에서 사용되는 경우가 많으며, 데이터 조각의 고유 식별자가 필요하지만 데이터의 정확한 내용을 비밀로 유지할 필요는 없습니다.

SQL 쿼리의 컨텍스트에서 xxhash64() 함수는 다음과 같은 다양한 용도로 사용할 수 있습니다.

- 테이블의 행에 대한 고유 식별자 생성
- 해시 값을 기반으로 데이터 분할
- 사용자 지정 인덱싱 또는 데이터 배포 전략 구현

특정 사용 사례는 애플리케이션의 요구 사항과 처리 중인 데이터에 따라 달라집니다.

구문

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

인수

expr1

모든 유형의 표현식입니다.

expr2

모든 유형의 표현식입니다.

반환

인수(BIGINT)의 64비트 해시 값을 반환합니다. 해시 시드는 42입니다.

예제

다음 예제에서는 제공된 입력을 기반으로 64비트 해시 값(5602566077635097486)을 생성합니다. 첫 번째 인수는 문자열 값입니다.이 경우 "Spark"라는 단어가 사용됩니다. 두 번째 인수는 단일 정수 값 123을 포함하는 배열입니다. 세 번째 인수는 해시 함수의 시드를 나타내는 정수 값입니다.

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

하이퍼로그 함수

SQL의 HyperLogLog(HLL) 함수는 실제 고유 요소 집합이 저장되지 않은 경우에도 대규모 데이터 세트의 고유 요소(카디널리티) 수를 효율적으로 추정하는 방법을 제공합니다.

HLL 함수 사용의 주요 이점은 다음과 같습니다.

- 메모리 효율성: HLL 스케치는 전체 고유 요소 세트를 저장하는 것보다 훨씬 적은 메모리가 필요하므로 대규모 데이터 세트에 적합합니다.
- 분산 컴퓨팅: HLL 스케치를 여러 데이터 소스 또는 처리 노드에서 결합할 수 있으므로 효율적인 분산 고유 수 추정이 가능합니다.
- 대략적인 결과: HLL은 정확도와 메모리 사용량 간의 조정 가능한 균형을 통해 대략적인 고유 수 추정을 제공합니다(정밀도 파라미터를 통해).

이러한 함수는 분석, 데이터 웨어하우징 및 실시간 스트림 처리 애플리케이션과 같이 고유한 항목의 수를 추정해야 하는 시나리오에서 특히 유용합니다.

AWS Clean Rooms 는 다음 HLL 함수를 지원합니다.

주제

- HLL_SKETCH_AGG 함수
- HLL_SKETCH_ESTIMATE 함수
- HLL_UNION 함수
- HLL_UNION_AGG 함수

HLL_SKETCH_AGG 함수

HLL_SKETCH_AGG 집계 함수는 지정된 열의 값에서 HLL 스케치를 생성합니다. 입력 표현식 값을 캡슐화하는 HLLSKETCH 데이터 형식을 반환합니다.

HLL_SKETCH_AGG 집계 함수는 모든 데이터 유형에서 작동하며 NULL 값을 무시합니다.

테이블에 행이 없거나 모든 행이 NULL이면 결과 스케치에 인덱스-값 페어(예: `{"version":1,"logm":15,"sparse":{"indices":[],"values":[]}}`)가 없습니다.

구문

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

인수

aggregate_expression

고유한 계산이 발생할 INT, BIGINT, STRING 또는 BINARY 유형의 표현식입니다. 모든 NULL 값은 무시됩니다.

lgConfigK

기본값이 12인 4~21 사이의 선택적 INT 상수입니다. K의 log-base-2입니다. 여기서 K는 스케치의 버킷 또는 슬롯 수입니다.

반환 타입

HLL_SKETCH_AGG 함수는 집계 그룹의 모든 입력 값을 소비하고 집계하기 때문에 계산된 HyperLogLog 스케치가 포함된 NULL이 아닌 BINARY 버퍼를 반환합니다.

예시

다음 예제에서는 HyperLogLog(HLL) 알고리즘을 사용하여 `col` 열의 고유한 값 수를 추정합니다. `hll_sketch_agg(col, 12)` 함수는 열의 값을 집계하여 정밀도 12를 사용하여 HLL 스케치를 생성합니다. 그런 다음 `hll_sketch_estimate()` 함수를 사용하여 생성된 HLL 스케치를 기반으로 고유한 값 수를 추정합니다. 쿼리의 최종 결과는 3이며, 이는 `col` 열의 예상 고유 값 수를 나타냅니다. 이 경우 고유 값은 1, 2, 3입니다.

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

다음 예제에서는 HLL 알고리즘을 사용하여 `col` 열의 고유한 값 수를 추정하지만 HLL 스케치에 대한 정밀도 값은 지정하지 않습니다. 이 경우 기본 정밀도인 14를 사용합니다. 이 `hll_sketch_agg(col)` 함수는 `col` 열의 값을 가져와서 HyperLogLog(HLL) 스케치를 생성합니다.이 스케치는 고유한 요소 수를 추정하는 데 사용할 수 있는 컴팩트한 데이터 구조입니다. `hll_sketch_estimate(hll_sketch_agg(col))` 함수는 이전 단계에서 생성된 HLL 스케치를 가져와 `col` 열의 고유한 값 수 추정치를 계산합니다. 쿼리의 최종 결과는 3이며, 이는 `col` 열의 예상 고유 값 수를 나타냅니다. 이 경우 고유 값은 1, 2, 3입니다.

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

HLL_SKETCH_ESTIMATE 함수

HLL_SKETCH_ESTIMATE 함수는 HLL 스케치를 가져와 스케치로 표시되는 고유 요소의 수를 추정합니다. HyperLogLog(HLL) 알고리즘을 사용하여 주어진 열의 고유 값 수에 대한 확률론적 근사치를 계산하고, HLL_SKETCH_AGG 함수에서 이전에 생성된 스케치 버퍼라고 하는 바이너리 표현을 사용하고, 결과를 큰 정수로 반환합니다.

HLL 스케치 알고리즘은 전체 고유 값 세트를 저장할 필요 없이 대규모 데이터 세트의 경우에도 고유한 요소의 수를 추정하는 효율적인 방법을 제공합니다.

또한 hll_union 및 hll_union_agg 함수는 이러한 버퍼를 입력으로 사용하고 병합하여 스케치를 결합할 수 있습니다.

구문

```
HLL_SKETCH_ESTIMATE (hllsketch_expression)
```

인수

hllsketch_expression

HLL_SKETCH_AGG에서 생성된 스케치를 포함하는 BINARY 표현식

반환 타입

HLL_SKETCH_ESTIMATE 함수는 입력 스케치로 표시되는 대략적인 고유 개수인 BIGINT 값을 반환합니다.

예시

다음 예제에서는 HyperLogLog(HLL) 스케치 알고리즘을 사용하여 col 열에 있는 값의 카디널리티(고유 개수)를 추정합니다. hll_sketch_agg(col, 12) 함수는 col 열을 가져와 12비트의 정밀도를 사용하여 HLL 스케치를 생성합니다. HLL 스케치는 집합의 고유 요소 수를 효율적으로 추정할 수 있는 대략적인 데이터 구조입니다. hll_sketch_estimate() 함수는에서 생성한 HLL 스케치를 가져hll_sketch_agg와 스케치로 표시되는 값의 카디널리티(고유 개수)를 추정합니다. 는 5개의 행이 있는 테스트 데이터 세트를 FROM VALUES (1), (1), (2), (2), (3) tab(col); 생성합니다. 여기서 col 열에는 1, 1, 2, 2, 3 값이 포함됩니다. 이 쿼리의 결과는 col 열에 있는 값의 예상 고유 개수인 3입니다.

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

다음 예제와 이전 예제의 차이점은 hll_sketch_agg 함수 호출에서 정밀도 파라미터(12비트)가 지정되지 않는다는 것입니다. 이 경우 기본 정밀도인 14비트가 사용되며, 이는 12비트의 정밀도를 사용한 이전 예제에 비해 고유 개수에 대한 보다 정확한 추정치를 제공할 수 있습니다.

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

HLL_UNION 함수

HLL_UNION 함수는 두 개의 HLL 스케치를 통합된 단일 스케치로 결합합니다. HyperLogLog(HLL) 알고리즘을 사용하여 두 개의 스케치를 단일 스케치로 결합합니다. 쿼리는 결과 버퍼를 사용하여 `hll_sketch_estimate` 함수를 사용하여 대략적인 고유 수를 긴 정수로 계산할 수 있습니다.

구문

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

인수

exprN

HLL_SKETCH_AGG에서 생성된 스케치를 포함하는 BINARY 표현식입니다.

allowDifferentLgConfigK

두 스케치를 서로 다른 lgConfigK 값과 병합할 수 있는지 여부를 제어하는 선택적 BOOLEAN 표현식입니다. 기본값은 `false`입니다.

반환 타입

HLL_UNION 함수는 입력 표현식을 결합한 결과로 계산된 HyperLogLog 스케치가 포함된 BINARY 버퍼를 반환합니다. `allowDifferentLgConfigK` 파라미터가 인 경우 `true`결과 스케치는 제공된 두 값 중 작은 `lgConfigK` 값을 사용합니다.

예시

다음 예제에서는 HyperLogLog(HLL) 스케치 알고리즘을 사용하여 데이터 세트의 `col1` 및 `col2`라는 두 열에서 값의 고유 수를 추정합니다.

`hll_sketch_agg(col1)` 함수는 `col1` 열의 고유 값에 대한 HLL 스케치를 생성합니다.

`hll_sketch_agg(col2)` 함수는 col2 열의 고유 값에 대한 HLL 스케치를 생성합니다.

`hll_union(...)` 함수는 1단계와 2단계에서 생성된 두 HLL 스케치를 통합된 단일 HLL 스케치로 결합합니다.

이 `hll_sketch_estimate(...)` 함수는 결합된 HLL 스케치를 가져와`col1`서 및 모두에서 고유한 값 수를 추정합니다`col2`.

FROM VALUES 절은 5개 행으로 구성된 테스트 데이터 세트를 생성합니다. 여기서 col1는 값 1, 1, 2, 2, 3을 포함하고는 값 4, 4, 5, 5, 6을 col2 포함합니다.

이 쿼리의 결과는 col1 및 모두에서 값의 예상 고유 개수이며col2, 이는 6입니다. HLL 스케치 알고리즘은 전체 고유 값 세트를 저장할 필요 없이 대규모 데이터 세트의 경우에도 고유한 요소의 수를 추정하는 효율적인 방법을 제공합니다. 이 예제에서는 hll_union 함수를 사용하여 두 열의 HLL 스케치를 결합합니다. 이를 통해 각 열에 대해 개별적으로가 아니라 전체 데이터 세트에서 고유한 수를 추정할 수 있습니다.

```
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
```

다음 예제와 이전 예제의 차이점은 hll_sketch_agg 함수 호출에서 정밀도 파라미터(12비트)가 지정되지 않는다는 것입니다. 이 경우 기본 정밀도인 14비트가 사용되며, 이는 12비트의 정밀도를 사용한 이전 예제에 비해 고유 개수에 대한 보다 정확한 추정치를 제공할 수 있습니다.

```
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 함수

HLL_UNION_AGG 함수는 여러 HLL 스케치를 통합된 단일 스케치로 결합합니다. HyperLogLog(HLL) 알고리즘을 사용하여 스케치 그룹을 단일 그룹으로 결합합니다. 쿼리는 결과 버퍼를 사용하여 hll_sketch_estimate 함수를 사용하여 대략적인 고유 수를 계산할 수 있습니다.

구문

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

인수

expr

HLL_SKETCH_AGG에서 생성된 스케치를 포함하는 BINARY 표현식입니다.

allowDifferentLgConfigK

두 스케치를 서로 다른 lgConfigK 값과 병합할 수 있는지 여부를 제어하는 선택적 BOOLEAN 표현식입니다. 기본값은 `false`입니다.

반환 타입

HLL_UNION_AGG 함수는 동일한 그룹의 입력 표현식을 결합한 결과로 계산된 HyperLogLog 스케치가 포함된 BINARY 버퍼를 반환합니다. `allowDifferentLgConfigK` 파라미터가 인 경우 `true`결과 스케치는 제공된 두 값 중 작은 `lgConfigK` 값을 사용합니다.

예시

다음 예제에서는 HyperLogLog(HLL) 스케치 알고리즘을 사용하여 여러 HLL 스케치의 고유한 값 수를 추정합니다.

첫 번째 예제에서는 데이터 세트의 고유한 값 수를 추정합니다.

```
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
```

내부 쿼리는 두 개의 HLL 스케치를 생성합니다.

- 첫 번째 SELECT 문은 단일 값 1에서 스케치를 생성합니다.
- 두 번째 SELECT 문은 1의 또 다른 단일 값에서 스케치를 생성하지만 정밀도는 20입니다.

외부 쿼리는 HLL_UNION_AGG 함수를 사용하여 두 스케치를 단일 스케치로 결합합니다. 그런 다음 HLL_SKETCH_ESTIMATE 함수를이 결합된 스케치에 적용하여 고유한 값 수를 추정합니다.

이 쿼리의 결과는 col 열에 있는 값의 예상 고유 개수인 입니다1. 즉, 두 입력 값 1은 값이 같더라도 고유한 것으로 간주됩니다.

두 번째 예제에는 HLL_UNION_AGG 함수에 대한 다른 정밀도 파라미터가 포함되어 있습니다. 이 경우 두 HLL 스케치는 모두 14비트의 정밀도로 생성되므로 true 파라미터hll_union_agg와를 사용하여 성공적으로 결합할 수 있습니다.

```
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
```

쿼리의 최종 결과는 예상 고유 수이며,이 경우 도 입니다1. 즉, 두 입력 값 1은 값이 같더라도 고유한 것으로 간주됩니다.

JSON 함수

비교적 작은 용량의 키-값 페어 집합을 저장해야 할 때는 JSON 형식으로 저장하면 공간을 절약할 수 있습니다. JSON 문자열은 단일 열에 저장할 수 있기 때문에 테이블 형식의 데이터 저장보다는 JSON을 사용하는 것이 더욱 효율적입니다.

Example

예를 들어, 가능한 모든 속성을 완벽하게 표현하기 위해 많은 열이 필요한 희소(sparce) 테이블이 있다고 가정해 보겠습니다. 그러나 주어진 행 또는 열에 대해 대부분의 열 값은 NULL입니다. 이때 JSON을 사용하면 행 데이터를 키:값 페어로 단일 JSON 문자열에 저장하여 희박하게 채워진 테이블 열을 제거할 수 있습니다.

또한 JSON 문자열은 수정이 쉽기 때문에 열을 테이블에 추가하지 않고도 키:값 페어를 더 저장할 수 있습니다.

JSON은 적게 사용하는 것이 바람직합니다. 특히 대용량의 데이터 집합을 저장할 때는 JSON을 사용하지 않는 것이 좋습니다. 이때는 이질적인 데이터가 단일 열에 저장되면서 JSON가 AWS Clean Rooms 열 저장 아키텍처를 사용하지 못하기 때문입니다.

JSON은 UTF-8로 인코딩된 텍스트 문자열을 사용합니다. 따라서 JSON 문자열은 CHAR 또는 VARCHAR 데이터 형식으로 저장될 수 있습니다. 문자열에 멀티바이트 문자가 포함된 경우에는 VARCHAR를 사용하십시오.

JSON 문자열은 다음 규칙에 따라 올바른 형식의 JSON이 되어야 합니다.

- 루트 레벨 JSON은 JSON 객체 또는 JSON 배열일 수 있습니다. JSON 객체는 쉼표로 구분된 키:값 페어의 집합으로서 순서 지정 없이 중괄호로 묶입니다.

  예: `{"one":1, "two":2}`
- JSON 배열은 쉼표로 구분된 값의 집합으로서 순서 지정과 함께 대괄호로 묶입니다.

  예제는 다음과 같습니다: `["first", {"one":1}, "second", 3, null]`
- JSON 배열은 0부터 시작되는 인덱스를 사용하기 때문에 배열의 첫 요소가 0 위치에 자리합니다. JSON 키:값 페어에서 키는 큰따옴표로 묶이는 문자열입니다.
- JSON 값은 다음 중 하나일 수 있습니다.
  - JSON 객체
  - JSON 배열
  - 큰따옴표로 묶이는 문자열
  - 숫자(정수 및 부동 소수점 수)
  - 불
  - Null
- 빈 객체와 빈 배열도 유효한 JSON 값입니다.
- JSON 필드는 대/소문자를 구분합니다.
- JSON 구조 요소 사이의 공백(`{ }, [ ]` 등)은 무시됩니다.

주제

- GET_JSON_OBJECT 함수
- TO_JSON 함수

GET_JSON_OBJECT 함수

GET_JSON_OBJECT 함수는에서 json 객체를 추출합니다`path`.

구문

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

인수

json_txt

잘 구성된 JSON을 포함하는 STRING 표현식입니다.

경로

잘 구성된 JSON 경로 표현식이 있는 STRING 리터럴입니다.

반환

STRING을 반환합니다.

객체를 찾을 수 없는 경우 NULL이 반환됩니다.

예제

다음 예시에서는 JSON 객체에서 값을 추출합니다. 첫 번째 인수는 단일 키-값 페어가 있는 간단한 객체를 나타내는 JSON 문자열입니다. 두 번째 인수는 JSON 경로 표현식입니다. $ 기호는 JSON 객체의 루트를 나타내며 .a 부분은 "a" 키와 연결된 값을 추출하도록 지정합니다. 함수의 출력은 입력 JSON 객체의 b"a" 키와 연결된 값인 "입니다.

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

TO_JSON 함수

TO_JSON 함수는 입력 표현식을 JSON 문자열 표현식으로 변환합니다. 함수는 다양한 데이터 유형(예: 숫자, 문자열 및 부울)을 해당 JSON 표현으로 변환하는 작업을 처리합니다.

TO_JSON 함수는 구조화된 데이터(예: 데이터베이스 행 또는 JSON 객체)를 JSON과 같은 보다 이동 가능한 자체 설명 형식으로 변환해야 할 때 유용합니다. 이는 JSON 형식 데이터를 기대하는 다른 시스템 또는 서비스와 상호 작용해야 할 때 특히 유용할 수 있습니다.

구문

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

인수

expr

JSON 문자열로 변환하려는 입력 표현식입니다. 값, 열 또는 기타 유효한 SQL 표현식일 수 있습니다.

options

JSON 변환 프로세스를 사용자 지정하는 데 사용할 수 있는 선택적 구성 옵션 세트입니다. 이러한 옵션에는 null 값 처리, 숫자 값 표현, 특수 문자 처리 등이 포함될 수 있습니다.

반환

지정된 구조체 값이 있는 JSON 문자열을 반환합니다.

예시

다음 예제에서는 명명된 구조체(구조화된 데이터 유형)를 JSON 문자열로 변환합니다. 첫 번째 인수 (named_struct('a', 1, 'b', 2))는 to_json() 함수에 전달되는 입력 표현식입니다. 값이 1인 "a"와 값이 2인 "b"라는 두 개의 필드가 있는 명명된 구조체를 생성합니다. to_json() 함수는 명명된 구조체를 인수로 받아 JSON 문자열 표현으로 변환합니다. 출력은 이름이 지정된 구조체{"a":1,"b":2}를 나타내는 유효한 JSON 문자열인 입니다.

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

다음 예제에서는 타임스탬프 값이 포함된 명명된 구조체를 사용자 지정 타임스탬프 형식의 JSON 문자열로 변환합니다. 첫 번째 인수(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')))는 타임스탬프 값이 포함된 단일 필드 'time'이 있는 명명된 구조체를 생성합니다. 두 번째 인수(map('timestampFormat', 'dd/MM/yyyy'))는 단일 키-값 페어로 맵(키-값 사전)을 생성합니다. 여기서 키는 'timestampFormat'이고 값은 'dd/MM/yyyy'입니다. 이 맵은 타임스탬프 값을 JSON으로 변환할 때 타임스탬프 값에 원하는 형식을 지정하는 데 사용됩니다. to_json() 함수는 명명된 구조체를 JSON 문자열로 변환합니다. 두 번째 인수인 맵은 타임스탬프 형식을 'dd/MM/yyyy'로 사용자 지정하는 데 사용됩니다. 출력은 이며{"time":"26/08/2015"}, 원하는 'dd/MM/yyyy' 형식의 타임스탬프 값을 포함하는 단일 필드 'time'이 있는 JSON 문자열입니다.

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

```
{"time":"26/08/2015"}
```

수학 함수

이 섹션에서는 AWS Clean Rooms Spark SQL에서 지원되는 수학 연산자 및 함수에 대해 설명합니다.

주제

- 수학 연산자 기호
- ABS 함수
- ACOS 함수
- ASIN 함수
- ATAN 함수
- ATAN2 함수
- CBRT 함수
- CEILING(또는 CEIL) 함수
- COS 함수
- COT 함수
- DEGREES 함수
- DIV 함수
- EXP 함수
- FLOOR 함수
- LN 함수
- LOG 함수
- MOD 함수
- PI 함수
- POWER 함수
- RADIANS 함수
- RAND 함수
- RANDOM 함수
- ROUND 함수

- SIGN 함수
- SIN 함수
- SQRT 함수
- TRUNC 함수

수학 연산자 기호

다음 표는 지원되는 수학 연산자를 나열한 것입니다.

지원되는 연산자

연산자	설명	예시	결과
+	더하기	2 + 3	5
-	빼기	2 - 3	-1
*	곱하기	2 * 3	6
/	나누기	/	2
%	모듈로	5 % 4	1
^	거듭제곱	2.0 ^ 3.0	8

예시

임의의 거래에서 취급 수수료 $2.00을 더하여 지불할 수수료를 계산합니다.

```
select commission, (commission + 2.00) as comm
from sales where salesid=10000;

commission | comm
-----------+-------
28.05      | 30.05
(1 row)
```

임의의 거래에서 판매 가격의 20%를 계산합니다.

```
select pricepaid, (pricepaid * .20) as twentypct
from sales where salesid=10000;

pricepaid | twentypct
----------+-----------
187.00    |    37.400
(1 row)
```

연속 성장 패턴에 따라 티켓 판매를 예측합니다. 이번 예에서는 하위 쿼리가 2008년 판매된 티켓 수량을 반환합니다. 그런 다음 그 결과를 10년 연속 성장률 5%와 거듭제곱합니다.

```
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)
```

날짜 ID가 2000보다 크거나 같은 판매의 가격 총액과 수수료 총액을 구합니다. 그런 다음 가격 총액에서 수수료 총액을 뺍니다.

```
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 함수

ABS는 절대 숫자 값을 계산합니다. 여기에서 숫자란 리터럴이거나, 혹은 숫자로 평가되는 표현식이 될 수 있습니다.

구문

```
ABS (number)
```

인수

number

숫자, 또는 숫자로 평가되는 표현식입니다. SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4, 또는 FLOAT8 형식일 수 있습니다.

반환 타입

ABS는 인수와 동일한 데이터 형식을 반환합니다.

예시

-38의 절대값을 계산합니다.

```
select abs (-38);
abs
-------
38
(1 row)
```

(14-76)의 절대값을 계산합니다.

```
select abs (14-76);
abs
-------
62
(1 row)
```

ACOS 함수

ACOS는 숫자의 아크 코사인을 반환하는 삼각 함수입니다. 반환 값은 라디안 단위이며 0과 PI 사이입니다.

구문

```
ACOS(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

-1의 아크 코사인을 반환하려면 다음 예제를 사용합니다.

```
SELECT ACOS(-1);

+-------------------+
|       acos        |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ASIN 함수

ASIN은 숫자의 아크 사인을 반환하는 삼각 함수입니다. 반환 값은 라디안 단위이며 PI/2과 -PI/2 사이입니다.

구문

```
ASIN(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

1의 아크 사인을 반환하려면 다음 예제를 사용합니다.

```
SELECT ASIN(1) AS halfpi;

+--------------------+
|       halfpi       |
+--------------------+
| 1.5707963267948966 |
+--------------------+
```

ATAN 함수

ATAN은 숫자의 아크 탄젠트를 반환하는 삼각 함수입니다. 반환 값은 라디안 단위이며 -PI과 PI 사이입니다.

구문

```
ATAN(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

1의 아크 탄젠트를 반환하여 4와 곱하려면 다음 예제를 사용합니다.

```
SELECT ATAN(1) * 4 AS pi;
```

```
+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

ATAN2 함수

ATAN2는 다른 숫자로 나눈 숫자의 아크 탄젠트를 반환하는 삼각 함수입니다. 반환 값은 라디안 단위이며 PI/2과 -PI/2 사이입니다.

구문

```
ATAN2(number1, number2)
```

인수

number1

DOUBLE PRECISION 수입니다.

number2

DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

2/2의 아크 탄젠트를 반환하여 4와 곱하려면 다음 예제를 사용합니다.

```
SELECT ATAN2(2,2) * 4 AS PI;

+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

CBRT 함수

CBRT 함수는 숫자의 세제곱근을 계산하는 수학 함수입니다.

구문

```
CBRT (number)
```

인수

CBRT는 배정밀도 숫자를 인수로 사용합니다.

반환 타입

CBRT는 배정밀도 숫자를 반환합니다.

예시

임의의 거래에서 지불되는 수수료의 세제곱근을 계산합니다.

```
select cbrt(commission) from sales where salesid=10000;

cbrt
------------------
3.03839539048843
(1 row)
```

CEILING(또는 CEIL) 함수

CEILING 또는 CEIL 함수는 숫자를 다음 정수(whole number)로 올림하는 데 사용됩니다. 반면 FLOOR 함수는 숫자를 다음 정수로 내림합니다.

구문

```
CEIL | CEILING(number)
```

인수

number

숫자 또는 숫자로 평가되는 표현식입니다. SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4 또는 FLOAT8 형식일 수 있습니다.

반환 타입

CEILING과 CEIL은 인수와 동일한 데이터 형식을 반환합니다.

예시

다음은 임의의 거래에서 지불되는 수수료의 상한을 계산하는 예입니다.

```
select ceiling(commission) from sales
where salesid=10000;

ceiling
---------
29
(1 row)
```

COS 함수

COS는 숫자의 코사인을 반환하는 삼각 함수입니다. 반환 값은 라디안 단위이며 -1과 1 사이(경계값 포함)입니다.

구문

```
COS(double_precision)
```

인수

number

입력 파라미터는 배정밀도 숫자입니다.

반환 타입

COS 함수는 배정밀도 숫자를 반환합니다.

예시

다음은 0의 코사인을 반환하는 예입니다.

```
select cos(0);
```

```
cos
-----
1
(1 row)
```

다음은 PI의 코사인을 반환하는 예입니다.

```
select cos(pi());
cos
-----
-1
(1 row)
```

COT 함수

COT는 숫자의 코탄젠트를 반환하는 삼각 함수입니다. 입력 파라미터는 0이 아닌 값이어야 합니다.

구문

```
COT(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

1의 코탄젠트를 반환하려면 다음 예제를 사용합니다.

```
SELECT COT(1);

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

```
| 0.6420926159343306 |
+--------------------+
```

DEGREES 함수

라디안 단위의 각도를 도 단위의 등가로 변환합니다.

구문

```
DEGREES(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

0.5 라디안에 상응하는 도수를 반환하려면 다음 예제를 사용합니다.

```
SELECT DEGREES(.5);

+-------------------+
|      degrees      |
+-------------------+
| 28.64788975654116 |
+-------------------+
```

PI 라디안을 도 단위로 변환하려면 다음 예제를 사용합니

```
SELECT DEGREES(pi());

+---------+
| degrees |
+---------+
|     180 |
```

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

DIV 함수

DIV 연산자는 디바이저로 나누기의 필수 부분을 반환합니다.

구문

```
dividend div divisor
```

인수

배당

숫자 또는 간격으로 평가되는 표현식입니다.

divisor

가 간격dividend인 경우 일치하는 간격 유형이고, 그렇지 않으면 숫자입니다.

반환 타입

BIGINT

예시

다음 예제에서는 스쿼럴 테이블에서 두 개의 열을 선택합니다. 각 스쿼럴의 고유 식별자가 포함된 id 열과 연령 calculated 열의 정수를 2로 나눈 열을 age div 2나타냅니다. age div 2 계산은 age 열에서 정수 분할을 수행하여 유효 기간을 가장 가까운 짝수로 내림합니다. 예를 들어 age 열에 3, 5, 7, 10과 같은 값이 포함된 경우 age div 2 열에는 각각 1, 2, 3, 5 값이 포함됩니다.

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

이 쿼리는 연령 범위에 따라 데이터를 그룹화하거나 분석해야 하고 연령 값을 가장 가까운 짝수 정수로 내림하여 간소화하려는 시나리오에서 유용할 수 있습니다. 결과 출력은 squirrels 테이블의 각 스쿼럴에 대해 id 및 수명을 2로 나눈 값을 제공합니다.

EXP 함수

EXP 함수는 숫자 표현식의 지수 함수, 또는 자연 알고리즘 기반인 거듭제곱된 e를 실행합니다. EXP 함수는 LN 함수의 역입니다.

구문

```
EXP (expression)
```

인수

expression

표현식의 데이터 형식은 INTEGER, DECIMAL 또는 DOUBLE PRECISION가 되어야 합니다.

반환 타입

EXP는 DOUBLE PRECISION 숫자를 반환합니다.

예시

EXP 함수를 사용하여 연속 성장 패턴에 따른 티켓 판매를 예측합니다. 이번 예에서는 하위 쿼리가 2008년 판매된 티켓 수량을 반환합니다. 그런 다음 그 결과를 EXP 함수 결과와 곱합니다. 이때 10년 연속 성장률은 7%로 지정합니다.

```
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 함수

FLOOR 함수는 숫자를 다음 정수(whole number)로 내림합니다.

구문

```
FLOOR (number)
```

인수

number

숫자 또는 숫자로 평가되는 표현식입니다. SMALLINT, INTEGER, BIGINT, DECIMAL, FLOAT4 또는 FLOAT8 형식일 수 있습니다.

반환 타입

FLOOR는 인수와 동일한 데이터 형식을 반환합니다.

예시

이 예에서는 FLOOR 함수를 사용하기 전후에 임의의 거래에서 지불되는 수수료의 값을 보여줍니다.

```
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 함수

LN 함수는 입력 파라미터의 자연 로그를 반환합니다.

구문

```
LN(expression)
```

인수

expression

함수가 실행되는 대상 열 또는 표현식입니다.

> **Note**
>
> 이 함수는 표현식이 AWS Clean Rooms 사용자 생성 테이블 또는 AWS Clean Rooms STL 또는 STV 시스템 테이블을 참조하는 경우 일부 데이터 유형에 대한 오류를 반환합니다.

다음과 같은 데이터 형식의 표현식은 사용자 생성 또는 시스템 테이블을 참조할 경우 오류를 나타냅니다.

- BOOLEAN
- CHAR
- 날짜
- DECIMAL 또는 NUMERIC
- TIMESTAMP
- VARCHAR

다음과 같은 데이터 형식의 표현식은 사용자 생성 테이블이나 STL 또는 STV 시스템 테이블에서 성공적으로 실행됩니다.

- BIGINT
- DOUBLE PRECISION
- INTEGER
- REAL
- SMALLINT

반환 타입

LN 함수는 표현식과 동일한 형식을 반환합니다.

예시

다음은 숫자 2.718281828의 자연 로그, 즉 밑이 e인 로그를 반환하는 예입니다.

```
select ln(2.718281828);
ln
--------------------
0.9999999998311267
(1 row)
```

반환되는 값은 거의 1에 일치합니다.

다음은 USERS 테이블에서 USERID 열의 값에 대한 자연 로그를 반환하는 예입니다.

```
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 함수

를 expr 사용하여의 로그를 반환합니다base.

구문

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

인수

expr

표현식의 데이터 형식은 정수, 소수 또는 부동 소수점이 되어야 합니다.

base

로그 계산의 기본값입니다. 배정밀도 데이터 형식의 양수(1 이하)여야 합니다.

반환 타입

LOG 함수는 배정밀도 숫자를 반환합니다.

예시

다음은 숫자 100의 밑이 10인 로그를 반환하는 예입니다.

```
select log(10, 100);
--------
2
(1 row)
```

MOD 함수

모듈로 연산이라고도 하는 두 숫자의 나머지를 반환합니다. 결과를 계산하려면 첫 번째 파라미터를 두 번째 파라미터로 나눕니다.

구문

```
MOD(number1, number2)
```

인수

number1

첫 번째 입력 파라미터는 INTEGER, SMALLINT, BIGINT 또는 DECIMAL 숫자입니다. 둘 중 한 파라미터가 DECIMAL 형식이라면 나머지 파라미터도 DECIMAL 형식이 되어야 합니다. 둘 중 한 파라미터가 INTEGER 형식이라면 나머지 파라미터는 INTEGER, SMALLINT 또는 BIGINT 형식이 될 수 있습니다. 두 파라미터 모두 SMALLINT 또는 BIGINT가 될 수 있지만 한 파라미터가 BIGINT라면 나머지 파라미터는 SMALLINT가 될 수 없습니다.

number2

두 번째 파라미터는 INTEGER, SMALLINT, BIGINT 또는 DECIMAL 숫자입니다. number2에도 number1과 동일한 데이터 형식 규칙이 적용됩니다.

반환 타입

유효한 반환 형식은 DECIMAL, INT, SMALLINT 및 BIGINT입니다. MOD 함수의 반환 형식은 두 입력 파라미터의 형식이 동일하다는 가정 하에 입력 파라미터와 동일한 숫자 형식입니다. 하지만 둘 중 한 파라미터가 INTEGER이라면 반환 형식도 INTEGER가 됩니다.

사용 노트

%를 모듈로 연산자로 사용할 수 있습니다.

예시

다음 예제에서는 숫자를 다른 숫자로 나눌 때 나머지를 반환합니다.

```
SELECT MOD(10, 4);

 mod
------
 2
```

다음 예제는 십진수 결과를 반환합니다.

```
SELECT MOD(10.5, 4);

 mod
------
 2.5
```

매개변수 값을 다음과 같이 변환할 수 있습니다.

```
SELECT MOD(CAST(16.4 as integer), 5);

 mod
------
 1
```

첫 번째 파라미터를 2로 나누어 짝수인지 확인합니다.

```
SELECT mod(5,2) = 0 as is_even;

 is_even
--------
 false
```

%를 모듈로 연산자로 사용할 수 있습니다.

```
SELECT 11 % 4 as remainder;
```

```
 remainder
-----------
 3
```

다음은 CATEGORY 테이블에서 홀수 번호 카테고리의 정보를 반환하는 예입니다.

```
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 함수

pi 함수는 PI 값을 소수점 14자리까지 반환합니다.

구문

```
PI()
```

반환 타입

DOUBLE PRECISION

예시

pi 값을 반환하려면 다음 예제를 사용합니다.

```
SELECT PI();
```

```
+-------------------+
|        pi         |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

POWER 함수

POWER 함수는 숫자 표현식을 두 번째 숫자 표현식의 거듭제곱으로 제곱하는 지수 함수입니다. 예를 들어 2의 세제곱은 POWER(2,3)으로 계산되어 8이라는 결과를 반환합니다.

구문

```
{POWER(expression1, expression2)
```

인수

expression1

제곱할 숫자 표현식입니다. 데이터 형식은 INTEGER, DECIMAL 또는 FLOAT여야 합니다.

expression2

expression1을 제곱할 거듭제곱입니다. 데이터 형식은 INTEGER, DECIMAL 또는 FLOAT여야 합니다.

반환 타입

DOUBLE PRECISION

예시

```
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 함수

RADIANS 함수는 도 단위의 각도를 라디안 단위의 등가로 변환합니다.

구문

```
RADIANS(number)
```

인수

number

입력 파라미터는 DOUBLE PRECISION 수입니다.

반환 타입

DOUBLE PRECISION

예시

라디안 환산 180도를 반환하려면 다음 예제를 사용합니다.

```
SELECT RADIANS(180);

+-------------------+
|      radians      |
+-------------------+
| 3.141592653589793 |
+-------------------+
```

RAND 함수

RAND 함수는 0에서 1 사이의 임의의 부동 소수점 숫자를 생성합니다. RAND 함수는 호출할 때마다 새 난수를 생성합니다.

구문

```
RAND()
```

반환 타입

RANDOM은 DOUBLE을 반환합니다.

예시

다음 예제에서는 squirrels 테이블의 각 행에 대해 0과 1 사이의 임의의 부동 소수점 숫자 열을 생성합니다. 결과 출력은 스쿼럴 테이블의 각 행에 대해 하나의 값이 있는 임의의 십진수 값 목록이 포함된 단일 열입니다.

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

이러한 유형의 쿼리는 예를 들어 무작위 이벤트를 시뮬레이션하거나 데이터 분석에 무작위성을 도입하기 위해 난수를 생성해야 할 때 유용합니다. squirrels 테이블의 컨텍스트에서 각 스쿼럴에 임의의 값을 할당하는 데 사용할 수 있으며, 이후 추가 처리 또는 분석에 사용할 수 있습니다.

RANDOM 함수

RANDOM 함수는 0.0(포함)과 1.0(제외) 사이에서 무작위로 값을 생성합니다.

구문

```
RANDOM()
```

반환 타입

RANDOM은 DOUBLE PRECISION 숫자를 반환합니다.

예시

1. 0과 99 사이에서 무작위로 값을 계산합니다. 무작위 숫자가 0~1이라면 다음 쿼리는 0과 100 사이에서 무작위 숫자를 생성합니다.

   ```
   select cast (random() * 100 as int);

   INTEGER
   ------
   24
   (1 row)
   ```

2. 10개 항목의 균일한 무작위 샘플을 검색합니다.

   ```
   select *
   from sales
   order by random()
   ```

```
limit 10;
```

이제 10개 항목의 무작위 샘플을 검색하지만, 가격에 비례하여 항목을 선택합니다. 예를 들어 다른 항목보다 가격이 두 배 높은 항목은 쿼리 결과에 나타날 가능성이 두 배 더 높습니다.

```
select *
from sales
order by log(1 - random()) / pricepaid
limit 10;
```

3. 다음은 RANDOM이 예측 가능한 순서로 숫자를 생성할 수 있도록 SET 명령을 사용하여 SEED 값을 설정하는 예입니다.

   먼저 SEED 값을 설정하지 않고 RANDOM 정수 3개를 반환합니다.

```
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)
```

   그런 다음 SEED 값을 .25로 설정한 후 RANDOM 숫자를 3개 더 반환합니다.

```
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)
```

마지막으로 SEED 값을 다시 .25로 설정한 후 RANDOM이 이전 세 번의 호출과 동일한 결과를 반환하는지 확인합니다.

```
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 함수

ROUND 함수는 숫자를 가장 가까운 정수 또는 소수로 반올림합니다.

ROUND 함수는 옵션으로 두 번째 인수를 추가할 수 있습니다. 이 두 번째 인수는 어느 방향이든 반올림할 소수 자릿수를 나타내는 정수입니다. 두 번째 인수를 제공하지 않으면 함수는 가장 가까운 정수로

반올림됩니다. 두 번째 인수 >n이 지정되면 함수는 전체 자릿수의 소수 자릿수가 n인 가장 가까운 숫자로 반올림됩니다.

구문

```
ROUND (number [ , integer ] )
```

인수

number

숫자 또는 숫자로 평가되는 표현식입니다. DECIMAL 또는 FLOAT8 유형일 수 있습니다.는 암시적 변환 규칙에 따라 다른 데이터 유형을 변환할 AWS Clean Rooms 수 있습니다.

integer(옵션)

어느 방향으로든 반올림을 위한 소수 자릿수를 나타내는 정수입니다.

반환 타입

ROUND는 입력 인수와 동일한 숫자 데이터 형식을 반환합니다.

예시

다음은 임의의 거래에서 지불되는 수수료를 가장 가까운 정수로 반올림하는 예입니다.

```
select commission, round(commission)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    28
(1 row)
```

다음은 임의의 거래에서 지불되는 수수료를 첫 번째 소수점 자리로 반올림하는 예입니다.

```
select commission, round(commission, 1)
from sales where salesid=10000;

commission | round
-----------+-------
```

```
     28.05 |  28.1
(1 row)
```

다음은 쿼리가 동일한 경우 정밀도를 반대 방향으로 연장하는 예입니다.

```
select commission, round(commission, -1)
from sales where salesid=10000;

commission | round
-----------+-------
     28.05 |    30
(1 row)
```

SIGN 함수

SIGN 함수는 숫자의 부호(양의 부호 또는 음의 부호)를 반환합니다. SIGN 함수의 결과는 인수 부호를 나타내는 1, -1 또는 0입니다.

구문

```
SIGN (number)
```

인수

number

숫자, 또는 숫자로 평가되는 표현식입니다. DECIMALor FLOAT8 유형일 수 있습니다.는 암시적 변환 규칙에 따라 다른 데이터 유형을 변환할 AWS Clean Rooms 수 있습니다.

반환 타입

SIGN은 입력 인수와 동일한 숫자 데이터 형식을 반환합니다. 입력이 DECIMAL이면 출력은 DECIMAL(1,0)입니다.

예시

판매 테이블에서 주어진 거래에 대해 지급된 수수료의 부호를 확인하려면 다음 예제를 사용합니다.

```
SELECT commission, SIGN(commission)
FROM sales WHERE salesid=10000;
```

```
+------------+------+
| commission | sign |
+------------+------+
|      28.05 |    1 |
+------------+------+
```

SIN 함수

SIN은 숫자의 사인을 반환하는 삼각 함수입니다. 반환 값은 -1과 1 사이입니다.

구문

```
SIN(number)
```

인수

number

라디안 단위의 DOUBLE PRECISION 숫자입니다.

반환 타입

DOUBLE PRECISION

예시

-PI의 사인을 반환하려면 다음 예제를 사용합니다.

```
SELECT SIN(-PI());

+-------------------------+
|           sin           |
+-------------------------+
| -0.00000000000000012246 |
+-------------------------+
```

SQRT 함수

SQRT 함수는 숫자 값의 제곱근을 반환합니다. 한 숫자에 동일한 숫자를 곱하면 지정된 값을 얻을 경우 해당 숫자를 제곱근이라고 합니다.

구문

```
SQRT (expression)
```

인수

expression

표현식의 데이터 형식은 정수, 소수 또는 부동 소수점이 되어야 합니다. 표현식에는 함수가 포함될 수 있습니다. 시스템에서 암시적 형식 변환을 수행할 수 있습니다.

반환 타입

SQRT는 DOUBLE PRECISION 숫자를 반환합니다.

예시

다음 예에서는 숫자의 제곱근을 반환합니다.

```
select sqrt(16);

sqrt
---------------
4
```

다음 예에서는 암시적 형식 변환을 수행합니다.

```
select sqrt('16');

sqrt
---------------
4
```

다음 예에서는 함수를 중첩하여 복잡한 작업을 수행합니다.

```
select sqrt(round(16.4));

sqrt
---------------
4
```

다음 예에서는 원의 면적을 지정했을 때 반지름의 길이를 산출합니다. 예를 들어 면적을 제곱 인치로 지정하면, 반지름을 인치 단위로 계산합니다. 샘플에서 면적은 20입니다.

```
select sqrt(20/pi());
```

이렇게 하면 값 5.0462650440403210이 반환됩니다.

다음은 SALES 테이블에서 COMMISSION 값의 제곱근을 반환하는 예입니다. COMMISSION 열은 DECIMAL 열입니다. 이 예에서는 복잡한 조건부 논리가 있는 쿼리에서 함수를 사용하는 방법을 보여 줍니다.

```
select sqrt(commission)
from sales where salesid < 10 order by salesid;

sqrt
------------------
10.4498803820905
3.37638860322683
7.24568837309472
5.1234753829798
...
```

다음은 동일한 COMMISSION 값 집합의 제곱근을 반올림하여 반환하는 쿼리입니다.

```
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
...
```

의 샘플 데이터에 대한 자세한 내용은 샘플 데이터베이스를 AWS Clean Rooms참조하세요.

TRUNC 함수

TRUNC 함수는 숫자를 이전 정수 또는 소수로 자릅니다.

TRUNC 함수는 옵션으로 두 번째 인수를 추가할 수 있습니다. 이 두 번째 인수는 어느 방향이든 반올림할 소수 자릿수를 나타내는 정수입니다. 두 번째 인수를 제공하지 않으면 함수는 가장 가까운 정수로 반올림됩니다. 두 번째 인수 >n이 지정되면 함수는 전체 자릿수의 소수 자릿수가 >n인 가장 가까운 숫자로 반올림됩니다. 이 함수는 타임스탬프를 잘라서 날짜를 반환하기도 합니다.

구문

```
TRUNC (number [ , integer ] |
timestamp )
```

인수

number

숫자 또는 숫자로 평가되는 표현식입니다. DECIMAL 또는 FLOAT8 유형일 수 있습니다.는 암시적 변환 규칙에 따라 다른 데이터 유형을 변환할 AWS Clean Rooms 수 있습니다.

integer(옵션)

어느 방향이든 정밀도의 소수점 자리 수를 나타내는 정수입니다. integer를 지정하지 않으면 숫자가 정수로 잘립니다. integer를 지정하면 숫자가 지정한 소수점 자리에서 절사됩니다.

timestamp

이 함수는 타임스탬프에서 날짜를 반환하기도 합니다. `00:00:00` 형식의 타임스탬프 값을 시간으로 반환하려면 함수 결과를 타임스탬프로 변환해야 합니다.

반환 타입

TRUNC는 첫 번째 입력 인수와 동일한 데이터 형식을 반환합니다. 타임스탬프일 때는 TRUNC가 날짜를 반환합니다.

예시

다음은 임의의 거래에서 지불되는 수수료를 절사하는 예입니다.

```
select commission, trunc(commission)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   111
```

```
(1 row)
```

다음은 동일한 수수료 값을 첫 번째 소수점 자리까지 절사하는 예입니다.

```
select commission, trunc(commission,1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 | 111.1

(1 row)
```

다음은 두 번째 인수 값을 음수로 하여 수수료를 절사하는 예입니다. 그 결과 111.15는 110으로 내림 처리됩니다.

```
select commission, trunc(commission,-1)
from sales where salesid=784;

commission | trunc
-----------+-------
    111.15 |   110
(1 row)
```

SYSDATE 함수(타임스탬프 반환)의 결과에서 날짜 구간을 반환합니다.

```
select sysdate;

timestamp
----------------------------
2011-07-21 10:32:38.248109
(1 row)

select trunc(sysdate);

trunc
------------
2011-07-21
(1 row)
```

다음은 TRUNC 함수를 TIMESTAMP 열에 적용하는 예입니다. 반환 형식은 날짜입니다.

```
select trunc(starttime) from event
order by eventid limit 1;

trunc
------------
2008-01-25
(1 row)
```

스칼라 함수

이 섹션에서는 AWS Clean Rooms Spark SQL에서 지원되는 스칼라 함수에 대해 설명합니다. 스칼라 함수는 하나 이상의 값을 입력으로 받아 단일 값을 출력으로 반환하는 함수입니다. 스칼라 함수는 개별 행 또는 요소에서 작동하며 각 입력에 대해 단일 결과를 생성합니다.

SIZE와 같은 스칼라 함수는 집계 함수(수, 합계, 평균) 및 테이블 생성 함수(폭발, 평면화)와 같은 다른 유형의 SQL 함수와 다릅니다. 이러한 다른 함수 유형은 여러 행에서 작동하거나 여러 행을 생성하는 반면, 스칼라 함수는 개별 행 또는 요소에서 작동합니다.

주제

- SIZE 함수

SIZE 함수

SIZE 함수는 기존 배열, 맵 또는 문자열을 인수로 받아 해당 데이터 구조의 크기 또는 길이를 나타내는 단일 값을 반환합니다. 새 데이터 구조는 생성되지 않습니다. 새 데이터 구조를 생성하는 대신 기존 데이터 구조의 속성을 쿼리하고 분석하는 데 사용됩니다.

이 함수는 배열의 요소 수 또는 문자열 길이를 결정하는 데 유용합니다. SQL에서 배열 및 기타 데이터 구조를 사용할 때 특히 유용할 수 있습니다. 데이터의 크기 또는 카디널리티에 대한 정보를 얻을 수 있기 때문입니다.

구문

```
size(expr)
```

인수

expr

ARRAY, MAP 또는 STRING 표현식입니다.

반환 타입

SIZE 함수는 INTEGER를 반환합니다.

예제

이 예제에서는 SIZE 함수가 배열에 적용`['b', 'd', 'c', 'a']`되고 배열의 요소 수`4`인 값을 반환합니다.

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

이 예제에서는 SIZE 함수가 맵에 적용`{'a': 1, 'b': 2}`되고 맵의 키-값 페어 수`2`인 값을 반환합니다.

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

이 예제에서는 SIZE 함수가 문자열에 적용`'hello world'`되고 문자열의 문자 수`11`인 값을 반환합니다.

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

문자열 함수

문자열 함수는 문자열을, 혹은 문자열로 평가되는 표현식을 처리 및 조작합니다. 이 함수에서 string 인수가 리터럴 값일 때는 작은따옴표로 묶어야 합니다. 지원되는 데이터 형식은 CHAR와 VARCHAR입니다.

다음 섹션에서는 함수 이름과 구문, 그리고 지원되는 함수에 대한 설명에 대해서 살펴보겠습니다. 문자열에 대한 오프셋은 모두 1부터 시작됩니다.

주제

- ||(연결) 연산자
- BTRIM 함수
- CONCAT 함수
- FORMAT_STRING 함수

- LEFT 및 RIGHT 함수
- LENGTH 함수
- LOWER 함수
- LPAD 및 RPAD 함수
- ltrim 함수
- POSITION 함수
- REGEXP_COUNT 함수
- REGEXP_INSTR 함수
- REGEXP_REPLACE 함수
- REGEXP_SUBSTR 함수
- REPEAT 함수
- REPLACE 함수
- REVERSE 함수
- RTRIM 함수
- SPLIT 함수
- SPLIT_PART 함수
- SUBSTRING 함수
- TRANSLATE 함수
- TRIM 함수
- UPPER 함수
- UUID 함수

||(연결) 연산자

|| 기호의 양쪽으로 두 표현식을 연결하여 연결된 표현식을 반환합니다.

연결 연산자는 CONCAT 함수와(과) 비슷합니다.

> **Note**
>
> CONCAT 함수와 연결 연산자 모두 표현식 중 하나 또는 둘 모두 NULL이면 결과도 NULL을 반환합니다.

구문

```
expression1 || expression2
```

인수

expression1, expression2

두 인수 모두 고정 길이 또는 가변 길이 문자열이거나 표현식이 될 수 있습니다.

반환 타입

|| 연산자는 문자열을 반환합니다. 문자열 형식은 입력 인수와 동일합니다.

예시

다음은 USERS 테이블에서 FIRSTNAME 필드와 LASTNAME 필드를 연결하는 예입니다.

```
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 값이 포함되었을 수도 있는 열을 연결하려면 NVL 및 COALESCE 함수 표현식을 사용해야 합니다. 다음은 NVL을 사용하여 NULL 값이 발견될 때마다 0을 반환하는 예입니다.

```
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 함수

BTRIM 함수는 선행 및 후행 공백을 제거하거나 옵션으로 지정하는 문자열과 일치하는 선행 및 후행 문자를 제거하여 문자열을 잘라냅니다.

구문

```
BTRIM(string [, trim_chars ] )
```

인수

string

잘라낼 입력 VARCHAR 문자열입니다.

trim_chars

일치시킬 문자가 포함된 VARCHAR 문자열입니다.

반환 타입

BTRIM 함수는 VARCHAR 문자열을 반환합니다.

예시

다음은 문자열 ' abc '에서 선행 및 후행 공백을 잘라내는 예입니다.

```
select '     abc    ' as untrim, btrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

다음은 문자열 'xyzaxyzbxyzcxyz'에서 선행 및 후행 'xyz' 문자열을 제거하는 예입니다. 결과를 보면 선행 및 후행 'xyz'만 제거되었고 문자열 내부에서는 제거되지 않았습니다.

```
select 'xyzaxyzbxyzcxyz' as untrim,
btrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | axyzbxyzc
```

다음 예제에서는 trim_chars 목록 'tes'의 모든 문자와 일치하는 문자열 'setuphistorycassettes'에서 선행 및 후행 부분을 제거합니다. 입력 문자열의 시작 또는 끝에서 trim_chars 목록에 없는 다른 문자 앞에 오는 모든 t, e 또는 s는 제거됩니다.

```
SELECT btrim('setuphistorycassettes', 'tes');

     btrim
-----------------
 uphistoryca
```

CONCAT 함수

CONCAT 함수는 두 표현식을 연결하고 결과 표현식을 반환합니다. 2개 이상의 표현식을 연결하려면 CONCAT 함수를 중첩시켜 사용합니다. 두 표현식 사이의 연결 연산자(||)는 CONCAT 함수와 동일한 결과를 반환합니다.

> **Note**
>
> CONCAT 함수와 연결 연산자 모두 표현식 중 하나 또는 둘 모두 NULL이면 결과도 NULL을 반환합니다.

구문

```
CONCAT ( expression1, expression2 )
```

인수

expression1, expression2

두 인수 모두 고정 길이 문자열, 가변 길이 문자열, 2진 표현식 또는 이러한 입력 중 하나로 평가되는 표현식이 될 수 있습니다.

반환 타입

CONCAT는 표현식을 반환합니다. 표현식의 데이터 유형은 입력 인수와 동일합니다.

입력 표현식의 유형이 다른 경우 암시적으로 입력하려고 AWS Clean Rooms 하면 표현식 중 하나가 캐스팅됩니다. 값을 캐스팅할 수 없는 경우 오류가 반환됩니다.

예시

다음 예에서는 문자열 리터럴 2개를 연결합니다:

```
select concat('December 25, ', '2008');

concat
-------------------
December 25, 2008
(1 row)
```

다음은 CONCAT이 아닌 || 연산자를 사용하여 동일한 결과를 반환하는 예입니다.

```
select 'December 25, '||'2008';

concat
-------------------
December 25, 2008
(1 row)
```

다음은 CONCAT 함수 2개를 사용하여 문자열 3개를 연결하는 예입니다.

```
select concat('Thursday, ', concat('December 25, ', '2008'));

concat
----------------------------
Thursday, December 25, 2008
(1 row)
```

NULL 값이 포함되었을 수도 있는 열을 연결하려면 NVL 및 COALESCE 함수을 사용해야 합니다. 다음은 NVL을 사용하여 NULL 값이 발견될 때마다 0을 반환하는 예입니다.

```
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)
```

다음은 VENUE 테이블에서 CITY 값과 STATE 값을 연결하는 쿼리입니다.

```
select concat(venuecity, venuestate)
from venue
where venueseats > 75000
order by venueseats;

concat
-------------------
DenverCO
Kansas CityMO
East RutherfordNJ
LandoverMD
(4 rows)
```

다음은 CONCAT 함수를 중첩시켜 사용하는 쿼리입니다. 이 쿼리는 VENUE 테이블에서 CITY 값과 STATE 값을 연결하지만 쉼표와 공백으로 결과 문자열을 구분합니다.

```
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 함수

FORMAT_STRING 함수는 템플릿 문자열의 자리 표시자를 제공된 인수로 대체하여 형식이 지정된 문자열을 생성합니다. printf 스타일 형식 문자열에서 형식이 지정된 문자열을 반환합니다.

FORMAT_STRING 함수는 템플릿 문자열의 자리 표시자를 인수로 전달된 해당 값으로 대체하여 작동합니다. 이러한 유형의 문자열 형식은 출력 메시지, 보고서 또는 기타 유형의 정보 텍스트를 생성할 때와 같이 정적 텍스트와 동적 데이터를 혼합하여 포함하는 문자열을 동적으로 구성해야 할 때 유용할 수 있습니다. FORMAT_STRING 함수는 이러한 형식의 문자열 유형을 생성하는 간결하고 읽기 쉬운 방법을 제공하므로 출력을 생성하는 코드를 더 쉽게 유지 관리하고 업데이트할 수 있습니다.

구문

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

인수

strfmt

STRING 표현식입니다.

obj

STRING 또는 숫자 표현식입니다.

반환 타입

FORMAT_STRING은 문자열을 반환합니다.

예시

다음 예제에는 10진수(정수) 값의 경우 , 문자열 값의 %d 경우의 두 자리 표시자를 포함하는 템플릿 문자열이 포함되어 %s 있습니다. %d 자리 표시자는 10진수(정수) 값()으로 대체100되고 %s 자리 표시자는 문자열 값()으로 대체됩니다"days". 출력은 제공된 인수 로 대체된 자리 표시자가 있는 템플릿 문자열입니다"Hello World 100 days".

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

LEFT 및 RIGHT 함수

이 두 함수는 문자열의 가장 왼쪽 또는 가장 오른쪽에서 지정한 만큼 문자 수를 반환합니다.

반환되는 문자 수는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다.

구문

```
LEFT ( string,  integer )

RIGHT ( string,  integer )
```

인수

string

문자열 또는 문자열로 평가되는 모든 표현식입니다.

integer

양의 정수입니다.

반환 타입

LEFT 및 RIGHT는 VARCHAR 문자열을 반환합니다.

예시

다음은 ID가 1000부터 1005 사이인 이벤트 이름에서 가장 왼쪽부터 5자, 그리고 가장 오른쪽부터 5자를 반환하는 예입니다.

```
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 함수

LOWER 함수

문자열을 소문자로 변환합니다. LOWER는 UTF-8 멀티바이트 문자를 지원하여 문자당 최대 4바이트까지 가능합니다.

구문

```
LOWER(string)
```

인수

string

입력 파라미터는 VARCHAR 문자열 또는 VARCHAR로 암시적으로 변환될 수 있는 CHAR와 같은 기타 데이터 형식입니다.

반환 타입

LOWER 함수는 입력 문자열과 데이터 형식이 동일한 문자열을 반환합니다.

예시

다음은 CATNAME 필드를 소문자로 변환하는 예입니다.

```
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 및 RPAD 함수

이 두 함수는 지정한 길이에 따라 문자열에 문자를 추가 또는 첨부합니다.

구문

```
LPAD (string1, length, [ string2 ])
```

```
RPAD (string1, length, [ string2 ])
```

인수

string1

문자열, 혹은 문자 열(character column)의 이름 같이 문자열로 평가되는 표현식입니다.

length

함수의 결과 길이를 정의하는 정수입니다. 문자열의 길이는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. string1이 지정한 길이보다 길면 오른쪽에서 절사됩니다. length가 음수이면 함수 결과로 빈 문자열이 반환됩니다.

string2

string1에 추가 또는 첨부되는 1개 이상의 문자입니다. 이 인수는 옵션이며, 지정하지 않으면 공백이 사용됩니다.

반환 타입

이 두 함수는 VARCHAR 데이터 형식을 반환합니다.

예시

다음은 지정한 이벤트 이름 집합을 20자로 절사한 후 공백을 포함해서 더욱 짧은 이름을 추가하는 예입니다.

```
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)
```

다음은 동일한 이벤트 이름 집합을 20자로 절사하지만 0123456789를 포함하여 더욱 짧은 이름을 첨부하는 예입니다.

```
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 함수

문자열의 시작 부분부터 문자를 잘라냅니다. 잘라낸 문자 목록에서 문자만 포함하는 가장 긴 문자열을 제거합니다. 트리밍 문자가 입력 문자열에 나타나지 않으면 트리밍이 완료된 것입니다.

구문

```
LTRIM( string [, trim_chars] )
```

인수

string

잘라낼 문자열 열, 표현식 또는 문자열 리터럴입니다.

trim_chars

문자열의 처음부터 잘라낼 문자를 나타내는 문자열 열, 표현식 또는 문자열 리터럴입니다. 지정하지 않으면 공백이 잘라내기 문자로 사용됩니다.

반환 타입

LTRIM 함수는 입력 문자열(CHAR 또는 VARCHAR)과 데이터 유형이 동일한 문자열을 반환합니다.

예시

다음은 `listime` 열에서 연도를 잘라내는 예입니다. 문자열 리터럴 `'2008-'`의 잘라내기 문자는 왼쪽부터 잘라낼 문자를 나타냅니다. 잘라내기 문자 `'028-'`을 사용하는 경우에도 동일한 결과를 얻을 수 있습니다.

```
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
```

LTRIM은 trim_chars의 문자가 string의 첫 문자이면 모두 제거합니다. 다음은 'C', 'D', 'G' 문자가 VARCHAR 열인 VENUENAME의 첫 문자일 때 각 문자를 잘라내는 예입니다.

```
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
```

다음 예제에서는 venueid 열에서 검색된 잘라내기 문자 2를 사용합니다.

```
select ltrim('2008-01-24 06:43:29', venueid)
from venue where venueid=2;

ltrim
------------------
008-01-24 06:43:29
```

다음 예제에서는 2가 '0' 잘라내기 문자 앞에서 발견되었기 대문에 어떤 문자도 잘라내지 않습니다.

```
select ltrim('2008-01-24 06:43:29', '0');

ltrim
-------------------
2008-01-24 06:43:29
```

다음 예제에서는 기본 공백 잘라내기 문자를 사용하여 문자열의 시작 부분부터 두 개의 공백을 잘라냅니다.

```
select ltrim('  2008-01-24 06:43:29');
```

```
ltrim
-------------------
2008-01-24 06:43:29
```

POSITION 함수

문자열 내에서 지정한 하위 문자열의 위치를 반환합니다.

구문

```
POSITION(substring IN string )
```

인수

substring

string 내에서 검색할 하위 문자열입니다.

string

검색할 문자열 또는 열입니다.

반환 타입

POSITION 함수는 하위 문자열의 위치에 해당하는 정수를 반환합니다(0이 아닌 1부터 시작). 이 위치는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다.

사용 노트

문자열 내에서 하위 문자열이 발견되지 않으면 POSITION이 0을 반환합니다.

```
select position('dog' in 'fish');

position
----------
 0
(1 row)
```

예시

다음은 단어 `fish` 내에서 문자열 `dogfish`의 위치를 나타내는 예입니다.

```
select position('fish' in 'dogfish');

position
----------
 4
(1 row)
```

다음은 SALES 테이블에서 COMMISSION이 999.00를 초과하는 거래의 수를 반환하는 예입니다.

```
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 함수

문자열에서 정규 표현식 패턴을 검색한 후 패턴 발생 횟수를 나타내는 정수를 반환합니다. 일치하는 결과가 발견되지 않으면 함수가 0을 반환합니다.

구문

```
REGEXP_COUNT ( source_string, pattern [, position [, parameters ] ] )
```

인수

source_string

열 이름 같이 검색할 문자열 표현식입니다.

패턴

정규 표현식 패턴을 나타내는 문자열 리터럴입니다.

position

source_string 내에서 검색을 시작할 위치를 나타내는 양의 정수입니다. 이 위치는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. 기본 값은 1입니다. position이 1보다 작으면 검색이 source_string의 첫 문자부터 시작됩니다. position이 source_string의 문자 수보다 크면 결과는 0이 됩니다.

parameters

함수가 패턴과 일치하는 방법을 나타내는 하나 이상의 문자열 리터럴입니다. 가능한 값은 다음과 같습니다.

- c - 대/소문자를 구분하여 일치시킵니다. 기본값은 대/소문자 구분 일치를 사용하는 것입니다.
- i - 대/소문자를 구분하지 않고 일치시킵니다.
- p – PCRE(Perl Compatible Regular Expression) 방언으로 패턴을 해석합니다.

반환 타입

Integer

예시

다음은 3자 시퀀스가 발생하는 횟수를 계산하는 예입니다.

```
SELECT regexp_count('abcdefghijklmnopqrstuvwxyz', '[a-z]{3}');

 regexp_count
 --------------
            8
```

다음은 최상위 도메인 이름이 org 또는 edu인 횟수를 계산하는 예입니다.

```
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
```

다음 예에서는 대/소문자를 구분하지 않는 일치를 사용하여 문자열 FOX의 발생 횟수를 계산합니다.

```
SELECT regexp_count('the fox', 'FOX', 1, 'i');

 regexp_count
 --------------
```

```
            1
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예에서는 대/소문자를 구분하여 일치하는 단어의 발생 횟수를 계산합니다.

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'p');

 regexp_count
 --------------
            2
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정한 의미를 지닌 ?= 연산자가 사용됩니다. 이 예는 이러한 단어의 발생 횟수를 계산하지만 대/소문자를 구분하지 않는 일치를 사용한다는 점에서 이전 예와 다릅니다.

```
SELECT regexp_count('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 'ip');

 regexp_count
 --------------
            3
```

REGEXP_INSTR 함수

문자열에서 정규 표현식 패턴을 검색하여 일치하는 하위 문자열의 시작 위치 또는 종료 위치를 나타내는 정수를 반환합니다. 일치하는 결과가 발견되지 않으면 함수가 0을 반환합니다. REGEXP_INSTR은 POSITION 함수와 비슷하지만 문자열에서 정규 표현식 패턴을 검색할 수 있습니다.

구문

```
REGEXP_INSTR ( source_string, pattern [, position [, occurrence] [, option
 [, parameters ] ] ] ] )
```

인수

source_string

열 이름 같이 검색할 문자열 표현식입니다.

패턴

정규 표현식 패턴을 나타내는 문자열 리터럴입니다.

position

source_string 내에서 검색을 시작할 위치를 나타내는 양의 정수입니다. 이 위치는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. 기본 값은 1입니다. position이 1보다 작으면 검색이 source_string의 첫 문자부터 시작됩니다. position이 source_string의 문자 수보다 크면 결과는 0이 됩니다.

발생

사용할 패턴 발생을 나타내는 양의 정수입니다. REGEXP_INSTR은 첫 번째 발생에서 1을 뺀 개수의 일치하는 항목을 건너뜁니다. 기본 값은 1입니다. 발생이 1보다 작거나 소스 문자열에 있는 문자 수보다 클 경우 검색이 무시되고 결과가 0이 됩니다.

option

일치하는 항목의 첫 번째 문자 위치(0)를 반환할지 일치하는 항목의 끝 다음에 나오는 첫 번째 문자의 위치(1)를 반환할지 여부를 나타내는 값입니다. 0이 아닌 값은 1과 같습니다. 기본값은 0입니다.

parameters

함수가 패턴과 일치하는 방법을 나타내는 하나 이상의 문자열 리터럴입니다. 가능한 값은 다음과 같습니다.

- c - 대/소문자를 구분하여 일치시킵니다. 기본값은 대/소문자 구분 일치를 사용하는 것입니다.
- i - 대/소문자를 구분하지 않고 일치시킵니다.
- e - 하위 표현식을 사용하여 하위 문자열을 추출합니다.

  패턴에 하위 표현식이 포함되어 있을 경우 REGEXP_INSTR은 패턴의 첫 번째 하위 표현식을 사용하여 하위 문자열과 일치시킵니다. REGEXP_INSTR은 첫 번째 하위 표현식만 고려하며 추가 하위 표현식은 무시됩니다. 패턴에 하위 표현식이 없으면 REGEXP_INSTR이 'e' 파라미터를 무시합니다.
- p – PCRE(Perl Compatible Regular Expression) 방언으로 패턴을 해석합니다.

반환 타입

Integer

예시

다음은 도메인 이름에서 첫 @ 문자를 검색하여 처음 일치하는 결과의 시작 위치를 반환하는 예입니다.

```
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
```

다음은 단어 Center와(과) 그 변형까지 검색하여 처음 일치하는 결과의 시작 위치를 반환하는 예입니다.

```
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
```

다음 예에서는 대/소문자를 구분하지 않는 일치 논리를 사용하여 문자열 FOX의 첫 번째 발생 시작 위치를 찾습니다.

```
SELECT regexp_instr('the fox', 'FOX', 1, 1, 0, 'i');

 regexp_instr
 --------------
            5
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예에서는 두 번째 단어의 시작 위치를 찾습니다.

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'p');
```

```
 regexp_instr
 --------------
           21
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예는 두 번째 단어의 시작 위치를 찾지만 대/소문자를 구분하지 않는 일치를 사용한다는 점에서 이전 예와 다릅니다.

```
SELECT regexp_instr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 0, 'ip');

 regexp_instr
 --------------
           15
```

REGEXP_REPLACE 함수

문자열에서 정규 표현식 패턴을 검색한 후 발견되는 모든 패턴을 지정한 문자열로 변경합니다. REGEXP_REPLACE는 REPLACE 함수과 비슷하지만 문자열에서 정규 표현식 패턴을 검색할 수 있습니다.

REGEXP_REPLACE는 TRANSLATE 함수 및 REPLACE 함수과 비슷합니다. 단, TRANSLATE는 단일 문자를 여러 차례 변경하고, REPLACE는 전체 문자열 하나를 다른 문자열로 변경하는 반면 REGEXP_REPLACE는 문자열에서 정규 표현식 패턴을 검색할 수 있습니다.

구문

```
REGEXP_REPLACE ( source_string, pattern [, replace_string [ , position [, parameters
 ] ] ] )
```

인수

source_string

열 이름 같이 검색할 문자열 표현식입니다.

패턴

정규 표현식 패턴을 나타내는 문자열 리터럴입니다.

replace_string

발견되는 패턴을 각각 변경할 문자열 표현식(열 이름 등)입니다. 기본값은 빈 문자열입니다("").

position

source_string 내에서 검색을 시작할 위치를 나타내는 양의 정수입니다. 이 위치는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. 기본 값은 1입니다. position이 1보다 작으면 검색이 source_string의 첫 문자부터 시작됩니다. position이 source_string의 문자 수보다 크면 결과는 source_string이 됩니다.

parameters

함수가 패턴과 일치하는 방법을 나타내는 하나 이상의 문자열 리터럴입니다. 가능한 값은 다음과 같습니다.

- c - 대/소문자를 구분하여 일치시킵니다. 기본값은 대/소문자 구분 일치를 사용하는 것입니다.
- i - 대/소문자를 구분하지 않고 일치시킵니다.
- p – PCRE(Perl Compatible Regular Expression) 방언으로 패턴을 해석합니다.

반환 타입

VARCHAR

pattern 또는 replace_string이 NULL이면 결과도 NULL이 됩니다.

예시

다음은 이메일 주소에서 @ 및 도메인 주소를 삭제하는 예입니다.

```
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
```

다음 예에서는 이메일 주소의 도메인 이름을 값 `internal.company.com`으로 바꿉니다.

```
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
```

다음 예에서는 대소문자를 구분하지 않는 일치를 사용하여 값 `quick brown fox` 내에서 문자열 `FOX`를 모두 바꿉니다.

```
SELECT regexp_replace('the fox', 'FOX', 'quick brown fox', 1, 'i');

   regexp_replace
---------------------
 the quick brown fox
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예에서는 해당 단어가 나타날 때마다 값 `[hidden]`으로 바꿉니다.

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'p');

        regexp_replace
------------------------------
 [hidden] plain A1234 [hidden]
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예는 해당 단어가 나타날 때마다 값 `[hidden]`으로 바꾸지만 대/소문자를 구분하지 않는 일치를 사용한다는 점에서 이전 예와 다릅니다.

```
SELECT regexp_replace('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 '[hidden]', 1, 'ip');
```

```
          regexp_replace
---------------------------------
 [hidden] plain [hidden] [hidden]
```

REGEXP_SUBSTR 함수

문자열에서 정규 표현식 패턴을 검색하여 문자를 반환합니다. REGEXP_SUBSTR은 SUBSTRING 함수 함수와 비슷하지만 문자열에서 정규 표현식 패턴을 검색할 수 있습니다. 함수에서 정규 표현식이 문자열의 어떤 문자와도 일치하지 않는 경우 빈 문자열이 반환됩니다.

구문

```
REGEXP_SUBSTR ( source_string, pattern [, position [, occurrence [, parameters ] ] ] )
```

인수

source_string

검색할 문자열 표현식입니다.

패턴

정규 표현식 패턴을 나타내는 문자열 리터럴입니다.

position

source_string 내에서 검색을 시작할 위치를 나타내는 양의 정수입니다. 이 위치는 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. 기본 값은 1입니다. position이 1보다 작으면 검색이 source_string의 첫 문자부터 시작됩니다. position이 source_string의 문자 수보다 크면 결과는 빈 문자열("")이 됩니다.

발생

사용할 패턴 발생을 나타내는 양의 정수입니다. REGEXP_SUBSTR은 첫 번째 발생에서 1을 뺀 개수의 일치하는 항목을 건너뜁니다. 기본 값은 1입니다. 발생이 1보다 작거나 source_string에 있는 문자 수보다 클 경우 검색이 무시되고 결과가 NULL이 됩니다.

parameters

함수가 패턴과 일치하는 방법을 나타내는 하나 이상의 문자열 리터럴입니다. 가능한 값은 다음과 같습니다.

- c - 대/소문자를 구분하여 일치시킵니다. 기본값은 대/소문자 구분 일치를 사용하는 것입니다.

- i - 대/소문자를 구분하지 않고 일치시킵니다.
- e - 하위 표현식을 사용하여 하위 문자열을 추출합니다.

  패턴에 하위 표현식이 포함되어 있을 경우 REGEXP_SUBSTR은 패턴의 첫 번째 하위 표현식을 사용하여 하위 문자열과 일치시킵니다. 하위 표현식은 괄호로 묶인 패턴 내 표현식입니다. 예를 들어 `'This is a (\\w+)'` 패턴은 첫 번째 표현식과 뒤에 단어가 오는 `'This is a '` 문자열을 일치시킵니다. e 매개 변수가 있는 REGEXP_SUBSTR은 패턴을 반환하는 대신 하위 표현식 내의 문자열만 반환합니다.

  REGEXP_SUBSTR은 첫 번째 하위 표현식만 고려하며 추가 하위 표현식은 무시됩니다. 패턴에 하위 표현식이 없으면 REGEXP_SUBSTR이 'e' 파라미터를 무시합니다.
- p – PCRE(Perl Compatible Regular Expression) 방언으로 패턴을 해석합니다.

반환 타입

VARCHAR

예시

다음은 @ 문자와 도메인 확장자 사이의 이메일 주소 구간을 반환하는 예입니다.

```
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
```

다음 예에서는 대/소문자를 구분하지 않는 일치를 사용하여 문자열 FOX의 첫 번째 발생에 해당하는 입력 부분을 반환합니다.

```
SELECT regexp_substr('the fox', 'FOX', 1, 1, 'i');

 regexp_substr
---------------
 fox
```

다음 예제에서는 소문자로 시작하는 입력의 첫 번째 부분을 반환합니다. 이는 c 파라미터가 없는 동일한 SELECT 문과 기능적으로 동일합니다.

```
SELECT regexp_substr('THE SECRET CODE IS THE LOWERCASE PART OF 1931abc0EZ.', '[a-z]+',
 1, 1, 'c');

 regexp_substr
---------------
 abc
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예에서는 두 번째 단어에 해당하는 입력 부분을 반환합니다.

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'p');

 regexp_substr
---------------
 a1234
```

다음 예에서는 PCRE 방언으로 작성된 패턴을 사용하여 하나 이상의 숫자와 하나의 소문자가 포함된 단어를 찾습니다. PCRE에서 특정 미리 보기 의미가 있는 ?= 연산자가 사용됩니다. 이 예는 두 번째 단어에 해당하는 입력 부분을 반환하지만 대/소문자를 구분하지 않는 일치를 사용한다는 점에서 이전 예와 다릅니다.

```
SELECT regexp_substr('passwd7 plain A1234 a1234', '(?=[^ ]*[a-z])(?=[^ ]*[0-9])[^ ]+',
 1, 2, 'ip');

 regexp_substr
---------------
 A1234
```

다음 예에서는 하위 표현식을 사용하여 `'this is a (\\w+)'` 패턴과 일치하는 두 번째 문자열을 찾습니다. 괄호를 친 하위 표현식을 반환합니다.

```
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 함수

문자열을 지정한 횟수만큼 반복합니다. 입력 파라미터가 숫자라고 해도 REPEAT는 이를 문자열로 처리합니다.

구문

```
REPEAT(string, integer)
```

인수

string

첫 번째 입력 파라미터는 반복할 문자열입니다.

integer

두 번째 파라미터는 문자열을 반복할 횟수를 나타내는 정수입니다.

반환 타입

REPEAT 함수는 문자열을 반환합니다.

예시

다음은 CATEGORY 테이블에서 CATID 열의 값을 3회 반복하는 예입니다.

```
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 함수

기존 문자열에서 발견되는 모든 문자 집합을 다른 지정 문자로 변경합니다.

REPLACE는 TRANSLATE 함수 및 REGEXP_REPLACE 함수과 비슷합니다. 단, TRANSLATE는 단일 문자를 여러 차례 변경하고, REGEXP_REPLACE는 문자열에서 정규 표현식 패턴을 검색하는 반면 REPLACE는 전체 문자열 하나를 다른 문자열로 변경합니다.

구문

```
REPLACE(string1, old_chars, new_chars)
```

인수

string

검색할 CHAR 또는 VARCHAR 문자열입니다.

old_chars

변경할 CHAR 또는 VARCHAR 문자열입니다.

new_chars

old_string을 변경할 새로운 CHAR 또는 VARCHAR 문자열입니다.

반환 타입

VARCHAR

old_chars 또는 new_chars가 NULL이면 결과도 NULL이 됩니다.

예시

다음은 CATGROUP 필드에서 문자열 Shows를 Theatre로 변환하는 예입니다.

```
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 함수

REVERSE 함수는 문자열에 대해 실행되며, 문자를 역순으로 반환합니다. 예를 들어, `reverse('abcde')`는 edcba를 반환합니다. 이 함수는 문자 데이터 형식 외에 숫자나 날짜 데이터 형식에서도 실행되지만 대부분은 문자열에서 실용적인 값을 갖습니다.

구문

```
REVERSE ( expression )
```

인수

expression

문자, 날짜, 타임스탬프, 숫자 데이터 형식 등 문자 반전의 대상이 되는 표현식입니다. 모든 표현식은 묵시적으로 가변 길이 문자열로 변환됩니다. 고정 길이 문자열에서는 후행 공백이 무시됩니다.

반환 타입

REVERSE는 VARCHAR를 반환합니다.

예시

다음은 USERS 테이블에서 각기 다른 도시 5곳을 선택한 후 각 도시의 이름을 반전시키는 예입니다.

```
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)
```

다음은 판매 ID 5개를 선택한 후 각 ID를 문자열로 반전시키는 예입니다.

```
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 함수

RTRIM 함수는 문자열 끝부터 지정된 문자 집합을 잘라냅니다. 잘라낸 문자 목록에서 문자만 포함하는 가장 긴 문자열을 제거합니다. 트리밍 문자가 입력 문자열에 나타나지 않으면 트리밍이 완료된 것입니다.

구문

```
RTRIM( string, trim_chars )
```

인수

string

잘라낼 문자열 열, 표현식 또는 문자열 리터럴입니다.

trim_chars

문자열의 끝부터 잘라낼 문자를 나타내는 문자열 열, 표현식 또는 문자열 리터럴입니다. 지정하지 않으면 공백이 잘라내기 문자로 사용됩니다.

반환 타입

string 인수와 동일한 데이터 형식의 문자열입니다.

예시

다음은 문자열 ' abc '에서 선행 및 후행 공백을 잘라내는 예입니다.

```
select '     abc    ' as untrim, rtrim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    |    abc
```

다음은 문자열 'xyzaxyzbxyzcxyz'에서 후행 'xyz' 문자열을 제거하는 예입니다. 결과를 보면 후행하는 'xyz'만 제거되었고 문자열 내부에서는 제거되지 않았습니다.

```
select 'xyzaxyzbxyzcxyz' as untrim,
rtrim('xyzaxyzbxyzcxyz', 'xyz') as trim;

     untrim      |   trim
-----------------+-----------
 xyzaxyzbxyzcxyz | xyzaxyzbxyzc
```

다음 예제에서는 trim_chars 목록의 모든 문자와 일치하는 문자열 'setuphistorycassettes'에서 후행 부분을 제거합니다. 입력 문자열의 끝에서 trim_chars 목록에 없는 다른 문자 앞에 오는 모든 t, e 또는 s는 제거됩니다.

```
SELECT rtrim('setuphistorycassettes', 'tes');

     rtrim
-----------------
 setuphistoryca
```

다음은 VENUENAME의 끝에서 있는 경우에 한해 문자 'Park'를 잘라내는 예입니다.

```
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
```

위 예를 보면 P, a, r 또는 k가 VENUENAME의 끝에 있을 경우 RTRIM이 각 문자를 모두 제거한 것을 알 수 있습니다.

SPLIT 함수

SPLIT 함수를 사용하면 더 큰 문자열에서 하위 문자열을 추출하여 배열로 사용할 수 있습니다. SPLIT 함수는 특정 구분 기호 또는 패턴을 기반으로 문자열을 개별 구성 요소로 나눌 때 유용합니다.

구문

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

인수

문자열

분할할 문자열 표현식입니다.

regex

정규식을 나타내는 문자열입니다. 정규식 문자열은 Java 정규식이어야 합니다.

limit

정규식이 적용되는 횟수를 제어하는 정수 표현식입니다.

- limit > 0: 결과 배열의 길이는 제한을 초과하지 않으며 결과 배열의 마지막 항목에는 마지막으로 일치하는 정규식 이후의 모든 입력이 포함됩니다.
- limit <= 0: 정규식은 가능한 한 여러 번 적용되며 결과 배열은 모든 크기일 수 있습니다.

반환 타입

SPLIT 함수는 ARRAY<STRING>을 반환합니다.

If `limit > 0`: 결과 배열의 길이는 제한을 초과하지 않으며 결과 배열의 마지막 항목에는 마지막으로 일치하는 정규식 이후의 모든 입력이 포함됩니다.

인 경우`limit <= 0`: 정규식이 가능한 한 많이 적용되며 결과 배열은 모든 크기일 수 있습니다.

예시

이 예제에서 SPLIT 함수는 문자 , `'A' 'B'`또는 `'C'` (정규 표현식 패턴 로 지정된 대로)가 발생할 `'oneAtwoBthreeC'` 때마다 입력 문자열을 분할합니다`'[ABC]'`. 결과 출력은 `"one"`, `"two"`, `"three"`, 빈 문자열의 네 가지 요소로 구성된 배열입니다`""`.

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

SPLIT_PART 함수

지정 구분자를 기준으로 문자열을 분할한 후 지정된 위치에 해당하는 부분을 반환합니다.

구문

```
SPLIT_PART(string, delimiter, position)
```

인수

string

분할할 문자열 열, 표현식 또는 문자열 리터럴입니다. 문자열은 CHAR 또는 VARCHAR가 될 수 있습니다.

delimiter

입력 문자열의 섹션을 나타내는 구분자 문자열입니다.

delimiter가 리터럴이면 작은따옴표로 묶어야 합니다.

position

반환할 문자열 구간의 위치입니다(1부터 시작). 반드시 0보다 큰 정수이어야 합니다. position이 문자열 구간의 수보다 크면 SPLIT_PART가 빈 문자열을 반환합니다. string에서 delimiter를 찾을 수 없는 경우 반환된 값에는 전체 string 또는 빈 값이 될 수 있는 지정된 부분의 내용이 포함됩니다.

반환 타입

string 파라미터와 동일한 CHAR 또는 VARCHAR 문자열입니다.

예시

다음 예제에서는 $ 구분 기호를 사용하여 문자열 리터럴을 여러 부분으로 분할하고 두 번째 부분을 반환합니다.

```
select split_part('abc$def$ghi','$',2)

split_part
----------
def
```

다음 예제에서는 $ 구분 기호를 사용하여 문자열 리터럴을 여러 부분으로 분할합니다. 4 부분을 찾을 수 없기 때문에 빈 문자열이 반환됩니다.

```
select split_part('abc$def$ghi','$',4)

split_part
----------

```

다음 예제에서는 # 구분 기호를 사용하여 문자열 리터럴을 여러 부분으로 분할합니다. 이 구분 기호를 찾을 수 없기 때문에 첫 번째 부분인 전체 문자열이 반환됩니다.

```
select split_part('abc$def$ghi','#',1)

split_part
------------
abc$def$ghi
```

다음은 타임스탬프 필드인 LISTTIME을 년, 월, 일 요소로 분할하는 예입니다.

```
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
```

다음은 LISTTIME 타임스탬프 필드를 선택하고, '-' 문자를 기준으로 필드를 분할하여 월(LISTTIME 문자열의 두 번째 구간)을 가져온 다음 각 월의 항목 수를 계산하는 예입니다.

```
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 함수

지정된 시작 위치를 기반으로 문자열의 하위 집합을 반환합니다.

입력이 문자열인 경우 추출된 문자의 시작 위치와 수는 바이트가 아닌 문자를 기준으로 하므로 멀티바이트 문자는 단일 문자로 계산됩니다. 입력이 이진 표현식인 경우 시작 위치와 추출된 하위 문자열은 바이트를 기반으로 합니다. 음의 길이는 지정할 수 없지만 음의 시작 위치는 지정 가능합니다.

구문

```
SUBSTRING(charactestring FROM start_position [ FOR numbecharacters ] )
```

```
SUBSTRING(charactestring, start_position, numbecharacters )
```

```
SUBSTRING(binary_expression, start_byte, numbebytes )
```

```
SUBSTRING(binary_expression, start_byte )
```

인수

문자

검색 대상의 문자열입니다. 문자가 아닌 데이터 형식도 문자열로 처리됩니다.

start_position

문자열 내에서 추출을 시작할 위치이며, 1부터 시작됩니다. start_position은 바이트가 아닌 문자 수를 기준으로 하기 때문에 멀티바이트 문자도 단일 문자로 계산됩니다. 이 수는 음의 값이 될 수 있습니다.

numbecharacters

추출할 문자 수(하위 문자열의 길이)입니다. 숫자 문자는 바이트가 아닌 문자 수를 기준으로 하므로 멀티바이트 문자는 단일 문자로 계산됩니다. 이 수는 음의 값이 될 수 없습니다.

start_byte

1에서 시작하여 추출을 시작할 이진 표현식 내의 위치입니다. 이 수는 음의 값이 될 수 있습니다.

numbebytes

추출할 바이트 수, 즉 하위 문자열의 길이입니다. 이 수는 음의 값이 될 수 없습니다.

반환 타입

VARCHAR

문자열에 대한 사용 참고 사항

다음은 6번째 문자부터 4자의 문자열을 반환하는 예입니다.

```
select substring('caterpillar',6,4);
substring
-----------
pill
(1 row)
```

start_position + numbecharacters가 문자열 길이를 초과하는 경우 SUBSTRING은 start_position부터 문자열 끝까지 하위 문자열을 반환합니다. 예:

```
select substring('caterpillar',6,8);
substring
-----------
pillar
(1 row)
```

start_position이 0 또는 음수인 경우에는 SUBSTRING 함수가 문자열의 첫 번째 문자부터 start_position + numbecharacters -1의 길이를 갖는 하위 문자열을 반환합니다. 예:

```
select substring('caterpillar',-2,6);
substring
-----------
cat
(1 row)
```

start_position + numbecharacters -1이 0보다 작거나 같으면 SUBSTRING이 빈 문자열을 반환합니다. 예:

```
select substring('caterpillar',-5,4);
substring
-----------

(1 row)
```

예시

다음은 LISTING 테이블의 LISTTIME 문자열에서 월을 반환하는 예입니다.

```
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)
```

다음은 위와 동일하지만 FROM...FOR 옵션을 사용하는 예입니다.

```
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)
```

멀티바이트 문자가 포함되었을 수도 있는 문자열에서는 접두사를 예측적으로 추출할 때 SUBSTRING 함수를 사용할 수 없습니다. 그 이유는 문자 수가 아닌 바이트 수를 기준으로 멀티바이트 문자열의 길이를 지정해야 하기 때문입니다. 바이트 길이를 기준으로 문자열의 시작 세그먼트를 추출하려면 문자열을 VARCHAR(byte_length)로 변환하여 절사할 수 있습니다. 여기에서 byte_length는 반드시 필요한 길이입니다. 다음은 문자열 `'Fourscore and seven'`에서 첫 5바이트를 추출하는 예입니다.

```
select cast('Fourscore and seven' as varchar(5));

varchar
-------
Fours
```

다음 예에서는 입력 문자열 `Silva, Ana`의 마지막 공백 뒤에 나타나는 첫 번째 이름 `Ana`를 반환합니다.

```
select reverse(substring(reverse('Silva, Ana'), 1, position(' ' IN reverse('Silva,
 Ana'))))

 reverse
-----------
 Ana
```

TRANSLATE 함수

임의의 표현식에서 발견되는 모든 지정 문자를 지정한 대체 문자로 변경합니다. 기존 문자는 characters_to_replace의 문자 위치와 characters_to_substitute 인수에 따라 변환 문자로 매핑됩니다. characters_to_replace 인수에서 지정하는 문자 수가 characters_to_substitute 인수에서 지정하는 문자 수보다 많으면 characters_to_replace 인수의 추가 문자가 반환 값에서 생략됩니다.

TRANSLATE는 REPLACE 함수 및 REGEXP_REPLACE 함수과 비슷합니다. 단, REPLACE는 전체 문자열 하나를 다른 문자열로 변경하고, REGEXP_REPLACE는 문자열에서 정규 표현식 패턴을 검색하는 반면 TRANSLATE는 단일 문자를 여러 차례 변경합니다.

인수가 NULL이면 반환되는 값도 NULL입니다.

구문

```
TRANSLATE ( expression, characters_to_replace, characters_to_substitute )
```

인수

expression

변환 대상인 표현식입니다.

characters_to_replace

변경 대상인 문자가 포함된 문자열입니다.

characters_to_substitute

대체할 문자가 포함된 문자열입니다.

반환 타입

VARCHAR

예시

다음은 문자열에서 일부 문자를 변경하는 예입니다.

```
select translate('mint tea', 'inea', 'osin');

translate
-----------
most tin
```

다음 예에서는 다음 열의 모든 값에서 at 기호(@)를 마침표로 변경합니다.

```
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
```

다음 예에서는 다음 열의 모든 값에서 공백을 밑줄로 변경하고 마침표를 제거합니다.

```
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 함수

선행 및 후행 공백을 제거하거나 옵션으로 지정하는 문자열과 일치하는 선행 및 후행 문자를 제거하여 문자열을 잘라냅니다.

구문

```
TRIM( [ BOTH ] [ trim_chars FROM ] string
```

인수

trim_chars

(옵션) 문자열에서 잘라낼 문자입니다. 이 파라미터를 생략하면 공백이 잘립니다.

string

자르기 대상이 되는 문자열입니다.

반환 타입

TRIM 함수는 VARCHAR 또는 CHAR 문자열을 반환합니다. TRIM 함수를 SQL 명령과 함께 사용하는 경우는 AWS Clean Rooms 결과를 VARCHAR로 묵시적으로 변환합니다. SQL 함수에 대해 SELECT 목록에서 TRIM 함수를 사용하는 경우는 결과를 암시적으로 변환하지 AWS Clean Rooms 않으며 데이터 형식 불일치 오류를 방지하기 위해 명시적 변환을 수행해야 할 수 있습니다. 명시적 변환에 대한 자세한 내용은 CAST 함수 함수를 참조하세요.

예시

다음은 문자열 ' abc '에서 선행 및 후행 공백을 잘라내는 예입니다.

```
select '     abc    ' as untrim, trim('     abc    ') as trim;

untrim    | trim
----------+------
   abc    | abc
```

다음은 문자열 "dog"에서 큰따옴표를 제거하는 예입니다.

```
select trim('"' FROM '"dog"');

btrim
-------
dog
```

TRIM은 trim_chars의 문자가 string의 첫 문자이면 모두 제거합니다. 다음은 'C', 'D', 'G' 문자가 VARCHAR 열인 VENUENAME의 첫 문자일 때 각 문자를 잘라내는 예입니다.

```
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 함수

문자열을 소문자로 변환합니다. UPPER는 UTF-8 멀티바이트 문자를 지원하여 문자당 최대 4바이트까지 가능합니다.

구문

```
UPPER(string)
```

인수

string

입력 파라미터는 VARCHAR 문자열 또는 VARCHAR로 암시적으로 변환될 수 있는 CHAR와 같은 기타 데이터 형식입니다.

반환 타입

UPPER 함수는 입력 문자열과 데이터 형식이 동일한 문자열을 반환합니다.

예시

다음은 CATNAME 필드를 대문자로 변환하는 예입니다.

```
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 함수

UUID 함수는 UUID(Universally Unique Identifier)를 생성합니다.

UUIDs는 다음과 같은 다양한 목적으로 고유 식별자를 제공하는 데 일반적으로 사용되는 전역 고유 식별자입니다.

- 데이터베이스 레코드 또는 기타 데이터 엔터티 식별.
- 파일, 디렉터리 또는 기타 리소스에 대한 고유한 이름 또는 키 생성.
- 분산 시스템에서 데이터를 추적하고 상호 연관시킵니다.
- 네트워크 패킷, 소프트웨어 구성 요소 또는 기타 디지털 자산에 대한 고유 식별자를 제공합니다.

UUID 함수는 분산 시스템에서도 장기간 동안 매우 높은 확률로 고유한 UUID 값을 생성합니다. UUIDs는 일반적으로 현재 타임스탬프, 컴퓨터의 네트워크 주소 및 기타 무작위 또는 의사 무작위 데이터의 조합을 사용하여 생성되므로 생성된 각 UUID가 다른 UUID와 충돌할 가능성이 매우 낮습니다.

SQL 쿼리의 컨텍스트에서 UUID 함수를 사용하여 데이터베이스에 삽입되는 새 레코드에 대한 고유 식별자를 생성하거나 데이터 파티셔닝, 인덱싱 또는 고유 식별자가 필요한 기타 목적을 위한 고유 키를 제공할 수 있습니다.

> **Note**
>
> UUID 함수는 비결정적입니다.

구문

```
uuid()
```

인수

UUID 함수는 인수를 취하지 않습니다.

반환 타입

UUID는 범용 고유 식별자(UUID) 문자열을 반환합니다. 값은 정식 UUID 36자 문자열로 반환됩니다.

예시

다음 예제에서는 UUID(Universally Unique Identifier)를 생성합니다. 출력은 범용 고유 식별자를 나타내는 36자 문자열입니다.

```
SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

개인 정보 보호 관련 함수

AWS Clean Rooms 는 다음 사양에 대한 개인 정보 보호 관련 규정 준수를 지원하는 함수를 제공합니다.

- 글로벌 프라이버시 플랫폼(GPP) - 온라인 프라이버시 및 데이터 사용을 위한 글로벌 표준화된 프레임워크를 설정하는 대화형 광고국(IAB)의 사양입니다. GPP의 기술 사양에 대한 자세한 내용은 GitHub의 글로벌 프라이버시 플랫폼 설명서를 참조하세요.
- 투명성 및 동의 프레임워크(TCF) - 2020년에 출시된 GPP의 주요 구성 요소로, 기업이 EU 일반 데이터 보호 규정(GDPR)과 같은 개인 정보 보호 규정을 준수하도록 지원하는 표준화된 기술 프레임워크를 제공합니다. TCF를 사용하면 고객이 데이터 수집 및 처리에 대한 동의를 부여하거나 보류할 수 있습니다. TCF의 기술 사양에 대한 자세한 내용은 GitHub의 TCF 설명서를 참조하세요.

주제

- consent_gpp_v1_decode 함수
- consent_tcf_v2_decode 함수

consent_gpp_v1_decode 함수

이 consent_gpp_v1_decode 함수는 글로벌 프라이버시 플랫폼(GPP) v1 동의 데이터를 디코딩하는 데 사용됩니다. 인코딩된 동의 문자열을 입력으로 받아 사용자의 개인 정보 기본 설정 및 동의 선택에 대한 정보가 포함된 디코딩된 동의 데이터를 반환합니다. 이 함수는 GPP v1 동의 정보가 포함된 데이터로 작업할 때 유용한데, 이를 통해 구조화된 형식으로 동의 데이터에 액세스하고 분석할 수 있기 때문입니다.

구문

```
consent_gpp_v1_decode(gpp_string)
```

인수

gpp_string

인코딩된 GPP v1 동의 문자열입니다.

반환

반환된 사전에는 다음과 같은 키-값 페어가 포함됩니다.

- version: 사용된 GPP 사양의 버전(현재 1).
- cmpId: 동의 문자열을 인코딩한 동의 관리 플랫폼(CMP)의 ID입니다.
- cmpVersion: 동의 문자열을 인코딩한 CMP의 버전입니다.
- consentScreen: 사용자가 동의한 CMP UI의 화면 ID입니다.
- consentLanguage: 동의 정보의 언어 코드입니다.
- vendorListVersion: 사용된 공급업체 목록의 버전입니다.
- publisherCountryCode: 게시자의 국가 코드입니다.
- purposeConsent: 사용자가 동의한 목적을 나타내는 정수 목록입니다.
- purposeLegitimateInterest: 사용자의 합법적인 이해 관계가 투명하게 전달된 용도 IDs 목록입니다.
- specialFeatureOptIns: 사용자가 옵트인한 특수 기능을 나타내는 정수 목록입니다.
- vendorConsent: 사용자가 동의한 공급업체 IDs.
- vendorLegitimateInterest: 사용자의 합법적인 이해 관계가 투명하게 전달된 공급업체 IDs 목록입니다.

예제

다음 예제에서는 인코딩된 동의 문자열인 단일 인수를 사용합니다. 사용자의 개인 정보 기본 설정, 동의 선택 및 기타 메타데이터에 대한 정보를 포함하여 디코딩된 동의 데이터가 포함된 사전을 반환합니다.

```
SELECT * FROM consent_gpp_v1_decode('ABCDEFGHIJK');
```

반환된 동의 데이터의 기본 구조에는 동의 문자열 버전, CMP(Consent Management Platform) 세부 정보, 다양한 용도 및 공급업체에 대한 사용자의 동의 및 합법적인 관심 선택, 기타 메타데이터에 대한 정보가 포함됩니다.

```
{
    "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 함수

consent_tcf_v2_decode 함수는 TCP(투명성 및 동의 프레임워크) v2 동의 데이터를 디코딩하는 데 사용됩니다. 인코딩된 동의 문자열을 입력으로 받아 사용자의 개인 정보 기본 설정 및 동의 선택에 대한 정보가 포함된 디코딩된 동의 데이터를 반환합니다. 이 함수는 TCF v2 동의 정보가 포함된 데이터로 작업할 때 유용한데, 이를 통해 구조화된 형식으로 동의 데이터에 액세스하고 분석할 수 있기 때문입니다.

구문

```
consent_tcf_v2_decode(tcf_string)
```

인수

tcf_string

인코딩된 TCF v2 동의 문자열입니다.

반환

consent_tcf_v2_decode 함수는 TCP(Transparency and Consent Framework) v2 동의 문자열에서 디코딩된 동의 데이터가 포함된 사전을 반환합니다.

반환된 사전에는 다음과 같은 키-값 페어가 포함됩니다.

코어 세그먼트

- version: 사용된 TCF 사양의 버전(현재 2).
- created: 동의 문자열이 생성된 날짜와 시간입니다.
- lastUpdated: 동의 문자열이 마지막으로 업데이트된 날짜와 시간입니다.
- cmpId: 동의 문자열을 인코딩한 동의 관리 플랫폼(CMP)의 ID입니다.
- cmpVersion: 동의 문자열을 인코딩한 CMP의 버전입니다.
- consentScreen: 사용자가 동의한 CMP UI의 화면 ID입니다.
- consentLanguage: 동의 정보의 언어 코드입니다.
- vendorListVersion: 사용된 공급업체 목록의 버전입니다.
- tcfPolicyVersion: 동의 문자열이 기반으로 하는 TCF 정책의 버전입니다.
- isServiceSpecific: 동의가 특정 서비스에만 적용되는지 또는 모든 서비스에 적용되는지 여부를 나타내는 부울 값입니다.
- useNonStandardStacks: 비표준 스택을 사용할지 여부를 나타내는 부울 값입니다.
- specialFeatureOptIns: 사용자가 옵트인한 특수 기능을 나타내는 정수 목록입니다.
- purposeConsent: 사용자가 동의한 목적을 나타내는 정수 목록입니다.
- purposesLITransparency: 사용자가 합법적인 관심 투명성을 부여한 목적을 나타내는 정수 목록입니다.
- purposeOneTreatment: 사용자가 "한 가지 용도 처리"를 요청했는지 여부를 나타내는 부울 값입니다(즉, 모든 용도가 동일하게 처리됨).
- publisherCountryCode: 게시자의 국가 코드입니다.
- vendorConsent: 사용자가 동의한 공급업체 IDs.

- vendorLegitimateInterest: 사용자의 합법적인 이해 관계가 투명하게 전달된 공급업체 IDs 목록입니다.
- pubRestrictionEntry: 게시자 제한 목록입니다. 이 필드에는 해당 목적 제한에 따른 목적 ID, 제한 유형 및 공급업체 IDs 목록이 포함되어 있습니다.

공개된 공급업체 세그먼트

- disclosedVendors: 사용자에게 공개된 공급업체를 나타내는 정수 목록입니다.

게시자 목적 세그먼트

- pubPurposesConsent: 사용자가 동의한 게시자별 목적을 나타내는 정수 목록입니다.
- pubPurposesLITransparency: 사용자가 합법적인 관심 투명성을 부여한 게시자별 목적을 나타내는 정수 목록입니다.
- customPurposesConsent: 사용자가 동의한 사용자 지정 목적을 나타내는 정수 목록입니다.
- customPurposesLITransparency: 사용자가 합법적인 관심 투명성을 부여한 사용자 지정 목적을 나타내는 정수 목록입니다.

이 자세한 동의 데이터는 개인 데이터로 작업할 때 사용자의 개인 정보 기본 설정을 이해하고 준수하는 데 사용할 수 있습니다.

예제

다음 예제에서는 인코딩된 동의 문자열인 단일 인수를 사용합니다. 사용자의 개인 정보 기본 설정, 동의 선택 및 기타 메타데이터에 대한 정보를 포함하여 디코딩된 동의 데이터가 포함된 사전을 반환합니다.

```
from aws_clean_rooms.functions import consent_tcf_v2_decode

consent_string = "CO1234567890abcdef"
consent_data = consent_tcf_v2_decode(consent_string)

print(consent_data)
```

반환된 동의 데이터의 기본 구조에는 동의 문자열 버전, CMP(Consent Management Platform) 세부 정보, 다양한 용도 및 공급업체에 대한 사용자의 동의 및 합법적인 관심 선택, 기타 메타데이터에 대한 정보가 포함됩니다.

```
    /** 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],
};
```

윈도우 함수

창 함수를 사용하면 사용자가 분석 비즈니스 쿼리를 보다 효율적으로 생성할 수 있습니다. 창 함수는 결과 집합의 파티션, 즉 "창"에서 실행되어 해당 창에 속하는 모든 행에 대한 값을 반환합니다. 이와는 반대로 창이 없는 함수는 결과 집합의 모든 행에 대해 계산을 실행합니다. 그 밖에도 결과 행을 집계하는 그룹 함수와 달리 창 함수에서는 테이블 표현식의 모든 행이 그대로 유지됩니다.

반환 값은 해당 창에 속한 행 집합의 값을 사용하여 계산됩니다. 창은 테이블의 각 행마다 추가 속성을 계산하는 데 사용되는 행 집합을 정의합니다. 창은 창 명세(OVER 절)를 사용하여 정의되며, 다음과 같이 세 가지 주요 개념을 근거로 합니다.

- 창 파티션 - 행 그룹을 형성합니다(PARTITION 절).
- 창 순서 지정 - 각 파티션의 행 순서 또는 시퀀스를 정의합니다(ORDER BY 절).
- 창 프레임 - 행 집합을 제한하기 위해 각 행마다 정의됩니다(ROWS 명세).

창 함수는 최종 ORDER BY 절을 제외하고 쿼리에서 실행되는 마지막 연산 집합입니다. 창 함수를 처리할 때는 그 전에 모든 조인을 비롯한 WHERE, GROUP BY 및 HAVING 절까지 모두 완료됩니다. 따라서 창 함수는 선택 목록 또는 ORDER BY 절에만 나타날 수 있습니다. 다른 프레임 절이 있는 단일 쿼리 내에서 여러 윈도 함수를 사용할 수 있습니다. CASE 등의 다른 스칼라 표현식에서 윈도 함수를 사용할 수도 있습니다.

창 함수 구문 요약

Window 함수는 다음과 같은 표준 구문을 따릅니다.

```
function (expression) OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list [ frame_clause ] ] )
```

여기서 함수는 이 섹션에서 설명하는 함수 중 하나입니다.

expr_list는 다음과 같습니다.

```
expression | column_name [, expr_list ]
```

order_list는 다음과 같습니다.

```
expression | column_name [ ASC | DESC ]
[ NULLS FIRST | NULLS LAST ]
[, order_list ]
```

frame_clause는 다음과 같습니다.

```
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 }}
```

인수

함수

창 함수 자세한 내용은 각 함수에 대한 설명을 참조하세요.

OVER

창 명세를 정의하는 절입니다. OVER 절은 창 함수에서 필수 인수로서 창 함수와 다른 SQL 함수를 구분하는 역할을 합니다.

PARTITION BY expr_list

(옵션) PARTITION BY 절은 결과 집합을 여러 파티션으로 분할한다는 점에서 GROUP BY 절과 매우 유사합니다. 파티션 절이 존재하는 경우에는 함수가 각 파티션의 행에 대해 계산됩니다. 반대로 파티션 절을 지정하지 않으면 전체 테이블이 단일 파티션으로 구성되어 함수가 해당하는 전체 테이블에 대해서 계산됩니다.

DENSE_RANK, NTILE, RANK, ROW_NUMBER 같은 순위 함수에서는 결과 집합의 모든 행을 전역적으로 비교해야 합니다. 이때 PARTITION BY 절을 사용하면 쿼리 옵티마이저가 워크로드를 파티션에 따라 다수의 조각으로 분산시키기 때문에 각 집계를 병렬 방식으로 실행할 수 있습니다. PARTITION BY 절을 사용하지 않으면 단일 조각에서 직렬 방식으로 집계를 실행해야 하기 때문에 특히 대용량의 클러스터에서는 성능에 매우 부정적인 영향을 끼치게 됩니다.

AWS Clean Rooms는 PARTITION BY 절에서 문자열 리터럴을 지원하지 않습니다.

ORDER BY order_list

(옵션) 윈도 함수는 ORDER BY의 순서 명세에 따라 정렬된 각 파티션의 행에 적용됩니다. 이 ORDER BY 절은 frame_clause의 ORDER BY 절과 구분되어 전혀 관련이 없습니다. 이러한 ORDER BY 절은 PARTITION BY 절 없이도 사용할 수 있습니다.

순위 함수에서는 ORDER BY 절이 순위 값의 기준을 식별하는 역할을 합니다. 집계 함수에서는 각 프레임에 대한 집계 함수 계산 이전에 파티션 행의 순서를 지정해야 합니다. 윈도 함수 형식에 대한 자세한 내용은 윈도우 함수 섹션을 참조하세요.

order list에는 열 식별자, 또는 열 식별자로 평가되는 표현식이 필요합니다. 열 이름 대신에 상수나 상수 표현식을 사용할 수도 없습니다.

NULLS 값은 자체 그룹으로 처리되어 NULLS FIRST 또는 NULLS LAST 옵션에 따라 정렬 후 순위가 결정됩니다. 기본적으로 NULL 값은 ASC 순서에서는 마지막에 정렬 후 순위가 결정되며, DESC 순서에서는 처음에 정렬 후 순위가 결정됩니다.

AWS Clean Rooms는 ORDER BY 절에서 문자열 리터럴을 지원하지 않습니다.

ORDER BY 절을 생략하면 행의 순서는 비확정적입니다.

> **Note**
>
> 와 같은 병렬 시스템에서 ORDER BY 절이 데이터의 고유하고 완전한 순서를 생성하지 않는 AWS Clean Rooms경우 행의 순서는 비결정적입니다. 즉, ORDER BY 표현식이 중복 값(부분 순서 지정)을 생성하는 경우 해당 행의 반환 순서는의 실행마다 다를 수 있습니다 AWS Clean Rooms. 그러면 창 함수 역시 예상하지 못하거나 일관적이지 못한 결과를 반환하게 됩니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

column_name

파티션으로 분할하거나 순서를 지정할 때 기준이 되는 열의 이름입니다.

ASC | DESC

표현식의 정렬 순서를 정의하는 옵션으로서 각각 다음과 같은 의미를 갖습니다.

- ASC: 오름차순(예: 숫자 값의 경우 낮은 값에서 높은 값 순, 문자열의 경우 'A'에서 'Z'의 순. 지정된 옵션이 없는 경우에는 데이터가 기본적으로 오름차순으로 정렬됩니다.
- DESC: 내림차순(숫자 값의 경우 높은 값에서 낮은 값 순, 문자열의 경우 'Z'에서 'A'의 순).

NULLS FIRST | NULLS LAST

NULLS의 순서를 NULL 값 이외의 값 이전에 결정할지, 혹은 이후에 결정할지 지정하는 옵션입니다. 기본적으로 ASC 순서에서는 마지막에 정렬 후 순위가 결정되며, DESC 순서에서는 처음에 정렬 후 순위가 결정됩니다.

frame_clause

집계 함수에서 프레임 절은 ORDER BY를 사용하여 함수의 창에 포함되는 행 집합을 추가적으로 정제하는 역할을 합니다. 이를 통해 순서가 지정된 결과 내에 행 집합을 추가하거나 제거할 수 있습니다. ROWS 키워드와 관련 지정자로 구성됩니다.

프레임 절은 순위 함수에 적용되지 않습니다. 또한 집계 함수의 OVER 절에 ORDER BY 절이 사용되지 않는 경우 프레임 절이 필요하지 않습니다. 집계 함수에서 ORDER BY 절이 사용되면 명시적인 프레임 절이 필요합니다.

ORDER BY 절을 지정하지 않으면 묵시적 프레임이 무제한이기 때문에 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING과 다름 없습니다.

ROWS

이 절은 현재 행에서 물리적 오프셋을 지정하여 창 프레임을 정의합니다.

이 절은 현재 창 또는 파티션에서 현재 행의 값이 결합되는 행을 지정합니다. 행의 위치는 인수를 사용하여 지정하며, 현재 행 앞 또는 뒤가 될 수 있습니다. 모든 창 프레임에서 기준점은 현재 행입니다. 각 행은 창 프레임이 파티션에서 밀려 앞으로 이동하면서 번갈아 현재 행이 됩니다.

프레임은 다음과 같이 현재 행까지 포함하여 단일 행 집합이 되거나,

```
{UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW}
```

혹은 다음과 같이 두 경계 사이의 행 집합이 될 수도 있습니다.

```
BETWEEN
{ UNBOUNDED PRECEDING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
AND
{ UNBOUNDED FOLLOWING | offset { PRECEDING | FOLLOWING } | CURRENT ROW }
```

UNBOUNDED PRECEDING은 파티션의 첫 행에서 창이 시작된다는 것을 나타내고, offset PRECEDING은 오프셋 값에 해당하는 행의 수만큼 현재 행 앞에서 창이 시작된다는 것을 나타냅니다. 기본값은 UNBOUNDED PRECEDING입니다.

CURRENT ROW는 창이 현재 행에서 시작하거나 끝난다는 것을 나타냅니다.

UNBOUNDED FOLLOWING은 파티션의 마지막 행에서 창이 끝나는 것을 나타내고, offset FOLLOWING은 오프셋 값에 해당하는 행의 수만큼 현재 행 뒤에서 창이 끝난다는 것을 나타냅니다.

offset은 현재 행 앞 또는 뒤로 물리적인 행의 수를 의미합니다. 이 경우에는 offset이 양의 숫자 값으로 평가되는 상수여야 합니다. 예를 들어 5 FOLLOWING일 때는 현재 행 뒤로 5개 행을 지나 프레임이 종료됩니다.

BETWEEN을 지정하지 않으면 묵시적이지만 프레임 경계가 현재 행으로 결정됩니다. 예를 들어 `ROWS 5 PRECEDING`은 `ROWS BETWEEN 5 PRECEDING AND CURRENT ROW`와 같습니다.

또한 ROWS UNBOUNDED FOLLOWING은 ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING과 같습니다.

> **Note**
>
> 시작 경계가 종료 경계보다 크게 프레임을 지정할 수는 없습니다. 예를 들어 다음과 같은 프레임은 지정할 수 없습니다.
>
> ```
> between 5 following and 5 preceding
> between current row and 2 preceding
> between 3 following and current row
> ```

창 함수 데이터에 대한 고유 순서 지정

윈도 함수의 ORDER BY 절이 데이터의 전체 순서를 고유하게 지정하지 않으면 행의 순서는 비확정적입니다. 다시 말해 ORDER BY 표현식에서 중복 값이 산출되면(부분 순서 지정) 여러 차례 실행할 때마다 해당 행의 반환 순서가 달라질 수 있습니다. 이 경우 윈도 함수 역시 예상하지 못하거나 일관적이지 못한 결과를 반환하게 됩니다.

예를 들어 다음 쿼리는 여러 실행에 대해 다른 결과를 반환합니다. 이러한 다른 결과는 order by dateid가 SUM 윈도 함수 데이터의 고유한 순서를 생성하지 않기 때문에 발생합니다.

```
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
...
```

이 경우에는 두 번째 ORDER BY 열을 윈도 함수에 추가하여 문제를 해결할 수 있습니다.

```
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
...
```

지원되는 함수

AWS Clean RoomsSpark SQL은 집계 및 순위라는 두 가지 유형의 창 함수를 지원합니다.

다음은 지원되는 집계 함수입니다.

- CUME_DIST 창 함수
- DENSE_RANK 창 함수
- FIRST 창 함수
- FIRST_VALUE 창 함수
- LAG 창 함수
- LAST 창 함수
- LAST_VALUE 창 함수
- LEAD 창 함수

다음은 지원되는 순위 함수입니다.

- DENSE_RANK 창 함수
- PERCENT_RANK 창 함수

- RANK 창 함수
- ROW_NUMBER 창 함수

창 함수 예제를 위한 샘플 테이블

각 함수 설명과 함께 특정 창 함수 예제를 찾을 수도 있습니다. 일부 예는 다음 테이블과 같이 11개의 행이 포함된 WINSALES라는 테이블을 사용합니다.

SALESID	DATEID	SELLERID	BUYERID	QTY	QTY_SHIPPED
30001	30001	3	B	10	10
10001	10001	1	C	10	10
10005	10005	1	A	30	
40001	40001	4	A	40	
10006	10006	1	C	10	
20001	20001	2	B	20	20
40005	40005	4	A	10	10
20002	20002	2	C	20	20
30003	30003	3	B	15	
30004	30004	3	B	20	
30007	30007	3	C	30	

CUME_DIST 창 함수

창 또는 파티션에 속하는 값의 누적 분포를 계산합니다. 오름차순을 가정했을 때 누적 분포는 다음과 같은 공식으로 결정됩니다.

```
count of rows with values <= x / count of rows in the window or partition
```

여기에서 x는 ORDER BY 절에서 지정하는 열의 현재 행 값과 동일합니다. 다음은 위와 같은 공식의 사용을 나타내는 데이터 세트입니다.

```
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
```

반환 값의 범위는 0부터 1까지입니다(1 포함).

구문

```
CUME_DIST ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

인수

OVER

창 파티션을 지정하는 절입니다. OVER 절에는 창 프레임 명세가 포함될 수 없습니다.

PARTITION BY partition_expression

선택 사항. OVER 절에서 각 그룹의 레코드 범위를 설정하는 표현식입니다.

ORDER BY order_list

누적 분포를 계산하기 위한 표현식입니다. 이 표현식은 숫자 데이터 형식을 갖거나, 혹은 묵시적으로 1로 변환될 수 있어야 합니다. 즉 ORDER BY가 생략되면 모든 행의 반환 값은 1입니다.

ORDER BY에서 고유한 순서를 지정하지 않으면 행의 순서는 비확정적입니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

반환 타입

FLOAT8

예제

다음은 각 판매자의 수량 누적 분포를 계산하는 예입니다.

```
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
```

요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

DENSE_RANK 창 함수

DENSE_RANK 창 함수는 OVER 절의 ORDER BY 표현식을 기준으로 값 그룹에 속한 값의 순위를 결정합니다. PARTITION BY 절(옵션)이 존재하면 각 행 그룹의 순위가 재설정됩니다. 순위 기준 값이 같은 행은 순위도 동일하게 결정됩니다. DENSE_RANK 함수는 한 가지 측면에서 RANK와 다릅니다. 즉 2개 이상의 행에서 순위가 동일하면 순위 값의 순서에서도 빈 자리가 없습니다. 예를 들어 두 행의 순위가 1로 결정되면 다음 순위는 2입니다.

순위 함수에서는 동일한 쿼리라고 해도 PARTITION BY 절과 ORDER BY 절을 다르게 사용할 수 있습니다.

구문

```
DENSE_RANK () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
```

```
)
```

인수

( )

함수에 인수가 없지만 빈 괄호가 필요합니다.

OVER

DENSE_RANK 함수의 창 절입니다.

PARTITION BY expr_list

선택 사항. 창을 정의하는 하나 이상의 표현식입니다.

ORDER BY order_list

선택 사항. 순위 값의 기준이 되는 표현식입니다. PARTITION BY를 지정하지 않으면 ORDER BY가 전체 테이블을 사용합니다. 즉 ORDER BY가 생략되면 모든 행의 반환 값은 1입니다.

ORDER BY에서 고유한 순서를 지정하지 않으면 행의 순서는 비확정적입니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

반환 타입

INTEGER

예제

다음 예에서는 판매 수량(내림차순)으로 테이블을 정렬하고 각 행에 밀집 순위와 정규 순위를 모두 할당합니다. 결과는 창 함수 결과를 적용한 후에 정렬됩니다.

```
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)
```

동일한 쿼리에서 DENSE_RANK와 RANK 함수를 함께 사용하여 같은 행 집합에 할당되는 순위의 차이를 기록합니다. 요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

다음 예에서는 SELLERID를 기준으로 테이블을 분할하여 수량에 따라 각 파티션의 순서(내림차순)를 지정한 후 밀집 순위를 각 행에 할당합니다. 결과는 창 함수 결과를 적용한 후에 정렬됩니다.

```
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)
```

요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

FIRST 창 함수

정렬된 행 집합이 주어지면 FIRST는 창 프레임의 첫 번째 행을 기준으로 지정된 표현식의 값을 반환합니다.

프레임의 마지막 행 선택에 대한 자세한 내용은 LAST 창 함수 섹션을 참조하세요.

구문

```
FIRST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

인수

expression

함수가 실행되는 대상 열 또는 표현식입니다.

IGNORE NULLS

이 옵션을 FIRST와 함께 사용하면 함수는 프레임에서 NULL이 아닌 첫 번째 값(또는 모든 값이 NULL인 경우 NULL)을 반환합니다.

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

함수에서 창 절을 삽입합니다.

PARTITION BY expr_list

하나 이상의 표현식과 관련하여 함수의 창을 정의합니다.

ORDER BY order_list

각 파티션의 행을 정렬합니다. PARTITION BY 절을 지정하지 않으면 ORDER BY가 전체 테이블을 정렬합니다. ORDER BY 절을 지정하면 frame_clause 역시 지정해야 합니다.

FIRST 함수의 결과는 데이터 순서에 따라 달라집니다. 다음과 같은 경우 함수 결과는 비확정적입니다.

- ORDER BY 절이 지정되지 않고 파티션에 다른 표현식 값 2개가 포함된 경우
- 표현식이 ORDER BY 목록에서는 동일한 값이지만 다른 값으로 평가되는 경우

frame_clause

집계 함수에서 ORDER BY 절이 사용되면 명시적인 프레임 절이 필요합니다. 프레임 절은 순서가 지정된 결과에 행 집합을 추가하거나 제거함으로써 함수의 창에 포함되는 행 집합을 정제하는 역할을 하며, ROWS 키워드와 관련 지정자로 구성됩니다. 창 함수 구문 요약을(를) 참조하세요.

반환 타입

이러한 함수는 기본 AWS Clean Rooms데이터 형식을 사용하는 표현식을 지원합니다. 반환 형식은 expression 데이터 형식과 동일합니다.

예제

다음은 VENUE 테이블에서 각 장소의 좌석 수용 능력을 반환하는 예로서 함수 결과의 순서(내림차순)는 좌석 수용 능력에 따라 지정됩니다. FIRST 함수는 프레임의 첫 번째 행에 해당하는 장소의 이름을 선택하는 데 사용됩니다.이 경우 좌석 수가 가장 많은 행입니다. 결과가 주를 기준으로 분할되어 있으므로 VENUESTATE 값이 바뀌면 첫 번째 값도 새롭게 선택됩니다. 여기에서는 창 프레임의 경계가 없기 때문에 각 파티션의 행마다 선택되는 첫 번째 값이 동일합니다.

California를 예로 들면, `Qualcomm Stadium`의 좌석 수(`70561`)가 가장 높기 때문에 이 장소의 이름이 `CA` 파티션의 모든 행에 대한 첫 번째 값에 해당합니다.

```
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 창 함수

행 집합의 순서가 지정되었다고 가정할 때 FIRST VALUE 함수는 창 프레임의 첫 번째 행과 관련하여 지정된 표현식의 값을 반환합니다.

프레임의 마지막 행 선택에 대한 자세한 내용은 LAST_VALUE 창 함수 섹션을 참조하세요.

구문

```
FIRST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

인수

expression

함수가 실행되는 대상 열 또는 표현식입니다.

IGNORE NULLS

FIRST_VALUE에서 이 옵션을 사용하면 프레임에서 NULL이 아닌 첫 번째 값을 반환합니다(또는 모든 값이 NULL이면 NULL을 반환합니다).

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

함수에서 창 절을 삽입합니다.

PARTITION BY expr_list

하나 이상의 표현식과 관련하여 함수의 창을 정의합니다.

ORDER BY order_list

각 파티션의 행을 정렬합니다. PARTITION BY 절을 지정하지 않으면 ORDER BY가 전체 테이블을 정렬합니다. ORDER BY 절을 지정하면 frame_clause 역시 지정해야 합니다.

FIRST_VALUE 함수의 결과는 데이터 순서에 따라 결정됩니다. 다음과 같은 경우 함수 결과는 비확정적입니다.

- ORDER BY 절이 지정되지 않고 파티션에 다른 표현식 값 2개가 포함된 경우
- 표현식이 ORDER BY 목록에서는 동일한 값이지만 다른 값으로 평가되는 경우

frame_clause

집계 함수에서 ORDER BY 절이 사용되면 명시적인 프레임 절이 필요합니다. 프레임 절은 순서가 지정된 결과에 행 집합을 추가하거나 제거함으로써 함수의 창에 포함되는 행 집합을 정제하는 역할을 하며, ROWS 키워드와 관련 지정자로 구성됩니다. 창 함수 구문 요약을(를) 참조하세요.

반환 타입

이러한 함수는 기본 AWS Clean Rooms데이터 형식을 사용하는 표현식을 지원합니다. 반환 형식은 expression 데이터 형식과 동일합니다.

예제

다음은 VENUE 테이블에서 각 장소의 좌석 수용 능력을 반환하는 예로서 함수 결과의 순서(내림차순)는 좌석 수용 능력에 따라 지정됩니다. FIRST_VALUE 함수는 프레임에서 첫 번째 행에 해당하는 장소의 이름을 선택할 때 사용됩니다. 이 경우에는 좌석 수가 가장 많은 행이 여기에 해당합니다. 결과가 주를 기준으로 분할되어 있으므로 VENUESTATE 값이 바뀌면 첫 번째 값도 새롭게 선택됩니다. 여기에서는 창 프레임의 경계가 없기 때문에 각 파티션의 행마다 선택되는 첫 번째 값이 동일합니다.

California를 예로 들면, `Qualcomm Stadium`의 좌석 수(`70561`)가 가장 높기 때문에 이 장소의 이름이 `CA` 파티션의 모든 행에 대한 첫 번째 값에 해당합니다.

```
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 창 함수

LAG 창 함수는 파티션에서 현재 행 위(앞)의 지정 오프셋에 위치한 행의 값을 반환합니다.

구문

```
LAG (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

인수

value_expr

함수가 실행되는 대상 열 또는 표현식입니다.

Offset

현재 행 앞으로 값을 반환할 행이 위치한 수를 지정하는 파라미터(옵션)입니다. 이 오프셋은 상수 정수 혹은 정수로 평가되는 표현식이 될 수 있습니다. 오프셋을 지정하지 않으면를 기본값1으로 AWS Clean Rooms사용합니다. 오프셋이 0이면 현재 행을 나타냅니다.

IGNORE NULLS

가 사용할 행을 결정할 때 null 값을 건너뛰어야 함을 나타내는 선택적 사양AWS Clean Rooms입니다. IGNORE NULLS를 지정하지 않으면 NULL 값이 포함됩니다.

> **Note**
>
> NVL 또는 COALESCE 표현식을 사용하여 NULL 값을 다른 값으로 변경할 수도 있습니다.

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

창 파티션 및 순서를 지정합니다. OVER 절에는 창 프레임 명세가 포함될 수 없습니다.

PARTITION BY window_partition

OVER 절에서 각 그룹의 레코드 범위를 설정하는 인수(옵션)입니다.

ORDER BY window_ordering

각 파티션의 행을 정렬합니다.

LAG 창 함수는 모든 AWS Clean Rooms데이터 유형을 사용하는 표현식을 지원합니다. 반환 형식은 value_expr 형식과 동일합니다.

예제

다음은 구매자 ID가 3인 구매자에게 팔린 티켓 수량과 구매자 3이 티켓을 구입한 시간을 나타내는 예입니다. 쿼리가 구매자 3의 각 판매 수량을 이전 판매 수량과 비교할 수 있도록 각 판매 수량에 대한 이전 판매 수량을 반환합니다. 2008년 1월 16일 이전에는 구매 기록이 없기 때문에 이전 판매 수량의 첫 번째 값은 NULL입니다.

```
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 창 함수

정렬된 행 집합이 주어지면 LAST 함수는 프레임의 마지막 행을 기준으로 표현식 값을 반환합니다.

프레임의 첫 번째 행 선택에 대한 자세한 내용은 FIRST 창 함수 섹션을 참조하세요.

구문

```
LAST( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

인수

expression

함수가 실행되는 대상 열 또는 표현식입니다.

IGNORE NULLS

프레임에서 NULL이 아닌 마지막 값을 반환합니다(또는 모든 값이 NULL이면 NULL을 반환합니다).

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

함수에서 창 절을 삽입합니다.

PARTITION BY expr_list

하나 이상의 표현식과 관련하여 함수의 창을 정의합니다.

ORDER BY order_list

각 파티션의 행을 정렬합니다. PARTITION BY 절을 지정하지 않으면 ORDER BY가 전체 테이블을 정렬합니다. ORDER BY 절을 지정하면 frame_clause 역시 지정해야 합니다.

결과는 데이터 순서에 따라 달라집니다. 다음과 같은 경우 함수 결과는 비확정적입니다.

- ORDER BY 절이 지정되지 않고 파티션에 다른 표현식 값 2개가 포함된 경우
- 표현식이 ORDER BY 목록에서는 동일한 값이지만 다른 값으로 평가되는 경우

frame_clause

집계 함수에서 ORDER BY 절이 사용되면 명시적인 프레임 절이 필요합니다. 프레임 절은 순서가 지정된 결과에 행 집합을 추가하거나 제거함으로써 함수의 창에 포함되는 행 집합을 정제하는 역할을 하며, ROWS 키워드와 관련 지정자로 구성됩니다. 창 함수 구문 요약을(를) 참조하세요.

반환 타입

이러한 함수는 기본 AWS Clean Rooms데이터 형식을 사용하는 표현식을 지원합니다. 반환 형식은 expression 데이터 형식과 동일합니다.

예제

다음은 VENUE 테이블에서 각 장소의 좌석 수용 능력을 반환하는 예로서 함수 결과의 순서(내림차순)는 좌석 수용 능력에 따라 지정됩니다. LAST 함수는 프레임의 마지막 행에 해당하는 장소의 이름을 선택하는 데 사용됩니다.이 경우 좌석 수가 가장 적은 행입니다. 결과가 주를 기준으로 분할되어 있으므로 VENUESTATE 값이 바뀌면 마지막 값도 새롭게 선택됩니다. 여기에서는 창 프레임의 경계가 없기 때문에 각 파티션의 행마다 선택되는 마지막 값이 동일합니다.

California를 보면, 파티션의 모든 행에 대해서 좌석 수(`Shoreline Amphitheatre`)가 가장 적은 `22000`가 반환됩니다.

```
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 창 함수

행 집합의 순서가 지정되었다고 가정할 때 LAST VALUE 함수는 프레임의 마지막 행과 관련하여 표현식의 값을 반환합니다.

프레임의 첫 번째 행 선택에 대한 자세한 내용은 FIRST_VALUE 창 함수 섹션을 참조하세요.

구문

```
LAST_VALUE( expression )[ IGNORE NULLS | RESPECT NULLS ]
OVER (
[ PARTITION BY expr_list ]
[ ORDER BY order_list frame_clause ]
)
```

인수

expression

함수가 실행되는 대상 열 또는 표현식입니다.

IGNORE NULLS

프레임에서 NULL이 아닌 마지막 값을 반환합니다(또는 모든 값이 NULL이면 NULL을 반환합니다).

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

함수에서 창 절을 삽입합니다.

PARTITION BY expr_list

하나 이상의 표현식과 관련하여 함수의 창을 정의합니다.

ORDER BY order_list

각 파티션의 행을 정렬합니다. PARTITION BY 절을 지정하지 않으면 ORDER BY가 전체 테이블을 정렬합니다. ORDER BY 절을 지정하면 frame_clause 역시 지정해야 합니다.

결과는 데이터 순서에 따라 달라집니다. 다음과 같은 경우 함수 결과는 비확정적입니다.

- ORDER BY 절이 지정되지 않고 파티션에 다른 표현식 값 2개가 포함된 경우
- 표현식이 ORDER BY 목록에서는 동일한 값이지만 다른 값으로 평가되는 경우

frame_clause

집계 함수에서 ORDER BY 절이 사용되면 명시적인 프레임 절이 필요합니다. 프레임 절은 순서가 지정된 결과에 행 집합을 추가하거나 제거함으로써 함수의 창에 포함되는 행 집합을 정제하는 역할을 하며, ROWS 키워드와 관련 지정자로 구성됩니다. 창 함수 구문 요약을(를) 참조하세요.

반환 타입

이러한 함수는 기본 AWS Clean Rooms데이터 형식을 사용하는 표현식을 지원합니다. 반환 형식은 expression 데이터 형식과 동일합니다.

예제

다음은 VENUE 테이블에서 각 장소의 좌석 수용 능력을 반환하는 예로서 함수 결과의 순서(내림차순)는 좌석 수용 능력에 따라 지정됩니다. LAST_VALUE 함수는 프레임에서 마지막 행에 해당하는 장소의 이름을 선택할 때 사용됩니다. 이 경우에는 좌석 수가 가장 적은 행이 여기에 해당합니다. 결과가 주를

기준으로 분할되어 있으므로 VENUESTATE 값이 바뀌면 마지막 값도 새롭게 선택됩니다. 여기에서는 창 프레임의 경계가 없기 때문에 각 파티션의 행마다 선택되는 마지막 값이 동일합니다.

California를 보면, 파티션의 모든 행에 대해서 좌석 수(Shoreline Amphitheatre)가 가장 적은 22000가 반환됩니다.

```
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 창 함수

LEAD 창 함수는 파티션에서 현재 행 아래(뒤)의 지정 오프셋에 위치한 행의 값을 반환합니다.

구문

```
LEAD (value_expr [, offset ])
[ IGNORE NULLS | RESPECT NULLS ]
```

```
OVER ( [ PARTITION BY window_partition ] ORDER BY window_ordering )
```

인수

value_expr

함수가 실행되는 대상 열 또는 표현식입니다.

Offset

현재 행 아래로 값을 반환할 행이 위치한 수를 지정하는 파라미터(옵션)입니다. 이 오프셋은 상수 정수 혹은 정수로 평가되는 표현식이 될 수 있습니다. 오프셋을 지정하지 않으면를 기본값1으로 AWS Clean Rooms사용합니다. 오프셋이 0이면 현재 행을 나타냅니다.

IGNORE NULLS

가 사용할 행을 결정할 때 null 값을 건너뛰어야 함을 나타내는 선택적 사양AWS Clean Rooms입니다. IGNORE NULLS를 지정하지 않으면 NULL 값이 포함됩니다.

> **Note**
>
> NVL 또는 COALESCE 표현식을 사용하여 NULL 값을 다른 값으로 변경할 수도 있습니다.

RESPECT NULLS

AWS Clean Rooms가 사용할 행을 결정할 때 null 값을 포함해야 함을 나타냅니다. RESPECT NULLS는 IGNORE NULLS를 지정하지 않은 경우 기본적으로 지원됩니다.

OVER

창 파티션 및 순서를 지정합니다. OVER 절에는 창 프레임 명세가 포함될 수 없습니다.

PARTITION BY window_partition

OVER 절에서 각 그룹의 레코드 범위를 설정하는 인수(옵션)입니다.

ORDER BY window_ordering

각 파티션의 행을 정렬합니다.

LEAD 창 함수는 모든 AWS Clean Rooms데이터 유형을 사용하는 표현식을 지원합니다. 반환 형식은 value_expr 형식과 동일합니다.

예제

다음은 SALES 테이블의 이벤트에 대해 2008년 1월 1일과 동년 1월 2일에 판매된 티켓 수수료와 후속 티켓 판매 수수료를 나타낸 예입니다.

```
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 창 함수

임의의 행의 백분율 순위를 계산합니다. 백분율 순위를 구하는 공식은 다음과 같습니다.

```
(x - 1) / (the number of rows in the window or partition - 1)
```

여기에서 x는 현재 행의 순위입니다. 다음은 위와 같은 공식의 사용을 나타내는 데이터 세트입니다.

```
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
```

반환 값의 범위는 0부터 1까지입니다(0과 1 포함). 모든 집합에서 첫 번째 행은 PERCENT_RANK가 0입니다.

구문

```
PERCENT_RANK ()
OVER (
[ PARTITION BY partition_expression ]
[ ORDER BY order_list ]
)
```

인수

( )

함수에 인수가 없지만 빈 괄호가 필요합니다.

OVER

창 파티션을 지정하는 절입니다. OVER 절에는 창 프레임 명세가 포함될 수 없습니다.

PARTITION BY partition_expression

선택 사항. OVER 절에서 각 그룹의 레코드 범위를 설정하는 표현식입니다.

ORDER BY order_list

선택 사항. 백분율 순위를 계산하기 위한 표현식입니다. 이 표현식은 숫자 데이터 형식을 갖거나, 혹은 묵시적으로 1로 변환될 수 있어야 합니다. 즉 ORDER BY가 생략되면 모든 행의 반환 값은 0입니다.

ORDER BY에서 고유한 순서를 지정하지 않으면 행의 순서는 비확정적입니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

반환 타입

FLOAT8

예제

다음은 각 판매자의 판매 수량에 대한 백분율 순위를 계산하는 예입니다.

```
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
```

요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

RANK 창 함수

RANK 창 함수는 OVER 절의 ORDER BY 표현식을 기준으로 값 그룹에 속한 값의 순위를 결정합니다. PARTITION BY 절(옵션)이 존재하면 각 행 그룹의 순위가 재설정됩니다. 순위 기준 값이 같은 행은 동일한 순위를 받습니다.는 묶인 행 수를 묶인 순위에 AWS Clean Rooms추가하여 다음 순위를 계산하므로 순위가 연속된 숫자가 아닐 수 있습니다. 예를 들어 두 행의 순위가 1로 결정되면 다음 순위는 3입니다.

RANK는 한 가지 측면에서 DENSE_RANK 창 함수와 다릅니다. 즉 DENSE_RANK에서는 2개 이상의 행에서 순위가 동일하면 순위 값의 순서에서도 빈 자리가 없습니다. 예를 들어 두 행의 순위가 1로 결정되면 다음 순위는 2입니다.

순위 함수에서는 동일한 쿼리라고 해도 PARTITION BY 절과 ORDER BY 절을 다르게 사용할 수 있습니다.

구문

```
RANK () OVER
```

```
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

인수

( )

함수에 인수가 없지만 빈 괄호가 필요합니다.

OVER

RANK 함수의 창 절입니다.

PARTITION BY expr_list

선택 사항. 창을 정의하는 하나 이상의 표현식입니다.

ORDER BY order_list

선택 사항. 순위 값의 기준이 되는 열을 정의합니다. PARTITION BY를 지정하지 않으면 ORDER BY가 전체 테이블을 사용합니다. 즉 ORDER BY가 생략되면 모든 행의 반환 값은 1입니다.

ORDER BY에서 고유한 순서를 지정하지 않으면 행의 순서는 비확정적입니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

반환 타입

INTEGER

예제

다음 예에서는 판매 수량에 따라 테이블의 순서(기본 오름차순)를 지정한 후 각 행마다 순위를 할당합니다. 순위 값 1은 가장 높은 순위의 값입니다. 결과는 창 함수 결과를 적용한 후에 정렬됩니다:

```
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)
```

이 예제의 외부 ORDER BY 절에는이 쿼리가 실행될 때마다가 일관되게 정렬된 결과를 AWS Clean Rooms반환하도록 하는 2열과 1열이 포함되어 있습니다. 예를 들어 판매 ID가 10001과 10006인 행은 QTY 및 RNK 값이 동일합니다. 이때 열 1에 따라 최종 결과 집합의 순서를 지정하면 10001 행이 항상 10006 행보다 앞에 위치할 수 있습니다. 요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

다음 예에서는 창 함수(`order by qty desc`)의 순서가 반전됩니다. 여기에서는 최고 순위 값이 가장 큰 QTY 값에 적용됩니다.

```
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)
```

요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

다음 예에서는 SELLERID를 기준으로 테이블을 분할하여 수량에 따라 각 파티션의 순서(내림차순)를 지정한 후 순위를 각 행에 할당합니다. 결과는 창 함수 결과를 적용한 후에 정렬됩니다.

```
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 창 함수

OVER 절의 ORDER BY 표현식을 기준으로 행 그룹 내에서 1부터 현재 행의 서수를 결정합니다. PARTITION BY 절(옵션)이 존재하면 각 행 그룹의 서수가 재설정됩니다. ORDER BY 표현식 값이 동일한 행이라고 해도 비확정적으로 다른 행 번호를 받습니다.

구문

```
ROW_NUMBER () OVER
(
[ PARTITION BY expr_list ]
[ ORDER BY order_list ]
)
```

인수

( )

함수에 인수가 없지만 빈 괄호가 필요합니다.

OVER

ROW_NUMBER 함수의 창 절입니다.

PARTITION BY expr_list

선택 사항. ROW_NUMBER 함수를 정의하는 하나 이상의 표현식입니다.

ORDER BY order_list

선택 사항. 행 번호의 기준이 되는 열을 정의한 표현식입니다. PARTITION BY를 지정하지 않으면 ORDER BY가 전체 테이블을 사용합니다.

ORDER BY가 고유한 순서를 지정하지 않거나 생략되면 행의 순서는 비확정적입니다. 자세한 내용은 창 함수 데이터에 대한 고유 순서 지정 단원을 참조하십시오.

반환 타입

BIGINT

예제

다음은 SELLERID를 기준으로 테이블을 분할하여 수량에 따라 각 파티션의 순서(오름차순)를 지정한 후 행 번호를 각 행에 할당하는 예입니다. 결과는 창 함수 결과를 적용한 후에 정렬됩니다.

```
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)
```

요청 데이터에 대한 설명은 창 함수 예제를 위한 샘플 테이블 섹션을 참조하세요.

AWS Clean Rooms Spark SQL 조건

조건은 1개 이상의 표현식과 논리 연산자로 이루어진 문장으로, true, false 또는 unknown으로 평가됩니다. 조건은 종종 조건자로 불리기도 합니다.

구문

```
comparison_condition
| logical_condition
| range_condition
| pattern_matching_condition
| null_condition
| EXISTS_condition
| IN_condition
```

> **Note**
>
> 모든 문자열 비교와 LIKE 패턴 일치는 대/소문자를 구분합니다. 예를 들어 'A'와 'a'는 일치하지 않습니다. 하지만 ILIKE 조건자를 사용하면 패턴 일치에서 대/소문자를 구분하지 않을 수도 있습니다.

AWS Clean Rooms Spark SQL에서는 다음 SQL 조건이 지원됩니다.

주제

- 비교 연산자
- 논리 조건
- 패턴 일치 조건
- BETWEEN 범위 조건

- NULL 조건
- EXISTS 조건
- IN 조건

비교 연산자

비교 조건이란 두 값의 논리적 관계를 말합니다. 모든 비교 조건은 반환 형식이 부울인 이진 연산자입니다.

AWS Clean Rooms Spark SQL은 다음 표에 설명된 비교 연산자를 지원합니다.

연산자	구문	설명
!	`!expression`	논리적 NOT 연산자입니다. 부울 표현식을 무효화하는 데 사용됩니다. 즉, 표현식 값의 반대를 반환합니다. ! 연산자를 AND 및 OR과 같은 다른 논리 연산자와 결합하여 더 복잡한 부울 표현식을 생성할 수도 있습니다.
<	`a < b`	보다 작음 비교 연산자입니다. 두 값을 비교하고 왼쪽의 값이 오른쪽의 값보다 작은지 확인하는 데 사용됩니다.
>	`a > b`	보다 큼 비교 연산자입니다. 두 값을 비교하고 왼쪽의 값이 오른쪽의 값보다 큰지 확인하는 데 사용됩니다.
<=	`a <= b`	보다 작거나 같은 비교 연산자입니다. 두 값을 비교하는 데 사용되며 왼쪽의 값이 오른쪽의 값보다 작거나 같`true`으면를

연산자	구문	설명
		반환하고, false 그렇지 않으면를 반환합니다.
>=	a >= b	보다 크거나 같은 비교 연산자입니다. 두 값을 비교하고 왼쪽의 값이 오른쪽의 값보다 크거나 같은지 확인하는 데 사용됩니다.
=	a = b	두 값을 비교하고 값이 같true으면를 반환하는 등식 비교 연산자이며, false 그렇지 않으면를 반환합니다.
<> 또는 !=	a <> b 또는 a != b	비교 연산자와 같지 않음 - 두 값을 비교하고 값이 같지 않으면를 반환하고 그렇지 false 않으면 true를 반환합니다.

연산자	구문	설명
==	a == b	두 값을 비교하고 값이 같`true`으면를 반환하는 표준 등식 비교 연산자입니다`false`. **Note** == 연산자는 문자열 값을 비교할 때 대/소문자를 구분합니다. 대/소문자를 구분하지 않는 비교를 수행해야 하는 경우 비교 전에 UPPER() 또는 LOWER()와 같은 함수를 사용하여 값을 동일한 대/소문자로 변환할 수 있습니다.

예시

다음은 몇 가지 간단한 비교 조건 예입니다.

```
a = 5
a < b
min(x) >= 5
qtysold = any (select qtysold from sales where dateid = 1882
```

다음 쿼리는 현재 포레이징되지 않는 모든 스쿼럴의 ID 값을 반환합니다.

```
SELECT id FROM squirrels
WHERE !is_foraging
```

다음은 VENUE 테이블에서 좌석 수가 10000석 이상인 장소를 반환하는 쿼리입니다.

```
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)
```

다음은 USERS 테이블에서 록 음악을 좋아하는 사용자(USERID)를 선택하는 예입니다.

```
select userid from users where likerock = 't' order by 1 limit 5;

userid
--------
3
5
6
13
16
(5 rows)
```

다음은 USER 테이블에서 록 음악을 좋아하는지 알 수 없는 사용자(USERID)를 선택하는 예입니다.

```
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 열이 있는 예

다음 예제 테이블 TIME_TEST에는 3개의 값이 삽입된 TIME_VAL(TIME 형식) 열이 있습니다.

```
select time_val from time_test;

time_val
---------------------
20:00:00
00:00:00.5550
00:58:00
```

다음 예에서는 각 timetz_val에서 시간을 추출합니다.

```
select time_val from time_test where time_val < '3:00';
   time_val
---------------
 00:00:00.5550
 00:58:00
```

다음 예에서는 2개의 시간 리터럴을 비교합니다.

```
select time '18:25:33.123456' = time '18:25:33.123456';
 ?column?
----------
 t
```

TIMETZ 열이 있는 예

다음 예제 테이블 TIMETZ_TEST에는 3개의 값이 삽입된 TIMETZ_VAL(TIMETZ 형식) 열이 있습니다.

```
select timetz_val from timetz_test;
```

```
timetz_val
------------------
04:00:00+00
00:00:00.5550+00
05:58:00+00
```

다음 예에서는 3:00:00 UTC보다 작은 TIMETZ 값만 선택합니다. 값을 UTC로 변환한 후 비교합니다.

```
select timetz_val from timetz_test where timetz_val < '3:00:00 UTC';

   timetz_val
---------------
 00:00:00.5550+00
```

다음 예에서는 2개의 TIMETZ 리터럴을 비교합니다. 비교를 위해 시간대는 무시됩니다.

```
select time '18:25:33.123456 PST' < time '19:25:33.123456 EST';

 ?column?
----------
 t
```

논리 조건

논리 조건은 두 조건 결과를 결합하여 단일 결과를 산출합니다. 모든 논리 조건은 반환 형식이 부울인 이진 연산자입니다.

구문

```
expression
{ AND | OR }
expression
NOT expression
```

논리 조건은 값이 3개인 부울 논리를 사용하며, 여기에서 NULL 값은 알 수 없는 관계를 의미합니다. 다음 표는 논리 조건 결과를 설명한 것으로서 E1과 E2는 표현식을 의미합니다.

E1	E	E1 AND E2	E1 OR E2	NOT E2
TRUE	TRUE	TRUE	TRUE	FALSE

E1	E	E1 AND E2	E1 OR E2	NOT E2
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
FALSE	TRUE	FALSE	TRUE	
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	
UNKNOWN	TRUE	UNKNOWN	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	

NOT 연산자는 AND 이전에, 그리고 AND 연산자는 OR 이전에 평가됩니다. 하지만 괄호를 사용하면 이러한 기본 평가 순서를 재정의할 수 있습니다.

예시

다음은 USERS 테이블에서 Las Vegas와 스포츠를 모두 좋아하는 사용자의 USERID 및 USERNAME을 반환하는 예입니다.

```
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)
```

다음은 USERS 테이블에서 Las Vegas 또는 스포츠를, 혹은 둘 다 좋아하는 사용자의 USERID 및 USERNAME을 반환하는 예입니다. 이 쿼리는 이전 예의 모든 출력에 더하여 Las Vegas 또는 스포츠만 좋아하는 사용자까지 반환합니다.

```
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)
```

다음은 OR 조건에 괄호를 사용하여 New York 또는 California에서 Macbeth를 공연한 장소를 찾는 예입니다.

```
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
...
```

위 예에서 괄호를 제거하면 쿼리의 논리 및 결과가 바뀝니다.

다음 예에서는 NOT 스크립트를 사용합니다:

```
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
...
```

다음은 NOT 조건에 이어 AND 조건을 사용하는 예입니다.

```
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)
```

패턴 일치 조건

패턴 일치 연산자는 문자열에서 조건식에 지정된 패턴을 검색하고 일치 항목을 찾았는지 여부에 따라 true 또는 false를 반환합니다. AWS Clean Rooms Spark SQL은 패턴 일치에 다음 메서드를 사용합니다.

- LIKE 표현식

  LIKE 연산자는 열 이름 같은 문자열 표현식과 %(퍼센트) 및 _(밑줄) 와일드카드 문자의 사용 패턴을 서로 비교합니다. LIKE 패턴 일치는 항상 전체 문자열을 검색합니다. LIKE는 대/소문자를 구분하여 일치를 수행합니다.

주제

- LIKE
- RLIKE

LIKE

LIKE 연산자는 열 이름 같은 문자열 표현식과 %(퍼센트) 및 _(밑줄) 와일드카드 문자의 사용 패턴을 서로 비교합니다. LIKE 패턴 일치는 항상 전체 문자열을 검색합니다. 문자열 내 아무 곳에서나 시퀀스를 일치시키려면 패턴이 퍼센트 기호로 시작해서 끝나야 합니다.

LIKE는 대/소문자를 구분합니다.

구문

```
expression [ NOT ] LIKE | pattern [ ESCAPE 'escape_char' ]
```

인수

expression

열 이름 같이 유효한 UTF-8 문자 표현식입니다.

LIKE

LIKE는 대/소문자를 구분하여 패턴을 일치시킵니다. 멀티바이트 문자에 대해 대/소문자를 구분하지 않는 패턴 일치를 수행하려면 LIKE 조건이 있는 expression 및 pattern에 LOWER 함수를 사용합니다.

= 및 <>와 같은 비교 조건자와 달리 LIKE 조건자는 후행 공백을 암시적으로 무시하지 않습니다. 후행 공백을 무시하려면 RTRIM을 사용하거나 CHAR 열을 VARCHAR로 명시적으로 캐스트합니다.

연~~산자는 LIKE와 동일합니다. 또한 !~~ 연산자는 NOT LIKE와 동일합니다.

패턴

일치시킬 패턴이 포함된, 유효한 UTF-8 문자 표현식입니다.

escape_char

패턴의 메타 문자를 이스케이프 처리하는 문자 표현식입니다. 기본값은 백슬래시 2개('\\')입니다.

pattern에 메타 문자가 포함되어 있지 않으면 패턴이 문자열 자체만 의미합니다. 이런 경우에는 LIKE가 등호 연산자와 동일한 역할을 합니다.

문자 표현식 중 하나는 CHAR 또는 VARCHAR 데이터 형식이 될 수 있습니다. 데이터 형식이 서로 다른 경우에는 AWS Clean Rooms 가 pattern을 expression의 데이터 형식으로 변환합니다.

LIKE에서 지원되는 패턴 일치 메타 문자는 다음과 같습니다.

연산자	설명
%	0개 이상의 문자 시퀀스를 일치시킵니다.
_	모든 문자를 일치시킵니다.

예시

다음 표는 LIKE를 사용한 패턴 일치의 예를 나타낸 것입니다.

표현식	반환
`'abc' LIKE 'abc'`	True
`'abc' LIKE 'a%'`	True
`'abc' LIKE '_B_'`	False
`'abc' LIKE 'c%'`	False

다음은 이름이 "E"로 시작하는 도시를 모두 찾는 예입니다.

```
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
...
```

다음은 성에 "ten"이 포함된 사용자를 찾는 예입니다.

```
select distinct lastname from users
where lastname like '%ten%' order by lastname;
lastname
-------------
Christensen
Wooten
...
```

다음 예제에서는 세 번째 및 네 번째 문자가 "ea"인 도시를 찾습니다.

```
select distinct city from users where city like '__EA%' order by city;
city
-------------
Brea
Clearwater
Great Falls
Ocean City
Olean
Wheaton
(6 rows)
```

다음은 기본 이스케이프 문자열(\\)을 사용하여 'start_'(텍스트 `start` 뒤에 밑줄 _가 붙음)가 포함된 문자열을 찾는 예입니다.

```
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)
```

다음은 '^'을 이스케이프 문자로 지정한 후 이 이스케이프 문자를 사용하여 'start_'(텍스트 `start` 뒤에 밑줄 _가 붙음)가 포함된 문자열을 찾는 예입니다.

```
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 연산자를 사용하면 문자열이 지정된 정규식 패턴과 일치하는지 확인할 수 있습니다.

str이와 일치하는 `true` 경우를 반환하고regexp, `false` 그렇지 않으면를 반환합니다.

구문

```
rlike(str, regexp)
```

인수

문자열

문자열 표현식

regexp

문자열 표현식입니다. 정규식 문자열은 Java 정규식이어야 합니다.

문자열 리터럴(정규식 패턴 포함)은 SQL 구문 분석기에서 이스케이프 처리되지 않습니다. 예를 들어 "\abc"와 일치시키려면 regexp의 정규식은 "^\abc$"일 수 있습니다.

예시

다음 예제에서는 spark.sql.parser.escapedStringLiterals 구성 파라미터의 값을 로 설정합니다true. 이 파라미터는 Spark SQL 엔진에만 해당됩니다. Spark SQL의 spark.sql.parser.escapedStringLiterals 파라미터는 SQL 구문 분석기가 이스케이프된 문자열 리터럴을 처리하는 방법을 제어합니다. 로 설정하면 true구문 분석기는 문자열 리터럴 내의 백슬래시 문자(\)를 이스케이프 문자로 해석하므로 문자열 값 내에 줄 바꿈, 탭 및 따옴표와 같은 특수 문자를 포함할 수 있습니다.

```
SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
```

예를 들어 에서는 SQL 쿼리에 다음 문자열 리터럴을 사용할 spark.sql.parser.escapedStringLiterals=true수 있습니다.

```
SELECT 'Hello, world!\n'
```

줄 바꿈 문자는 출력에서 리터럴 줄 바꿈 문자로 해석\n됩니다.

다음 예제에서는 정규식 패턴 일치를 수행합니다. 첫 번째 인수는 RLIKE 연산자에게 전달됩니다. 실제 사용자 이름이 '****' 패턴으로 대체되는 파일 경로를 나타내는 문자열입니다. 두 번째 인수는 일치에 사용되는 정규식 패턴입니다. 출력(true)은 첫 번째 문자열('%SystemDrive%\Users****')이 정규식 패턴()과 일치함을 나타냅니다'%SystemDrive%\\Users.*'.

```
SELECT rlike('%SystemDrive%\Users\John', '%SystemDrive%\Users.*');
true
```

BETWEEN 범위 조건

BETWEEN 조건은 키워드 BETWEEN과 AND를 사용하여 값의 범위에 대한 표현식의 포함 여부를 테스트합니다.

구문

```
expression [ NOT ] BETWEEN expression AND expression
```

표현식은 숫자, 문자 또는 날짜/시간 데이터 형식이 될 수 있지만 서로 호환 가능해야 합니다. 범위는 모든 값을 포함합니다.

예시

다음은 티켓 2장, 3장 또는 4장 중에서 판매가 등록된 티켓의 거래 수를 계산하는 예입니다.

```
select count(*) from sales
where qtysold between 2 and 4;

count
--------
104021
(1 row)
```

범위 조건에는 시작 값과 종료 값이 모두 포함됩니다.

```
select min(dateid), max(dateid) from sales
where dateid between 1900 and 1910;

min  | max
-----+-----
1900 | 1910
```

범위 조건에서 첫 번째 표현식은 두 번째 표현식보다 값이 작아야 하고, 두 번째 표현식은 첫 번째 표현식보다 값이 커야 합니다. 다음은 표현식의 값으로 인해 항상 0개의 행을 반환하는 예입니다.

```
select count(*) from sales
where qtysold between 4 and 2;
```

```
count
-------
0
(1 row)
```

하지만 NOT 한정자를 적용하면 논리가 반전되어 모든 행의 수를 반환합니다.

```
select count(*) from sales
where qtysold not between 4 and 2;

count
--------
172456
(1 row)
```

다음은 좌석 수가 20,000~50,000석인 장소 목록을 반환하는 쿼리입니다.

```
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)
```

다음 예에서는 날짜 값에 BETWEEN을 사용하는 방법을 보여 줍니다.

```
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
```

BETWEEN의 범위는 포함되지만 날짜는 기본적으로 00:00:00의 시간 값을 가집니다. 샘플 쿼리에 대해 유일하게 유효한 1월 3일 행은 판매 시간이 `1/3/2008 00:00:00`인 행입니다.

NULL 조건

NULL 조건은 값이 누락되거나 알 수 없는 경우 NULL 여부를 테스트합니다.

구문

```
expression IS [ NOT ] NULL
```

인수

expression

열 같은 모든 표현식입니다.

IS NULL

표현식의 값이 NULL일 때는 true이고, 표현식에 값이 있을 때는 false입니다.

IS NOT NULL

표현식의 값이 NULL일 때는 false이고, 표현식에 값이 있을 때는 true입니다.

예제

다음은 SALES 테이블의 QTYSOLD 필드에 NULL이 포함된 횟수를 나타내는 예입니다.

```
select count(*) from sales
where qtysold is null;
count
-------
0
```

```
(1 row)
```

EXISTS 조건

EXISTS 조건은 하위 쿼리에 대한 행의 존재 유무를 테스트한 후 하위 쿼리에서 행이 1개 이상 존재하면 true를 반환합니다. NOT을 지정하는 경우에는 하위 쿼리에 행이 없을 때 true를 반환합니다.

구문

```
[ NOT ] EXISTS (table_subquery)
```

인수

exists

table_subquery가 행을 1개 이상 반환하면 true입니다.

not_exists

table_subquery가 행을 하나도 반환하지 않으면 true입니다.

table_subquery

열이 1개 이상, 그리고 행이 1개 이상 포함된 테이블로 평가되는 하위 쿼리입니다.

예제

다음은 유형에 상관없이 판매가 이루어진 날짜마다 각각 한 번씩 날짜 식별자를 모두 반환하는 예입니다.

```
select dateid from date
where exists (
select 1 from sales
where date.dateid = sales.dateid
)
order by dateid;

dateid
--------
1827
1828
1829
```

```
...
```

IN 조건

IN 조건은 값 집합 또는 하위 쿼리에서 값의 멤버십 여부를 테스트합니다.

구문

```
expression [ NOT ] IN (expr_list | table_subquery)
```

인수

expression

expr_list 또는 table_subquery를 대상으로 평가되는 숫자, 문자 또는 날짜/시간 표현식으로서 해당 목록이나 하위 쿼리의 데이터 형식과 호환되어야 합니다.

expr_list

쉼표로 구분된 표현식 1개 이상, 또는 쉼표로 구분된 표현식 집합 1개 이상이며 괄호로 경계를 표시합니다.

table_subquery

행이 1개 이상 포함되어 있지만 select 목록의 열은 1개로 제한된 테이블로 평가되는 하위 쿼리입니다.

IN | NOT IN

IN은 표현식이 표현식 목록 또는 쿼리의 멤버일 때 true를 반환합니다. NOT IN은 표현식이 멤버가 아닐 때 true를 반환합니다. IN과 NOT IN은 expression이 NULL을 산출하는 경우, 혹은 expr_list 또는 table_subquery 값이 하나도 일치하지 않고 두 비교 행 중 적어도 하나가 NULL을 산출하는 경우에는 NULL과 함께 아무런 행도 반환되지 않습니다.

예시

다음 조건은 나열된 값일 때만 true로 평가됩니다.

```
qtysold in (2, 4, 5)
date.day in ('Mon', 'Tues')
date.month not in ('Oct', 'Nov', 'Dec')
```

대용량 IN 목록의 최적화

값이 10개 이상인 IN 목록은 쿼리 성능의 최적화를 위해 내부에서 스칼라 배열로 평가됩니다. 값이 10개 미만인 IN 목록은 OR 조건자의 연속으로 평가됩니다. 이러한 최적화는 SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, BOOLEAN, CHAR, VARCHAR, DATE, TIMESTAMP 및 TIMESTAMPTZ 데이터 형식에서 지원됩니다.

이러한 최적화의 효과는 다음 쿼리에서 EXPLAIN 출력을 보면 알 수 있습니다. 예:

```
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)
```

중첩된 데이터 쿼리

AWS Clean Rooms 는 관계형 및 중첩 데이터에 대한 SQL 호환 액세스를 제공합니다.

AWS Clean Rooms 는 중첩된 데이터에 액세스할 때 경로 탐색에 점선 표기법 및 배열 하위 스크립트를 사용합니다. 또한 FROM 절 항목이 배열을 반복하고 중첩 해제 작업에 사용할 수 있습니다. 다음 항목에서는 배열/구조체/맵 데이터 유형의 사용과 경로 및 배열 탐색, 중첩 해제 및 조인을 결합하는 다양한 쿼리 패턴에 대해 설명합니다.

주제

- 탐색
- 쿼리 중첩 해제
- Lax 의미 체계
- 내부 검사 유형

탐색

AWS Clean Rooms 를 사용하면 각각 `[...]` 대괄호와 점 표기법을 사용하여 배열과 구조를 탐색할 수 있습니다. 또한 점 표기법을 사용하는 구조와 대괄호 표기법을 사용하는 배열을 혼합하여 탐색할 수 있습니다.

Example

예를 들어, 다음 예제 쿼리에서는 `c_orders` 배열 데이터 열이 구조가 있는 배열이고 속성의 이름이 `o_orderkey`라고 가정합니다.

```
SELECT cust.c_orders[0].o_orderkey FROM customer_orders_lineitem AS cust;
```

필터링, 조인 및 집계와 같은 모든 형식의 쿼리에 점 및 대괄호 표기법을 사용할 수 있습니다. 일반적으로 열 참조가 있는 쿼리에서 이러한 표기법을 사용할 수 있습니다.

Example

다음 예에서는 결과를 필터링하는 SELECT 문을 사용합니다.

```
SELECT count(*) FROM customer_orders_lineitem WHERE c_orders[0].o_orderkey IS NOT NULL;
```

Example

다음 예에서는 GROUP BY 및 ORDER BY 절에 괄호와 점 탐색을 사용합니다.

```
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;
```

쿼리 중첩 해제

쿼리를 중첩 해제하려면 배열을 통해 반복을 AWS Clean Rooms 활성화합니다. 쿼리의 FROM 절로 배열을 탐색하여 이를 수행합니다.

Example

다음 예에서는 이전 예를 사용하여 `c_orders`에 대한 속성 값을 반복합니다.

```
SELECT o FROM customer_orders_lineitem c, c.c_orders o;
```

중첩 해제 구문은 FROM 절의 확장입니다. 표준 SQL에서 FROM 절 x (AS) y는 y가 x 관계에 있는 각 튜플을 반복함을 의미합니다. 이 경우 x는 관계를 나타내고, y는 관계 x에 대한 별칭을 나타냅니다. 마찬가지로 FROM 절 항목 x (AS) y를 사용하여 중첩을 해제하는 구문은 y가 배열 표현식 x의 각 값을 반복한다는 의미입니다. 이 경우 x는 배열 표현식이고, y는 x에 대한 별칭입니다.

왼쪽 피연산자는 일반 탐색을 위해 점 및 대괄호 표기법을 사용할 수도 있습니다.

Example

이전 예시를 참고하세요.

- `customer_orders_lineitem c`은 `customer_order_lineitem` 기본 테이블에 대한 반복입니다
- `c.c_orders o`는 `c.c_orders array`에 대한 반복입니다

배열 내의 배열인 `o_lineitems` 속성을 반복하려면 여러 절을 추가해야 합니다.

```
SELECT o, l FROM customer_orders_lineitem c, c.c_orders o, o.o_lineitems l;
```

AWS Clean Rooms 는 AT 키워드를 사용하여 배열을 반복할 때 배열 인덱스도 지원합니다. 절 x AS y AT z는 배열 x를 반복하고 배열 인덱스인 필드 z를 생성합니다.

Example

다음 예에서는 배열 인덱스의 작동 방식을 보여줍니다.

```
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

다음 예에서는 스칼라 배열을 반복합니다.

```
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

다음 예에서는 여러 수준의 배열을 반복합니다. 이 예제에서는 여러 unnest 절을 사용하여 가장 안쪽 배열로 반복합니다. f.multi_level_array AS 배열은 multi_level_array를 반복합니다.. 배열 AS 요소는 multi_level_array 이내의 배열에 대한 반복입니다.

```
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 의미 체계

기본적으로 중첩된 데이터 값에 대한 탐색 작업은 탐색이 유효하지 않은 경우 오류를 반환하는 대신 null을 반환합니다. 중첩된 데이터 값이 객체가 아니거나 중첩된 데이터 값이 객체이지만 쿼리에 사용된 속성 이름이 포함되어 있지 않은 경우 객체 탐색은 유효하지 않습니다.

Example

예를 들어, 다음 쿼리는 중첩된 데이터 열 `c_orders`의 잘못된 속성 이름에 액세스합니다:

```
SELECT c.c_orders.something FROM customer_orders_lineitem c;
```

배열 탐색은 중첩된 데이터 값이 배열이 아니거나 배열 인덱스가 범위를 벗어난 경우 null을 반환합니다.

Example

다음 쿼리는 `c_orders[1][1]`가 범위를 벗어났기 때문에 null을 반환합니다.

```
SELECT c.c_orders[1][1] FROM customer_orders_lineitem c;
```

내부 검사 유형

중첩된 데이터 유형 열은 값에 대한 유형 및 기타 유형 정보를 반환하는 검사 함수를 지원합니다. AWS Clean Rooms는 중첩된 데이터 열에 대해 다음과 같은 부울 함수를 지원합니다.

- 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

입력 값이 null인 경우 이러한 함수는 모두 false를 반환합니다. IS_SCALAR, IS_OBJECT 및 IS_ARRAY는 상호 배타적이며 null을 제외한 모든 가능한 값을 포함합니다. 데이터에 해당하는 유형을 추론하려면 다음 예제와 같이 중첩된 데이터 값 유형(상위 수준)을 반환하는 JSON_TYPEOF 함수를 AWS Clean Rooms 사용합니다.

```
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 참조의 문서 기록

다음 표에서는 AWS Clean Rooms SQL 참조에 대한 설명서 릴리스를 설명합니다.

이 설명서에 대한 업데이트 알림을 받으려면 RSS 피드에 가입하면 됩니다. RSS 업데이트를 구독하려면 사용 중인 브라우저에서 RSS 플러그인을 활성화해야 합니다.

변경 사항	설명	날짜
Spark SQL에서 힌트 지원	AWS Clean Rooms Spark SQL은 쿼리 성능을 최적화하고 컴퓨팅 비용을 줄이기 위해 쿼리 힌트를 지원합니다.	2026년 1월 20일
Spark SQL에서 캐시 테이블 지원	AWS Clean Rooms Spark SQL은 고객이 기존 테이블을 캐싱하거나 쿼리 결과에서 새 테이블을 생성 및 캐싱하여 쿼리 성능을 개선할 수 있는 CACHE TABLE 명령을 지원합니다.	2025년 10월 22일
Spark SQL에서 FIRST 및 LAST Window 함수 지원	AWS Clean Rooms Spark SQL은 FIRST 및 LAST와 같은 창 함수를 지원합니다.	2025년 6월 12일
Spark SQL 함수 설명서 업데이트	지원되는 Spark SQL 함수를 정확하게 반영하기 위한 설명서 전용 업데이트입니다. <=> 연산자, SIMILAR TO, LISTAGG 및 ARRAY_INSERT를 포함하여 지원되지 않는 25개의 함수에 대한 설명서를 제거했습니다. 함수 이름을 DATEADD에서 DATE_ADD로, DATEDIFF에서 DATE_DIFF로, ISNULL에서 IS_NULL로, ISNOTNULL에	2025년 5월 20일

	서 IS_NOT_NULL로 수정했습니다. DATE_PART 예제의 오타를 수정했습니다.	
AWS Clean Rooms Spark SQL	이제 고객은 Spark SQL 분석 엔진에서 지원되는 일부 SQL 조건, 함수, 명령 및 규칙을 사용하여 쿼리를 실행할 수 있습니다.	2024년 10월 29일
SQL 명령 및 SQL 함수 - 업데이트	JOIN 절, EXCEPT 집합 연산자, CASE 조건식, ANY_VALUE, NVL 및 COALESCE, NULLIF, CAST, CONVERT, CONVERT_TIMEZONE, EXTRACT, MOD, SIGN, CONCAT, FIRST_VALUE 및 LAST_VALUE 함수에 대한 예제가 추가되었습니다.	2024년 2월 28일
SQL 함수 - 업데이트	AWS Clean Rooms 는 이제 배열, SUPER 및 VARBYTE와 같은 SQL 함수를 지원합니다. 이제 지원되는 수학 함수는 ACOS, ASIN, ATAN, ATAN2, COT, DEXP, PI, POW, RADIANS 및 SIN 등입니다. 이제 CAN_JSON_PARSE, JSON_PARSE 및 JSON_SERIALIZE와 같은 JSON 함수가 지원됩니다.	2023년 10월 6일
중첩된 데이터 유형 지원	AWS Clean Rooms 는 이제 중첩된 데이터 형식을 지원합니다.	2023년 8월 30일

SQL 명명 규칙 - 업데이트	예약 열 이름을 명확히 하기 위해 문서 전용으로 변경되었습니다.	2023년 8월 16일
정식 출시	이제 AWS Clean Rooms SQL 참조를 정식 버전으로 사용할 수 있습니다.	2023년 7월 31일

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.