Red Hat Integration 2020.q1

Data Virtualization のリファレンス

テクノロジープレビュー - データ仮想化の参照

Last Updated: 2023-02-28

Red Hat Integration 2020.q1 Data Virtualization のリファレンス

テクノロジープレビュー - データ仮想化の参照

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

Data Virtualization の一般的な参照情報。

目次

第1章 DATA VIRTUALIZATION のリファレンス

Data Virtualization は、情報の統合にスケーラビリティーが高く、高性能なソリューションを提供します。統合データおよびエンリッチされたデータを、複数のプロトコルで JSON、XML、およびその他の形式としてリレーショナルデータベースして使用することができます。Data Virtualization は、開発者のデータアクセスやアプリケーションの消費を簡素化します。

Red Hat は、商用開発サポート、実稼働サポート、および Data Virtualization の試験が可能です。Data Virtualization は、企業のオープンソースプロジェクトと Red Hat データ統合の重要なコンポーネントです。

Data Virtualization を調べる前に、Data Virtualization の基本的な構成を理解することは非常に重要です。たとえば、仮想データベースとはモデルとは詳細は、Teiid Basics を参照してください。

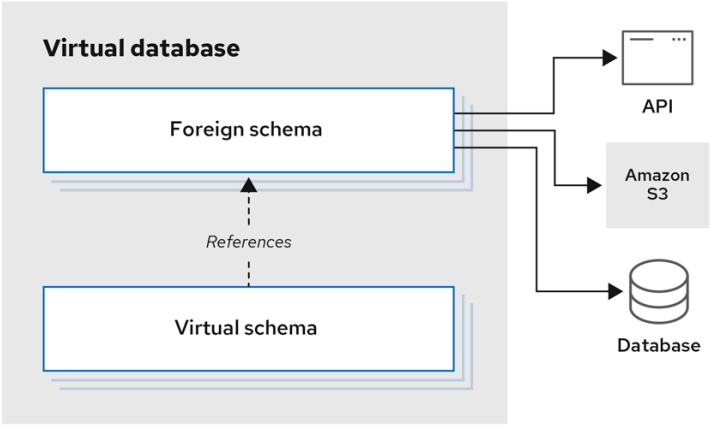
指定されていない場合、本書で参照されるバージョンは Teiid プロジェクトバージョンを参照します。各種プラットフォームで実行される Teiid または Data Virtualization には、プラットフォームと製品固有のバージョン管理の両方があります。

重要

Data Virtualization はテクノロジープレビューの機能です。テクノロジープレビュー機能は、Red Hat の本番環境のサービスレベルアグリーメント (SLA) ではサポートされず、機能的に完全ではないことがあるため、Red Hat は本番環境での使用は推奨しません。Red Hat は実稼働環境でこれらを使用することを推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

第2章 仮想データベース

仮想データベース(VDB)は、複数のデータソースからデータを統合するために使用されるコンポーネントのメタデータコンテナーであり、単一の統一された API を介して統合方法でアクセスできるようにします。



仮想データベースには、通常複数のスキーマコンポーネント（モデルとも呼ばれます）が含まれ、各スキーマにはメタデータ（テーブル、手順、関数）が含まれます。スキーマには、以下の 2 つのタイプがあります。

外部スキーマ

外部スキーマは **ソース** または **物理** スキーマとも呼ばれ、外部スキーマは Oracle、Db2、MySQL などのリレーショナルデータベース、CSV、Microsoft Excel、または SOAP や REST などの Web サービスなどの外部データソースまたはリモートデータソースを表します。

仮想スキーマ

外部スキーマのスキーマオブジェクトを使用して定義されるビュー層または論理 **スキーマレイヤー**。たとえば、複数のソースから複数の外部テーブルを集約するビューテーブルを作成すると、生成されるビューは、ビューを定義するデータソースの複雑度からユーザーになります。

留意すべき重要な点の 1 つは、仮想データベースにはメタデータだけが含まれていることです。Data Virtualization に関連するユースケースには、開始する仮想データベースモデルが必要です。そのため、VDB を設計および開発する方法を理解することが重要です。

以下の仮想データベースモデルの例は、PostgreSQL データベースへの接続を行う単一の外部スキーマコンポーネントを定義します。

この例の SQL DDL コマンドは、SQL/MED 仕様を実装します。

```
CREATE DATABASE my_example;
USE DATABASE my_example;
CREATE SERVER pgsql
   VERSION 'one' FOREIGN DATA WRAPPER postgresql
   OPTIONS (
      "resource-name" 'java:/postgres-ds'
   );
CREATE SCHEMA test SERVER pgsql;
IMPORT FOREIGN SCHEMA public FROM SERVER pgsql INTO test
   OPTIONS(
```

```
        importer.useFullSchemaName false,
        importer.tableTypes 'TABLE,VIEW'
);
```

以下のセクションでは、前述の例のステートメントを使用して仮想データベースを定義する方法を説明します。前は、**ソーススキーマ** コンポーネントのさまざまな要素について学ぶ必要があります。

外部データソース

仮想データベースの「ソーススキーマ」コンポーネントは、外部データソースのメタデータをローカルで表現するスキーマオブジェクト、テーブル、手順、および関数のコレクションです。この例では、スキーマオブジェクトは直接定義されていませんが、サーバーからインポートされます。外部データソースへの接続の詳細は、**resource-name** を介して提供されます。これは、外部データソースへの名前付きの接続参照です。

Data Virtualization の目的で、クエリーを接続および実行してこれらの外部データソースからメタデータを取得するため、Data Virtualization は 2 種類のリソースを定義/提供します。

トランスレーター

DATA WRAPPER とも呼ばれるトランスレーターは、Data Virtualization クエリーエンジンと物理データソース間の抽象化レイヤーを提供するコンポーネントです。トランスレーターは、クエリーコマンドを Data Virtualization からソース固有のコマンドに変換し、実行する方法を認識します。トランスレーターには、物理ソースが返すデータを Data Virtualization クエリーエンジンが処理できる形式に変換する情報もあります。たとえば、Web サービストランスレーターを使用すると、トランスレーターは Data Virtualization レイヤーからの SQL 手順を HTTP 呼び出しに変換し、JSON 応答は表形式で変換されます。

Data Virtualization は、システムの一部としてさまざまなトランスレーターを提供するか、提供された java ライブラリーを使用して開発することもできます。利用可能な翻訳者の詳細は、「翻訳者」を参照して ください。

2.1. 仮想データベースのプロパティー

DATABASE プロパティー

- **domain-ddl**
- **schema-ddl**
- **query-timeout** は、この VDB に対して実行されるクエリーのデフォルトクエリーのタイムアウトをミリ秒単位で設定します。**0** は、サーバーのデフォルトクエリータイムアウトを使用する必要があることを示します。デフォルトは 0 です。サーバーのデフォルトクエリータイムアウトが小さい値に設定されている場合は有効ではありません。クライアントは引き続き、クライアント側で管理する独自のタイムアウトを設定できます。
- **connection.XXX**: デフォルトの connection/execution プロパティーを設定するために ODBC トランスポートおよび OData で使用する場合。関連するプロパティーの詳細は、『クライアント開発者ガイド』の「**ドライバー接続** 」を参照し てください。これらは、確立後に接続に設定されることに注意してください。

```
CREATE DATABASE vdb OPTIONS ("connection.partialResultsMode" true);
```

- **authentication-type**

設定されたセキュリティードメインの認証タイプ。現在使用できる値は（GSS、USERPASSWORD）です。デフォルトではトランスポート（通常は USERPASSWORD）に設定されます。

- **password-pattern**

USERPASSWORD 認証が使用されるかどうかを判断する接続ユーザー名と照合される正規表現。**password-pattern** は **authentication-type** よりも優先されます。デフォルトは **authentication-type** です。

- **gss-pattern**

GSS 認証が使用されているかどうかを決定する接続ユーザー名と照合される正規表現。**GSS-pattern** は **password-pattern** よりも優先されます。デフォルトは **password-pattern** です。

- **max-sessions-per-user** (11.2+)

この VDB のユーザー名で識別される各ユーザーに許可される最大セッション数。何も設定されず、負の値の場合には、ユーザーの最大値が表示されますが、セッションサービスの最大値は引き続き適用されます。これはクラスター内の各 Data Virtualization サーバーメンバーに対して実施され、クラスター全体のデータ仮想化サーバーメンバーには適用されません。既存のセッション下のタスク用に作成される派生セッションは、この最大値に対してカウントされません。

- **model.visible**

インポートされた vdb モデルの可視性を上書きするのに使用します。

- **include-pg-metadata**

デフォルトでは、**org.teiid.addPGMetadata** プロパティーを false に設定していない限り、PostgreSQL メタデータは常に VDB に追加されます。このプロパティーにより、VDB ごとに PG メタデータを追加できます。詳細は、『管理者ガイド』の「**システムプロパティー**」を参照してください。ODBC を使用して VDB にアクセスする場合には、VDB には PG メタデータが含まれている必要があります。

- **lazy-invalidate**

デフォルトでは、TTL の有効期限が無効になります。詳細は、『キャッシュガイド』の「**内部ドラフト**」を参照してください。lazy-invalidate を **true** に設定すると、TTL の更新は非検証になります。

- **deployment-name**

実質的に予約されています。サーバーによるデプロイ時、サーバーデプロイメントの名前に設定されます。

スキーマおよびモデルのプロパティー

- **visible**

値が **true** の場合にスキーマに表示済みとしてマークします（デフォルト設定）。**表示される** フラグが **false** に設定されている場合、スキーマのメタデータはメタデータ要求から非表示になります。プロパティーを **false** に設定しても、このスキーマに対してクエリーを発行することは禁止されません。データへのアクセスを制御する方法は、「データ ロール」を参照し てください。

- **multisource**

スキーマをマルチソースモードに設定し、データを複数の異なるソースのパーティションに存在するように設定します。スキーマのメタデータがすべてのデータソースで同じであることを前提とします。

- **multisource.columnName**

マルチソーススキーマでは、パーティションを指定する追加の列が暗黙的にすべてのテーブルに追加され、ソースを特定します。このプロパティーは列の名前を定義します。型は常に **String** になります。

- **multisource.addColumn**

このフラグは、このスキーマのすべてのテーブルに暗黙的なパーティション列を追加するように指定します。**true** 値により列が追加されます。デフォルトは **false** です。

- **allowed-languages**

VDB の任意の目的に使用できるプログラミング言語のコンマ区切りリストを指定します。名前は大文字と小文字を区別し、リストにはエントリー間の空白を含めることはできません。例： &**lt;property name="allowed-languages" value="javascript"/>**

- **allow-** languages は、ロールに **allowed-languages** プロパティーに記載されている言語を使用するパーミッションがあることを指定します。たとえば、以下の抜粋の **allow-language** プロパティーは、**RoleA** ロールを持つユーザーに Javascript を使用するパーミッションがあることを指定します。

```
<data-role name="RoleA">
    <description>Read and javascript access.</description>

    <permission>
      <resource-name>modelName</resource-name>
      <allow-read>true</allow-read>
    </permission>

    <permission>
      <resource-name>javascript</resource-name>
      <allow-language>true</allow-language>
    </permission>

    <mapped-role-name>role1</mapped-role-name>

  </data-role>
```

2.2. スキーマオブジェクトの DDL メタデータ

テーブルとビューがスキーマの同じ namespace に存在する。インデックスはスキーマスコープオブジェクトとはみなされませんが、定義されたテーブルまたはビューに対してスコープ付けされます。手順と機能は別の namespace で定義されていますが、仮想手順言語で定義される機能は、関数と同じ名前の手順の両方として存在します。ドメインタイプはスキーマスコープではありません。それらは VDB 全体のスコープになります。

データ型

データ型の詳細は、SQL 文法の BNF の **単純なデータタイプ** を参照してください。

外部テーブル

FOREIGN テーブルは、Oracle、Microsoft SQL Server などのソースデータベースの実際のリレーショナルデータベースの実際のリレーショナルデータベースを表すソーススキーマで定義されるテーブルです。リレーショナルデータベースの場合、既存のスキーマを自動インポートする必要がある場合に、Data Virtualization は VDB のデプロイメント時にデータベーススキーマ情報を自動的に取得できます。

ただし、ユーザーは PHYSICAL スキーマでテーブルを明示的に定義したい場合や、カスタム変換機能でリレーショナルデータベース以外のデータをリレーショナルデータベースとして表現する場合は、以下の FOREIGN テーブルセマンティクスを使用できます。

例： 外部テーブルの作成（PHYSICAL モデルで作成）

```
CREATE FOREIGN TABLE {table-name} (
   <table-element> (,<table-element>)*
   (,<constraint>)*
) [OPTIONS (<options-clause>)]


<table-element> ::=
   {column-name} <data-type> <element-attr> <options-clause>

<data-type> ::=
   varchar | boolean | integer | double | date | timestamp .. (see Data Types)

<element-attr> ::=
   [AUTO_INCREMENT] [NOT NULL] [PRIMARY KEY] [UNIQUE] [INDEX] [DEFAULT {expr}]

<constraint> ::=
   CONSTRAINT {constraint-name} (
      PRIMARY KEY <columns> |
      FOREIGN KEY (<columns>) REFERENCES tbl (<columns>)
      UNIQUE <columns> |
      ACCESSPATTERN <columns>
      INDEX <columns>

<columns> ::=
   ( {column-name} [,{column-name}]* )

<options-clause> ::=
   <key> <value>[,<key>, <value>]*
```

外部テーブルの作成に関する詳細は、BNF for SQL grammar の **CREATE TABLE** を参照してください。

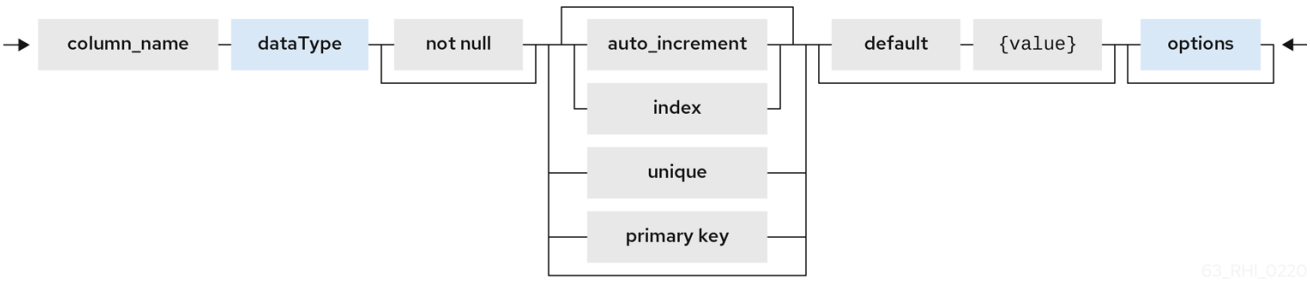
例： 外部テーブルの作成（PHYSICAL モデルで作成）

```
CREATE FOREIGN TABLE Customer (
   id integer PRIMARY KEY,
   firstname varchar(25),
   lastname varchar(25),
   dob timestamp);

CREATE FOREIGN TABLE Order (
   id integer PRIMARY KEY,
   customerid integer OPTIONS(ANNOTATION 'Customer primary key'),
   saledate date,
   amount decimal(25,4),
   CONSTRAINT CUSTOMER_FK FOREIGN KEY(customerid) REFERENCES Customer(id)
 ) OPTIONS(UPDATABLE true, ANNOTATION 'Orders Table');
```

TABLE OPTIONS:（以下のオプションはよく知られており、定義されたその他のプロパティーは拡張メタデータとみなされます）

プロパティー	データタイプまたは許可される値	説明
UUID	string	ビューの一意識別子。
CARDINALITY	int	コスト情報。テーブルの行数。計画の目的で使用されます。
更新可能	'TRUE'	'FALSE'
ビューの更新が許可されるかどうかを定義します。	アノテーション	string
ビューの説明。	DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC



COLUMN OPTIONS:（以下のオプションはよく知られており、その他のプロパティーは拡張メタデータとみなされます）。

プロパティー	データタイプまたは許可される値	説明
UUID	string	列の一意識別子。
NAMEINSOURCE	string	これが FOREIGN テーブルの列名である場合、この値はソースデータベースの列の名前を表します。省略すると、ソースに対するデータのクエリー時に列名が使用されます。
CASE_SENSITIVE	'TRUE'|'FALSE'	
選択可能	'TRUE'|'FALSE'	この列がユーザークエリーから選択できる場合は TRUE。

プロパティー	データタイプまたは許可される値	説明
更新可能	'TRUE'|'FALSE'	列が updatable であるかを定義します。view/table が updatable の場合、デフォルトは true に設定されます。
署名あり	'TRUE'|'FALSE'	
通貨	'TRUE'|'FALSE'	
FIXED_LENGTH	'TRUE'|'FALSE'	
検索可能	'SEARCHABLE'|'UNSEARCHABLE'|'LIKE_ONLY'|'ALL_EXCEPT_LIKE'	列の検索性。通常、データタイプにより指定されます。
MIN_VALUE		
MAX_VALUE		
CHAR_OCTET_LENGTH	integer	
アノテーション	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	Long	コスト情報。この列の NULLS 数。
DISTINCT_VALUES	Long	コスト情報。この列の一意の値の数。

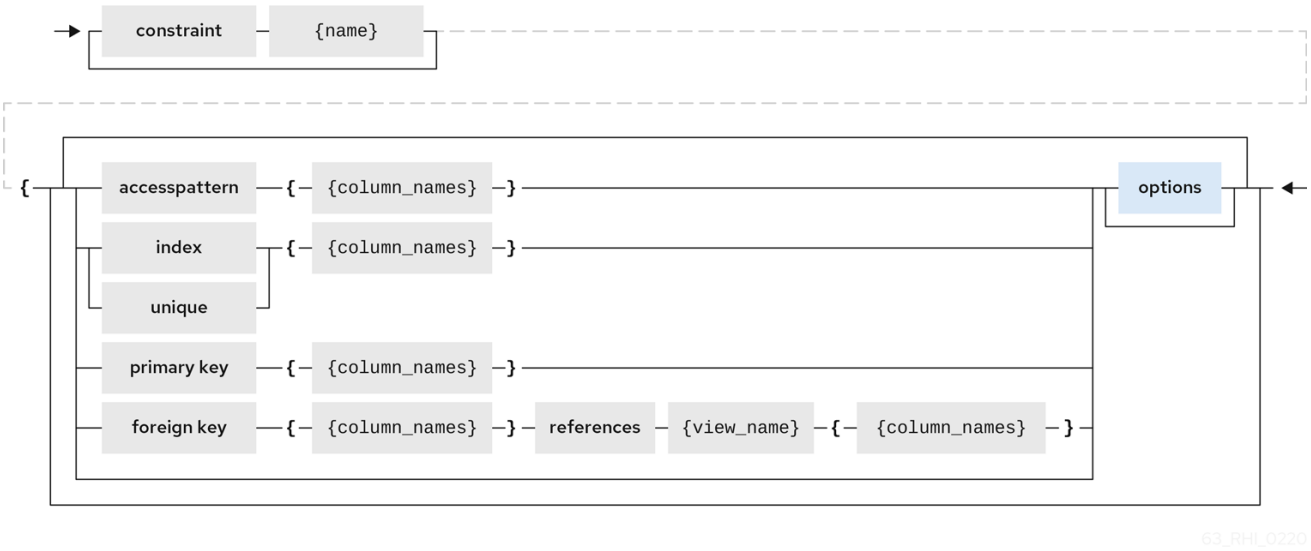
列は NOT NULL、auto_increment、または DEFAULT の値とマークすることもできます。

大きな型の列は、精度/スケールなしで 10 進数または 10 進数の列を宣言できます。デフォルトは、半分スケールの精度に対する内部の最大値です。あるいは、デフォルトでスケールが 0 になる精度を使用します。

タイプタイムスタンプのコラムは、スケールなしで宣言できます。これは、デフォルトで内部最大の 9 分秒になります。

テーブルの制約

テーブル/ビューで制約を定義して、インデックスや他のテーブル/ビューへの関係を定義できます。この情報は、Data Virtualization オプティマイザーによってクエリーを計画するか、マテリアル化テーブルのインデックスを使用してデータへのアクセスを最適化します。



CONSTRAINTS は gitops で定義できるものと同じです。

CONSTRAINTs の例

```
CREATE FOREIGN TABLE Orders (
    name varchar(50),
    saledate date,
    amount decimal,
    CONSTRAINT CUSTOMER_FK FOREIGN KEY(customerid) REFERENCES Customer(id)
    ACCESSPATTERN (name),
    PRIMARY KEY ...
    UNIQUE ...
    INDEX ...
```

テーブルの変更

ALTER TABLE ステートメントの完全な SQL 文法は、BNF for SQL 文法 の **ALTER TABLE** を参照してください。

ALTER コマンドを使用すると、列の追加、変更、削除、および任意の OPTIONS の値の変更、および制約の追加を行うことができます。以下の例は、ALTER コマンドを使用してテーブルオブジェクトを変更する方法を示しています。

```
-- add column to the table
ALTER FOREIGN TABLE "Customer" ADD COLUMN address varchar(50) OPTIONS(SELECTABLE
true);

-- remove column to the table
ALTER FOREIGN TABLE "Customer" DROP COLUMN address;

-- adding options property on the table
ALTER FOREIGN TABLE "Customer" OPTIONS (ADD CARDINALITY 10000);

-- Changing options property on the table
ALTER FOREIGN TABLE "Customer" OPTIONS (SET CARDINALITY 9999);

-- Changing options property on the table's column
ALTER FOREIGN TABLE "Customer" ALTER COLUMN "name" OPTIONS(SET UPDATABLE
```

```
FALSE)

-- Changing table's column type to integer
ALTER FOREIGN TABLE "Customer" ALTER COLUMN "id" TYPE bigdecimal;

-- Changing table's column column name
ALTER FOREIGN TABLE "Customer" RENAME COLUMN "id" TO "customer_id";

-- Adding a constraint
ALTER VIEW "Customer_View" ADD PRIMARY KEY (id);
```

ビュー

ビューは仮想テーブルです。ビューには、実際のテーブルなどの行と列が含まれます。ビューの列は、ソースまたは他のビューモデルの1つ以上の実際のテーブルのコラムです。また、複数の列または集約された列で構成される式を使用することもできます。列定義が view テーブルで定義されない場合は、**AS** キーワードの後に定義されるビューの選択変換の展開された列から派生します。

データが1つのテーブルから送信されるかのように、関数、JOIN ステートメント、および WHERE 句をビューデータに追加できます。

現在、アクセスパターンは表示には意味がありませんが、文法で引き続き許可されます。ビューの他の制約も適用されません。内部マテリアルビューで指定されていない限り、それらをマテリアル化ターゲットテーブルに自動的に追加されます。ただし、アクセス以外のパターンビュー制約は、最適化やクライアントによる検出の関係を伝えるなど、他の目的でも便利です。

BNF - CREATE VIEW

```
CREATE VIEW {table-name} [(
    <view-element> (,<view-element>)*
    (,<constraint>)*
)] [OPTIONS (<options-clause>)]
    AS {transformation_query}


<table-element> ::=
    {column-name} [<data-type> <element-attr> <options-clause>]

<data-type> ::=
    varchar | boolean | integer | double | date | timestamp .. (see Data Types)

<element-attr> ::=
    [AUTO_INCREMENT] [NOT NULL] [PRIMARY KEY] [UNIQUE] [INDEX] [DEFAULT {expr}]

<constraint> ::=
    CONSTRAINT {constraint-name} (
        PRIMARY KEY <columns> |
        FOREIGN KEY (<columns>) REFERENCES tbl (<columns>)
        UNIQUE <columns> |
        ACCESSPATTERN <columns>
        INDEX <columns>

<columns> ::=
    ( {column-name} [,{column-name}]* )
```

```
<options-clause> ::=
   <key> <value>[,<key>, <value>]*
```

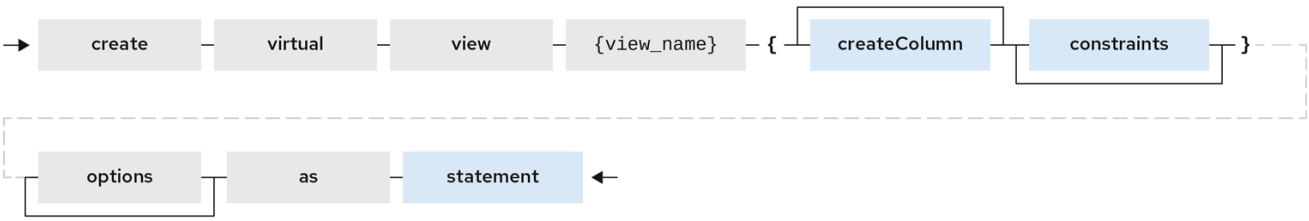


表2.1 VIEW OPTIONS:（これらのプロパティーは CREATE TABLE で定義されるプロパティーの他に、これらのプロパティーも CREATE TABLE で定義されます）

プロパティー	データタイプまたは許可される値	説明
マテリアル化	'TRUE'|'FALSE'	テーブルがマテリアル化されているかどうかを定義します。
MATERIALIZED_TABLE	'table.name'	このビューが外部データベースにマテリアル化されている場合は、マテリアル化されたテーブルの名前を定義します。

例： ビューテーブルの作成（VIRTUAL スキーマで作成される）

```
CREATE VIEW CustomerOrders
  AS
  SELECT concat(c.firstname, c.lastname) as name,
       o.saledate as saledate,
       o.amount as amount
  FROM Customer C JOIN Order o ON c.id = o.customerid;
```

重要

列は変換クエリー（SELECT ステートメント）によって暗黙的に定義されることに注意してください。列はインラインでも定義できますが、定義した場合はプロパティーの変更のみが可能です。ADD または DROP の新規列は追加できません。

テーブルの変更

The BNF for ALTER VIEW, refer to ALTER TABLE（ALTER VIEW の BNF。ALTER TABLE を参照）

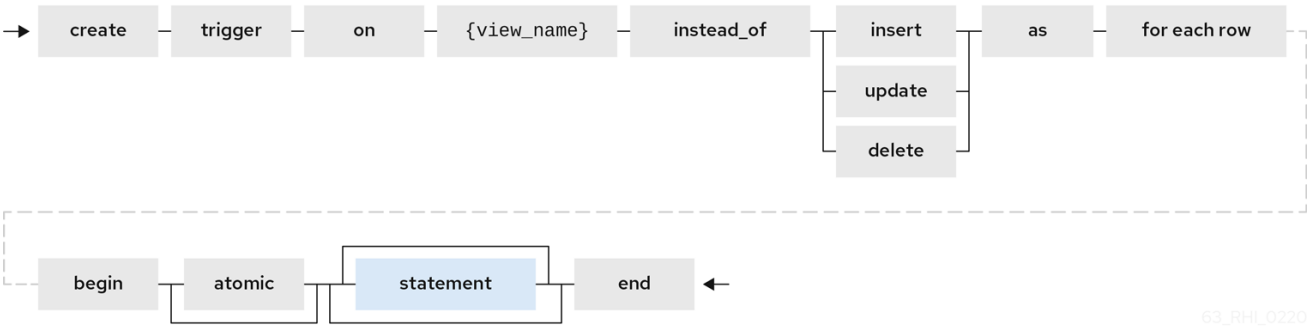
ALTER COMMAND を使用すると、VIEW の変換クエリーを変更できます。列 **情報は変更できません。** 変換クエリーは有効である必要があります。

```
ALTER VIEW CustomerOrders
   AS
   SELECT concat(c.firstname, c.lastname) as name,
```

```
        o.saledate as saledate,
        o.amount as amount
    FROM Customer C JOIN Order o ON c.id = o.customerid
    WHERE saledate < TIMESTAMPADD(now(), -1, SQL_TSI_MONTH)
```

VIEW での INSTEAD OF のトリガー（VIEW の更新）

複数のベーステーブルを構成するビューは、レコードを挿入し、更新を適用し、テーブルの参照データを削除するために **INSTEAD OF** トリガーを使用する必要があります。VIEW の **UPDATABLE** OPTION が **TRUE** に設定されている場合、一部の変換の複雑さに基づいて、ユーザーに INSTEAD OF TRIGGERS が自動的に提供されます。ただし、CREATE TRIGGER メカニズムを使用すると、デフォルトの動作を提供/上書きすることができます。



例： INSERT のビューでの INSTEAD OF トリガーの定義

```
CREATE TRIGGER ON CustomerOrders INSTEAD OF INSERT AS
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO Customer (...) VALUES (NEW.name ...);
    INSERT INTO Orders (...) VALUES (NEW.value ...);
  END
```

更新の場合

例： UPDATE の View でトリガーではなく定義

```
CREATE TRIGGER ON CustomerOrders INSTEAD OF UPDATE AS
  FOR EACH ROW
  BEGIN ATOMIC
    IF (CHANGING.saledate)
    BEGIN
      UPDATE Customer SET saledate = NEW.saledate;
      UPDATE INTO Orders (...) VALUES (NEW.value ...);
    END
  END
```

更新すると、列の以前の値と新しい値にアクセスできます。更新手順の詳細は、「手順の更新」を参照してください。???

ソーステーブルでの AFTER トリガー

ソーステーブルには、変更データキャプチャーシステムによって報告される変更イベントを処理するために登録される一意の名前付きトリガーを含めることができます。

表示するのと同様に、AFTER insert は NEW グループを介して新しい値へのアクセスを提供し、AFTER delete は OLD グループ経由で以前の値へのアクセスを提供し、AFTER の更新により両方のアクセスが提供されます。

例： カスタマーでの AFTER トリガー

```
CREATE TRIGGER ON Customer AFTER INSERT AS
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO CustomerOrders (CustomerName, CustomerID) VALUES (NEW.Name, NEW.ID);
  END
```

通常、操作ごとにハンドラーを定義します(INSERT/UPDATE/DELTE)。

更新手順の詳細は、「 更新手順」を参照してください。

手順/機能の作成

ユーザーは以下の機能のいずれかを定義できます。

Source Procedure("CREATE FOREIGN PROCEDURE")

ソースのストアドプロシージャー。

Source Function("CREATE FOREIGN FUNCTION")

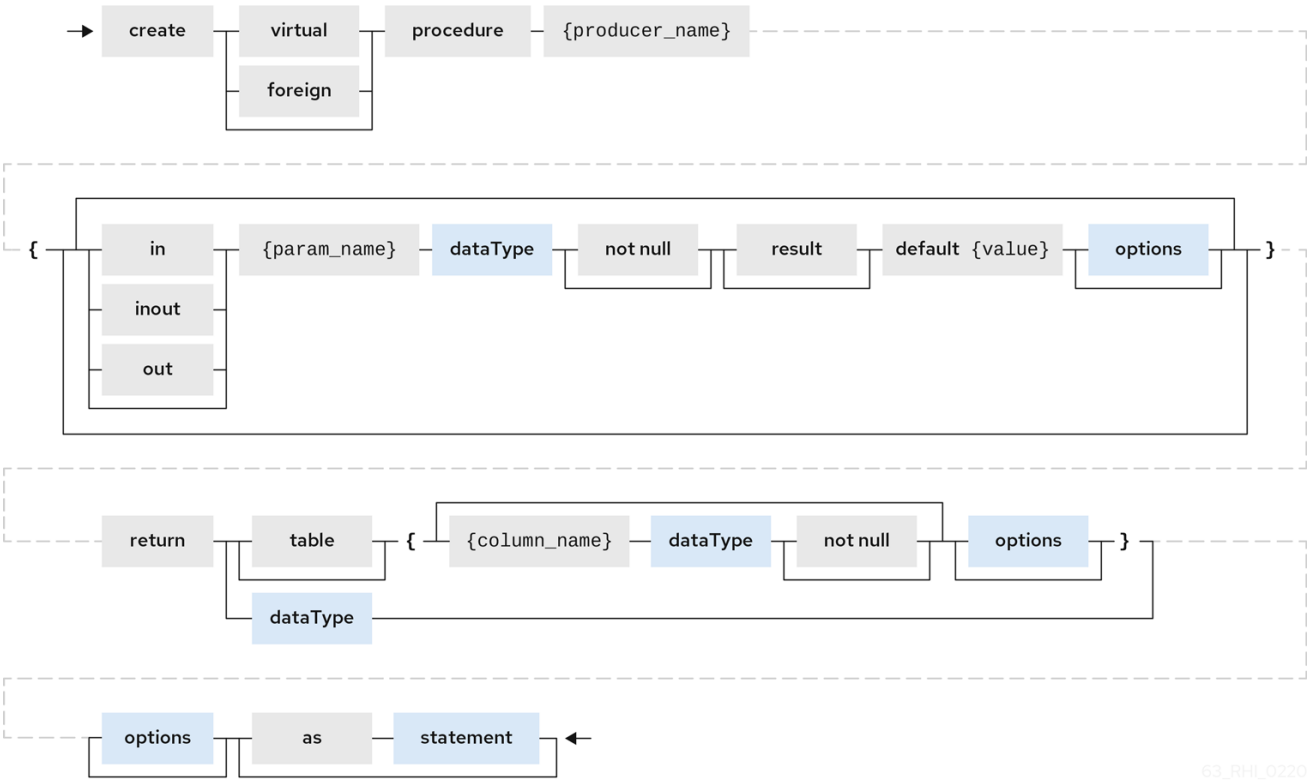
データソースの機能に依存し、Data Virtualization エンジンの評価ではなく、Data Virtualization がソースにプッシュされる関数。

仮想手順（「CREATE VIRTUAL PROCEDURE」）

ストアドプロシージャーと同様に、これは Data Virtualization の Procedure 言語を使用して定義され、Data Virtualization エンジンで評価されます。

function/UDF("CREATE VIRTUAL FUNCTION")

Teiid 手順言語を使用して定義できるユーザー定義の関数や、Java クラスによる実装の定義はできます。UDF の Java コードの作成に関する詳細は、『 Translator Development Guide 』の「 **Support for user-defined functions(un-pushdown)**」を参照してください。



関数または手順 の詳細は、SQL 文法の BNF を参照してください。

変数引数

IN パラメーターだけを使用する代わりに、最後のオプション以外のパラメーターで VARIADIC を宣言して、手順が呼び出される際に 0 以上の回数を繰り返すことができます。

例： Vararg 手順

```
CREATE FOREIGN PROCEDURE proc (x integer, VARIADIC z integer)
    RETURNS (x string);
```

FUNCTION OPTIONS:（以下はよく知られており、その他のプロパティーは拡張メタデータとみなされます）

プロパティー	データタイプまたは許可される値	説明
UUID	string	一意識別子
NAMEINSOURCE	これがソース機能である場合、物理ソースの名前（上記の論理名と異なる場合）になります。	
アノテーション	string	関数/手順の説明
CATEGORY	string	関数カテゴリー

プロパティー	データタイプまたは許可される値	説明
DETERMINISM	NONDETERMINISTIC, COMMAND_DETERMINISTIC, SESSION_DETERMINISTIC, USER_DETERMINISTIC, VDB_DETERMINISTIC, DETERMINISTIC	仮想手順では使用されません
NULL-ON-NULL	'TRUE'|'FALSE'	
JAVA_CLASS	string	UDF の場合にメソッドを定義する Java クラス
JAVA_METHOD	string	上記で定義された UDF 実装の Java メソッド名
VARARGS	'TRUE'|'FALSE'	関数の最後の引数をいつでも 0 に繰り返すことができることを示します。デフォルトは false です。VARIADIC パラメーターを使用する方が適切です。
AGGREGATE	'TRUE'|'FALSE'	関数がユーザー定義の集約機能であることを示します。集約固有のプロパティーを以下に示します。

NULL-ON-NULL、VARARGS、およびすべての AGGREGATE プロパティーは、関数としてマークされたソースの手順で使用できる有効なリレーショナルデータベースメタデータプロパティーでもあることに注意してください。

ソース固有の関数に基づく FOREIGN 関数を作成することもできます。データソースによって提供される関数を使用する外部関数の作成に関する詳細は、『Translator Development guide』の「**Source supported functions**」を参照してください。

.AGGREGATE 関数のオプション

プロパティー	データタイプまたは許可される値	説明
分析	'TRUE'|'FALSE'	集約関数のウィンドウが必要であることを示します。デフォルト値は **false** です。
ALLOW-ORDERBY	'TRUE'|'FALSE'	集約関数が ORDER BY 句を使用できることを示します。デフォルト値は **false** です。

プロパティー	データタイプまたは許可される値	説明
ALLOWS-DISTINCT	'TRUE'|'FALSE'	aggregate 関数が **DISTINCT** キーワードを使用できることを示します。デフォルト値は **false** です。
DECOMPOSABLE	'TRUE'|'FALSE'	1 つの引数集約関数をデータのサブセットに対して agg（agg(x)）として破棄できることを示します。デフォルト値は **false** です。
USES-DISTINCT-ROWS	'TRUE'|'FALSE'	集約関数がすべての行ではなく、個別の行を効果的に使用することを示します。デフォルト値は **false** です。

Teiid 手順言語を使用して定義される Virtual Function は集約機能できないことに注意してください。

注記

JAR ライブラリーの指定: Teiid 手順の定義なしで UDF（仮想）関数を定義した場合は、Java での実装を反映する必要があります。Java ライブラリーを VDB への依存関係として設定する方法については、『Translator Development Guide』の「**Support for User-Defined Functions**」を参照してください。

PROCEDURE OPTIONS:（以下のオプションはよく知られており、定義されたその他のプロパティーは拡張メタデータとみなされます）

プロパティー	データタイプまたは許可される値	説明
UUID	string	一意識別子
NAMEINSOURCE	string	ソースの場合
アノテーション	string	手順の説明
UPDATECOUNT	int	この手順が基礎となるソースを更新する場合、更新数が >1 の場合に、実行用に XA プロトコルが適用されます。

例： 仮想手順の定義

```
CREATE VIRTUAL PROCEDURE CustomerActivity(customerid integer)
  RETURNS (name varchar(25), activitydate date, amount decimal)
  AS
```

```
BEGIN
  ...
END
```

仮想手順および仮想手順言語の詳細は、「仮想手順」および「 手順 言語」を参照 し てください。

例： 仮想機能の定義

```
CREATE VIRTUAL FUNCTION CustomerRank(customerid integer)
  RETURNS integer AS
  BEGIN
    DECLARE integer result;
    ...
    RETURN result;
  END
```

手順列は NOT NULL としてマークすることも、DEFAULT の値で指定することもできます。ソース手順でパラメーターをデフォルトに可能で、Data Virtualization でデフォルト値を指定しない場合は、パラメーターでエクステンションプロパティー teiid_rel:default_handling を省略するように設定される必要があります。

単一の RESULT パラメーターしか存在できず、**out** パラメーターである必要があります。RESULT パラメーターは、Stable 以外の RETURNS 型が 1 つ必要です。両方が宣言された場合は、例外が発生します。他方が正確ではありません。「RETURNS type」は、特に関数のための構文は短くなりますが、パラメーターの形式は追加のメタデータ（標準名、拡張メタデータ、戻り値テーブルの定義も定義）に便利です。

return パラメーターは、引数リストに表示される場所にかかわらず、ランタイム時に手順の最初のパラメーターとして扱われます。これは、"? = EXEC ..." 形式で割り当てが想定される Data Virtualization および JDBC 呼び出しセマンティクスと一致します。

.relational 拡張 OPTIONS:

プロパティー	データタイプまたは許可される値	説明
native-query	パラメーター化された文字列	関数と手順の両方に適用されます。標準の接頭辞形式ではなく、関数構文の代わりに括弧を使用します。詳細は「 Translators でパラメーター化 **可能なネイティブクエリー** 」を参照してください。
non-prepared	boolean	native-query オプションを使用して JDBC 手順に適用されます。true の場合、PreparedStatement はネイティブクエリーの実行には使用されません。

例： ネイティブクエリー

```
CREATE FOREIGN FUNCTION func (x integer, y integer)
  RETURNS integer OPTIONS ("teiid_rel:native-query" '$1 << $2');
```

例： ネイティブクエリーのシーケンス

```
CREATE FOREIGN FUNCTION seq_nextval ()
  RETURNS integer
  OPTIONS ("teiid_rel:native-query" 'seq.nextval');
```

ヒント

ソース関数表現を使用して、シーケンス機能を公開します。

拡張メタデータ

カスタムトランスフォーマーの場合、エクステンションメタデータを定義するとき、テーブル/ビュー/手順/列のプロパティーは必要なものになります。プロパティーに関連する内容を示す一貫性のある接頭辞を使用することが推奨されます。teiid_ で始まる接頭辞は、Data Virtualization で使用するために予約されます。プロパティーキーはランタイム API 経由でアクセスする場合は大文字と小文字を区別しませんが、SYS.PROPERTIES にアクセスする場合に大文字と小文字が区別されます。

警告

カスタムプレフィックスまたは namespace への SET NAMESPACE の使用が許可されなくなりました。

```
CREATE VIEW MyView (...)
  OPTIONS ("my-translator:mycustom-prop" 'anyvalue')
```

表2.2 組み込みプレフィックス

プレフィックス	説明
teiid_rel	リレーショナルデータベースの拡張。include 関数およびネイティブクエリーメタデータを使用
teiid_sf	Salesforce エクステンション。
teiid_mongo	MongoDB エクステンション
teiid_odata	OData エクステンション
teiid_accumulo	Accumulo 拡張
teiid_excel	Excel エクステンション
teiid_ldap	LDAP 拡張

プレフィックス	説明
teiid_rest	REST エクステンション
teiid_pi	PI データベースの拡張

2.3. ドメインの DDL メタデータ

ドメインは、特定のタイプ名の有効な値のセットを定義する単純なタイプ宣言です。データベースレベルでのみ作成できます。

ドメインの作成

```
CREATE DOMAIN <Domain name> [ AS ] <data type>
    [ [NOT] NULL ]
```

ドメイン名にはキーワード以外の識別子を使用できます。

データ型については「BNF」を参照してください。

ドメインを定義したら、列、パラメーターなどのデータ型として参照できます。

例： Virtual database DDL

```
CREATE DOMAIN mychar AS VARCHAR(1000);

CREATE VIRTUAL SCHEMA viewLayer;
SET SCHEMA viewLayer;
CREATE VIEW v1 (col1 mychar) as select 'value';
...
```

システムメタデータのクエリー時に、列のタイプがドメイン名として表示されます。

制限事項

データ型が想定される以下の場所でドメイン名が認識されない場合があります。

- 一時テーブルの作成
- 即時実行
- arraytable
- objecttable
- texttable
- xmltable

pg_attribute をクエリーすると、ODBC/pg メタデータにはドメイン名ではなく、ベースタイプの名前が表示されます。

第3章 SQL 互換性

Data Virtualization は、SQL-92 DML のほとんどすべての機能を提供します。SQL-99 以降の機能は、コミュニティーのニーズに基づいて常に追加されます。以下では、SQL の網羅的な説明は試行されず、SQL を Data Virtualization 内でどのように使用するかを説明します。Data Virtualization が受け入れる SQL の正確な形式に関する詳細は、BNF for SQL grammar を参照してください。

3.1. 識別子

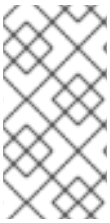
SQL コマンドには、テーブルと列への参照が含まれます。これらの参照は識別子の形式にあり、コマンドのコンテキストでテーブルと列を一意に識別します。すべてのクエリーは、仮想データベースまたは VDB のコンテキストで処理されます。情報は複数のソースでフェデレーションできるため、競合を回避するためにテーブルと列を何らかの方法でスコープする必要があります。このスコープは、各データソースまたはビューのセットに関する情報が含まれるスキーマによって提供されます。

完全修飾テーブルおよび列名の形式は以下のとおりです。識別子の個別の「パート」はピリオドで区切られます。

- TABLE: <schema_name>.<table_spec>
- COLUMN: <schema_name>.<table_spec>.<column_name>

構文ルール

- 識別子は英数字またはアンダースコア(_)文字で構成され、アルファベットで開始する必要があります。Unicode 文字は識別子で使用できます。
- 二重引用符の識別子にはあらゆるコンテンツがあります。二重引用符は、追加の二重引用符でエスケープされている場合に使用できます（例： **「some "" id」** など）。
- データソースは異なる方法でテーブルを整理し、一部の先頭のカタログ、スキーマ、またはユーザー情報を含むため、Data Virtualization ではテーブル仕様をドットで区切られたコンストラクトにすることができます。



注記

テーブル仕様にドット解決機能が含まれる場合、名前の任意の数の終了セグメントに対して部分的な名前が一致します。たとえば、完全修飾名 **vdbname."sourceschema.sourcetable"** を持つテーブルが部分的な名前 **ソーステーブル** にマッチします。

- 列、列エイリアス、およびスキーマにはドット(.)文字を含めることはできません。
- 引用符で囲まれていても、Data Virtualization で大文字と小文字が区別されません。

有効な完全修飾テーブル識別子の例は次のとおりです。

- MySchema.Portfolios
- "MySchema.Portfolios"
- MySchema.MyCatalog.dbo.Authors

有効な完全修飾列識別子の例は次のとおりです。

- MySchema.Portfolios.portfolioID
- "MySchema.Portfolios"."portfolioID"
- MySchema.MyCatalog.dbo.Authors.lastName

完全修飾識別子は常に SQL コマンドで使用できます。コマンドのコンテキストでは、結果となる名前があいまいである限り、部分的にまたは非修飾形式を使用することもできます。同じクエリーで異なる形式の修飾を混在させることができます。

ピリオド(.)文字を含むエイリアスを使用する場合は、エイリアス名が修飾名と同じ処理され、完全修飾オブジェクト名と競合する可能性があるという既知の問題です。

予約された単語

Data Virtualization の予約された語には、標準の SQL336 Foundation、SQL/MED、および SQL/XML の予約単語と、BIGINTEGER、BIGDECIMAL、MAKEDEP などの Data Virtualization 固有の単語が含まれます。予約されている単語の詳細は、SQL 文法の「BNF for SQL 文法」の 「**Reserved Keywords and Reserved Keywords for Future Use** 」の セクションを参照してください。

3.2. OPERATOR の優先順位

Data Virtualization は、優先順位が低い演算子よりも先に、優先順位の高い演算子を解析して評価します。優先順位が等しい演算子は left-associative（左から右）です。以下の表では、演算子の優先順位を high から low に示します。

Operator	説明
[]	array element reference
+,-	正の値式/負の値式
*,/	多重/障害障害
+,-	追加/サブアクション
||	concat
基準	詳細は「条件」を参照し て ください。

3.3. 式

識別子、リテラル、関数を式に統合できます。式は、SELECT、FROM（結合基準）、WHERE、GROUP BY、HAVING、または ORDER BY を含むほぼすべてのキーワードを含むクエリーで使用できます。

Data Virtualization では、以下のタイプの式を使用できます。

- 列識別子
- リテラル

- 集約関数
- ウィンドウ機能
- ケースと検索
- scalar サブキュー
- パラメーターの参照
- 配列
- 基準
- スカラー関数

3.3.1. 列識別子

列識別子は、SELECT ステートメントの出力列、INSERT および UPDATE ステートメントの列と、WHERE および FROM 句で使用される基準を指定するために使用されます。これらは GROUP BY、HAVING、および ORDER BY 句でも使用されます。列識別子の構文は、上記の Identifiers セクションで定義されています。

3.3.2. リテラル

リテラル値は固定値を表します。これらは 'standard' のデータ型のいずれかです。データ型の詳細は、「データタイプ」を参照 し てください。

構文ルール

- 整数値には、値（整数、long、または大きな整数）を保持するのに十分なデータ型が割り当てられます。
- 浮動小数点の値は常に二重に解析されます。
- キーワード「null」は、存在しない値または不明な値を表すために使用され、本質的に入力されません。多くの場合、null リテラル値にはコンテキストに基づいて暗黙的なタイプが割り当てられます。たとえば、関数 '5 + null' では、null 値には、値 '5' のタイプに一致する「integer」タイプが割り当てられます。暗黙的なコンテキストを持たないクエリーの SELECT 句で使用される null リテラルは 'string' タイプに割り当てられます。

簡単なリテラル値の例は次のとおりです。

```
'abc'
```

例： エスケープされた単一ティック

```
'isn''t true'
```

```
5
```

例： Scientific 表記

```
-37.75e01
```

-

例： 完全数値型 BigDecimal

```
100.0
```

```
true
```

```
false
```

例： Unicode 文字

```
'\u0027'
```

例： バイナリー

```
X'0F0A'
```

日付/タイムリテラルは、JDBC Escaped リテラル構文 のいずれかを使用することができます。

例： Date リテラル

```
{d'...'}
```

例： Time リテラル

```
{t'...'}
```

例： Timestamp リテラル

```
{ts'...'}
```

または、ANSI キーワード構文です。

例： Date リテラル

```
DATE '...'
```

例： Time リテラル

```
TIME '...'
```

例： Timestamp リテラル

```
TIMESTAMP '...'
```

いずれの方法でも、式の文字列リテラル値の部分は定義された形式（「yyyy-MM-dd」、日付の場合は "hh:mm:ss"、および "yyyy-MM-dd[hh:mm:ss[.fff...]]]"（タイムスタンプ）に従う必要があります。

集約関数

集約関数は、明示的なグループまたは暗黙的な GROUP BY によって生成されるグループからの値セットを取得し、グループから計算された単一のスカラー値を返します。

Data Virtualization で以下の集約機能を使用できます。

COUNT(*)

グループの値（null および重複を含む）をカウントします。整数を返します。大きな数が計算されると例外が発生します。

COUNT(x)

グループの値の数をカウントします（null を除く）。整数を返します。大きな数が計算されると例外が発生します。

COUNT_BIG(*)

グループの値（null および重複を含む）をカウントします。long - より大きい数が計算されると例外がスローされます。

COUNT_BIG(x)

グループの値の数をカウントします（null を除く）。long - より大きい数が計算されると例外がスローされます。

SUM(x)

グループの値の合計（null を除く）

AVG(x)

グループの値（null を除く）の平均。

MIN(x)

グループの最小値（null を除く）。

MAX(x)

グループの最大値（null を除く）。

ANY(x)/SOME(x)

グループのいずれかの値が TRUE の場合（null を除く）、TRUE を返します。

EVERY(x)

グループのすべての値が TRUE の場合（null を除く）、TRUE を返します。

VAR_POP(x)

biased variance(excluding null)logically equals（sum(x^2)- sum(x)^2/count(x)）/count(x); returns a double; null if count = 0.

VAR_SAMP(x)

Example variance(null)logically equals（sum(x^2)- sum(x)^2/count(x)）/（count(x)- 1）; returns a double; null if count < 2.

STDDEV_POP(x)

標準偏差（null を除く）は SQRT（VAR_POP(x)）に論理的に等しくなります。

STDDEV_SAMP(x)

標準偏差の例（null を除く）は SQRT（VAR_SAMP(x)）に論理的に等しくなります。

TEXTAGG(expression [as name], ... [DELIMITER char] [QUOTE char | NO QUOTE] [HEADER] [ENCODING id] [ORDER BY ...])

グループの各行のすべての式の CSV テキスト集計。DELIMITER が指定されていない場合、デフォルトでコンマ(,)が区切り文字として使用されます。null 以外の値はすべて引用符で囲まれます。二重引用符(")はデフォルトの引用符文字です。QUOTE を使用して別の値を指定するか、または値の引

用なしで NO QUOTE を指定します。HEADER が指定されている場合、結果にはヘッダーの行が最初の行として含まれます。グループ内に行がない場合でも、ヘッダー行が表示されます。この集約は Blob を返します。

```
TEXTAGG(col1, col2 as name DELIMITER '|' HEADER ORDER BY col1)
```

- XMLAGG(xml_expr [ORDER BY ...])- グループ内のすべての XML 式の XML 連結（null を除く）ORDER BY 句はエイリアス名を参照したり、位置の順序を使用したりできません。
- JSONARRAY_AGG(x [ORDER BY ...])- null 値を含む Clob として JSON 配列の結果を作成します。ORDER BY 句はエイリアス名を参照したり、位置の順序を使用したりできません。詳細は、「 JSONARRAY 関数 」を参照してください。

例： 整数値式

```
jsonArray_Agg(col1 order by col1 nulls first)
```

戻る可能性がある

```
[null,null,1,2,3]
```

- STRING_AGG(x, delim)- 区切り文字 delim を使用して x の連結から誤った結果を作成します。いずれの引数も null の場合、値は連結されません。どちらの引数も文字(string/clob)またはバイナリー（varbinary、Blob）となり、結果はそれぞれ CLOB または BLOB になります。DISTINCT および ORDER BY は STRING_AGG で許可されます。

例： 文字列の集約式

```
string_agg(col1, ',' ORDER BY col1 ASC)
```

戻る可能性がある

```
'a,b,c'
```

- LIST_AGG(x [, delim])WITHIN GROUP(ORDER BY ...)- Oracle と同じ構文を使用する STRING_AGG の形式。ここで、 X は文字列に変換できる任意の型にすることができます。**delim** 値（指定されている場合）はリテラルでなければならず、**ORDER BY** 値が必要です。これは、同等の **string_agg** 式の解析エイリアスのみです。

例： 集約式の一覧表示

```
listagg(col1, ',') WITHIN GROUP (ORDER BY col1 ASC)
```

戻る可能性がある

```
'a,b,c'
```

- ARRAY_AGG(x [ORDER BY ...])- 式 x に一致するベースタイプでアレイを作成します。ORDER BY 句はエイリアス名を参照したり、位置の順序を使用したりできません。
- agg([DISTINCT|ALL] arg ... [ORDER BY ...])- A user defined aggregate function.

構文ルール

- 一部の集約関数には式の前にキーワード「DISTINCT」が含まれている可能性があり、重複式の値を無視するべきであることを示します。DISTINCT は COUNT(*)で許可されず、MIN または MAX（結果の変更）では意味がないため、COUNT、SUM、および AVG で使用できます。
- 集約関数は、クエリー式に干渉しない FROM、GROUP BY、または WHERE 句で使用できません。
- 集約関数は、クエリー式に干渉しない別の集約関数内で入れ子にすることはできません。
- 集約関数は、他の関数内で入れ子にすることができます。
- すべての集約関数は、オプションの FILTER 句の形式の FILTER 句を取ることができます。

```
FILTER ( WHERE condition )
```

条件は、サブクエリーまたは相関変数を含まないブール値式にすることができます。フィルターは、グループ化操作の前に各行に対して論理的に評価されます。false の場合、集約関数は指定の行の値を累積しません。

集計についての詳細は、GROUP BY または HAVING のセクションを参照してください。

3.3.3. ウィンドウ機能

Data Virtualization は ANSI SQL336 ウィンドウ機能を提供します。window 関数は、**GROUP BY** 句を使用せずに、集約関数を結果セットのサブセットに適用できます。window 関数は集約関数と似ていますが、**OVER** 句またはウィンドウ仕様を使用する必要があります。

使用方法

```
  aggregate [FILTER (WHERE ...)] OVER ( [partition] [ORDER BY ...] [frame] )
| FIRST_VALUE(val) OVER ( [partition] [ORDER BY ...] [frame] )
| LAST_VALUE(val) OVER ( [partition] [ORDER BY ...] [frame] )
| analytical OVER ( [partition] [ORDER BY ...] )

partition := PARTITION BY expression [, expression]*

frame := range_or_rows extent

range_or_rows := RANGE | ROWS

extent :=
    frameBound
  | BETWEEN frameBound AND frameBound

frameBound :=
    UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | n PRECEDING
  | n FOLLOWING
  | CURRENT ROW
```

上記の構文では、**集約** は任意の 集約関数 を参照できます。キーワードは、以下の分析関数 ROW_NUMBER、RANK、DENSE_RANK、PERCENT_RANK、CUME_DIST に存在します。

FIRST_VALUE、LAST_VALUE、LEAD、LAG、NTH_VALUE、および NTILE analytical 関数もあります。詳細は、「collect al functions definitions」 を参照してください。

構文ルール

- ウィンドウ関数は、クエリー式の SELECT 句および ORDER BY 句でのみ表示されます。
- ウィンドウ関数は、互いにネストできません。
- 式によるパーティション設定や順序にサブクィーター参照を含めることはできません。
- ウィンドウ表示時に aggregate ORDER BY 句を使用できません。
- ウィンドウ仕様 ORDER BY 句はエイリアス名を参照したり、位置の順番を使用したりできません。
- ウィンドウ仕様が順序付けされている場合は、ウィンドウアグリゲートで DISTINCT を使用しない場合があります。
- Analytical value 関数は DISTINCT を使用せず、ウィンドウの仕様で順序付けを使用する必要があります。
- RANGE または ROWS では、ORDER BY 句を指定する必要があります。指定がない場合はデフォルトのフレームが RANGE UNBOUNDED PRECEDING です。指定しないと、デフォルトは CURRENT ROW になります。開始前と終了の組み合わせは許可されていません。たとえば、UNBOUNDED FOLLOWING は開始前も終了として許可される UNBOUNDED PRECEDING は許可されていません。
- RANGE は n PRECEDING または n FOLLOWING を使用することはできません。

分析関数定義

ランク付け機能

- RANK（）: 1 で始まる各パーティション内の一意の順序付け値ごとに数字を割り当てます。これにより、次のランクが前の行の数と同じになります。
- DENSE_RANK（）: 次のランクが連続するように、各パーティション内の一意の順序付け値ごとに数字を割り当てます。
- PERCENT_RANK（）:(RANK - 1)/(RC - 1)でコンピュートされます。RC はパーティションの合計行数です。
- CUME_DIST（）: PR / RC として計算されます。PR はピアを含む行のランクであり、RC はパーティションの総行数です。
  デフォルトでは、すべての値が整数になります。大きな値が必要な場合は例外が発生します。システムの org.teiid.longRanks を使用して、代わりに RANK、DENSE_RANK、および ROW_NUMBER は long 値を返します。

値関数

- FIRST_VALUE(val)- 指定の順序付けを持つウィンドウフレームの最初の値を返します。
- LAST_VALUE(val)- 指定の順序付けのあるウィンドウフレームで最後に確認された値を返します。

- LEAD(val [, offset [, default]]): 現在の行の前にあるオフセット行であるウインドウで順序付けされた値にアクセスします。このような行がない場合は、デフォルト値が返されます。指定されていない場合、オフセットは 1 で、デフォルトは null です。
- LAG(val [, offset [, offset [, default]]): 現在の行の背後にあるオフセット行であるウインドウで順序付けされた値にアクセスします。このような行がない場合は、デフォルト値が返されます。指定されていない場合、オフセットは 1 で、デフォルトは null です。
- NTH_VALUE(val, n)- ウィンドウフレームの nth val を返します。インデックスは 0 を超える必要があります。そのような値が存在しない場合は、null を返します。

行値関数

- ROW_NUMBER（）- **1** で始まるパーティションの各行に数字を割り当てます。
- NTILE(n)- パーティションを、最大 **1** でサイズ異なる n タイルに分割します。大きなタイルは、最初に順番に作成されます。**N** は **0** より大きい値である必要があります。

処理

ウィンドウ関数は、SELECT 句から出力を作成する直前に論理的に処理されます。GROUP BY 句が存在する場合、ウィンドウ関数はネストされた集約を使用できます。ウィンドウ関数の有無により、出力の順序には保証されません。SELECT ステートメントには、順序が予測できるように ORDER BY 句が必要です。

注記

OVER 句の ORDER BY は、上レベルの ORDER BY と同じルールのプッシュダウンと処理ルールに従います。通常、null 処理はエンジンとプッシュダウン処理によって異なる可能性があるため、NULLS FIRST/LAST を指定する必要があります。また、異なるデフォルトの動作が異なる場合に、ソート動作を制御するシステムプロパティーも参照してください。

Data Virtualization は、同じウィンドウ仕様を持つすべてのウィンドウ関数を処理します。通常、一意のウィンドウ仕様ごとに SELECT 句に送信される行値を完全に渡す必要があります。ウィンドウの指定ごとに、この値は PARTITION BY 句に従ってグループ化されます。PARTITION BY 句が指定されていない場合、入力全体が単一のパーティションとして処理されます。

出力値のフレームは、analytical 関数または **ROWS/RANGE** 句の定義に基づいて決定されます。デフォルトのフレームは **RANGE UNBOUNDED PRECEDING です。RANGE** は行とそのピアを一緒に計算します。**ROWS** は、全行で計算します。**ROW_NUMBER** などのほとんどの分析機能には暗黙の **RANGE/ROWS** があります。そのため、異なる関数を指定できません。たとえば、**ROW_NUMBER（）OVER(order)** は代わりに **count(*)OVER(order ROWS UNBOUNDED PRECEDING AND CURRENT ROW)とし** て表現できます。したがって、ピアの数に関係なく、すべての行に異なる値を割り当てます。

例： ウィンドウ結果

```
SELECT name, salary, max(salary) over (partition by name) as max_sal,
        rank() over (order by salary) as rank, dense_rank() over (order by salary) as dense_rank,
        row_number() over (order by salary) as row_num FROM employees
```

name	salary	max_sal	ランク	dense_rank	row_num
John	100000	100000	2	2	2
Henry	50000	50000	5	4	5
John	60000	100000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

3.3.4. ケースと検索

Data Virtualization で、スカラー式に条件ロジックを含めるには、以下の 2 つの形式の CASE 式を使用できます。

- **CASE <expr> (WHEN <expr> THEN <expr>)+ [336E expr] END**
- **CASE(WHEN <criteria> THEN <expr>)+ [gitopsE expr] END**

各フォームは条件付きロジックに基づいて出力を許可します。最初のフォームは最初の式で始まり、値が一致するまで WHEN 式を評価し、THEN 式を出力します。WHEN が一致しないと、ELSE 式が出力されます。WHEN が一致せず、ELSE が指定されていない場合には、null リテラル値が出力されます。2 つ目のフォーム（検索されたケース式）は、評価する任意の基準を指定する WHEN 句を検索します。いずれかの条件が true と評価されると、THEN 式が評価され、出力されます。WHEN が true の場合は、評価または NULL が出力され、存在しない場合は NULL が出力されます。

ケースステートメントの例

```
SELECT CASE columnA WHEN '10' THEN 'ten' WHEN '20' THEN 'twenty' END AS myExample

SELECT CASE WHEN columnA = '10' THEN 'ten' WHEN columnA = '20' THEN 'twenty' END AS
myExample
```

3.3.5. scalar サブキュー

Subqueries は、SELECT、WHERE、または HAVING 句でのみ単一のスカラー値を生成するために使用できます。スカラーサブクエリーは SELECT 句に単一の列を持つ必要があり、0 または 1 行のいずれかを返す必要があります。行が返されなければ、null は scalar サブクエリーの値として返されます。他の種類のサブキューに関する情報は、Subqueries を参照してください。

3.3.6. パラメーターの参照

パラメーターは **?** 記号を使用して指定されます。パラメーターは、JDBC の **PreparedStatement** または **CallableStatements** でのみ使用できます。各パラメーターは、JDBC API の 1 ベースのインデックスによって指定された値にリンクされます。

3.3.7. 配列

アレイの値は、オプションの末尾のコンマまたは明示的な ARRAY コンストラクターで式リストを括弧で囲むことで構築できます。

例： 空のアレイ

```
()
(,)
ARRAY[]
```

例： 単一要素アレイ

```
(expr,)
ARRAY[expr]
```

注記

パーサーが、単純なネストされた式ではなく、括弧が付いた配列として認識するには、末尾のコンマが必要です。

例： 一般的なアレイ構文

```
(expr, expr ... [,])
ARRAY[expr, ...]
```

アレイ内のすべての要素が同じタイプである場合、アレイは一致するベースタイプを持ちます。要素タイプが異なる場合、配列ベースタイプは object になります。

array 要素参照は以下の形式を取ります。

```
array_expr[index_expr]
```

index_expr は整数値に解決される必要があります。この構文は、**array_get** システム関数と事実上同じで、1 ベースのインデックスを想定しています。

3.4. 基準

条件には、以下のいずれかの項目を使用できます。

- true または false に評価される述語。
- 条件を組み合わせる論理条件（AND、OR、not）。
- 型ブール値の値式。

用途

```
criteria AND|OR criteria
```

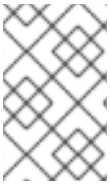
```
NOT criteria
```

```
(criteria)
```

expression (=|<>|!=|<|>|<=|>=) (expression|((ANY|ALL|SOME) subquery|(array_expression)))

expression IS [NOT] DISTINCT FROM expression

IS DISTINCT FROM は null 値を同等とみなし、UNKNOWN 値を生成しません。



注記

オプティマイザーは **IS DISTINCT FROM** を処理するように調整されていないため、プッシュされていない結合述語で使用すると、作成される計画も通常の比較は実行されません。

expression [NOT] IS NULL

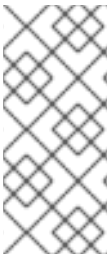
expression [NOT] IN (expression [,expression]*)|subquery

expression [NOT] LIKE pattern [ESCAPE char]

LIKE は、指定の文字列パターンに対して文字列式と一致します。パターンには **%** を含めることで任意の数の文字に一致させ、任意の 1 文字に一致する _ を指定できます。エスケープ文字を使用すると、一致文字 **%** および _ をエスケープできます。

expression [NOT] SIMILAR TO pattern [ESCAPE char]

SIMILAR TO は、LIKE と標準の正規表現構文との間の相互です。.* および . ではなく、**%** と _ が使用され ます。



注記

Data Virtualization は、**SIMILAR TO** パターン値を網羅的に検証しません。代わりに、パターンは同等の正規表現に変換されます。**SIMILAR TO** を使用する場合は、一般的な正規表現機能に依存しないでください。追加の機能が必要な場合は、**LIKE_REGEX** を使用します。Data Virtualization では SQL のプッシュダウン述語を処理する機能が制限されているため、literal 以外のパターンは使用しないでください。

expression [NOT] LIKE_REGEX pattern

照合には、**LIKE_REGEX** を標準の正規表現構文と共に使用できます。これは、エスケープ文字が使用されなくなった点で **SIMILAR TO** および **LIKE** とは異なります。\ は正規表現の標準的なエスケープメカニズムで、**%'** と _ には特別な意味がありません。ランタイムエンジンは正規表現の JRE 実装を使用します。詳細は、java.util.regex.Pattern クラスを参照してください。

注記

データ仮想化は、**LIKE_REGEX** パターン値を網羅的に検証しません。SQL 仕様で指定されていない JRE のみの正規表現機能を使用できます。また、すべてのソースが同じ正規表現のフレーバーまたは拡張機能を使用できる訳ではありません。プッシュダウンの状況では、使用するパターンが Data Virtualization と該当するすべてのソースで同じ意味を持つように注意してください。

> EXISTS (subquery)

> expression [NOT] BETWEEN minExpression AND maxExpression

Data Virtualization は、**BETWEEN** を同等の形式 **式である minExpression AND expression 00:00:0 maxExpression** に変換します。

> expression

式 のタイプが Boolean になります。

構文ルール

- 最も低いものから高い優先順位の順番は、compare、not、AND、OR です。
- 括弧でネストされた基準は、親基準を評価する前に論理的に評価されます。

有効な基準の例をいくつか示します。

- **(balance > 2500.0)**
- **100*(50 - x)/(25 - y) > z**
- **concat（areaCode,concat('-',phone)）LIKE '313361'**

NULL 値の比較

null 値は不明な値を表します。null 値と比較すると **不明な** と評価されます。これは、使用されて **いない** 場合でも true にすることはできません。

条件の優先順位

Data Virtualization は、優先順位が低い条件よりも先に、優先順位の高い条件を解析して評価します。優先順位が等しい条件は left-associative です。以下の表は、high から low までの条件の一覧です。

状態	説明
SQL 演算子	式を参照してください。
EXISTS, LIKE_REGEX, BETWEEN, IN, IN, IS NULL, IS DISTINCT, <,GITOPS, >, >=, =, <>	Comparison
NOT	否定
AND	接続詞
あるいは	disjunction

注記

先行さを防ぐために、パーサーは考えられる基準シーケンスをすべて受け入れません。たとえば、left-associative 解析では、最初に **= を認識してから一般的な値式 を探すため、= b は null** になりません。**B は** 有効な一般的な値式ではありません。 したがって、ネストを使用する必要があります。たとえば、**(a = b)は null です**。 ルール の解析に関する詳細は、「SQL 文法の BNF 」を参照してください。

3.5. スカラー関数

Data Virtualization は、ビルトインのスカラー機能を多数提供します。詳細は、「 DML コマンド 」および「 データタイプ 」を参照してください。さらに、Data Virtualization はユーザー定義の関数または UDF の機能を提供します。UDF の追加に関する詳細は、『Translator Development Guide』の「 **User-defined functions** 」を参照してください。UDF を追加したら、他の関数を呼び出すのと同じ方法で呼び出すことができます。

3.5.1. 数値関数

数値関数は数値を返します（整数、long、float、ダブル、大きな整数、大きい 10 進数）。通常、これらは文字列を取りますが、数値は入力として取ります。

機能	定義	データタイプ制約
+ - * /	標準の数値演算子	x in {integer, long, float, double, biginteger, bigdecimal}, return type is x [a]
ABS(x)	x の絶対値	上記の標準的な数値演算子を参照してください。
ACOS(x)	arc cosine of x	x in {double, bigdecimal}, return type is double
ASIN(x)	x のアーティファクト	x in {double, bigdecimal}, return type is double
ATAN(x)	x の ArC tangent	x in {double, bigdecimal}, return type is double
ATAN2(x,y)	x および y の ArC tangent	X, y in {double, bigdecimal}, return type is double
CEILING(x)	x の Ceiling	X in {double, float}, return type is double
COS(x)	cosine of x	x in {double, bigdecimal}, return type is double
COT(x)	x の Cotangent of x	x in {double, bigdecimal}, return type is double

機能	定義	データタイプ制約
DEGREES(x)	x degrees を radians に変換する	x in {double, bigdecimal}, return type is double
EXP(x)	e^x	X in {double, float}, return type is double
FLOOR(x)	x floor of x	X in {double, float}, return type is double
FORMATBIGDECIMAL(x, y)	y 形式を使用した x のフォーマット	X は大きい 10 進数で、y は文字列で、文字列を返します。
FORMATBIGINTEGER(x, y)	y 形式を使用した x のフォーマット	X は大型整数で、y は文字列で、文字列を返します。
FORMATDOUBLE(x, y)	y 形式を使用した x のフォーマット	X は double で、y は文字列を返します。文字列を返します。
FORMATFLOAT(x, y)	y 形式を使用した x のフォーマット	X は浮動小数点、y は文字列で、文字列を返します。
FORMATINTEGER(x, y)	y 形式を使用した x のフォーマット	X は整数で、y は文字列です。文字列を返します。
FORMATLONG(x, y)	y 形式を使用した x のフォーマット	X は long、y は文字列で、文字列を返します。
LOG(x)	x（ベース e）の自然なログ	X in {double, float}, return type is double
LOG10(x)	Log of x(base 10)	X in {double, float}, return type is double
MOD(x, y)	modulus（x / y のメイン元）	x in {integer, long, float, double, biginteger, bigdecimal}, return type is x と同じです。
PARSEBIGDECIMAL(x, y)	y 形式を使用した x の解析	X、y は文字列で、大きい 10 進数を返します。
PARSEBIGINTEGER(x, y)	y 形式を使用した x の解析	X、y は文字列で、大きな整数を返します。
PARSEDOUBLE(x, y)	y 形式を使用した x の解析	X、y は文字列で、ダブルを返します。

機能	定義	データタイプ制約
PARSEFLOAT(x, y)	y 形式を使用した x の解析	X、y は文字列で、浮動小数点を返します。
PARSEINTEGER(x, y)	y 形式を使用した x の解析	X、y は文字列で、整数を返します。
PARSELONG(x, y)	y 形式を使用した x の解析	X、y は文字列で、long を返します。
PI()	Pi の値	戻り値は double です
POWER(x,y)	X から y の電源へ	x in {double, bigdecimal, biginteger}, return is the same type as x
RADIANS(x)	x radians をレベルに変換する	x in {double, bigdecimal}, return type is double
RAND()	必要時にクエリーで確立されたジェネレーターやシステムクロックで初期化されたジェネレーターを使用して、乱数を返します。	double を返します。
RAND(x)	x でシードされた新しいジェネレーターを使用して乱数を返します。通常、これは初期化クエリーで呼び出される必要があります。これは、Data Virtualization RAND 関数によって返されるランダムな値にのみ影響し、ソースによって評価される RAND 関数からの値は影響を与えません。	X は整数で、二重を返します。
ROUND(x,y)	y の位置に丸めた x を丸めます。負の値は、y を小数点の左側にある場所を示します。	x in {integer, float, double, bigdecimal} y は整数で、戻り値は x と同じタイプです。
SIGN(x)	x > 0 の場合は 1、x = 0 の場合は 0（x < 0 の場合は -1）	x in {integer, long, float, double, biginteger, bigdecimal}, return type is integer
SIN(x)	x の Sine 値	x in {double, bigdecimal}, return type is double
SQRT(x)	x の平方ルート	x in {long, double, bigdecimal}, return type is double

機能	定義	データタイプ制約
TAN(x)	x の Tangent	x in {double, bigdecimal}, return type is double
BITAND(x, y)	x と y のビット単位の AND	X, y in {integer}, return type is integer
HadoopTOR(x, y)	x および y のビット単位の OR	X, y in {integer}, return type is integer
giveTXOR(x, y)	x および y のビット単位の XOR	X, y in {integer}, return type is integer
BITNOT(x)	x をビットにしない	{integer} の x 戻り値の型は整数です。

[a] 非あいまいな関数関数の精度とスケールは、Java のものと一致します。大きな 10 進数操作の結果は、単位を除き Java に一致します。ただし、max(16, dividend.scale + divisor.precision + 1)の推奨スケールを使用し、スケールを max(dividend.scale, normalized scale)に設定して末尾のゼロを削除します。

文字列からの数値データ型の解析

Data Virtualization は、文字列の数字を解析するのに使用できる関数のセットを提供します。各文字列に文字列の形式を指定する必要があります。これらの関数は、java.text.DecimalFormat クラスによって確立された規則を使用して、これらの関数で使用することのできる形式を定義します。Sun Java の URL で Sun Java Web サイトにアクセスすると、このクラスが数値の文字列形式を定義する方法が確認できます。

たとえば、これらの関数呼び出しを java.text.DecimalFormat 規則に準拠するフォーマット文字列とともに使用して、文字列を解析し、必要なデータタイプを返すことができます。

入力文字列	関数呼び出しのフォーマット文字列	出力値	出力データタイプ
'$25.30'	parseDouble(cost, '$,0.00;($,0.00)')	25.3	double
'25%'	parseFloat(percent, ',#0%')	25	float
'2,534.1'	parseFloat(total, ',0.;-,0.')	2534.1	float
'1.234E3'	parseLong(amt, '0.###E0')	1234	Long

入力文字列	関数呼び出しのフォーマット文字列	出力値	出力データタイプ
'1,234,567'	parseInteger(total, ',0;-,0')	1234567	integer

数値データ型を文字列としてフォーマットする

Data Virtualization は、数値のデータタイプを文字列に変換するために使用できる関数のセットを提供します。各文字列にフォーマットを指定する必要があります。これらの関数は、java.text.DecimalFormat クラス内で確立された規則を使用して、これらの関数で使用することのできる形式を定義します。Sun Java の URL で Sun Java Web サイトにアクセスすると、このクラスが数値の文字列形式を定義する方法が確認できます。

たとえば、java.text.DecimalFormat 規則に準拠するフォーマット文字列を使用して、これらの関数呼び出しを使用し、数値のデータ型を文字列にフォーマットできます。

入力値	入力データタイプ	関数呼び出しのフォーマット文字列	出力文字列
25.3	double	formatDouble(cost, '$,0.00;($,0.00)')	'$25.30'
25	float	formatFloat(percent, ',#0%')	'25%'
2534.1	float	formatFloat(total, ',0.;-,0.')	'2,534.1'
1234	Long	formatLong(amt, '0.###E0')	'1.234E3'
1234567	integer	formatInteger(total, ',0;-,0')	'1,234,567'

3.5.2. 文字列関数

文字列関数は一般的に文字列を入力として取得し、文字列を出力として返します。

指定されていない場合、以下の表のすべての引数および戻り値の型は文字列であり、すべてのインデックスは 1 をベースとします。0 インデックスは文字列の開始前に考慮されます。

機能	定義	データタイプ制約
x || y	Concatenation Operator	X,y in {string, clob}, return type is string or character large object(CLOB)

機能	定義	データタイプ制約
ASCII(x)	x の左文字の ASCII 値を指定します[1]。入力として空の文字列は **null** を返します。	戻り値の型は整数です
CHR(x) CHAR(x)	ASCII 値の **x** [a] に文字[1] を指定します。	x in {integer} [1] エンジンの ASCII 関数および CHR 関数の実装では、文字は UCS2 値のみに制限されます。プッシュダウンでは、文字値のソースがコード 255 以外の文字値の整合性はほとんどありません。
CONCAT(x, y)	ANSI セマンティクスで x および y を連結します。x や y が null の場合は、null を返します。	X, y（{string} の場合）
CONCAT2(x, y)	非ANSI null セマンティクスで x および y を連結します。x および y が null の場合は、null を返します。x または y のみが null の場合は、他の値を返します。	X, y（{string} の場合）
ENDSWITH(x, y)	y が x で終わるかどうかを確認します。x または y が null の場合は、null を返します。	X, y in {string}, returns boolean
INITCAP(x)	文字列 x 大文字、およびその他の小文字で、各単語の最初の文字を作成します。	x in {string}
INSERT(str1, start, length, str2)	string2 を string1 に挿入します。	str1 in {string}, start in {integer}, length in {integer}, str2 in {string}
LCASE(x)	x の小文字	x in {string}
LEFT(x, y)	get left y characters of x	X in {string}, y in {integer}, return string
LENGTH(x) CHAR_LENGTH(x) CHARACTER_LENGTH(x)	x の長さ	戻り値の型は整数です
LOCATE(x, y)POSITION(x IN y)	y の開始時に y の位置を見つけます。	X in {string}, y in {string}, return integer

機能	定義	データタイプ制約
LOCATE(x, y, z)	z から、y で x の位置を見つけます。	X in {string}, y in {string}, z in {integer}, return integer
LPAD(x, y)	y の長さの空白があるパッド入力文字列 x。	X in {string}, y in {integer}, return string
LPAD(x, y, z)	文字 z を使用して y の長さのパッド入力文字列 x。	X in {string}, y in {string}, z in {character}, return string
LTRIM(x)	空の文字の左のトリム。	X in {string}, return string
QUERYSTRING(path [, expr [AS name] ...])	指定のパスに追加される適切にエンコードされたクエリー文字列を返します。null 値式は省略され、次に null パスは '' として処理されます。名前は列参照式では任意となります。たとえば、**QUERYSTRING('path', 'value' as "&x", ' & ' as y, null as z)returns 'path?%26x=value&y=%20%233620'**です。	{string} のパス（例： name）は識別子です。
REPEAT(str1,instances)	string1 を指定した回数で繰り返します。	{string} の str1、{integer} のインスタンス文字列を返します。
RIGHT(x, y)	get right y characters of x	X in {string}, y in {integer}, return string
RPAD(input string x, pad length y)	y の長さの右側のスペースを含むパッド入力文字列 x	X in {string}, y in {integer}, return string
RPAD(x, y, z)	文字 z を使用した y の長さのパッド入力文字列 x	X in {string}, y in {string}, z in {character}, return string
RTRIM(x)	空の文字の右側のトリミング	X は文字列で、返される文字列です。
SPACE(x)	スペース文字 x の回数を繰り返す	X は整数で、返される文字列です。
SUBSTRING(x, y)SUBSTRING(x FROM y)	[b] Get substring from x, from the position y to the end of x	{integer} の Y
SUBSTRING(x, y, z)SUBSTRING(x FROM y FOR z)	[b] 長さ z で位置 y から x のサブ文字列を取得	Y, z in {integer}

機能	定義	データタイプ制約
TRANSLATE(x, y, z)	y の各文字を同じ位置の z の文字に置き換えることで、変換文字列 x を使用します。	x in {string}
TRIM([[LEADING|TRAILING|BOTH] [x] FROM] y)	先頭、末尾、または文字 x の文字列の y の末尾をトリミングします。LEADING/TRAILING/BOTH が指定されていない場合は、BOTH が使用されます。トリム文字 x を指定しないと、空白のスペース「 が使用されます。	x in {character}, y in {string}
UCASE(x)	x の大文字	x in {string}
UNESCAPE(x)	x のエスケープされていないバージョン。エスケープシーケンスは \b - バックスペース、\t - タブ、\n - 改行、\f - フォームフィード、\r - キャリッジリターンです。エスケープ文字の後に他の文字が表示されると、その文字は出力に表示され、エスケープ文字は無視されます。	x in {string}

[a] これらの関数で使用される非 ASCII 範囲文字または整数は、関数が評価される場所(Data Virtualization vs. source)に応じて異なる結果または例外を生成する場合があります。Data Virtualization は、UTF16 値で実行される Java のデフォルト int を使用して、char および char を int 変換に使用します。

[b] ソースに依存するサブ文字列関数には、負の引数/長さの引数に対する一貫した動作や、/length 引数のバインド外に関連する動作がありません。Data Virtualization のデフォルト動作は次のとおりです。

- from 値がバインド不足、または長さが 0 未満の場合は null 値を返します。
- インデックスのゼロは 1 と同じように有効です。
- インデックスから負の値は、最初に文字列の最後からカウントされます。

ただし、一部のソースは **null** ではなく空の文字列を返すことができ、一部のソースは負のインデックスと互換性がありません。

TO_CHARS

指定されたエンコーディングでバイナリー大規模なオブジェクト(BLOB)から CLOB を返します。

```
TO_CHARS(x, encoding [, wellformed])
```

BASE64、HEX、UTF-8-BOM、および組み込みの Java Charset 名はエンコーディング [b] の有効な値です。x は BLOB で、エンコーディングは文字列で、適切にフォームはブール値で、CLOB を返します。デフォルトでは、2 つの引数形式はデフォルトで wellformed=true に設定されます。適切な情報が

false である場合、変換関数は即座に結果を検証し、不適切な文字や不正な入力で例外が発生します。

TO_BYTES

指定されたエンコーディングで CLOB から BLOB を返します。

```
TO_BYTES(x, encoding [, wellformed])
```

BASE64、HEX、UTF-8-BOM、および組み込み Java Charset 名はエンコーディングの有効な値です [b]。CLOB の x は文字列で、エンコードはブール値で、BLOB を返します。デフォルトでは、2 つの引数形式はデフォルトで wellformed=true に設定されます。適切な情報が false である場合、変換関数は即座に結果を検証し、不適切な文字や不正な入力で例外が発生します。適切ではない文字が true の場合、文字セットのデフォルト置換文字に置き換えられます。BASE64 や HEX などのバイナリー形式は、適切なパラメーターに関係なく、正確性の有無がチェックされます。

[b] Charset 名の詳細は、Charset docs を参照してください。

REPLACE

指定の文字列のすべての出現箇所を別の文字列に置き換えます。

```
REPLACE(x, y, z)
```

x で y のすべての出現箇所を z に置き換えます。x、y、z は文字列で、戻り値は文字列です。

REGEXP_REPLACE

特定のパターンの 1 つまたはすべての出現箇所を別の文字列に置き換えます。

```
REGEXP_REPLACE(str, pattern, sub [, flags])
```

str 内のパターンの 1 つ以上を sub に置き換えます。すべての引数は文字列で、戻り値は文字列です。

pattern パラメーターは有効な Java 正規表現であることが想定されます。

flags 引数は、以下の意味を持つ有効なフラグのいずれかを連結できます。

フラグ	名前	意味
g	グローバル	最初のものだけではなく、すべての出現箇所を置き換えます。
m	複数行	複数の行で照合します。
i	大文字と小文字を区別しない	ケースの機密性なしで一致します。

使用方法

以下は、グローバルおよび大文字と小文字を区別しないオプションを使用して「xxbye Wxx」を返します。

regexp_replace の例

```
regexp_replace('Goodbye World', '[g-o].', 'x', 'gi')
```

3.5.3. 日付および時刻の関数

日付、時刻、またはタイムスタンプで日時関数が戻るか、または操作します。

日時関数は、java.text.SimpleDateFormat クラス内で確立された規則を使用して、これらの関数と共に使用できる形式を定義します。このクラスがどのように形式を定義するかについては、SimpleDateFormat の Javadocs を参照してください。

機能	定義	データタイプ制約
CURDATE() CURRENT_DATE[()]	現在の日付を返します。ユーザーコマンド内のすべての呼び出しに同じ値が返されます。	日付を返します。
CURTIME()	現在の時間を返します。ユーザーコマンド内のすべての呼び出しに同じ値を返します。CURRENT_TIME も参照してください。	時間を返します。
NOW()	現在のタイムスタンプ（ミリ秒の精度あり）を返します。ユーザーコマンドまたは手順命令のすべての呼び出しに同じ値を返します。CURRENT_TIMESTAMP も参照してください。	タイムスタンプを返します。
CURRENT_TIME[(precision)]	現在の時間を返します。ユーザーコマンド内のすべての呼び出しに同じ値を返します。Data Virtualization の時間タイプは少秒をトラッキングしないため、精度の引数が事実上無視されます。精度がない場合、CURTIME（）と同じです。	時間を返します。
CURRENT_TIMESTAMP[(precision)]	現在のタイムスタンプ（ミリ秒の精度あり）を返します。ユーザーコマンドまたは手順命令で、同じ精度を持つすべての呼び出しに同じ値を返します。精度がない場合、NOW（）と同じです。現在のタイムスタンプには、デフォルトでミリ秒の精度しかないため、精度を 3 よりも大きく設定すると効果はありません。	タイムスタンプを返します。
DAYNAME(x)	デフォルトのロケールで日名を返します。	X in {date, timestamp}, returns string

機能	定義	データタイプ制約
DAYOFMONTH(x)	戻る日	X in {date, timestamp}（整数を返します）
DAYOFWEEK(x)	曜日を返します（日： 平日=1, Saturday=7）	X in {date, timestamp}（整数を返します）
DAYOFYEAR(x)	日付番号を年で返します。	X in {date, timestamp}（整数を返します）
EPOCH(x)	マイクロ秒の精度を持つ unix エポックからの経過時間（秒単位）	X in {date, timestamp}, returns double
EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND|QUARTER|EPOCH FROM x)	日付値 x から指定のフィールド値を返します。関連する YEAR、MONTH、DAYOFMONTH、HOUR、MINUTE、SECOND、QUARTER、EPOCH 関数と同じ結果を生成します。SQL 仕様では、TIMEZONE_HOUR および TIMEZONE_MINUTE を抽出ターゲットとして許可します。Data Virtualization では、日付の値はすべてサーバーのタイムゾーンになります。	X in {date, time, timestamp}, epoch returns double, other return integer
FORMATDATE(x, y)	y 形式を使用した日付 x の形式。	X は date、y は文字列で、文字列を返します。
FORMATTIME(x, y)	y 形式を使用した時間 x のフォーマット。	X は時間で、y は文字列を返します。文字列を返します。
FORMATTIMESTAMP(x, y)	y 形式を使用したタイムスタンプ x の形式。	X はタイムスタンプで、y は文字列を返します。文字列を返します。
FROM_MILLIS (millis)	指定のミリ秒の Timestamp 値を返します。	長い UTC タイムスタンプ（ミリ秒単位）
FROM_UNIXTIME (unix_timestamp)	デフォルトの yyyy/mm/dd hh:mm:ss で、Unix タイムスタンプを String 値として返します。	長い Unix タイムスタンプ（秒単位）
hOUR(x)	時間を返します（通常の 24 時間形式）。	X in {time, timestamp}（整数を返します）
MINUTE(x)	分を返します。	X in {time, timestamp}（整数を返します）

機能	定義	データタイプ制約
MODIFYTIMEZONE（timestamp、startTimeZone、endTimeZone）	開始タイムゾーンと終了タイムゾーン間で異なる値に調整された受信タイムスタンプに基づいてタイムスタンプを返します。 サーバーが GMT-6 にある場合は、Fixzone({ts '2006-01-10 04:00:00.0'},'GMT-7', 'GMT-8') はタイムスタンプ {ts '2006-01-10 05:00:00.0'} を返します。この値は、GMT-7 と GMT-8 の相違点を補正するために 1 時間前に調整されています。	startTimeZone および endTimeZone は文字列で、タイムスタンプを返します。
MODIFYTIMEZONE(timestamp, endTimeZone)	modifytimezone(timestamp, startTimeZone, endTimeZone)と同様にタイムスタンプを返しますが、startTimeZone がサーバープロセスと同じであると仮定します。	タイムスタンプはタイムスタンプで、endTimeZone は文字列で、タイムスタンプを返します。
MONTH(x)	戻る月。	X in {date, timestamp}（整数を返します）
MONTHNAME(x)	デフォルトのロケールで月を返します。	X in {date, timestamp}, returns string
PARSEDATE(x, y)	y 形式を使用して、x から date を解析します。	X, y in {string}, returns date
PARSETIME(x, y)	y 形式を使用して、x からの時間を解析します。	X, y in {string}, returns time
PARSETIMESTAMP(x,y)	y 形式を使用して、x からのタイムスタンプを解析します。	X, y in {string}, returns timestamp
QUARTER(x)	送金を返します。	X in {date, timestamp}（整数を返します）
SECOND(x)	戻り値の秒数。	X in {time, timestamp}（整数を返します）
TIMESTAMPCREATE(date, time)	日付と時刻からタイムスタンプを作成します。	日付 in {date}, time in {time}, return timestamp
TO_MILLIS (timestamp)	UTC タイムスタンプをミリ秒単位で返します。	タイムスタンプ値

機能	定義	データタイプ制約
UNIX_TIMESTAMP (unix_timestamp)	長い Unix タイムスタンプ（秒単位）を返します。	yyyy/mm/dd hh:mm:ss のデフォルト形式の unix_timestamp 文字列
WEEK(x)	1 ~53 年で週を返します。カスタマイズ情報は、『管理者ガイド』の「**システムプロパティー**」を参照してください。	X in {date, timestamp}（整数を返します）
YEAR(x)	4 桁の年を返します。	X in {date, timestamp}（整数を返します）

Timestampadd

指定の間隔をタイムスタンプに追加します。

構文

```
TIMESTAMPADD(interval, count, timestamp)
```

引数

名前	説明
interval	日時間隔の単位は、以下のキーワードのいずれかになります。 • SQL_TSI_FRAC_SECOND - 分秒（秒） • SQL_TSI_SECOND - seconds • SQL_TSI_MINUTE - 分 • SQL_TSI_HOUR - 時間 • SQL_TSI_DAY - days • SQL_TSI_WEEK - 初日に日曜日を使用する週 • SQL_TSI_MONTH - months • SQL_TSI_QUARTER - 最初の月が 3 ヶ月という月（3 カ月） • SQL_TSI_YEAR - 年
count	タイムスタンプに追加するユニットの長い数または整数数。負の値は、その単位数に減算します。Long 値は TIMESTAMPDIFF とのシンボリメトに許可されますが、有効な範囲は整数値に制限されます。
timestamp	日時式。

例

```
SELECT TIMESTAMPADD(SQL_TSI_MONTH, 12,'2016-10-10')
SELECT TIMESTAMPADD(SQL_TSI_SECOND, 12,'2016-10-10 23:59:59')
```

Timestampdiff

2 つのタイムスタンプが長い値を返す間に超過した日付部分の間隔を計算します。

構文

```
TIMESTAMPDIFF(interval, startTime, endTime)
```

引数

名前	説明
interval	日時間隔の単位。Timestampadd で使用されるキーワードと同じです。
startTime	日時式。
endTime	日時式。

例

```
SELECT TIMESTAMPDIFF(SQL_TSI_MONTH,'2000-01-02','2016-10-10')
SELECT TIMESTAMPDIFF(SQL_TSI_SECOND,'2000-01-02 00:00:00','2016-10-10 23:59:59')
SELECT TIMESTAMPDIFF(SQL_TSI_FRAC_SECOND,'2000-01-02 00:00:00.0','2016-10-10
23:59:59.999999')
```

注記

(endTime > startTime)の場合は、負の値以外の数値が返されます。(endTime < startTime)の場合は、正の値以外の数値が返されます。日付の部分の差異は、タイムスタンプの近づく方法に関係なくカウントされます。たとえば、「2000-01-02 00:00:00.0」は「2000-01-01 23:59:59.999999」よりも 1 時間前に考慮されます。

互換性の問題

- SQL では、Timestampdiff は通常整数を返します。ただし、Data Virtualization の実装は長く返されます。プッシュされたタイムスタンプdiff から整数値が整数の範囲から外すことが予想されると例外が発生することがあります。
- Teiid 8.2 以前のバージョンでのタイムスタンプの diff の実装は、正規間隔の概算数（1 年で 365 日、月に 30 日、30 日単位など）に基づいた値を返します。たとえば、2013-03-24 から 2013-04-01 の月間における差異は 0 でしたが、相互にまたがる日付部分を基にしたのは 1 です。後方互換性に関する情報は、『Adminstrator 's Guide』の 「 **System Properties** 」を参照してください。

文字列からの日付データタイプの解析

データ仮想化は、'19970101' や '31/1/1996' など、異なる形式で表示される日付が含まれる文字列を暗黙的に変換しません。ただし、次のセクションで説明する parseDate、parseTime、および parseTimestamp 関数を使用して、異なる形式の文字列を明示的に適切なデータタイプに変換できます。これらの関数は、java.text.SimpleDateFormat クラス内で確立された規則を使用して、これらの関数で使用することのできる形式を定義します。このクラスによる日時の文字列形式の定義方法に関する詳細は、「 Javadocs for SimpleDateFormat 」を参照してください。フォーマット文字列は、お使いの Java のデフォルトロケールに固有のものであることに注意してください。

たとえば、これらの関数呼び出しに java.text.SimpleDateFormat 規則に準拠するフォーマット文字列を指定して、文字列を解析し、必要なデータタイプを返すことができます。

文字列	文字列を解析する関数呼び出し
'1997010'	parseDate(myDateString, 'yyyyMMdd')
'31/1/1996'	parseDate(myDateString, 'dd''/''MM''/''yyyy')
'22:08:56 CST'	parseTime (myTime, 'HH:mm:ss z')
'03.24.2003 at 06:14:32'	parseTimestamp(myTimestamp, 'MM.dd.yyyy''at''hh:mm:ss')

タイムゾーンの指定

タイムゾーンは複数の形式で指定できます。「Eastern 標準時間」の EST などの一般的な省略は許可されますが、曖昧になってしまう可能性があるため、使用は推奨されません。あいまいなタイムゾーンは、continent または ocean/largest city 形式で定義されます。たとえば、America/New_York、America/Buenos_Aires、または Europe/London です。さらに、GMT オフセット GMT[+/-]HH:MM でカスタムのタイムゾーンを指定できます。

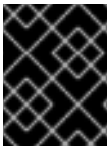
例： GMT-05:00

3.5.4. 型変換関数

クエリー内で、CONVERT または CAST キーワードを使用してデータタイプ間で変換できます。詳細は、「 タイプ変換」を参照してください。

機能	定義
CONVERT(x, type)	x を type に変換します。type は Data Virtualization Base タイプです。
CAST(x AS type)	x を type に変換します。type は Data Virtualization Base タイプです。

これらの関数は構文以外と同じです。CAST は標準の SQL 構文で、CONVERT は標準の JDBC/ODBC 構文です。



重要

length、精度、スケールなど、型で指定したオプションは事実上無視されます。ランタイムは、単純にあるオブジェクトタイプから別のオブジェクトタイプに変換されます。

3.5.5. 選択関数

選択関数では、値のいずれかの特徴に基づいて 2 つの値から選択できます。

機能	定義	データタイプ制約
COALESCE(x,y+)	null 以外の最初のパラメーターを返します。	X およびすべての y のタイプには、互換性があります。
IFNULL(x,y)	x が null の場合は y を返します。それ以外の場合は x を返します。	X、y、および戻り値の型は同じタイプである必要がありますが、任意のタイプにすることができます。
NVL(x,y)	x が null の場合は y を返します。それ以外の場合は x を返します。	X、y、および戻り値の型は同じタイプである必要がありますが、任意のタイプにすることができます。
NULLIF(param1, param2)	(param1 = param2)、null else param1 と同等です。	param1 および param2 は互換性を持つタイプである必要があります。

IFNULL および NVL は相互のエイリアスです。これらは同じ機能です。

3.5.6. デコード関数

デコード機能により、Data Virtualization サーバーが結果セット内の列の内容を確認し、変更またはデコードし、アプリケーションが結果をより適切に使用できるようにできます。

機能	定義	データタイプ制約
DECODESTRING(x, y [, z])	任意の区切り文字 **z** と y の文字列を使用して列 **x** をデコードし、デコードされた列を文字列として返します。区切り文字が指定されていない場合は、コンマ(**,**)が使用されます。**Y** の形式は **SearchDelimResultDelimSearchDelimResult[DelimDefault]** です。一致するものがない場合は、指定または **x** の場合は Default を返します。**非推奨。代わりに CASE 式を使用してください。**	すべての文字列

機能	定義	データタイプ制約
DECODEINTEGER(x, y [, z])	任意の区切り文字 z と **y** の文字列を使用して列 **x** をデコードし、デコードされた列を整数として返します。区切り文字が指定されていない場合は、コンマ(**,**)が使用されます。**Y** の形式は **SearchDelimResultDelimSearchDelimResult[DelimDefault]** です。一致するものがない場合は、指定または **x** の場合は Default を返します。**非推奨。代わりに CASE 式を使用してください。**	すべての文字列パラメーター、整数を返す

各関数呼び出しには、以下の引数が含まれます。

1. **X** はデコード操作の入力値です。通常、これは列名です。
2. Y は、入力値と出力値の **区** 切ったセットが含まれるリテラル文字列です。
3. **z** は、これらのメソッドのオプションのパラメーターで、y で指定された文字列がどの区切り文字であるかを指定できます。

たとえば、アプリケーションは、**IS_IN_STOCK** という列が含まれる **PARTS** というテーブルにクエリーを実行します。これには、アプリケーションが処理するために整数に変更する必要のあるブール値が含まれます。この場合、**DECODEINTEGER** 関数を使用してブール値を整数に変更できます。

```
SELECT DECODEINTEGER(IS_IN_STOCK, 'false, 0, true, 1') FROM PartsSupplier.PARTS;
```

Data Virtualization システムが結果セットで **false** の値に遭遇すると、値は 0 に置き換えられます。

整数を使用する代わりに、アプリケーションに文字列の値が必要な場合は、**DECODESTRING** 関数を使用して必要な文字列値を返すことができます。

```
SELECT DECODESTRING(IS_IN_STOCK, 'false, no, true, yes, null') FROM PartsSupplier.PARTS;
```

このサンプルクエリーは、2 つの入力/出力値のペアに加えて、列に前述の入力値が含まれない場合に使用する値を提供します。IS_IN_STOCK 列の行に true または false が含まれていない場合、Data Virtualization サーバーは null を結果セットに挿入します。

これらの DECODE 関数を使用する場合は、文字列内で必要な入力/出力値のペアをいくつでも提供できます。デフォルトでは、Data Virtualization システムはコンマ区切りの区切り文字を想定しますが、関数呼び出しに 3 番目のパラメーターを追加して別の区切り文字を指定することができます。

```
SELECT DECODESTRING(IS_IN_STOCK, 'false:no:true:yes:null',':') FROM PartsSupplier.PARTS;
```

DECODE 文字列のキーワード **null** を入力値または出力値のいずれかとして使用し、null 値を表すことができます。ただし、リテラル文字列 **null** を入力または出力値として使用する必要がある場合（つまり null という単語が列に表示され、null 値ではない）は、単語を引用符（ **「null」** ）に記述できます。

```
SELECT DECODESTRING( IS_IN_STOCK, 'null,no,"null",no,nil,no,false,no,true,yes' ) FROM
PartsSupplier.PARTS;
```

DECODE 関数が列に一致する出力値が見つからない場合で、デフォルト値を指定しない場合、DECODE 関数はその列にある Data Virtualization サーバーが元の値を返します。

3.5.7. lookup 関数

Lookup 関数は、参照テーブルから値へのアクセスを迅速化する手段を提供します。Lookup 関数はすべてのキーを自動的にキャッシュし、参照されるテーブルの関数で宣言された列のペアを返します。同じキーと戻り値の列を使用して、同じテーブルに対して後続のルックアップを行うと、キャッシュされた値を使用します。このキャッシュは、ビジネス用語でコードまたは参照テーブルとしても知られる、ルックアップテーブルを使用するクエリーに対する応答時間を加速します。

```
LOOKUP(codeTable, returnColumn, keyColumn, keyValue)
```

lookup テーブル codeTable で、keyColumn に値 keyValue がある行を見つけ、一致する keyValue が見つからない場合は、関連付けられた returnColumn 値または null を返します。codeTable はターゲットテーブルの完全修飾名である文字列リテラルである必要があります。returnColumn および keyColumn も文字列リテラルで、codeTable の対応する列名と一致させる必要があります。keyValue には、keyColumn のデータタイプに一致する必要がある任意の式を指定できます。return datatype matches that of returnColumn.

国コード検索

```
lookup('ISOCountryCodes', 'CountryCode', 'CountryName', 'United States')
```

ISOCountryCodes テーブルを使用して、国名を ISO 国コードに翻訳します。1 列の CountryName は keyColumn を表します。次の列の CountryCode は、国の ISO コードを含む returnColumn を表します。そのため、ここで lookup 機能を使用すると CountryName が提供されます。これは、上記の 'United States' として示され、応答に CountryCode の値を想定しています。

この関数を codeTable、returnColumn、および keyColumn の組み合わせで呼び出すと、Data Virtualization System は結果をキャッシュします。Data Virtualization System は、この検索テーブルに後でアクセスされるすべてのセッションで、このキャッシュをすべてのクエリーに使用します。通常、更新対象となるデータの lookup 機能を使用することや、行ベースのセキュリティーや列マスク効果を含むセッション/ユーザー固有の場合があります。Lookup 関数でのキャッシュに関する詳細は、『キャッシング ガイド』を参照 してください。

keyColumn には、対応する codeTable の一意の値が含まれることが予想されます。keyColumn に重複値が含まれる場合、例外が発生します。

3.5.8. システム機能

システム機能は、クエリー内から Data Virtualization システムの情報へのアクセスを提供します。

COMMANDPAYLOAD

コマンドペイロードから文字列を取得します。コマンドペイロードが指定されていない場合は null になります。コマンドペイロードは、クエリーごとに Data Virtualization JDBC API エクステンションの **TeiidStatement.setPayload** メソッドで設定されます。

COMMANDPAYLOAD([key])

key パラメーターを指定すると、コマンドペイロードオブジェクトは java.util.Properties オブジェクトにキャストされ、キーに対応するプロパティー値が返されます。キーが指定されていない場合、戻り値はコマンドペイロードオブジェクト toString 値になります。

key、戻り値は文字列です。

ENV

システムプロパティーを取得します。この関数は名前がなく、レガシーの互換性のために含まれています。より適切に名前付き関数については、ENV_VAR および SYS_PROP を参照してください。

ENV(key)

ENV('KEY')を使用して呼び出し、値を文字列として返します。ex: ENV('PATH')。渡されたキーで値が見つからなかった場合、小文字を使ったキーのバージョンも試行されます。この機能は、実行時にシステムプロパティーを設定することはできますが、決定論的に処理されます。

ENV_VAR

環境変数を取得します。

ENV_VAR(key)

ENV_VAR('KEY')を使用して呼び出し、値を文字列として返します。ex: ENV_VAR('USER')この機能の動作は、ケースの機密性に関してプラットフォームに依存します。この機能は、実行時に環境変数を変更することはできますが、決定論的に処理されます。

SYS_PROP

システムプロパティーを取得します。

SYS_PROP(key)

SYS_PROP('KEY')を使用して呼び出し、値を文字列として返します。ex: SYS_PROP('USER')。この機能は実行時にシステムプロパティーを変更できる場合でも、決定論的として処理されます。

NODE_ID

ノード ID を取得します。通常、Data Virtualization に埋め込まれない「jboss.node.name」のシステムプロパティーの値を取得します。

NODE_ID()

戻り値は string です。

SESSION_ID

現在のセッション ID の文字列形式を取得します。

SESSION_ID()

戻り値は string です。

USER

クエリーを実行しているユーザーの名前を取得します。

USER([includeSecurityDomain])

includeSecurityDomain はブール値で、戻り値は string です。

includeSecurityDomain が省略されたり、true であった場合、ユーザー名は @security-domain が付加された状態で返されます。

CURRENT_DATABASE

データベースのカタログ名を取得します。VDB 名は常にカタログ名になります。

CURRENT_DATABASE()

戻り値は string です。

TEIID_SESSION_GET

セッション変数を取得します。

TEIID_SESSION_GET(name)

name は文字列で、戻り値はオブジェクトです。

null 名は null 値を返します。通常、get wrap を CAST でラップして必要なタイプに変換します。

TEIID_SESSION_SET

セッション変数を設定します。

TEIID_SESSION_SET(name, value)

name は文字列で、value はオブジェクトで、戻り値はオブジェクトです。

キーまたは null の以前の値が返されます。セットは現在のトランザクションには影響がなく、コミット/ロールバックの影響を受けません。

GENERATED_KEY

生成されたキーを返すこのセッションの最後の insert ステートメントで生成されたキーから列値を取得します。

通常、この機能は手順の範囲内でのみ使用され、挿入から生成されたキー値を決定します。すべてのソースが生成されたキーを返すわけではないため、すべての挿入が生成されたキーを提供するわけではありません。

GENERATED_KEY()

戻り値は long です。

最後に生成されたキーの最初のコラムを long 値として返します。そのような生成されたキーがない場合は null が返されます。

GENERATED_KEY(column_name)`

column_name は文字列です。戻り値は type オブジェクトです。

複数の生成された列または long 以外のタイプがある場合は、より一般的な **GENERATED_KEY** を使用できます。このような生成されたキーや一致するキー列がない場合は、null を返します。

3.5.9. XML 関数

XML 関数は、XML データを操作する機能を提供します。詳細は、「JSON 関数 の **JSONTOXML** 」を参照してください。

サンプル用のデータのサンプル

XML 関数で提供される例では、以下の表構造を使用します。

```
TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    CustomerName varchar(25),
    ContactName varchar(25)
    Address varchar(50),
    City varchar(25),
    PostalCode varchar(25),
    Country varchar(25),
);
```

データを使用

customerid	CustomerName	ContactName	アドレス	City	PostalCode	国
87	Wartian Herku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	フィンランド
88	Wellington Importadora	親の親	Rua do Mercado, 12	Resende	08737-363	ブラジル
89	white CloverMarks	karl Jablonski	305 - 14th Ave.S. Suite 3B	Seattle	98128	USA

XMLCAST

XML へ/からキャストします。

```
XMLCAST(expression AS type)
```

式または型は XML である必要があります。戻り値は、型として **入力** されます。これは、**XMLTABLE** が必要なランタイムタイプに値を変換するために使用する機能と同じですが、**XMLCAST** は配列型のターゲットとは機能しません。

XMLCOMMENT

XML コメントを返します。

```
XMLCOMMENT(comment)
```

comment は文字列です。戻り値は XML です。

XMLCONCAT

指定の XML 型を連結した XML を返します。

```
XMLCONCAT(content [, content]*)
```

コンテンツは XML です。戻り値は XML です。

値が null の場合は無視されます。すべての値が null の場合は、null を返します。

2 つ以上の XML フラグメントを連結します。

```
SELECT XMLCONCAT(
        XMLELEMENT("name", CustomerName),
        XMLPARSE(CONTENT '<a>b</a>' WELLFORMED)
      )
FROM   Customer c
WHERE  c.CustomerID = 87;

============================================================
<name>Wartian Herkku</name><a>b</a>
```

XMLELEMENT

指定の名前と内容を含む XML 要素を返します。

```
XMLELEMENT([NAME] name [, <NSP>] [, <ATTR>][, content]*)

ATTR:=XMLATTRIBUTES(exp [AS name] [, exp [AS name]]*)

NSP:=XMLNAMESPACES((uri AS prefix | DEFAULT uri | NO DEFAULT))+
```

コンテンツの値が XML 以外のタイプの場合は、親要素に追加されるときにエスケープされます。Null コンテンツの値は無視されます。XML またはコンテンツの文字列値の空白は保持されますが、コンテンツ値の間には空白は追加されません。

XMLNAMESPACES は namespace 情報を提供します。NO DEFAULT は、デフォルトの名前空間を null uri - xmlns="" と同等です。DEFAULT または DEFAULT の名前空間項目を 1 つだけ指定できます。名前空間の接頭辞 xmlns と xml は予約されています。

属性名が指定されていない場合は、式が列参照である必要があります。その場合、属性名は列名になります。null 属性値は無視されます。

name、prefix は識別子です。uri は文字列リテラルです。コンテンツはどのタイプでも指定できます。戻り値は XML です。戻り値はドキュメントが想定される場所での使用に有効です。

簡単な例

```
SELECT XMLELEMENT("name", CustomerName)
FROM   Customer c
WHERE  c.CustomerID = 87;

============================================================
<name>Wartian Herkku</name>
```

複数の列

```
SELECT XMLELEMENT("customer",
        XMLELEMENT("name", c.CustomerName),
        XMLELEMENT("contact", c.ContactName))
FROM   Customer c
```

```
WHERE  c.CustomerID = 87;

============================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

属性としての列

```
SELECT XMLELEMENT("customer",
        XMLELEMENT("name", c.CustomerName,
          XMLATTRIBUTES(
               "contact" as c.ContactName,
               "id" as c.CustomerID
          )
        )
      )
FROM   Customer c
WHERE  c.CustomerID = 87;

============================================================
<customer><name contact="Pirkko Koskitalo" id="87">Wartian Herkku</name></customer>
```

XMLFOREST

各コンテンツアイテムの XML 要素の連結を返します。

```
XMLFOREST(content [AS name] [, <NSP>] [, content [AS name]]*)
```

NSP - XMLNAMESPACES の定義については、「XML 関数 の **XMLELEMENT** 」を参照してください。

name は識別子です。コンテンツには任意のタイプを指定できます。戻り値は XML です。

コンテンツアイテムに名前が指定されていない場合は、式が列参照である必要があります。この場合、要素名は部分的にエスケープされた列名のバージョンになります。

XMLFOREST を使用して、複数の XMLELEMENTS の宣言を簡素化できます。XMLFOREST 関数を使用すると、一度に複数の列を処理できます。

例

```
SELECT XMLELEMENT("customer",
        XMLFOREST(
          c.CustomerName AS "name",
          c.ContactName AS "contact"
        ))
FROM   Customer c
WHERE  c.CustomerID = 87;

============================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

XMLAGG

XMLAGG は XML 要素のコレクションを取得し、集約された XML ドキュメントを返す集約関数です。

```
XMLAGG(xml)
```

上記の例の XML 要素では、条件に一致する行が複数ある場合に、カスタマーテーブルの各行が XML の行を生成します。有効な XML を生成しますが、root 要素がないため、適切に形成されません。XMLAGG を使用して修正できます。

例

```
SELECT XMLELEMENT("customers",
       XMLAGG(
         XMLELEMENT("customer",
           XMLFOREST(
             c.CustomerName AS "name",
             c.ContactName AS "contact"
           )))
FROM   Customer c


============================================================
<customers>
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
<customer><name>Wellington Importadora</name><contact>Paula Parente</contact></customer>
<customer><name>White Clover Markets</name><contact>Karl Jablonski</contact></customer>
</customers>
```

XMLPARSE

文字列値式の XML 型表現を返します。

```
XMLPARSE((DOCUMENT|CONTENT) expr [WELLFORMED])
```

{string, clob, blob, varbinary} の exprの戻り値は XML です。

DOCUMENT が指定されている場合、式には単一のルート要素が必要であり、XML 宣言が含まれる場合とそうでない場合があります。

WELLFORMED を指定すると検証は省略されます。これは、すでに有効なことが認識される CLOB および BLOB に特に便利です。

```
SELECT XMLPARSE(CONTENT '<customer><name>Wartian Herkku</name><contact>Pirkko
Koskitalo</contact></customer>' WELLFORMED);

Will return a SQLXML with contents
============================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact></customer>
```

XMLPI

XML 処理命令を返します。

```
XMLPI([NAME] name [, content])
```

name は識別子です。Content は文字列です。戻り値は XML です。

XMLQUERY

指定の xquery を評価する XML 結果を返します。

```
XMLQUERY([<NSP>] xquery [<PASSING>] [(NULL|EMPTY) ON EMPTY]]

PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

NSP - XMLNAMESPACES の定義については、「**XMLELEMENT** in XML 関数」を参照してください。

また、名前空間は xquery Prolog で直接宣言することもできます。

オプションの PASSING 句は、名前のないコンテキスト項目と、グローバル変数値という名前を提供するために使用されます。xquery がコンテキスト項目を使用し、指定がない場合は例外が発生します。指定できるのは 1 つのコンテキスト項目のみで、XML 型でなければなりません。コンテキスト以外の XML 以外のパス値はすべて適切な XML 型に変換されます。コンテキストアイテムが null と評価されると、null が返されます。

ON EMPTY 句は、評価シーケンスが空の場合に結果を指定するために使用されます。デフォルトである EMPTY ON EMPTY は、空の XML 結果を返します。NULL ON EMPTY は null 結果を返します。

XQuery（文字列）。戻り値は XML です。

XMLQUERY は SQL/XML の検出仕様の一部です。

詳細は、「XMLTABLE in FROM clause」を参照してください。

注記

「XQuery の最適化」も参照してください。

XMLEXISTS

指定の xquery を評価して空でないシーケンスが返されると true を返します。

```
XMLEXISTS([<NSP>] xquery [<PASSING>]]

PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

NSP - XMLNAMESPACES の定義については、「**XMLELEMENT** in XML 関数」を参照してください。

また、名前空間は xquery Prolog で直接宣言することもできます。

オプションの PASSING 句は、名前のないコンテキスト項目と、グローバル変数値という名前を提供するために使用されます。xquery がコンテキスト項目を使用し、指定がない場合は例外が発生します。指定できるのは 1 つのコンテキスト項目のみで、XML 型でなければなりません。コンテキスト以外の XML 以外のパス値はすべて適切な XML 型に変換されます。コンテキストアイテムが null と評価されると、null/Unknown が返されます。

XQuery（文字列）。戻り値はブール値です。

XMLEXISTS は SQL/XML Hadoop 仕様の一部です。

注記

「XQuery の最適化」も参照してください。

XMLSERIALIZE

XML 式の文字型表現を返します。

```
XMLSERIALIZE([(DOCUMENT|CONTENT)] xml [AS datatype] [ENCODING enc] [VERSION ver]
[(INCLUDING|EXCLUDING) XMLDECLARATION])
```

戻り値はデータタイプと一致します。 データタイプが指定されていない場合、clob が想定されます。

タイプは文字（string、varchar、clob）またはバイナリー（blob、varbinar）にすることができます。CONTENT がデフォルトです。DOCUMENT を指定し、XML が有効なドキュメントまたはフラグメントでない場合は、例外が発生します。

エンコーディングの暗号は識別子として指定されます。文字のシリアライズではエンコーディングを指定できません。 version ver は文字列リテラルとして指定されます。特定の XMLDECLARATION が指定されていない場合は、UTF-8/UTF-16 またはバージョン 1.0 以外のドキュメントシリアライゼーションを実行する場合や、基礎となる XML に宣言がある場合のみ宣言が含まれます。 CONTENT がシリアライズされる場合は、値がドキュメントまたは要素でない場合に宣言は省略されます。

FE FF および XML 宣言の適切なバイト順序マークを含む、UTF-16 に XML の BLOB を生成する以下の例を参照してください。

バイナリーシリアライゼーションの例

```
XMLSERIALIZE(DOCUMENT value AS BLOB ENCODING "UTF-16" INCLUDING
XMLDECLARATION)
```

XMLTEXT

XML テキストを返します。

```
XMLTEXT(text)
```

text は文字列です。戻り値は XML です。

XSLTRANSFORM

XSL スタイルシートを指定されたドキュメントに適用します。

```
XSLTRANSFORM(doc, xsl)
```

doc, XSL in {string, clob, xml}戻り値は clob です。

いずれの引数も null の場合、結果は null になります。

XPATHVALUE

XPATH 式をドキュメントに適用し、最初に一致した結果の文字列値を返します。結果および XQuery をより制御するには、XMLQUERY 関数を使用します。詳細は「XML 関数 の **XMLQUERY** 」を参照してください。

```
XPATHVALUE(doc, xpath)
```

{string, clob, blob, xml}. xpath のドキュメントは文字列です。戻り値は文字列です。

テキスト以外のノードと一致すると、依然としてすべての子テキストノードを含む文字列の結果が生成されます。単一要素が xsi:nil でマークされた場合は、null を返します。

入力ドキュメントで名前空間を使用する場合は、名前空間を無視する XPATH を指定する必要がある場合があります。

xpathValue Ignoring Namespaces のサンプル XML

```
<?xml version="1.0" ?>
  <ns1:return xmlns:ns1="http://com.test.ws/exampleWebService">Hello<x> World</x></return>
```

関数：

xpathValue Ignoring Namespaces のサンプル

```
xpathValue(value, '/*[local-name()="return"]')
```

Hello Worldの結果

例： フラットデータ構造からの階層 XML の生成

以下の表とそのコンテンツを使用

```
Table {
 x string,
 y integer
}
```

以下のような XML を生成する場合は ['a', 1], ['a', 2], ['b', 3], ['b', 4] などのデータ

```
<root>
   <x>
      a
      <y>1</y>
      <y>2</y>
   </x>
   <x>
      b
      <y>3</y>
      <y>4</y>
   </x>
</root>
```

以下のように Data Virtualization の SQL ステートメントを使用します。

```
select xmlelement(name "root", xmlagg(p))
   from (select xmlelement(name "x", x, xmlagg(xmlelement(name "y", y)) as p from tbl group by x)) as
v
```

その他の例は、http://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql.phpを参照してください。

3.5.10. JSON 関数

JSON 関数は、JSON (JavaScript Object Notation)データを操作する機能を提供します。

サンプル用のデータのサンプル

XML 関数で提供される例は、以下のテーブル構造を使用します。

```
TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    CustomerName varchar(25),
    ContactName varchar(25)
    Address varchar(50),
    City varchar(25),
    PostalCode varchar(25),
    Country varchar(25),
);
```

データを使用

customerid	CustomerName	ContactName	アドレス	City	PostalCode	国
87	Wartian Herku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	フィンランド
88	Wellington Importadora	親の親	Rua do Mercado, 12	Resende	08737-363	ブラジル
89	white CloverMarks	karl Jablonski	305 - 14th Ave.S. Suite 3B	Seattle	98128	USA

JSONARRAY

JSON アレイを返します。

```
JSONARRAY(value...)
```

value は、JSON 値に変換できるオブジェクトです。詳細は、「 JSON 関数 」を参照してください。
戻り値は JSON です。

null 値は結果に null リテラルとして含まれます。

混合値の例

```
jsonArray('a"b', 1, null, false, {d'2010-11-21'})
```

戻り値

```
["a\"b",1,null,false,"2010-11-21"]
```

テーブルでの JSONARRAY の使用

```
SELECT JSONARRAY(CustomerId, CustomerName)
FROM   Customer c
WHERE  c.CustomerID >= 88;
```

```
==========================================================
[88,"Wellington Importadora"]
[89,"White Clover Markets"]
```

JSONOBJECT

JSON オブジェクトを返します。

```
JSONARRAY(value [as name] ...)
```

value は、JSON 値に変換できるオブジェクトです。詳細は、「 JSON 関数 」を参照してください。戻り値は JSON です。

null 値は結果に null リテラルとして含まれます。

名前が指定されておらず、式が列参照である場合は、exprN が使用されます。ここで、N は JSONARRAY 式の値の 1 ベースのインデックスです。

混合値の例

```
jsonObject('a"b' as val, 1, null as "null")
```

戻り値

```
{"val":"a\"b","expr2":1,"null":null}
```

テーブルでの JSONOBJECT の使用

```
SELECT JSONOBJECT(CustomerId, CustomerName)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"CustomerId":88, "CustomerName":"Wellington Importadora"}
{"CustomerId":89, "CustomerName":"White Clover Markets"}
```

別の例

```
SELECT JSONOBJECT(JSONOBJECT(CustomerId, CustomerName) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"Customer":{"CustomerId":88, "CustomerName":"Wellington Importadora"}}
{"Customer":{"CustomerId":89, "CustomerName":"White Clover Markets"}}
```

別の例

```
SELECT JSONOBJECT(JSONARRAY(CustomerId, CustomerName) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
==========================================================
{"Customer":[88, "Wellington Importadora"]}
{"Customer":[89, "White Clover Markets"]}
```

JSONPARSE

JSON 結果を検証し、返します。

```
JSONPARSE(value, wellformed)
```

値 は、適切な JSON バイナリーエンコーディング（UTF-8、UTF-16、または UTF-32）を持つ Blob です。適切にフォーマットされたものは、検証をスキップする必要があることを示すブール値です。戻り値は JSON です。

いずれかの入力の null は null を返します。

JSON 単純なリテラル値の解析

```
jsonParse('{"Customer":{"CustomerId":88, "CustomerName":"Wellington Importadora"}}', true)
```

JSONARRAY_AGG

null 値を含む Clob として JSON アレイの結果を作成します。これは JSONARRAY と似ていますが、その内容を単一のオブジェクトに集約します。

```
SELECT JSONARRAY_AGG(JSONOBJECT(CustomerId, CustomerName))
FROM   Customer c
WHERE  c.CustomerID >= 88;
============================================================
[{"CustomerId":88, "CustomerName":"Wellington Importadora"}, {"CustomerId":89,
"CustomerName":"White Clover Markets"}]
```

アレイを以下のようにラップすることもできます。

```
SELECT JSONOBJECT(JSONARRAY_AGG(JSONOBJECT(CustomerId as id, CustomerName as
name)) as Customer)
FROM   Customer c
WHERE  c.CustomerID >= 88;
============================================================
{"Customer":[{"id":89,"name":"Wellington Importadora"},{"id":100,"name":"White Clover Markets"}]}
```

JSON への変換

適切な JSON ドキュメントフォームに値を変換するのに、単純な仕様に準拠する変換が使用されます。

- null 値は null リテラルとして含まれます。
- JSON として解析された値、または JSON コンストラクト関数（JSONPARSE、JSONARRAY、JSONARRAY_AGG）から返される値は、JSON 結果に直接追加されます。
- ブール値は true/false リテラルとして含まれます。
- 数値はデフォルトの文字列変換として含まれます。数字または + インフィナリティーの結果が許可されていない場合は、無効な JSON を取得できる場合があります。
- 文字列の値はエスケープまたは引用形式に含まれます。
- バイナリーの値は暗黙的に JSON 値に変換可能で、JSON に含まれる前に特定の値を必要とします。
- その他の値はすべて、適切なエスケープ/引用形式で文字列変換として含まれます。

JSONTOXML

JSON から XML ドキュメントを返します。

```
JSONTOXML(rootElementName, json)
```

rootElementName は文字列で、**json** は {clob, blob} にあります。戻り値は XML です。

適切な UTF エンコーディング（8、16LE）。16BE、32LE、32BE）が JSON Blob について検出されます。別のエンコーディングを使用する場合は、String 関数の TO_CHARS 関数 を参照してください。

結果は常に適切な形式の XML ドキュメントになります。

XML へのマッピングでは、以下のルールを使用します。

- 現在の要素名は最初に rootElementName であり、JSON 構造が通過するのでオブジェクト値名になります。
- すべての要素名は有効な XML 1.1 名である必要があります。無効な名前は、SQLXML 仕様に従って完全にエスケープされます。
- 各オブジェクトまたはプリミティブ値は、現在の名前で要素で囲まれます。
- 配列の値がルートでない限り、追加要素で囲まれません。
- null 値は、xsi:nil="true" 属性を持つ空の要素によって表されます。
- boolean および number の値要素では、属性 xsi:type はそれぞれ boolean と 10 進数に設定されます。

JSON:

jsonToXml('person', x)の XML へのサンプル

```
{"firstName" : "John" , "children" : [ "Randy", "Judy" ]}
```

XML:

jsonToXml('person', x)の XML へのサンプル

```
<?xml version="1.0" ?>
  <person>
    <firstName>John</firstName>
    <children>Randy</children>
    <children>Judy<children>
  </person>
```

JSON:

ルートアレイを使用した jsonToXml('person', x)の XML へのサンプル

```
[{"firstName" : "George" }, { "firstName" : "Jerry" }]
```

XML（より適切に形式の XML を維持するために、追加の "person" wrapping 要素がある点に注意してください）。

ルートアレイを使用した jsonToXml('person', x)の XML へのサンプル

```
<?xml version="1.0" ?>
<person>
  <person>
    <firstName>George</firstName>
  </person>
  <person>
    <firstName>Jerry</firstName>
  </person>
</person>
```

JSON:

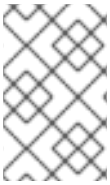
無効な名前を持つ jsonToXml('root', x)の XML へのサンプル

```
{"/invalid" : "abc" }
```

XML:

無効な名前を持つ jsonToXml('root', x)の XML へのサンプル

```
<?xml version="1.0" ?>
<root>
  <_x002F_invalid>abc</_x002F_invalid>
</root>
```



注記

以前のリリース以前は、**xXXXX** ではなく **uXXXX** エスケープを使用しないと誤って使用していました。その動作に依存する必要がある場合は、org.teiid.useXMLxEscape システムプロパティーを参照してください。

JsonPath

JsonPath 式の処理は Jayway JsonPath によって提供されます。1 ベースのインデックスではなく、0 ベースのインデックスを使用することに注意してください。さまざまなパス式で想定された戻り値を熟知していることを確認してください。たとえば、行の JsonPath 式がアレイを提供することを想定している場合は、その配列が不要なパス式によって自動的に返されるアレイではなく、希望の配列であることを確認します。

パス名で '.' などの予約文字が使用される状況が発生した場合は、任意のキー（$['.key'] など）を許可するため、括弧付きの JsonPath 表記を使用する必要があります。

詳細は、「 JSONTABLE 」を参照してください。

JSONPATHVALUE

単一の JSON 値を文字列として抽出します。

```
JSONPATHVALUE(value, path [, nullLeafOnMissing])
```

値 は clob JSON ドキュメントで、 **path** は JsonPath 文字列で、 **nullLeafOnMissing** はブール値です。戻り値は、結果となる JSON の文字列値です。

nullLeafOnMissing が false（デフォルト）の場合は、見つからないリーフに評価されるパスによって例外が発生します。**nullLeafOnMissing** が true の場合、null 値が返されます。

値が、indefinite パス式によって生成された配列である場合、最初の値のみが返されます。

```
jsonPathValue('{"key":"value"}' '$.missing', true)
```

戻り値

```
null
```

```
jsonPathValue('[{"key":"value1"}, {"key":"value2"}]' '$..key')
```

戻り値

```
value1
```

JSONQUERY

JSON ドキュメントに対して JsonPath 式を評価し、JSON 結果を返します。

```
JSONQUERY(value, path [, nullLeafOnMissing])
```

値 は clob JSON ドキュメントで、 **path** は JsonPath 文字列で、 **nullLeafOnMissing** はブール値です。戻り値は JSON 値です。

nullLeafOnMissing が false（デフォルト）の場合は、見つからないリーフに評価されるパスによって例外が発生します。**nullLeafOnMissing** が true の場合、null 値が返されます。

```
jsonPathValue('[{"key":"value1"}, {"key":"value2"}]' '$..key')
```

戻り値

```
["value1","value2"]
```

3.5.11. セキュリティー機能

セキュリティー機能は、セキュリティーシステムまたはハッシュ/暗号化の値と対話する機能を提供します。

HASROLE

現在の呼び出し元に Data Virtualization データロール **roleName** があるかどうか。

```
hasRole([roleType,] roleName)
```

roleName は文字列でなければならず、戻り値の型は Boolean です。

後方互換性を確保するために、2 つの引数フォームが提供されます。**roleType** は文字列で、「data」でなければなりません。

ロール名は大文字と小文字を区別し、Data Virtualization Data ロール のみに一致します。同じ名前の対応するデータロールがない場合は、外部/JAAS ロール/グループ名はこの機能で有効ではありません。

MD5

値の MD5 ハッシュを計算します。

```
MD5(value)
```

値 は文字列または varbinary である必要があります。戻り値のタイプは varbinary です。文字列の値は、最初に UTF-8 バイト表現に変換されます。

SHA1

値の SHA-1 ハッシュを計算します。

```
SHA1(value)
```

値 は文字列または varbinary である必要があります。戻り値のタイプは varbinary です。文字列の値は、最初に UTF-8 バイト表現に変換されます。

SHA2_256

値の SHA-2 256 ビットハッシュを計算します。

```
SHA2_256(value)
```

値 は文字列または varbinary である必要があります。戻り値のタイプは varbinary です。文字列の値は、最初に UTF-8 バイト表現に変換されます。

SHA2_512

値の SHA-2 512 ビットハッシュを計算します。

```
SHA2_512(value)
```

値 は文字列または varbinary である必要があります。戻り値のタイプは varbinary です。文字列の値は、最初に UTF-8 バイト表現に変換されます。

AES_ENCRYPT

```
aes_encrypt(data, key)
```

AES_ENCRYPT（）は、明示的な初期化ベクトルで暗号アルゴリズムの AES(Advanced Encryption Standard)アルゴリズム、16 バイト（128 ビット）キー長、および AES/CBC/PKCS5Padding 暗号アルゴリズムを使用したデータの暗号化を許可します。

AES_ENCRYPT（） は BinaryType 暗号化データを返します。引数 **データ** は暗号化する BinaryType データで、引数 **キー** は暗号化で使用される BinaryType です。

AES_DECRYPT

```
aes_decrypt(data, key)
```

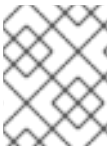
AES_DECRYPT（）は公式の AES(Advanced Encryption Standard)アルゴリズム、16 バイト（128 ビット）キーの長さ、および AES/CBC/PKCS5Padding 暗号アルゴリズムを使用してデータの復号化を許可します。

AES_DECRYPT（） は BinaryType 復号化データを返します。引数 **データ** は復号化する BinaryType データで、引数 **キー** は復号化で使用される BinaryType です。

3.5.12. 空間関数

空間関数は、地理データの作業機能 を提供 します。Data Virtualization は JTS Topology Suite に依存して、SQL Revision 1.1 の OpenGIS Simple Features Specification との部分的な互換性を提供します。特定の機能の詳細は、Open GIS 仕様または PostGIS マニュアル を参照してください。

ほとんどの Geometry 機能は、WKB 形式および WKT 形式による 2 つのディメンションに制限されます。



注記

Data Virtualization とプッシュダウンの結果には若干の違いがあり、さらに改良する必要があります。

ST_GeomFromText

WKT 形式の Clob からジオメトリーを返します。

```
ST_GeomFromText(text [, srid])
```

テキスト は CLOB の **srid** で、空間参照識別子(SRID)を表す任意の整数です。戻り値はジオメトリーです。

ST_GeogFromText

(E)WKT 形式の Clob から地理を返します。

```
ST_GeogFromText(text)
```

テキスト は CLOB で、 **srid** は任意の整数です。戻り値は地理です。

ST_GeomFromWKB/ST_GeomFromBinary

WKB 形式の BLOB からジオメトリーを返します。

```
ST_GeomFromWKB(bin [, srid])
```

bin は BLOB で、 **srid** はオプションの整数です。戻り値はジオメトリーです。

ST_GeomFromEWKB

EWKB 形式の BLOB からジオメトリーを返します。

```
ST_GeomFromEWKB(bin)
```

bin は BLOB です。戻り値はジオメトリーです。このバージョンのトランスレーターは、2 つのディメンションでのみ機能します。

ST_GeogFromWKB

(E)WKB 形式の BLOB からジオグラフを返します。

```
ST_GeomFromEWKB(bin)
```

bin は BLOB です。戻り値は地理です。このバージョンのトランスレーターは、2 つのディメンションでのみ機能します。

ST_GeomFromEWKT

EWKT 形式の文字大きなオブジェクト(CLOB)からジオメトリーを返します。

```
ST_GeomFromEWKT(text)
```

テキスト は CLOB です。戻り値はジオメトリーです。このバージョンのトランスレーターは、2 つのディメンションでのみ機能します。

ST_GeomFromGeoJSON

GeoJSON 形式の CLOB からジオメトリーを返します。

```
ST_GeomFromGeoJson(`text` [, srid])
```

テキスト は CLOB で、 **srid** は任意の整数です。戻り値はジオメトリーです。

ST_GeomFromGML

GML2 形式の CLOB からジオメトリーを返します。

```
ST_GeomFromGML(text [, srid])
```

テキスト は CLOB で、 **srid** は任意の整数です。戻り値はジオメトリーです。

ST_AsText

```
ST_AsText(geom)
```

ジオメトリーは ジオメトリーです。戻り値は、WKT 形式の CLOB です。

ST_AsBinary

```
ST_AsBinary(geo)
```

geo はジオメトリーまたはジオグラフです。戻り値は、WKB 形式のバイナリー大規模なオブジェクト(BLOB)です。

ST_AsEWKB

```
ST_AsEWKB(geom)
```

ジオメトリーは ジオメトリーです。戻り値は EWKB 形式の BLOB です。

ST_AsGeoJSON

```
ST_AsGeoJSON(geom)
```

ジオメトリーは ジオメトリーです。戻り値は GeoJSON 値を持つ CLOB です。

ST_AsGML

```
ST_AsGML(geom)
```

ジオメトリーは ジオメトリーです。戻り値は GML2 値を持つ CLOB です。

ST_AsEWKT

```
ST_AsEWKT(geo)
```

geo はジオメトリーまたはジオグラフです。戻り値は、EWKT 値を持つ CLOB です。EWKT の値は、SRID プレフィックスを持つ WKT 値です。

ST_AsKML

```
ST_AsKML(geom)
```

ジオメトリーは ジオメトリーです。戻り値は、KML 値を持つ CLOB です。KML の値は、簡素化された GML 値であり、SRID 4326 に展開されます。

&&

geom1 および **geom2** intersect の境界ボックスに true を返します。

```
geom1 && geom2
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Contains

geom1 に **geom2** が含まれる場合に true を返します。

```
ST_Contains(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Crosses

ジオメンションをまたがると、true を返します。

```
ST_Crosses(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Disjoint

ジオメンションがない場合は true を返します。

```
ST_Disjoint(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Distance

2 つの地理エントリー間の距離を返します。

```
ST_Distance(geo1, geo2)
```

geo1、**geo2** は共にジオメーターまたは地理的な点があります。戻り値は double です。geography バリアントは評価用にプッシュする必要があります。

ST_DWithin

地理的なエントリーがある間隔内にある場合は true を返します。

```
ST_DWithin(geom1, geom2, dist)
```

geom1、**geom2** は geometries です。 **dist** は二重です。戻り値はブール値です。

ST_Equals

2 つの地理的な値が等しい場合は true を返します。Point と order は異なる場合がありますが、ジオメトリーは他方の外部に置くことはできません。

```
ST_Equals(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Intersects

ジオメンティンで交差部分がある場合は true を返します。

```
ST_Intersects(geo1, geo2)
```

geo1、**geo2** は共にジオメーターまたは地理的な点があります。戻り値はブール値です。geography バリアントは評価用にプッシュする必要があります。

ST_OrderingEquals

geom1 と **geom2** が同じ構造を持ち、同じポイントの順序がある場合は true を返します。

```
ST_OrderingEquals(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Overlaps

地理エントリーが重複している場合、true を返します。

```
ST_Overlaps(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Relate

geom1 および geom2 の交差部分をテストするか、または返します。

```
ST_Relate(geom1, geom2, pattern)
```

geom1、**geom2** は geometries です。 **Pattern** は、DE-9IM パターン文字列 9 文字です。戻り値はブール値です。

```
ST_Relate(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値は、DE-9IM の交差部分文字列 9 文字です。

ST_Touches

ジオメトンが連絡すると true を返します。

```
ST_Touches(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Within

geom1 が **geom2** 内に完全にある場合は true を返します。

```
ST_Within(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はブール値です。

ST_Area

ジオムのエリアを返します。

```
ST_Area(geom)
```

ジオメトリーは ジオメトリーです。戻り値は double です。

ST_CoordDim

ジオムのコーディネートディメンションを返します。

```
ST_CoordDim(geom)
```

ジオメトリーは ジオメトリーです。戻り値は 0 から 3 の間の整数です。

ST_Dimension

ジオメトリーのディメンションを返します。

```
ST_Dimension(geom)
```

ジオメトリーは ジオメトリーです。戻り値は 0 から 3 の間の整数です。

ST_EndPoint

LineString geom の終了ポイントを返します。**geom** が LineString ではない場合に null を返します。

```
ST_EndPoint(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_ExteriorRing

ポリジーンジオムの exterior リングまたは shell LineString を返します。**ジオム** がポリモーンではない場合に null を返します。

```
ST_ExteriorRing(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_GeometryN

nth geometry をジオムの指定の 1 ベースのインデックスで返します。指定のインデックスのジオメトリーが存在しない場合に null を返します。コレクションでないタイプは、最初のインデックスで自身を返します。

```
ST_GeometryN(geom, index)
```

geom はジオメトリーです。インデックスは整数です。戻り値はジオメトリーです。

ST_GeometryType

ジオム のタイプ名を ST_name として返します。name は LineString、Posplygon、Point などになります。

```
ST_GeometryType(geom)
```

ジオメトリーは ジオメトリーです。戻り値は文字列です。

ST_HasArc

ジオメトリーに円形の文字列があるかどうかをテストします。トランスレーターは曲線化されたジオメトリータイプで機能しないため、**false** を報告します。

```
ST_HasArc(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_InteriorRingN

nth interior ring LinearString geometry を、ジオムの指定の 1 ベースのインデックスで返します。指定されたインデックスのジオメトリーが存在しない場合や、**ジオム** がポリモーンではない場合に null を返します。

```
ST_InteriorRingN(geom, index)
```

geom はジオメトリーです。インデックスは整数です。戻り値はジオメトリーです。

ST_IsClosed

LineString **geom** が閉じられている場合に true を返します。**geom** が LineString でない場合、false を返します。

```
ST_IsClosed(geom)
```

ジオメトリーは ジオメトリーです。戻り値はブール値です。

ST_IsEmpty

ポイントのセットが空の場合は true を返します。

```
ST_IsEmpty(geom)
```

ジオメトリーは ジオメトリーです。戻り値はブール値です。

ST_IsRing

LineString **geom** がリングの場合、true を返します。**geom** が LineString ではない場合は false を返します。

```
ST_IsRing(geom)
```

ジオメトリーは ジオメトリーです。戻り値はブール値です。

ST_IsSimple

ジオム が単純な場合は true を返します。

```
ST_IsSimple(geom)
```

ジオメトリーは ジオメトリーです。戻り値はブール値です。

ST_IsValid

ジオム が有効な場合は **true** を返します。

```
ST_IsValid(geom)
```

ジオメトリーは ジオメトリーです。戻り値はブール値です。

ST_Length

(Multi)LineString の長さを返します。それ以外の場合は 0 を返します。

```
ST_Length(geo)
```

geo はジオメトリーまたはジオグラフです。戻り値は double です。geography バリアントは評価用にプッシュする必要があります。

ST_NumGeometries

ジオメーム内の地理エントリーの数を返し **ます**。ジオメトリーコレクションでなければ、1 を返します。

```
ST_NumGeometries(geom)
```

ジオメトリーは ジオメトリーです。戻り値は整数です。

ST_NumInteriorRings

ポリgonジオメトリー内の内部リングの数を返します。**ジオム** がポリモーンではない場合に null を返します。

```
ST_NumInteriorRings(geom)
```

ジオメトリーは ジオメトリーです。戻り値は整数です。

ST_NunPoints

ジオム 内のポイント数を返します。

```
ST_NunPoints(geom)
```

ジオメトリーは ジオメトリーです。戻り値は整数です。

ST_PointOnSurface

地図の地図に確実に配置できるポイントを返します。

```
ST_PointOnSurface(geom)
```

ジオメトリーは ジオメトリーです。戻り値はポイントジオメトリーです。

ST_Perimeter

(Multi)Polygon geom の境界を返します。**geom** が(Multi)Polygon ではない場合に 0 を返します。

```
ST_Perimeter(geom)
```

ジオメトリーは ジオメトリーです。戻り値は double です。

ST_PointN

指定の 1 ベースのインデックスの n 番目のポイントをジオムで返します。指定されたインデックスのポイントが存在しない場合や、**geom** が LineString ではない場合に null を返します。

```
ST_PointN(geom, index)
```

geom はジオメトリーです。インデックスは整数です。戻り値はジオメトリーです。

ST_SRID

ジオメトリーの SRID を返します。

```
ST_SRID(geo)
```

geo はジオメトリーまたはジオグラフです。戻り値は整数です。null ではなく 0 値は、null 以外のジオメトリーの不明な SRID に対して返されます。

ST_SetSRID

指定のジオメトリーの SRID を設定します。

```
ST_SetSRID(geo, srid)
```

geo はジオメトリーまたはジオグラフです。**Srid** は整数です。戻り値は **geo** の値と同じです。SRID メタデータのみが変更されます。変換は実行されません。

ST_StartPoint

LineString geom の開始点を返します。**geom** が LineString ではない場合に null を返します。

-

```
ST_StartPoint(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_X

X ordinate 値を返します。ポイントが空の場合は null を返します。ジオメトリーがポイントではない場合に例外が発生します。

```
ST_X(geom)
```

ジオメトリーは ジオメトリーです。戻り値は double です。

ST_Y

Y ordinate 値を返します。ポイントが空の場合は null を返します。ジオメトリーがポイントではない場合に例外が発生します。

```
ST_Y(geom)
```

ジオメトリーは ジオメトリーです。戻り値は double です。

ST_Z

Z ordinate 値を返します。ポイントが空の場合は null を返します。ジオメトリーがポイントではない場合に例外が発生します。通常、トランスレーターは 2 つ以上のディメンションと機能しないため、**null** を返します。

```
ST_Z(geom)
```

ジオメトリーは ジオメトリーです。戻り値は double です。

ST_Boundary

指定のジオメトリーの境界を計算します。

```
ST_Boundary(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_Buffer

ジオム の距離内にあるジオメトリーを計算します。

```
ST_Buffer(geom, distance)
```

ジオメトリーは ジオメトリーです。**距離** は 2 倍です。戻り値はジオメトリーです。

ST_Centroid

地理的な地理的な中央ポイントを計算します。

```
ST_Centroid(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_ConvexHull

ジオメトリーのすべてのポイントを含む最小のコンバージョンを返します。

```
ST_ConvexHull(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_CurveToLine

CircularString/CurvedPolygon を LineString/Polygon に変換します。Data Virtualization に現在実装されていません。

```
ST_CurveToLine(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_Difference

geom2 にない **geom1** に含まれるポイントのセットの明確さを計算します。

```
ST_Difference(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はジオメトリーです。

ST_Envelope

指定のジオメトリーの 2D 境界ボックスを計算します。

```
ST_Envelope(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_Force_2D

z コーディネート値が存在する場合は削除します。

```
ST_Force_2D(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_Intersection

geom1 および **geom2** に含まれるポイントのポイントセット交差部分を計算します。

```
ST_Intersection(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はジオメトリーです。

ST_Simplify

Douglas-Peucker アルゴリズムを使用してジオメトリーを簡素化しますが、無効なジオメトリーまたは空のジオメトリーを単純化します。

```
ST_Simplify(geom, distanceTolerance)
```

ジオメトリーは ジオメトリーです。**distanceTolerance** は 2 倍です。戻り値はジオメトリーです。

ST_SimplifyPreserveTopology

Douglas-Peucker アルゴリズムを使用してジオメトリーを簡素化します。有効なジオメトリーを常に返します。

```
ST_SimplifyPreserveTopology(geom, distanceTolerance)
```

ジオメトリーは ジオメトリーです。**distanceTolerance** は 2 倍です。戻り値はジオメトリーです。

ST_SnapToGrid

ジオメトリー内のすべてのポイントを、指定したサイズのグリッドへすべて移動します。

```
ST_SnapToGrid(geom, size)
```

ジオメトリーは ジオメトリーです。サイズは二重です。戻り値はジオメトリーです。

ST_SymDifference

geom2 と交差しない geom1 の一部を返します。

```
ST_SymDifference(geom1, geom2)
```

geom1、**geom2** はジオメトリーです。戻り値はジオメトリーです。

ST_Transform

ジオメトリーの値をあるコーディネートシステムから別のシステムに変換します。

```
ST_Transform(geom, srid)
```

ジオメトリーは ジオメトリーです。**Srid** は整数です。戻り値はジオメトリーです。SPATIAL_REF_SYS ビューに、**srid** の値と geometry 値の SRID が存在している。

ST_Union

すべての **geom1** および **geom2** を含むポイントセットを表すジオメトリーを返します。

```
ST_Union(geom1, geom2)
```

geom1、**geom2** は geometries です。戻り値はジオメトリーです。

ST_Extent

すべてのジオメトリーの値について 2D の境界ボックスを計算します。すべての値に同じ SRID が必要です。

```
ST_Extent(geom)
```

ジオメトリーは ジオメトリーです。戻り値はジオメトリーです。

ST_Point

指定のコーディネートの Point を上書きします。

-

```
ST_Point(x, y)
```

X と y は 2 倍です。戻り値は Point geometry です。

ST_Polygon

指定されたシェルおよび SRID で Polygon を返します。

```
ST_Polygon(geom, srid)
```

ジオム は線形リングジオメトリーで、**srid** は整数です。戻り値は Polygon geometry です。

3.5.13. その他の機能

では、追加の機能と、他のプロジェクトが提供する機能について説明します。

array_get

指定されたアレイインデックスでオブジェクト値を返します。

```
array_get(array, index)
```

array はオブジェクトタイプで、**index** は整数で、戻り値の型はオブジェクトです。

1 ベースのインデックスが使用されます。実際の配列の値は、java.sql.Array または java 配列型である必要があります。引数が null の場合や、インデックスがバインドされていないと null が返されます。

array_length

指定のアレイの長さを返します。

```
array_length(array)
```

array はオブジェクトタイプで、戻り値のタイプは integer です。

実際の配列の値は、java.sql.Array または java 配列型である必要があります。配列の値が誤った型である場合、例外が発生します。

uuid

ユニバーサル一意識別子を返します。

```
uuid()
```

戻り値の型は string です。

暗号化で強力な乱数ジェネレーターを使用して 4（疑似ランダムに生成された）UUID を生成します。形式は XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX で、各 X は 16 進数の数字になります。

データ品質機能

データ品質機能は ODDQ プロジェクトによって提供されます。この関数は **osdq.** のプレフィックスが付けられますが、接頭辞なしで呼び出すことができます。

osdq.random

ランダム化された文字列を返します。たとえば、**jboss teiid** は **jtids soibe** にランダム化できます。

```
random(sourceValue)
```

sourceValue はランダム化される文字列です。

osdq.digit

文字列の数字を返します。たとえば、**a1 b2 c3 d4** は **1234** になります。

```
digit(sourceValue)
```

sourceValue は、数字の抽出元となる文字列です。

osdq.whitespaceIndex

最初の空白のインデックスを返します。たとえば、**jboss teiid** は **5** を返します。

```
whitespaceIndex(sourceValue)
```

sourceValue は、空白文字インデックスを検索する文字列です。

osdq.validCreditCard

クレジットカード番号が有効かどうかを確認します。クレジットカードロジックとチェックサムと一致する場合は **true** を返します。

```
validCreditCard(cc)
```

cc は、確認するクレジットカード番号の文字列です。

osdq.validSSN

ソーシャルセキュリティー番号(SSN)が有効かどうかを確認します。SSN ロジックと一致する場合は **true** を返します。

```
validSSN(ssn)
```

SSN は、チェックするソーシャルセキュリティー番号の文字列です。

osdq.validPhone

電話番号が有効かどうかを確認します。数字が電話番号のロジックと一致する場合は **true** を返します。番号には 8 を超える文字が含まれる必要があり、**000** 文字未満で開始することはできません。

```
validPhone(phone)
```

'phone is the phone number string must to check.

osdq.validEmail

メールアドレスが有効かどうかを確認します。有効な場合は **true** を返します。

```
validEmail(email)
```

email は確認するメールアドレス文字列です。

osdq.cosineDistance

Cosine Similarity アルゴリズムに基づいて 2 つの文字列間の浮動小数点距離を返します。

```
cosineDistance(a, b)
```

a および **b** は、距離を計算する文字列です。

osdq.jaccardDistance

Jaccard similarity アルゴリズムに基づいて、2 つの文字列間の浮動小数点距離を返します。

```
jaccardDistance(a, b)
```

a および **b** は、距離を計算する文字列です。

osdq.jaroWinklerDistance

Jaro-Winkler アルゴリズムに基づいて 2 つの文字列間の浮動小数点距離を返します。

```
jaroWinklerDistance(a, b)
```

a および **b** は、距離を計算する文字列です。

osdq.levenshteinDistance

Levenshtein アルゴリズムに基づいて 2 つの文字列間の浮動小数点距離を返します。

```
levenshteinDistance(a, b)
```

a および **b** は、距離を計算する文字列です。

osdq.intersectionFuzzy

1 番目のセットから、2 番目のセットのすべてのメンバーに、指定された値よりも間隔が差し引いた一意の要素のセットを返します。

```
intersectionFuzzy(a, b)
```

A および **b** は文字列の配列です。**c** は距離を表す浮動小数点であるため、0.0 または less は any と一致し、>1.0 は完全に一致します。

osdq.minusFuzzy

1 番目のセットから、2 番目のセットのすべてのメンバーに、指定された値よりも間隔が差し引いた一意の要素のセットを返します。

```
minusFuzzy(a, b, c)
```

A および **b** は文字列の配列です。**c** は距離を表す浮動小数点であるため、0.0 または less は any と一致し、>1.0 は完全に一致します。

osdq.unionFuzzy

最初のセットからのメンバーおよび 2 番目のセットのメンバーが含まれる一意の要素のセットを返します。

```
unionFuzzy(a, b, c)
```

A および **b** は文字列の配列です。**c** は距離を表す浮動小数点であるため、0.0 または less は any と一致し、>1.0 は完全に一致します。

3.5.14. 非決定的関数処理

データ仮想化は、さまざまな決定論による機能を分類します。関数が評価され、結果をキャッシュできるエクステントは決定レベルに基づいています。

決定論的

この関数は、指定の入力に同じ結果を常に返します。決定論的な関数は、すべての入力値が認識されるとすぐにエンジンによって評価されます。これは書き換えフェーズの直後に発生する可能性があります。lookup 関数などの一部の機能は、実際には決定論的ではありませんが、パフォーマンスなどのように処理されます。このリストの残りの項目に従って分類されていない関数は、すべて決定論的とみなされます。

ユーザーの決定論的

この関数は、同じユーザーの指定された入力に同じ結果を返します。これには、**hasRole** および **user** 関数が含まれます。決定論的関数は、すべての入力値が認識されるとすぐにエンジンによって評価されます。これは書き換えフェーズの直後に発生する可能性があります。ユーザー決定論的な関数が準備済み処理計画の作成時に評価された場合、作成されるプランはユーザーに対してのみキャッシュされます。

セッション決定

この関数は、同じユーザーセッションで指定された入力に同じ結果を返します。このカテゴリーには **env** 関数が含まれます。セッション決定関数は、すべての入力値が認識されるとすぐにエンジンによって評価されます。準備済み処理計画の作成時にセッション決定論的な機能が評価されると、作成されるプランはユーザーのセッションに対してのみキャッシュされます。

コマンド決定

関数評価の結果は、user コマンドのスコープ内でのみ決定論的です。このカテゴリーには、**curdate** 関数、**curtime** 関数、**現在は commandpayload** 関数が含まれます。コマンド決定論的な関数は、これらの関数を使用する準備されたプランが関連する値で実行されるように処理されるまで評価に遅延します。コマンド決定的な関数評価は、プッシュダウンの前に行われます。ただし、同じコマンド決定論的な時間関数が複数表示される場合は、同じ値に評価することは保証されません。

非決定的

関数評価の結果は完全に非決定論的です。このカテゴリーには、実行され **た** 関数と **nondeterministic** のマークが付けられた UDF が含まれます。非決定的関数は、プッシュダウンを優先して処理するまで評価の遅れます。関数がプッシュされない場合、その実行コンテキストのすべての行に対して評価できます（たとえば、関数が select 句で使用される場合）。

注記

Uncorrelated subqueries は、それらで使用される関数に関係なく、決定論的として処理されます。

3.6. DML コマンド

Data Virtualization で SQL を使用してクエリーを発行し、ビュー変換を定義できます。仮想手順および更新手順で SQL を使用する方法の詳細は、「手順 言語」を参照し てください。これらの機能のほとんどは標準の SQL 構文と機能を使用しているため、すべての SQL 参照を使用して詳細を確認できます。

SQL のデータを操作する基本的なコマンドとして、create、read、update、および delete(CRUD)操作（INSERT、SELECT、UPDATE、DELETE）に対応しています。MERGE ステートメントは INSERT と UPDATE の組み合わせとして機能します。

EXECUTE コマンド、手順のリレーショナルデータベースコマンドを使用して、手順を実行することもできます。詳細は、「Proced uralational command」または「 Anonymous procedure block 」を参照してください。

3.6.1. 設定操作

Data Virtualization の SQL **UNION** ALL、**UNION ALL**、INTTERSEC**T、** および **EXCEPT** セット操作を使用して、クエリー式の結果を組み合わせることができます。

使用方法

```
queryExpression (UNION|INTERSECT|EXCEPT) [ALL] queryExpression [ORDER BY...]
```

構文ルール：

- 出力コラムは、最初の set 操作ブランチの出力列によって名前が付けられます。
- 各 **SELECT** には、各相対列に同じ数の出力列と互換性のあるデータタイプがなければなりません。データ型が一貫性がなく、暗黙的な変換が存在する場合は、データ型変換が実行されます。
- **UNION**、INTTERSEC**T、** または **EXCEPT** が **all** なしで指定される場合、出力列は同じタイプである必要があります。
- SQL **INTERSECT ALL** 演算子または **EXCEPT ALL** 演算子は使用できません。

3.6.2. SELECT コマンド

SELECT コマンドは、任意の数の関係のレコードを取得するために使用されます。

SELECT コマンドには以下の句を含めることができます。

- ...
- ...
- FROM ...
- ...
- グループ化...
- ...
- ORDER BY ...
- （制限 ...） |([OFFSET ...] [FETCH ...])
- オプション ...

OPTION 句を除き、前述のすべての句は SQL 仕様で定義されます。この仕様は、これらの句が論理的に処理される順序も指定します。処理はステージごとに行われ、各ステージは一連の行を以下のステージに渡します。処理モデルは論理的で、データベースエンジンが処理を実行する方法を示していませんが、SQL の仕組みを理解するのに便利なモデルです。SELECT コマンドは、以下の段階における句を処理します。

ステージ 1: WITH 句

一覧表示されている順序ですべての項目からすべての行を収集します。後続の WITH 項目とメインのクエリーは、テーブルであるかのように WITH 項目を参照できます。

ステージ 2: FROM 句

クエリーに関連するすべてのテーブルからすべての行を収集し、それらを Cartesian 製品に論理的に結合し、すべてのテーブルのすべての列が含まれる 1 つの大きなテーブルを生成します。結合条件および結合条件は、結合構造に一致しない行をフィルターするために適用されます。

ステージ 3: WHERE 句

FROM ステージのすべての出力行に基準を適用し、行数を減らします。

ステージ 4: GROUP BY 句

GROUP BY 列の一致する値を持つ行のセットをグループ化します。

ステージ 5: HAVING 句

行の各グループに基準を適用します。基準は、グループ内の定数値を持つ列にのみ適用できます（グループ化列またはグループ全体に適用される集約関数）。

ステージ 6: SELECT 句

クエリーから返される列式を指定します。式は、行のグループに基づいて集約関数を含め、この時点以降は存在しなくなります。出力列は、列のエイリアスまたはエンジンによって決定される暗黙的な名前を使用して名前が付けられます。SELECT DISTINCT が指定されている場合は、SELECT ステージから返される行で重複削除が実行されます。

ステージ 7: ORDER BY 句

SELECT ステージから返された行を必要に応じて並べ替えます。指定した順序で複数のコラムのソート（昇順または降順）をサポートします。出力列は SELECT ステージから返される列と同じで、名前は同じになります。

ステージ 8: LIMIT 句

指定された行のみを返します（skip 値および制限値）。

上記のモデルは、SQL の仕組みを理解するのに役立ちます。たとえば、SELECT 句がエイリアスを列に割り当てると、後続の ORDER BY 句がこれらのエイリアスを使用して列を参照する必要があることが理にかなっています。処理モデルに関する知識がないと、混乱を生じさせる可能性があります。モデルを若干見ると、HER ステージは SELECT ステージの後に発生した唯一のステージで、このステージには列の名前が付けられていることが明確になります。WHERE 句は SELECT の前に処理されるため、列には名前が付けられておらず、エイリアスは認識されていません。

ヒント

明示的な表構文 **TABLE x** は **SELECT * FROM x** のショートカットとして使用できます。

3.6.3. VALUES コマンド

VALUES コマンドは、単純なテーブルの構築に使用されます。

構文の例

```
VALUES (value,...)
```

```
VALUES (value,...), (valueX,...) ...
```

単一の値が設定された VALUES コマンドは **SELECT 値 ...** と 同等です。複数の値セットを持つ VALUES コマンドは、単純な SELECT の UNION ALL と同じです（例： **SELECT value ...）。UNION ALL SELECT valueX, ...**.

3.6.4. コマンドの更新

更新コマンドを更新して、整数の更新数を報告します。更新コマンドを更新して、整数値(2^31-1)を報告できます。行数を増やすと、コマンドは最大整数値を報告します。

3.6.4.1. INSERT コマンド

INSERT コマンドは、レコードをテーブルに追加するために使用されます。

構文の例

```
INSERT INTO table (column,...) VALUES (value,...)
```

```
INSERT INTO table (column,...) query
```

3.6.4.2. UPDATE コマンド

UPDATE コマンドは、テーブルのレコードを変更するために使用されます。この操作により、1つ以上のレコードが更新されるか、基準に一致しない場合はレコードが更新されません。

構文の例

```
UPDATE table [[AS] alias] SET (column=value,...) [WHERE criteria]
```

3.6.4.3. DELETE コマンド

DELETE コマンドは、テーブルからレコードを削除するために使用されます。この操作により、1つ以上のレコードが削除されるか、基準に一致しない場合はレコードは削除されません。

構文の例

```
DELETE FROM table [[AS] alias] [WHERE criteria]
```

3.6.4.4. UPSERT(MERGE)コマンド

UPSERT （または **MERGE**）コマンドは、レコードの追加または更新に使用されます。Data Virtualization に実装されるANSI 以外のバージョンの **UPSERT** は、ターゲットテーブルにプライマリーキーが必要となり、ターゲット列がプライマリーキーに対応する変更済みの INSERT ステートメントです。**INSERT** を実行する前に、**UPSERT** 操作は行が存在するかどうかを確認し、存在する場合は、**UPSERT** が新しい行を挿入せずに現在の行を更新します。

構文の例

```
UPSERT INTO table [[AS] alias] (column,...) VALUES (value,...)
```

```
UPSERT INTO table (column,...) query
```

UPSERT プッシュダウン

UPSERT ステートメントがソースにプッシュされていない場合、これはそれぞれの INSERT および UPDATE 操作に分割されます。ターゲットデータベースシステムは、トランザクションの原子性を保証するために拡張アーキテクチャー(XA)をサポートする必要があります。

3.6.4.5. EXECUTE コマンド

EXECUTE コマンドは、仮想手順やストアドプロシージャーなどの手順を実行するために使用されます。この手順には、ゼロ以上のスカラー入力パラメーターを指定できます。手順からの戻り値は結果セット、または inout/out/return スケーラーのセットになります。

EXECUTE コマンドの以下の短い形式を使用できます。

- EXEC
- 呼び出し

構文の例

```
EXECUTE proc()
```

```
CALL proc(value, ...)
```

名前付きパラメーター構文

```
EXECUTE proc(name1=>value1,name4=>param4, ...)
```

構文ルール

- デフォルトのパラメーター仕様の順序は、手順定義の定義方法と同じです。
- 名前でパラメーターを指定できます。デフォルト値が指定されたパラメーター、またはメタデータで null 可能なパラメーターは、名前付きパラメーター呼び出しから省略でき、実行時に適切な値を渡します。
- デフォルト値があるか、またはメタデータで null 可能な位置パラメーターは、パラメーター一覧の最後から省略でき、実行時に適切な値を渡します。
- この手順で結果セットが返されない場合は、インラインビュークエリーとして使用すると、RETURN、OUT、および IN_OUT パラメーターの値が 1 行として返されます。
- VARIADIC パラメーターは、最後の位置引数として 0 以上を繰り返すことができます。

3.6.4.6. 手順リレーショナルデータベースコマンド

手順のリレーショナルデータベースでは、SELECT の構文を使用して EXEC をエミュレートします。手順関連付けられたコマンドでは、手順グループ名がテーブルの代わりに FROM 句で使用されます。この手順に対して、必要な入力値がすべて条件にある場合に、通常のテーブルアクセスの代わりに、この手順が実行されます。条件にある入力値の組み合わせはそれぞれ、手順が実行されます。

構文の例

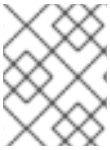
```
select * from proc
```

```
select output_param1, output_param2 from proc where input_param1 = 'x'
```

```
select output_param1, output_param2 from proc, table where input_param1 = table.col1 and
input_param2 = table.col2
```

構文ルール

- テーブルプロジェクトとして、入力パラメーターを追加して EXEC と同じ列である手順。結果セットが返されない手順の場合、IN_OUT 列は 2 つの列として展開されます。
  - 出力値を表す 1 つ。
  - パラメーターの入力を表す {column name}_IN という名前のもの。
- 入力値は、基準を介して渡されます。値は **=** か、**null または** 述語 **で** 渡すことができます。Disjuncts は使用できません。また、等価述語を介して非機密列の値を渡すこともできます。
- 手順ビューでは、IN パラメーターおよび IN_OUT パラメーターにアクセスパターンが自動的に付けられます。アクセスパターンを使用すると、必要に応じて手順ビューを依存結合として正しく計画したり、十分な基準が見つからない場合は失敗します。
- パラメーター（IN、IN_OUT、OUT、RETURN）と結果セットのコラムの間に重複する名前が含まれる手順は、proceduralational コマンドで使用できません。
- 手順と同じ名前のテーブルまたはビューがすでにある場合は、手順のリレーショナルデータベース構文を使用して呼び出すことはできません。
- 指定の入力に基準がない場合に IN または IN_OUT パラメーターのデフォルト値は使用されません。デフォルト値は、名前付き手順の構文にのみ有効です。詳細は、「 EXECUTE 」を参照してください。



注記

上記の問題は、ネストされたテーブル参照を使用する場合に適用されません。詳細は、「 **Nested table reference**in FROM clause 」を参照してください。

複数の実行

in または join の条件を使用すると、手順が複数回実行されます。

3.6.4.7. Anonymous procedure block

手順言語ブロックはユーザーコマンドとして実行できます。これは、仮想手順が存在しない状況で利点がありますが、一連のプロセスをサーバー側で実行できます。仮想手順を定義するための言語の詳細は、「手順 言語」を参照し てください。

構文の例

```
begin insert into pm1.g1 (e1, e2) select ?, ?; select rowcount; end;
```

構文ルール

- 前述の例に示すよう **に**、準備済み/呼び出し可能なステートメントパラメーターを使用してパラメーターで使用できます。これには **?** パラメーターが使用されています。
- 匿名手順ブロックでは **out** パラメーターは使用できません。回避策として、必要に応じてセッション変数を使用できます。
- Anonymous procedure blocks not return data as output parameters.
- ステートメントのいずれかが結果セットを返すと、単一の結果が返されます。戻り可能な結果セットには、一致する列と型の数がなければなりません。ステートメントが結果セットを提供することが意図されていないことを示すには、WITHOUT RETURN 句を使用します。

3.6.5. Subqueries

サブクエリーは、別の SQL クエリーに埋め込まれた SQL クエリーです。subquery を含むクエリーは、外部クエリーです。

サブクエリータイプ：

- scalar サブクエリー：1 つの値を持つ単一の列のみを返すサブクエリー。スカラーサブキューは式のタイプであり、単一の値式が想定される場合に使用できます。
- 相関サブクエリー： 外側のクエリーからのコラム参照を含むサブクエリー。
- Uncorrelated subquery - outer サブクエリーへの参照が含まれないサブクエリーです。

インラインビュー

外部クエリーの FROM 句（「インラインビュー」とも呼ばれる）の Subqueries は、任意の数の行と列を返すことができます。このサブクエリーのタイプは常にエイリアスを指定する必要があります。インラインビューは従来のビューとほぼ同じです。WITH 句 も併せて参照してください。

FROM 句のサブクエリーの例（インラインビュー）

```
SELECT a FROM (SELECT Y.b, Y.c FROM Y WHERE Y.d = '3') AS X WHERE a = X.c AND b = X.b
```

Subqueries は、式または基準が想定される場所に表示されます。

subqueries は、定量化された基準、**EXISTS** 述語、**IN** 述語、および Scalar サブ キューとして使用できます。

EXISTS を使用した WHERE のサブクエリーの例

```
SELECT a FROM X WHERE EXISTS (SELECT 1 FROM Y WHERE c=X.a)
```

定量化された比較サブキューの例

```
SELECT a FROM X WHERE a >= ANY (SELECT b FROM Y WHERE c=3)
SELECT a FROM X WHERE a < SOME (SELECT b FROM Y WHERE c=4)
SELECT a FROM X WHERE a = ALL (SELECT b FROM Y WHERE c=2)
```

IN サブクエリーの例

```
SELECT a FROM X WHERE a IN (SELECT b FROM Y WHERE c=3)
```

Subquery Optimization も参照してください。

3.6.6. WITH 句

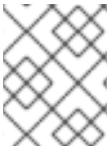
Data Virtualization は、**WITH** 句を介して一般的なテーブル式へのアクセスを提供します。 **WITH** 句の項目をテーブルとして参照するには、後続の WITH 句項目とメインのクエリーで使用することができます。**WITH** 句は、クエリースコープの一時テーブルを提供することと考えることができます。

用途

```
WITH name [(column, ...)] AS [/*+ no_inline|materialize */] (query expression) ...
```

構文ルール

- 展開された列名はすべて一意でなければなりません。一意でない場合は、列名の一覧を指定する必要があります。
- WITH 句項目の列が宣言されている場合は、クエリー式によって展開された列の数に一致する必要があります。
- 各 WITH 句項目には一意の名前を指定する必要があります。
- オプションの **no_inline** ヒントは、クエリー式が参照されている場所のインラインビューとして置き換えないようにするオプティマイザーを示しています。ソースクエリーで必要な共通テーブルの複数の評価には、no_inline を使用できます。
- オプションの **マテリアル化** ヒントでは、共通テーブルを Data Virtualization の一時的なテーブルとして作成する必要があります。これにより、共通テーブルの単一の評価が強制されます。



注記

WITH 句も最適化の対象であり、後続のクエリーでは必要ありませんが、そのエントリーは処理されないことがあります。

注記

一般的なテーブルは、プッシュダウンの可能性を強化するために積極的にインライン化されます。共通テーブルがメインのクエリーで 1 回のみ参照されている場合は、インラインになる可能性が高くなります。共通のテーブルを使用してプッシュ以外の相関サブクエリーの n 多重処理を行わない場合などに、**no_inline** や **materialize** ヒントを含める必要がある場合があります。

例

```
WITH n (x) AS (select col from tbl) select x from n, n as n1
```

```
WITH n (x) AS /*+ no_inline */ (select col from tbl) select x from n, n as n1
```

再帰的な共通テーブル式

再帰的な共通テーブル式は、それ自体を参照して、再帰的または反復方式で完全な共通テーブルを構築するために許可される一般的なテーブル式の特別な形式です。

用途

```
WITH name [(column, ...)] AS (anchor query expression UNION [ALL] recursive query expression) ...
```

再帰クエリー式は、名前で共通テーブルを参照できます。アンカーのクエリー式は、処理中に最初に実行されます。結果は共通のテーブルに追加され、再帰クエリー式の実行に対して参照されます。このプロセスは、中間の結果がなくなるまで、新しい結果に対して繰り返し行われます。

重要

再帰的でない共通テーブル式により、過剰な処理が発生する可能性があります。

デフォルトでは、再帰的な共通テーブル式の runaway 処理を防ぐために、処理は 10000 回に制限されます。プッシュされる再帰的な共通テーブル式はこの制限の対象ではありませんが、他のソース固有の制限が適用される可能性があります。セッション変数 **teiid.maxRecursion** を大きな整数値に設定することで、制限を変更できます。制限を超えると、例外が発生します。

再帰制限が処理の完了前に到達するため、以下の例は失敗します。

```
SELECT teiid_session_set('teiid.maxRecursion', 25);
WITH n (x) AS (values('a') UNION select chr(ascii(x)+1) from n where x < 'z') select * from n
```

3.6.7. SELECT 句

SELECT キーワードで始まり、**SELECT ステートメント** と呼ばれる SQL クエリー。Data Virtualization では、ほとんどの標準の SQL クエリー構成を使用できます。

用途

```
SELECT [DISTINCT|ALL] ((expression [[AS] name])|(group identifier.STAR))*|STAR ...
```

構文ルール

- エイリアス化された式は、出力列名と ORDER BY 句にのみ使用されます。クエリーの他の句では使用できません。
- DISTINCT は、SELECT シンボルが比較されている場合にのみ指定できます。

3.6.8. FROM 句

FROM 句は SELECT、UPDATE、DELETE ステートメントのターゲットテーブルを指定します。

構文の例：

- FROM table [[AS] alias]
- FROM table1 [INNER|LEFT OUTER|RIGHT OUTER|FULL OUTER] JOIN table2 ON join-criteria
- FROM table1 CROSS JOIN table2
- FROM(subquery)[AS] エイリアス

- FROM TABLE(subquery)[AS] エイリアス。詳細は、「ネストされたテーブル」を参照してください。
- FROM table1 JOIN /*+ MAKEDEP */ table2 ON join-criteria
- FROM table1 JOIN /*+ MAKENOTDEP */ table2 ON join-criteria
- FROM /*+ MAKEIND */ table1 JOIN table2 ON join-criteria
- FROM /*+ NO_UNNEST */ vw1 JOIN table2 ON join-criteria
- FROM table1 left outer join /*+ optional */ table2 ON join-criteria詳細は、「Federated optimizations」の **オプションで参加** します。
- FROM TEXTTABLE... 詳細は、TEXTTABLE を参照してください。
- FROM XMLTABLE... 詳細は「XMLTABLE」を参照してください。
- FROM ARRAYTABLE... 詳細は、ARRAYTABLE を参照してください。
- FROM OBJECTTABLE... 詳細は、「OBJECTTABLE」を参照してください。
- FROM JSONTABLE... 詳細は JSONTABLE を参照してください。
- FROM SELECT... 詳細は「Subqueries」の「**Inline views**」を参照してください。

From 句のヒント

from 句のヒントは、通常、影響を受ける句の前にあるコメントブロックに指定されます。MAKEDEP および MAKENOTDEP は、影響を受ける句の後にはcomment でない形式でも表示される可能性があります。複数のヒントをその句に適用する場合は、ヒントを同じコメントブロックに配置する必要があります。

ヒントの例

> FROM */*+ MAKEDEP PRESERVE */* (tbl1 inner join tbl2 inner join tbl3 on tbl2.col1 = tbl3.col1 on tbl1.col1 = tbl2.col1), tbl3 WHERE tbl1.col1 = tbl2.col1

依存する結合ヒント

MAKEIND、M**MAKEDEP**、および **MAKENOTDEP** は、依存する結合動作を制御するのに使用できるヒントです。オプティマイザーがクエリー構造、メタデータ、および費用情報に基づいて最も最適な計画を選択しない場合にのみ使用してください。このヒントは、**FROM** キーワードに続くコメントに表示されます。このヒントは、名前付きテーブルだけでなく、**FROM** 句に対して指定できます。

MAKEIND
: 句が依存する参加の独立（実行）側のものである必要があることを示します。

MAKEDEP
: 句が参加に依存する（フィルターされた）側でなければならないことを示します。

MAKENOTDEP
: 句が参加に依存する（フィルター処理）されないようにします。

MAKEDEP および **MAKEIND** で以下のオプションの **MAX** 引数および **JOIN** 引数を使用できます。

MAKEDEP(JOIN)

結合全体をプッシュする必要があることを示します。

MAKEDEP(NO JOIN)

結合全体をプッシュしないことを示します。

MAKEDEP(MAX:val)

独立した結合は、独立した側からの値の最大数より少ない場合にのみ実行する必要があります。

他のヒント

NO_UNNEST は、サブクエリー FROM 句またはビューに対して指定でき、周りのクエリーでネストされた SQL をマージしないよう planner に指示します。このプロセスは、ビューフラット化 と呼ばれます。このヒントは、Data Virtualization の計画にのみ適用され、ソースクエリーに渡されません。NO_UNNEST は、FROM キーワードに続くコメントに表示されます。

PRESERVE ヒントは、ANSI 結合ツリーに対して使用して、Data Virtualization オプティマイザーが結合を並べ替えることができるようにするのではなく、結合の構造を保持することができます。これは、Oracle ORDERED または MySQL STRAIGHT_JOIN ヒントと同様に機能します。

PRESERVE ヒントの例

```
FROM /*+ PRESERVE */ (tbl1 inner join tbl2 inner join tbl3 on tbl2.col1 = tbl3.col1 on tbl1.col1 =
tbl2.col1)
```

3.6.8.1. ネストされたテーブル

ネストされたテーブルは、**TABLE** キーワードを含む **FROM** 句で表示できます。これらは、通常の join セマンティクスでビューを使用する代わりに使用されます。ネストされたテーブルに含まれるコマンドから展開された列は、結合基準、WHERE 句、およびその他のコンテキストで使用できます。ここでは、FROM 句を展開された列を使用できます。

ネストされたテーブルは、**INNER** および **LEFT OUTER** 結合が使用される限り、前述の **FROM** 句の列参照に相関させることができます。これは、ネストされた式が手順または関数呼び出しである場合に特に便利です。

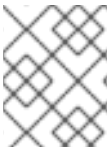
有効なネストされたテーブルの例

```
select * from t1, TABLE(call proc(t1.x)) t2
```

無効なネストされたテーブルの例

以下のネストされたテーブルの例は無効です。これは、**t1** は FROM 句のネストされたテーブルの後に表示されるためです。

```
select * from TABLE(call proc(t1.x)) t2, t1
```



複数の実行

相関ネストされたテーブルを使用すると、相関行ごとに複数のテーブル式の実行が可能になります。

3.6.8.2. XMLTABLE

XMLTABLE 関数は、XQuery を使用して表形式出力を生成します。XMLTABLE 関数は暗黙的にネストされたテーブルで、FROM 句内で使用できます。XMLTABLE は SQL/XML336 仕様の一部です。

用途

```
XMLTABLE([<NSP>,] xquery-expression [<PASSING>] [COLUMNS <COLUMN>, ... ]) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [DEFAULT expression] [PATH string]))
```

NSP - XMLNAMESPACES の定義については、「**XMLELEMENT** in XML 関数 」を参照してください。
PASSING の定義については、XML 関数 の **XMLQUERY** を参照してください。

注記

「 XQuery の最適化 」も参照してください。

パラメーター

- オプションの XMLNAMESPACES 句は、XQuery および COLUMN パス式で使用できる namepaces を指定します。
- xquery-expression は有効な XQuery である必要があります。xquery によって返される各シーケンスアイテムは、COLUMNS 句で定義されている値行を作成するために使用されます。
- COLUMNS が指定されていない場合は、以下の例のように、アイテム全体を XML 値として返す COLUMNS 句と同等です。

  ```
  "COLUMNS OBJECT_VALUE XML PATH '.'"
  ```

- FOR ORDINALITY 列は整数として入力され、1 ベースのアイテム番号を値として返します。
- 各ordinalityの列は型を指定し、オプションで PATH 式と DEFAULT 式を指定します。
- PATH が指定されていない場合、パスは列名と同じです。

構文ルール

- FOR ORDINALITY 列のみを指定できます。
- 列名に重複を含めることはできません。
- バイナリー大規模なオブジェクト(BLOB)データ型を使用できますが、**xs:hexBinary** 値には互換性が同梱されます。xs:base64Binary の場合は、明示的な値コンストラクター（ **xs:base64Binary(<path>)を使用する PATH の回避策を使用** します。
- アレイ以外のタイプが予想される場合は、列式は単一の値に評価される必要があります。
- 配列型を指定すると、シーケンスに要素がない場合は null 値が返されます。
- 値は空の文字列であるため、空の要素は有効な null 値ではありません。**xsi:nil** 属性を使用して要素に null 値を指定します。

XMLTABLE の例

PASSING の使用（1 行を返します [1]）

```
select * from xmltable('/a' PASSING xmlparse(document '<a id="1"/>') COLUMNS id integer PATH '@id') x
```

ネストされたテーブルとして

```
select x.* from t, xmltable('/x/y' PASSING t.doc COLUMNS first string, second FOR ORDINALITY) x
```

無効な多値

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer PATH 'b/@id') x
```

アレイ多値

```
select * from xmltable('/a' PASSING xmlparse(document '<a><b id="1"/><b id="2"/></a>') COLUMNS id integer[] PATH 'b/@id') x
```

nil 要素

```
select * from xmltable('/a' PASSING xmlparse(document '<a xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><b xsi:nil="true"/></a>') COLUMNS id integer PATH 'b') x
```

注記

上記の例では、**nil** 属性(**xsi:nil="true"**)が指定されていない場合に例外がスローされ、**b** を整数値に変換します。

3.6.8.3. ARRAYTABLE

ARRAYTABLE 関数は、表形式出力を生成する配列の入力を処理します。関数自体はプロジェクトの列を定義します。ARRAYTABLE 関数は暗黙的にネストされたテーブルであり、FROM 句内で使用できます。

用途

```
ARRAYTABLE([ROW|ROWS] expression COLUMNS <COLUMN>, ...) AS name
COLUMN := name datatype
```

パラメーター

expression

処理する配列。java.sql.Array または java 配列の値である必要があります。

ROW|ROWS

ROW（デフォルト）が指定されている場合、指定のアレイから 1 行のみが生成されます（通常は 1 つの単独配列）。ROWS が指定されている場合は、複数の行が生成されます。マルチディメンションアレイが予想され、外部アレイの null 以外の要素ごとに 1 行が生成されます。

式が null の場合、行は生成されません。

構文ルール

- 列名に重複を含めることはできません。

アレイテーブルの例

- ネストされたテーブルとして以下を実行します。

```
select x.* from (call source.invokeMDX('some query')) r, arraytable(r.tuple COLUMNS first string, second bigdecimal) x
```

ARRAYTABLE は、ネストされたテーブルで **array_get** 関数を使用するためのショートカットです。

たとえば、以下の ARRAYTABLE 関数は以下のようになります。

```
ARRAYTABLE(val COLUMNS col1 string, col2 integer) AS X
```

array_get 関数を使用する以下のステートメントと同じです。

```
TABLE(SELECT cast(array_get(val, 1) AS string) AS col1, cast(array_get(val, 2) AS integer) AS col2) AS X
```

3.6.8.4. OBJECTTABLE

OBJECTTABLE 関数は、オブジェクト入力を処理して表形式で出力を生成します。関数自体はプロジェクトの列を定義します。OBJECTTABLE 関数は暗黙的にネストされたテーブルであり、FROM 句内で使用できます。

用途

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS colName colType colScript [DEFAULT defaultExpr] ...) AS id
```

パラメーター

lang

処理するスクリプトの大文字と小文字を区別する言語名である任意の文字列リテラル。スクリプトエンジンは、JSR-223 ScriptEngineManager ルックアップを介して利用できるようにする必要があります。

pidUAGE が指定されていない場合は、デフォルトの 'teiid_script' が使用されます。name:: **val** 式の値をスクリプト context. rowScript:: にバインドする識別子。rowScript:: 行の値を作成して行値を作成する文字列リテラル。Iterator が生成する null 以外のアイテムごとに、列が評価されます。列の colName/colType:: ID/data タイプで、任意で DEFAULT 句式 defaultExpr 表現 **defaultExpr** を指定できます。cocoScript:: 列の値に評価されるスクリプトを指定する文字列リテラル。

構文ルール

- 列名に重複を含めることはできません。
- Data Virtualization は、スクリプト実行コンテキストにいくつかの特殊な変数を配置します。CommandContext は、**teiid_context** として利用できます。さらに、**coScripts** は **teiid_row** および **teiid_row_number** にアクセスできます。 **teiid_row** は、行スクリプトで生成される現

在の行オブジェクトです。 **teiid_row_number** は、現在の 1 ベースの行番号です。

- **rowScript** は Iterator に評価されます。結果がすでに Iterator の場合、これは直接使用されます。評価の結果が Iteratable、Array、または Array タイプである場合、イテレーターが取得されます。他のオブジェクトは、単一アイテムの Iterator として処理されます。いずれの場合も、null 行の値はスキップされます。

注記

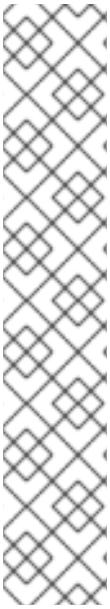
変数の名前付けには制限はありませんが、ターゲット言語で識別子として参照できる名前を選択することが推奨されます。

OBJECTTABLE の例

- 特殊な変数へのアクセス：

```
SELECT x.* FROM OBJECTTABLE('teiid_context' COLUMNS "user" string 'teiid_row.userName',
row_number integer 'teiid_row_number') AS x
```

結果は、それぞれユーザー名と 1 を含む 2 つの列が含まれる行になります。



注記

teiid_script 以外の言語では、通常、Java 機能への無制限のアクセスが許可されます。その結果、デフォルトでは使用は制限されます。**allowed-languages** プロパティーで名前別に使用可能な言語を宣言することで、制限を上書きできます。 通常パーミッションチェックの対象でないビュー定義であっても、OBJECTTABLE を使用するには、**allowed-languages** プロパティーを定義する必要があります。また、ユーザーアカウントに言語アクセス権を設定して、ユーザーが OBJECTTABLE 関数を処理できるようにする必要があります。

- teiid_script の詳細は、次のセクションを参照してください。
- teiid_script 以外の言語の使用を有効にする方法は、「 仮想データベースプロパティー で **allowed-languages** 」を参照してください。
- ユーザーアカウントのパーミッションの設定に関する詳細は、「パーミッション」の「ユーザー **クエリーのパーミッション** 」を参照し て ください。

teiid_script

teiid_script は、アレイ/リストでオブジェクトおよびインデックス化された値の 0 引数以外のメソッドへのアクセスを可能にする単純なスクリプト式言語です。teiid_script 式は、pass または special 変数を参照して始まります。次に、任意の数の **.** アクセサーをチェーンして、式を別の値に評価できます。メソッドには、getFoo ではなく foo などのプロパティー名でアクセスできます。オブジェクトに **getFoo**（）メソッドと **foo** （）メソッドの両方が含まれる場合、accessor **foo** 参照 **foo**（） と **getFoo** を使用して getter を呼び出す必要があります。同じ **.** アクセッサー構文を使用して、1 ベースの正の値を使用してアレイまたはリストインデックスにアクセスします。システム関数 **array_get** と同じロジックが使用されます。つまり、インデックスがバインド外である場合は、例外ではなく **null** を返します。

teiid_script は、起動時に入力が実行されると、実質的に動的に入力されます。アクセサーがオブジェクトに存在しない場合や、メソッドにアクセスできない場合は、例外が発生します。アクセサーチェーン内のどの時点でも null 値に対して評価されると、null が返されます。

teiid_script の例

- VDB の記述文字列を取得するには、以下を実行します。

```
teiid_context.session.vdb.description
```

- VDB の記述文字列の最初の文字を取得するには、以下を実行します。

```
teiid_context.session.vdb.description.toCharArray.1
```

3.6.8.5. TEXTTABLE

TEXTTABLE 関数は、文字の入力を処理して表形式出力を生成します。固定されたファイルフォーマット解析と、ファイルフォーマットの解析の両方を提供します。関数自体はプロジェクトの列を定義します。TEXTTABLE 関数は暗黙的にネストされたテーブルで、FROM 句内で使用できます。

用途

```
TEXTTABLE(expression [SELECTOR string] COLUMNS <COLUMN>, ... [NO ROW DELIMITER |
ROW DELIMITER char] [DELIMITER char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP
integer] [NO TRIM]) AS name
```

ここで、<COLUMN>

```
COLUMN := name (FOR ORDINALITY | ([HEADER string] datatype [WIDTH integer [NO TRIM]]
[SELECTOR string integer]))
```

パラメーター

expression

処理するテキストコンテンツ。文字大きなオブジェクト(CLOB)に変換できます。

セレクター

複数のタイプの行が含まれるファイルで使用されます（例： order ヘッダー、詳細、概要）。
TEXTTABLE SELECTOR は、出力に追加する行を指定します。一致する行はセレクター文字列で開始する必要があります。列区切りファイルのセレクターの後に列区切り文字が続く必要があります。
TEXTTABLE SELECTOR が指定された場合、列の値に SELECTOR を指定することもできます。列 SELECTOR 引数は、指定した SELECTOR 接頭辞が付いた直近のテキスト行を選択し、指定された 1 ベースの整数の位置（セレクター自体を含む）の値を選択します。指定の行を持つそのようなテキスト行や位置が存在しない場合は、null 値が生成されます。SELECTOR 列は、固定幅解析では有効ではありません。

行区切り文字なし

固定の解析で、改行行区切り文字があることを想定すべきではないことを指定します。

行区切り文字

行区切り文字 / 改行を別の文字に設定します。デフォルトは改行文字です。キャリッジリターンの改行を 1 文字として処理するための組み込み処理機能を使用する。ROW DELIMITER が指定された場合、カリッジリターンには特別な対応はありません。

DELIMITER

使用するフィールド区切り文字文字を設定します。デフォルトは 、 です。

QUOTE

フィールド値をラップするために使用される引用符、または修飾子を設定します。デフォルトは 「 です。

ESCAPE

引用符が使用されていない場合に使用するエスケープ文字を設定します。これは、区切り文字や改行文字が先頭文字でエスケープされる場合に使用します（例：\）。

ヘッダー

列名が発生するテキスト行番号（改行ごとにカウント）を指定します。列の HEADER オプションが指定された場合、これは予想されるヘッダー名として使用されます。ヘッダーの前にある行はすべてスキップされます。HEADER が指定されている場合は、ヘッダー行を使用して、大文字と小文字を区別しない名前が一致する TEXTTABLE 列の位置を判断します。これは、列のサブセットのみが必要な状況で特に便利です。HEADER 値が指定されていない場合、デフォルトは 1 に設定されます。HEADER が指定されていない場合、列はテキストの内容と位置に一致することが予想されます。

SKIP

コンテンツを解析する前にスキップするテキスト行の数（新しい行をすべてカウント）を指定します。HEADER は SKIP で指定できます。

FOR ORDINALITY

整数として入力され、1 ベースの項目番号を値として返します。

WIDTH

列の固定幅の長さ（バイト単位ではなく）を指定します。デフォルトの ROW DELIMITER では、CRNL シーケンスは単一の文字としてカウントされます。

トリムなし

TEXTTABLE に指定すると、すべての列およびヘッダーの値に影響します。NO TRIM が列に指定された場合、固定または非修飾テキストの値は先頭および末尾の空白をトリミングされません。

構文ルール

- 幅が 1 つの列に指定される場合は、すべての列に指定し、負の値ではない整数にする必要があります。
- 幅が指定されている場合は、固定の幅解析が使用され、ESCAPE、QUOTE、列 SELECTOR、または HEADER は指定しないでください。
- 幅が指定されていない場合は、NO ROW DELIMITER を使用することはできません。
- 列名に重複を含めることはできません。
- QUOTE、DELIMITER、および ROW DELIMITER に指定される文字はすべて異なっている必要があります。

TEXTTABLE の例

- HEADER パラメーターを使用すると、1 行の ['b'] を返します。

```
SELECT * FROM TEXTTABLE(UNESCAPE('col1,col2,col3\na,b,c') COLUMNS col2 string HEADER)
x
```

- 固定の幅を使用し、2 行 ['a', 'b', 'c'], ['d', 'e', 'f'] を返します。

```
SELECT * FROM TEXTTABLE(UNESCAPE('abc\ndef') COLUMNS col1 string width 1, col2 string width 1, col3 string width 1) x
```

- 行区切り文字なしで固定の幅を使用すると、3 行 ['a'], ['b'], ['c'] を返します。

```
SELECT * FROM TEXTTABLE('abc' COLUMNS col1 string width 1 NO ROW DELIMITER) x
```

- ESCAPE パラメーターを使用すると、1 行 ['a,', 'b'] を返します。

```
SELECT * FROM TEXTTABLE('a:,,b' COLUMNS col1 string, col2 string ESCAPE ':') x
```

- ネストされたテーブルとして以下を実行します。

```
SELECT x.* FROM t, TEXTTABLE(t.clobcolumn COLUMNS first string, second date SKIP 1) x
```

- SELECTORs の使用。2 行 ['c', 'd', 'b'], ['c', 'f', 'b'] を返します。

```
SELECT * FROM TEXTTABLE('a,b\nc,d\nc,f' SELECTOR 'c' COLUMNS col1 string, col2 string col3 string SELECTOR 'a' 2) x
```

3.6.8.6. JSONTABLE

JSONTABLE 関数は JsonPath を使用して表形式出力を生成します。JSONTABLE 機能は暗黙的にネストされたテーブルで、FROM 句内で使用できます。

用途

```
JSONTABLE(value, path [, nullLeafOnMissing] COLUMNS <COLUMN>, ... ) AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [PATH string]))
```

JsonPathも参照してください。

パラメーター

value

有効な JSON ドキュメントを含む clob。

nullLeafOnMissing

false（デフォルト）の場合は、欠落しているリーフに評価されるパスにより例外が発生します。nullLeafOnMissing が true の場合、null 値が返されます。

PATH

文字列は有効な JsonPath である必要があります。配列の値が返されると、行の生成に null 以外の各要素が使用されます。そうでないと、null 以外の項目 1 つを使用して行を作成します。

FOR ORDINALITY

整数として入力された列。1 ベースのアイテム番号をその値として返します。

- 各非理論の列は型を指定し、オプションで PATH を指定します。

- PATH が指定されていない場合、パスは @['name'] コラム名から生成されます。PATH が指定されている場合は、@ で始まる必要があります。これは、パスが現在の行コンテキスト項目の相対パスで処理されることを意味します。

構文ルール

- 列名に重複を含めることはできません。
- JSONTABLE 関数で配列タイプを使用することはできません。

JSON 表の例

渡すと 1 行を返します [1]。

```
select * from jsontable('{"a": {"id":1}}}', '$.a' COLUMNS id integer) x
```

ネストされたテーブルとして以下を実行します。

```
select x.* from t, jsontable(t.doc, '$.x.y' COLUMNS first string, second FOR ORDINALITY) x
```

より複雑なパス：

```
select x.* from jsontable('[{"firstName": "John", "lastName": "Wayne", "children": []}, {"firstName":
"John", "lastName": "Adams", "children":["Sue","Bob"]}]', '$.*' COLUMNS familyName string path
'@.lastName', children integer path '@.children.length()' ) x
```

XMLTABLE との相違点

JSON から表への結果の処理は、以前は JSONTOXML で XMLTABLE を使用することで推奨されていました。ほとんどのタスクでは、JSONTABLE はより簡単な構文を提供します。ただし、以下のような違いがいくつかあります。

- JSONTABLE は、JSON を完全に解析します。XMLTABLE はストリーミング処理を使用して、メモリーのオーバーヘッドを減らします。
- JSONPath は、XQuery より強力なものではありません。XQuery/XPath には多くの関数および操作があり、JsonPath では使用できません。
- JSONPath では、列パスの親参照は許可されません。親階層のルートまたは任意の部分を参照する機能はありません（XPath 内の）。

3.6.9. WHERE 句

WHERE 句は、SELECT、UPDATE、DELETE ステートメントの影響を受けるレコードを制限する基準を定義します。

WHERE の一般的な形式は次のとおりです。

- WHERE 条件

3.6.10. GROUP BY 句

GROUP BY 句は、指定した式の値に基づいて行をグループ化する必要があることを示します。各グループに 1 つの行が返され、必要に応じて HAVING 句に基づいてこれらの集約行をフィルターします。

GROUP BY の一般的な形式は次のとおりです。

```
GROUP BY expression [,expression]*
```

```
GROUP BY ROLLUP(expression [,expression]*)
```

構文ルール

- SELECT 句のエイリアス名に対しては、グループの列参照を作成できません。
- 別のグループで使用される式は、select 句に表示されるはずです。
- グループごとに使用されない SELECT/HAVING/ORDER BY 句の列参照と式は集約関数に表示されるはずです。
- 集約関数が SELECT 句で使用され、GROUP BY が指定されていない場合、暗黙的な GROUP BY は結果を単一グループとして設定された状態で実行されます。この場合、SELECT のすべての列は集約関数である必要があります。これは、グループ全体で他の列の値が修正されないためです。
- GROUP BY 列は、同じタイプである必要があります。

Rollups

通常のグループと同様に、ROLLUP 処理は、HAVING 句が処理される前に論理的に実行されます。式の ROLLUP は、より高い集約レベルで計算された値の集約値を追加して、正規表現と同じ出力を生成します。ROLLUP の N 式では、(expr1)、（expr1、expr1、expr1、expprN-1）などの集計は、出力に他のグルーピング式を null 値として提供します。以下の例では、通常の集約クエリーを使用します。

```
SELECT country, city, sum(amount) from sales group by country, city
```

クエリーは以下のデータを返します。

表3.1 通常の集計クエリーによって返されるデータ

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
UK	birmingham	50000
UK	ロンドン	75000

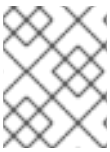
一方、以下の例ではロールアップクエリーを使用します。

ロールアップクエリーから返されるデータ

```
SELECT country, city, sum(amount) from sales group by rollup(country, city)
```

戻り値は以下のようになります。

country	city	sum(amount)
US	St. Louis	10000
US	Raleigh	150000
US	Denver	20000
US	**<null>**	180000
UK	birmingham	50000
UK	ロンドン	75000
UK	**<null>**	125000
<null>	**<null>**	305000



注記

すべてのソースが ROLLUP と互換性がなく、通常の集約処理と比較すると、ROLLUP を使用すると一部の最適化が抑制される可能性があります。

現在、Data Virtualization での ROLLUP の使用は、SQL 仕様と比較して制限されます。

3.6.11. HAVING 句

HAVING 句は **WHERE** 句としてそのまま動作しますが、**GROUP BY** の出力で動作します。**WHERE** 句と同様に、**HAVING** 句で同じ構文を使用できます。

構文ルール

- GROUP BY 句で使用される式には、集約関数（**COUNT** COUNT、**AVG**、**SUM**、**MIN**、**MAX**）、またはグルーピング式のいずれかが含まれている必要があります。

3.6.12. ORDER BY 句

ORDER BY 句は、レコードをソートする方法を指定します。オプションは ASC(ascending)または DESC(descending)です。

用途

```
ORDER BY expression [ASC|DESC] [NULLS (FIRST|LAST)], ...
```

構文ルール

- 並べ替え列は、1 ベースの位置整数、SELECT 句のエイリアス名、SELECT 句式、または関連の式で指定できます。
- 列参照は、エイリアスされた列の式として SELECT 句に表示されるか、FROM 句のテーブルから列を参照できます。列参照が SELECT 句にない場合、クエリーはセット操作にできず、SELECT DISTINCT を指定したり、GROUP BY 句を含んでもできません。
- 関連のない式で、select 句にエイリアス付き式として表示されない式は、設定されていない QUERY の ORDER BY 句で許可されます。式で参照される列は、from 句のテーブル参照から取得する必要があります。列参照はエイリアス名または位置にすることはできません。
- ORDER BY 列は、同じ型である必要があります。
- ORDER BY が LIMIT 句なしでインラインビューまたは表示定義で使用される場合は、Data Virtualization オプティマイザーによって削除されます。
- NULLS FIRST/LAST が指定されている場合は、null は最初または最後にソートされることが保証されます。null 順序が指定されていない場合、結果は通常 null で低い値（Data Virtualization の内部ソート動作）としてソートされます。ただし、すべてのソースが、デフォルトで低い値としてソートされた null で結果を返す訳ではなく、Data Virtualization は、異なる null 順序が異なる結果を返す可能性があります。

警告

位置順序の使用は ANSI SQL 標準ではサポートされなくなり、Data Virtualization で非推奨になった機能です。ORDER BY 句でエイリアス名を使用することが推奨されます。

3.6.13. LIMIT 句

LIMIT 句は SELECT コマンドから返されるレコードの数の制限を指定します。任意のオフセット（スキップする行数）を指定できます。LIMIT 句は、SQL 2008 OFFSET/FETCH FIRST 句を使用して指定することもできます。ORDER BY も指定された場合、OFFSET/LIMIT が適用される前に適用されます。ORDER BY が指定されていない場合、通常は行のサブセットが返す保証はありません。

用途

```
LIMIT [offset,] limit
```

```
LIMIT limit OFFSET offset
```

```
[OFFSET offset ROW|ROWS] [FETCH FIRST|NEXT [limit] ROW|ROWS ONLY]
```

構文ルール

- LIMIT/OFFSET 式は、負の値ではない整数またはパラメーター参照(**?**)である必要があります。**0** のオフセットは無視されます。制限が **0** の場合は、行が返されません。
- FIRST/NEXT という用語は交換可能で、ROW/ROWS です。

- LIMIT 句は、結果が制限の論理アプリケーションと一致しない場合でも、プッシュ操作を抑制すべきではないことを示すオプションの NON_STRICT ヒントを取ることができます。ヒントは、順序のない制限（ **"SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2"** など）でのみ必要です。

LIMIT 句の例

- **LIMIT 100** は最初の 100 レコードを返します（行 1-100）。
- **LIMIT 500, 100** は 500 レコードを省略し、次の 100 レコード（行 501-600）を返します。
- **OFFSET 500 ROWS** は 500 レコードを省略します。
- **OFFSET 500 ROWS FETCH NEXT 100 ROWS ONLY** は 500 レコードのみをスキップし、次の 100 レコードを返します（行 501-600）。
- **FETCH FIRST ROW ONLY** は最初のレコードのみを返します。

3.6.14. INTO 句

警告

テーブルに挿入する INTO 句の使用が非推奨になりました。代わりに query コマンドが含まれる INSERT を使用する必要があります。INSERT の使用方法は、「INSERT コマンド」を参照してください。

into 句が SELECT で指定されると、クエリーの結果は指定されたテーブルに挿入されます。多くの場合、これはレコードを一時テーブルに挿入するために使用されます。INTO 句は FROM 句の直前に付けられます。

用途

```
INTO table FROM ...
```

構文ルール

- **INTO** 句は、**ORDER BY** 句および **LIMIT** 句の後、処理で論理的に適用されます。
- **SELECT INTO** の Data Virtualization のサポートは Microsoft SQL Server に似ています。 **INTO** 句のターゲットは、**SELECT** コマンドの結果が挿入されるテーブルです。
  たとえば、以下のステートメントがあります。

  ```
  SELECT col1, col2 INTO targetTable FROM sourceTable
  ```

  sourceTable から **targetTable** に **col1** と **col2** を挿入します。
- SELECT INTO を UNION クエリーと組み合わせることはできません。
  つまり、**targetTable** に挿入するために、**sourceTable UNION** クエリーから結果を選択できません。

3.6.15. OPTION 句

OPTION キーワードは、ユーザーがコマンドで渡すことができるオプションを示します。これらのオプションは Data Virtualization 固有のもので、SQL 仕様では対応していません。

用途

```
OPTION option (, option)*
```

サポートされているオプション

MAKEDEP テーブル(,table)*

参加に依存するソーステーブルを指定します。

MAKEIND テーブル(,table)*

結合で独立すべきソーステーブルを指定します。

MAKENOTDEP テーブル(,table)*

依存する結合が使用されないようにします。

NOCACHE [table (,table)*]

キャッシュがすべてのテーブルまたは指定のテーブルに使用されるのを防ぎます。

例

```
OPTION MAKEDEP table1
```

```
OPTION NOCACHE
```

OPTION 句で指定されたすべてのテーブルは完全修飾する必要があります。ただし、テーブル名は完全修飾名またはエイリアス名のいずれかと一致します。

MAKEDEP および MAKEIND のヒントは、オプションの引数を取り、依存する参加を制御することができます。拡張ヒントの形式は以下のとおりです。

```
MAKEDEP tbl([max:val] [[no] join])
```

- TBL **(JOIN)は参加全体をプッシュする必要が** あることを意味します。
- TBL **(NO JOIN)は、結合全体をプッシュするべきではないこと** を意味します。
- TB **L(MAX:val)は、独立した結合が** 独立側からの値の最大数より少ない場合にのみ実行されるべきことを意味します。

ヒント

Data Virtualization は、OPTION 句の PLANONLY、DEBUG、および SHOWPLAN の引数を受け入れません。これらのオプションにより以前に提供されていた機能の実行方法は、『クライアント開発者ガイド』を参照してください。

注記

MAKEDEP および MAKENOTDEP のヒントは、**@view1.view2...table の形式でテーブル** 名を取ることができます。たとえば、インラインビュー **"SELECT * FROM(SELECT * FROM tbl1, tbl2 WHERE tbl1.c1 = tbl2.c2)AS v1 OPTION MAKEDEP @v1.tbl1"** の場合、ヒントは v1 ビューに適用するものとして理解されます。

3.7. DDL コマンド

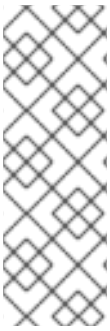
Data Virtualization は、一時テーブルを作成または破棄し、実行時に手順および定義を表示する DDL コマンドのサブセットと互換性があります。現在、一時的なメタデータエントリーを任意にドロップしたり、作成したりすることはできません。 仮想データベースでスキーマを定義するのに使用できる DDL ステートメントの詳細は、「 DDL メタデータ 」を参照してください。

3.7.1. 一時的なテーブル

Data Virtualization で一時（一時）テーブルを作成して使用できます。 *一時的なテーブルは動的に作成されますが、他の物理テーブルとして扱われます。*

3.7.1.1. ローカルの一時テーブル

ローカルの一時テーブルは、INSERT ステートメントで参照するか、CREATE TABLE ステートメントで明示的に定義することで定義できます。暗黙的に作成された一時テーブルには、# で始まる名前が必要です。



注記

Data Virtualization は、一時テーブルが作成される仮想手順のセッションまたはブロックに一時テーブルがスコープされることを意味します。この解釈は SQL 仕様とは異なり、他のデータベースベンダーが実装する解釈とは異なります。ブロックを終了するか、セッションを終了すると、テーブルが破棄されます。呼び出し手順が作成するセッションテーブルおよびその他の一時テーブルは、呼び出された手順には表示されません。同じ名前の一時的なテーブルが呼び出された手順で作成されると、新規インスタンスが作成されます。

作成構文

ローカルの一時テーブルを明示的に作成することも、暗黙的に作成することもできます。

明示的な作成構文

ローカル一時テーブルは、以下の例のように CREATE TABLE ステートメントで明示的に定義できます。

```
CREATE LOCAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY (column, ...)]) [ON COMMIT PRESERVE ROWS]
```

- *SERIAL データ型を使用して NOT NULL および auto-incrementing INTEGER 列を指定します。SERIAL 列の開始値は 1 です。*

暗黙的な作成構文

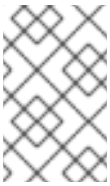
ローカルの一時テーブルは、INSERT ステートメントで参照することで暗黙的に定義できます。

```
INSERT INTO #name (column, ...) VALUES (value, ...)
INSERT INTO #name [(column, ...)] select c1, c2 from t
```

注記

#name が存在しない場合は、値式の指定の列名とタイプを使用して定義されます。

```
INSERT INTO #name (column, ...) VALUES (value, ...)
INSERT INTO #name [(column, ...)] select c1, c2 from t
```



注記

#name が存在しない場合は、ターゲット列名とクエリーベースの列の型を使用して定義されます。ターゲット列が指定されていない場合、列名はクエリーから派生した列名と一致します。

ドロップの構文

```
DROP TABLE name
```

+ 以下の例では、一連のステートメントが 2 つのソースからの一時テーブルを読み込み、レコードを手動で挿入してから、SELECT クエリーで一時テーブルを使用します。

例： ローカルの一時テーブル

```
CREATE LOCAL TEMPORARY TABLE TEMP (a integer, b integer, c integer);
SELECT * INTO temp FROM Src1;
SELECT * INTO temp FROM Src2;
INSERT INTO temp VALUES (1,2,3);
SELECT a,b,c FROM Src3, temp WHERE Src3.a = temp.b;
```

ローカルの一時テーブルの使用の詳細は、「仮想手順」を参照し てください。

3.7.1.2. グローバルな一時テーブル

グローバルな一時テーブルは、デプロイ時に Data Virtualization に提供するメタデータから作成されます。ローカルの一時テーブルとは異なり、実行時にグローバル一時テーブルを作成することはできません。グローバル一時テーブルは、スキーマエントリーで共通の定義を共有します。ただし、各セッションに一時テーブルの新しいインスタンスが作成されます。セッションが終了すると、テーブルが破棄されます。明示的なドロップサポートはありません。グローバル一時テーブルの一般的な用途は、結果を手順から除外することです。

作成構文

```
CREATE GLOBAL TEMPORARY TABLE name (column type [NOT NULL], ... [PRIMARY KEY
(column, ...)]) OPTIONS (UPDATABLE 'true')
```

SERIAL データ型を使用する場合、グローバル一時テーブルの各セッションのインスタンスには独自のシーケンスが設定されます。

一時テーブルを更新する場合は、UPDATABLE を明示的に指定する必要があります。

構文オプションの詳細は、スキーマオブジェクトの DDL メタデータ の **CREATE TABLE** DDL ステートメントを参照してください。

3.7.1.3. グローバルおよびローカルの一時テーブルの一般的な機能

グローバルおよびローカルの一時テーブルは、一般的な機能の一部を共有します。

プライマリーキーの使用

- すべてのキー列は比較する必要があります。
- プライマリーキーを使用する場合は、SQL 比較演算子および IN、LIKE、および ORDER BY 演算子の検索の改善を可能にするクラスター化されたインデックスを作成します。
- **Null** はプライマリーキーの値として使用できますが、すべての null キーを持つ 1 行のみが必要になります。

トランザクション

- **READ_UNCOMMITED** トランザクション分離レベルがあります。分離レベルを高くすることのできるロックメカニズムはありません。また、ロールバックの結果は複数のトランザクション間で一貫性がなくなる可能性があります。同時トランザクションが同じローカル一時テーブルまたはセッションに関連付けられていない場合、トランザクションの分離レベルは効果的にシリアライズ可能です。ローカルの一時テーブルと完全な一貫性が必要な場合は、1 度に 1 つのトランザクションとの接続のみを使用します。この操作のモードは、トランザクションによる接続を追跡する接続プールにより保証されます。

制限事項

- **CREATE TABLE** 構文では、基本的なテーブル定義（列名、タイプ、および null 可能な情報）と任意のプライマリーキーのみを指定できます。グローバル一時テーブルの場合、ephemeral ステートメントの追加メタデータは、一時テーブルインスタンスの作成時に事実上無視されます。ただし、他のテーブルエントリーと同様の計画でメタデータが使用される可能性があります。
- **ON COMMIT PRESERVE ROWS** を使用できます。他の **ON COMMIT** アクションを使用することはできません。
- DROP ステートメントに「drop behavior」オプションを使用することはできません。
- 一時テーブルは、フェイルオーバーセーフではありません。
- インラインではない LOB 値（XML、CLOB、BLOB、JSON、ジオメトリー）は、一時的なテーブルの値ではなく、参照によって追跡されます。一時テーブルに外部ソースからの LOB 値を挿入すると、関連するステートメントまたは接続が閉じられると読み取れない場合があります。

3.7.1.4. 外部一時テーブル

ローカルまたはグローバルな一時テーブルとは異なり、外部一時テーブルは、メタデータの負荷ではなく、ランタイム時に作成される実際のソーステーブルへの参照となります。

外部一時テーブルには、明示的な作成構文が必要です。

-

```
CREATE FOREIGN TEMPORARY TABLE name ... ON schema
```

テーブル作成ボディー構文は標準の CREATE FOREIGN TABLE DDL ステートメントと同じです。詳細は、DDL メタデータ を参照してください。通常、ソースの名前の設定、最新の状態、ネイティブタイプなど、ソーステーブルに適切にアクセスするには DDL OPTION 句の使用が必要になる場合があります。

スキーマ名は、VDB に既存のスキーマ/モデルを指定する必要があります。テーブルには、そのソースにあるかのようにアクセスされます。ただし、Data Virtualization 内では、一時的なテーブルのスコープは、フォレンジではない一時テーブルと同じスコープになります。つまり、外部一時テーブルは Data Virtualization スキーマに属しず、作成先のセッションまたは手順ブロックにスコープ指定されます。

外部一時テーブルの DROP 構文は、foreign 以外の一時テーブルの場合と同じです。

外部一時テーブルの CREATE や対応する DROP はいずれも、pushdown コマンドを発行しません。このメカニズムは、Data Virtualization 内での一時的なソーステーブルを公開します。

FOREIGN TEMPORARY TABLE には 2 つの使用シナリオがあります。1 つ目は、ソースの追加テーブルに動的にアクセスすることです。もう 1 つは、パフォーマンス上の理由から、Data Virtualization のローカル一時テーブルの使用を置き換えます。後者の場合の使用方法パターンは次のようになります。

```
//- create the source table
source.native("CREATE GLOBAL TEMPORARY TABLE name IF NOT EXISTS ... ON COMMIT
DELETE ROWS");
//- bring the table into Data Virtualization
CREATE FOREIGN TEMPORARY TABLE name ... OPTIONS (UPDATABLE true)
//- use the table
...
//- forget the table
DROP TABLE name
```

ネイティブ手順を使用してソース固有の CREATE DDL をソースに渡すことに注意してください。Data Virtualization は現在、CREATE ステートメントに基づいて一時的なテーブルのソース作成のプッシュを試行しません。上記のネイティブ手順などの他のメカニズムでは、最初にテーブルを作成しておく必要があります。また、DDL の定義テーブルはデフォルトでデータ可能ではないので、テーブルは明示的に updatable とマークされることに注意してください。

ソースの一時テーブルの処理は、これを意図した通りに機能させるためにも理解しておく必要があります。すべてのセッションに同じ GLOBAL テーブル定義を使用するソース（Oracle など）、またはセッションスコープの一時テーブルを使用するソース（PostgreSQL など）はトランザクションでアクセスされると機能します。以下の理由によりトランザクションが必要です。

- コミットの動作（DELETE ROWS または DROP など）のソースによってクリーンアップが行われます。Data Virtualization ドロップはソースコマンドを実行せず、保証されていないことに注意してください（例外が発生した場合、データベース接続の損失、ハードシャットダウンなど）。
- トランザクションによる接続を追跡する場合、Data Virtualization によるそのソースの複数の使用が同じ接続/セッションを使用するようにし、同じ一時テーブルとデータを使用します。

ヒント

Data Virtualization では **ON COMMIT** 句を使用することはできません。その結果、ローカルの一時テーブルでは、ソーステーブルの **ON COMMIT** 動作がデフォルトの **PRESERVE ROWS** とは異なる可能性があります。

3.7.2. 変更ビュー

用途

```
ALTER VIEW name AS queryExpression
```

構文ルール

- modify クエリー式の前に、マテリアル化されたビュー定義用のキャッシュヒントを付けることができます。ヒントは、次にマテリアル化されたビューテーブルが読み込まれる際に有効になります。

3.7.3. 手順の変更

用途

```
ALTER PROCEDURE name AS block
```

構文ルール

- ALTER ブロックには **CREATE VIRTUAL PROCEDURE** を含めないでください。
- ALTER ブロックの前に、キャッシュ手順のキャッシュヒントを付けることができます。

3.7.4. 変更トリガー

用途

```
ALTER TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE (AS FOR EACH ROW block)
| (ENABLED|DISABLED)
```

構文ルール

- ターゲット **名** は、更新可能なビューである必要があります。
- Trigger は true スキーマオブジェクトではありません。これらはビューにのみスコープが設定され、名前はありません。
- 更新手順は、指定のトリガーイベントにすでに存在している必要があります。詳細は、「Triggers」を参照してください。

3.8. 手順

Data Virtualization の手順言語を使用して外部手順を呼び出し、仮想手順とトリガーを定義できます。

3.8.1. 手順言語

Data Virtualization で手順言語を使用して、仮想手順を定義することができます。これは、リレーショナルデータベース管理システムに保存された手順と類似しています。この言語を使用して、ビューに対して INSERT、UPDATE、DELETE コマンドを置く変換ロジックを定義できます。これらは更新手順として知られています。詳細は、「 仮想手順および 更新手順 (Triggers) 」を参照してください。

3.8.1.1. コマンドステートメント

コマンドステートメントは、1 つ以上のデータソースに対して DML コマンド、DDL コマンド、または動的 SQL を実行します。詳細は、「DML コマンド および DDL コマンド」を参照してください。

用途

```
command [(WITH|WITHOUT) RETURN];
```

コマンドステートメントの例

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100 WITHOUT RETURN;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

構文ルール

- EXECUTE コマンドステートメントは、IN/OUT、OUT、および RETURN パラメーターにアクセスできます。戻り値にアクセスするには、ステートメントの形式は **var = EXEC proc…** になります。パラメーター構文の名前が OUT または IN/OUT の値にアクセスするには、使用する必要があります。たとえば、**EXEC proc(in_paramgitops'1', out_paramgitopsvar)** は out パラメーターの値を変数 var に割り当てます。パラメーターのデータタイプは、暗黙的に変数のデータ型に変換できることが予想されます。EXECUTE コマンドステートメントの詳細は、「EXECUTE コマンド」を参照してください。

- RETURN 句は、コマンドの結果が手順から返すことができるかどうかを判断します。WITH RETURN がデフォルトです。コマンドが結果セットを返さないか、手順が結果セットを返さないと、RETURN 句は無視されます。WITH RETURN が指定されている場合は、コマンドの結果セットは、予想される手順の結果セットと一致する必要があります。手順結果セットとして、WITH RETURN を実行して成功した最後のステートメントのみが返されます。戻り可能な結果セットがなく、この手順で結果セットが返されることを宣言すると、空の結果セットが返されます。

注記

*INTO 句はテーブルへの挿入にのみ使用されます。'SELECT ... INTO table ... **は、'INSERT INTO table SELECT ...** 変数を割り当てる必要がある場合は、以下のいずれかの方法を使用できます。*

scalar サブクエリーで割り当てステートメントを使用します。

```
DECLARE string var = (SELECT col ...);
```

一時テーブルの使用

```
INSERT INTO #temp SELECT col1, col2 ...;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

アレイの使用

```
DECLARE string[] var = (SELECT (col1, col2) ...);
DECLARE string col1val = var[1];
```

3.8.1.2. 動的 SQL コマンド

動的 SQL では、仮想手順で任意の SQL コマンドの実行が可能になります。動的 SQL は、正確なコマンド形式が実行前に認識されない状況で有用です。

用途

```
EXECUTE IMMEDIATE <sql expression> AS <variable> <type> [, <variable> <type>]* [INTO
<variable>] [USING <variable>=<expression> [,<variable>=<expression>]*] [UPDATE <literal>]
```

構文ルール

- *SQL 式は、262144 文字未満の CLOB または文字列の値である必要があります。*
- ***AS** 句は、実行された SQL 文字列によって返される Projected シンボル名とタイプを定義するために使用されます。**AS** 句シンボルは、実行された SQL 文字列によって返されるシンボルと位置的に一致します。変換できないタイプや、実行された SQL 文字列によって返された列が多すぎると、エラーが発生します。*
- ***INTO** 句は動的 SQL を指定された temp テーブルにプロジェクトします。**INTO** 句を指定すると、Dynamic コマンドは実際に QUERY EXPRESSION を持つ INSERT のように動作するステートメントを実行します。dynamic SQL コマンドが **INTO** 句で一時テーブルを作成する場合は、テーブルのメタデータを定義するために **AS** 句が必要になります。*
- ***USING** 句を使用すると、動的な SQL 文字列に、実行時に指定された値にバインドされる変数参照を含めることができます。これにより、周りの手順の変数名および入力名から SQL 文字列を独立させることができます。動的コマンド **USING** 句では、各変数は短縮名でのみ指定されます。ただし、動的な SQL では、**USING** 変数は **DVAR** に対して完全修飾する必要があります。**USING** 句は、動的 SQL で有効な式として使用される値のみに使用されます。**USING** 句を使用してテーブル名、キーワードなどを置き換えることはできません。これにより、準備済みステートメントで通常のバインド(?)式と同等のシンボルが使用されます。**USING** 句は、必要な文字列操作の量を減らすのに役立ちます。USING 句の値にバインドされていない SQL 文字列の **USING** 記号に参照が行われると、例外が発生します。*

- **UPDATE** 句は、更新モデル数を指定するために使用されます。許可される値は(0,1,*) です。0 は、句が指定されていない場合のデフォルト値です。詳細は、「モデル数の更新」を参照してください。

例： 動的SQL

```
...
/* Typically complex criteria would be formed based upon inputs to the procedure.
 In this simple example the criteria is references the using clause to isolate
 the SQL string from referencing a value from the procedure directly */

DECLARE string criteria = 'Customer.Accounts.Last = DVARS.LastName';

/* Now we create the desired SQL string */
DECLARE string sql_string = 'SELECT ID, First || " " || Last AS Name, Birthdate FROM
Customer.Accounts WHERE ' || criteria;

/* The execution of the SQL string will create the #temp table with the columns (ID, Name, Birthdate).
  Note that we also have the USING clause to bind a value to LastName, which is referenced in the
criteria. */
EXECUTE IMMEDIATE sql_string AS ID integer, Name string, Birthdate date INTO #temp USING
LastName='some name';

/* The temp table can now be used with the values from the Dynamic SQL */
loop on (SELCT ID from #temp) as myCursor
...
```

以下は、動的なSQL 文字列の条件を構築するより複雑な方法の例になります。つまり、仮想手順の **AccountAccess.GetAccounts** には入力 **ID**、**LastName**、および **bday** があります。**ID** に値を指定すると、動的なSQL 条件で使用される唯一の値になります。または、**LastName** に値を指定すると、値が検索文字列であるかどうかを検出します。**LastName** に加えて **bday** を指定すると、**LastName** で複合基準を形成するために使用されます。

例： USING 句を使用した動的SQL と動的に構築された基準文字列

```
...
DECLARE string crit = null;

IF (AccountAccess.GetAccounts.ID IS NOT NULL)
 crit = '(Customer.Accounts.ID = DVARS.ID)';
ELSE IF (AccountAccess.GetAccounts.LastName IS NOT NULL)
BEGIN
 IF (AccountAccess.GetAccounts.LastName == '%')
   ERROR "Last name cannot be %";
 ELSE IF (LOCATE('%', AccountAccess.GetAccounts.LastName) < 0)
   crit = '(Customer.Accounts.Last = DVARS.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE DVARS.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
END
ELSE
 ERROR "ID or LastName must be specified.";

EXECUTE IMMEDIATE 'SELECT ID, First || " " || Last AS Name, Birthdate FROM
```

```
Customer.Accounts WHERE ' || crit USING ID=AccountAccess.GetAccounts.ID,
LastName=AccountAccess.GetAccounts.LastName, BirthDay=AccountAccess.GetAccounts.Bday;
...
```

動的SQL の制限および回避策

dynamic SQL コマンドを使用すると、一時的なテーブルの使用を必要とする割り当てステートメントが生成されます。

割り当ての例

```
EXECUTE IMMEDIATE <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

*条件の一部が存在しない場合、適切な基準の構築は複雑になります。たとえば、**基準** がすでに NULL であった場合、以下の例では **条件** に NULL が残されます。*

例： 危険な NULL 処理

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

条件が使用前に NULL ではないことを確認することが推奨されます。これができない場合は、以下の例のようにデフォルトを指定できます。

例： NULL 処理

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and (Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

*dynamic SQL が **UPDATE** コマンド、**DELETE** コマンド、または **INSERT** コマンドである場合、ステートメントの行数は rowcount 変数から取得できます。*

例： AS および INTO 句

```
/* Execute an update */
EXECUTE IMMEDIATE <expression>;
```

3.8.1.3. 宣言ステートメント

宣言ステートメントは、変数とそのタイプを宣言します。変数を宣言すると、手順内のそのブロックとサブブロックで使用できます。変数はデフォルトで null に初期化されますが、宣言ステートメントの一部として式の値を割り当てることもできます。

用途

```
DECLARE <type> [VARIABLES.]<name> [= <expression>];
```

構文の例

```
declare integer x;
declare string VARIABLES.myvar = 'value';
```

構文ルール

- *サブブロックでは、重複する名前の変数をリダクションできません。*
- *VARIABLES グループが指定されていない場合でも常に暗示されます。*
- *割り当て値は、Assignments ステートメントと同じルールに従います。*
- *標準タイプに加えて、例外変数を宣言する場合は EXCEPTION を指定できます。*

3.8.1.4. 割り当てステートメント

割り当てステートメントは、式を評価することで値を変数に割り当てます。

用途

```
<variable reference> = <expression>;
```

構文の例

```
myString = 'Thank you';
VARIABLES.x = (SELECT Column1 FROM MySchema.MyTable);
```

*割り当ての有効な変数には、宣言ステートメントで宣言された in-scope 変数、または **in_out** および **out** パラメーターが含まれます。**In_out** パラメーターおよび **out** パラメーターは、完全修飾名でアクセスできます。*

例： Out パラメーター

```
CREATE VIRTUAL PROCEDURE proc (OUT STRING x, INOUT STRING y) AS
BEGIN
  proc.x = 'some value ' || proc.y;
  y = 'some new value';
END
```

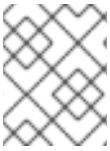
3.8.1.5. 特別な変数

***VARIABLES.ROWCOUNT** 整数変数には、最後に実行した INSERT、UPDATE、または DELETE コマンドステートメントの影響を受ける行の数が含まれます。into 句で動的 SQL **に** より処理される挿入により、**ROWCOUNT** も更新されます。*

サンプル使用例

```
...
UPDATE FOO SET X = 1 WHERE Y = 2;
DECLARE INTEGER UPDATED = VARIABLES.ROWCOUNT;
...
```

*更新以外のコマンドステートメント（**WITH** または **WITHOUT RETURN**）は **ROWCOUNT** を 0 にリセットします。*



注記

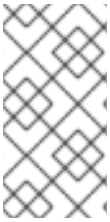
*適切な **ROWCOUNT** 値を取得するには、コマンド文の直後に **ROWCOUNT** を変数に保存します。*

3.8.1.6. 複合ステートメント

複合ステートメントまたはブロックは、一連のステートメントを論理的にグループ化します。複合ステートメントで作成される一時的なテーブルおよび変数は、そのブロックにのみローカルであり、ブロックを終了すると破棄されます。

用途

```
[label :] BEGIN [[NOT] ATOMIC]
    statement*
[EXCEPTION ex
    statement*
]
 END
```



注記

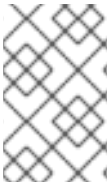
***IF**、**LOOP**、**WHILE** などによってブロックが予想されると、パーサーによって単一のステートメントも使用できます。ブロック **BEGIN** または **END** は想定されていませんが、**BEGIN** と **END** のペアでラップされているかのようにステートメントが実行されます。*

構文ルール

- ***NOT ATOMIC** または **ATOMIC** 句が指定されていない場合、ブロックはアトミックに実行されます。*
- ***ATOMIC** 句が指定されている場合は、ブロックをアトミックに実行する必要があります。トランザクションがすでにスレッドに関連付けられている場合は、追加のアクションは実行されません。保存ポイントまたはサブトランザクションは現在使用されていません。より高いレベルのトランザクションが使用され、ブロックによって例外処理の一部がない状態では、トランザクションはロールバックのみとしてマークされます。そうでない場合は、トランザクションがブロックの実行に関連付けられます。ブロックが正常に完了すると、トランザクションはコミットされます。*
- *ラベルは、このラベルを含むステートメントで使用されるラベルと同じにすることはできません。*
- *変数の割り当てと、暗黙的な結果のカーソルはロールバックによる影響を受けません。ブロックが正常に完了しない場合は、割り当ては引き続き影響を受けます。*

例外処理

***EXCEPTION** 句が複合ステートメント内で使用されている場合、ステートメントから出力された処理例外は、**EXCEPTION** ステートメントに転送される実行フローと共に取得されます。このブロックによって開始されるブロックレベルのトランザクションは、例外ハンドラーが正常に完了するとコミットされます。例外ハンドラーまたは元の例外が例外ハンドラーから出力された場合、トランザクションはロールバックされます。例外ハンドラーステートメントでは、BLOCK に固有の一時テーブルまたは変数は使用できなくなります。*



注記

*例外を処理するだけで、通常ソースで発信されたエラーや関数実行で発生します。低レベルの内部 Data Virtualization エラーまたは Java **RuntimeException** はキャッチされません。*

*キャッチされた例外の処理を支援するために、**EXCEPTION** 句は例外の重要なフィールドを公開するグループ名を指定します。以下の表には、例外グループに含まれる変数をまとめています。*

変数	タイプ	説明
状態	string	SQL 状態
ERRORCODE	integer	エラーまたはベンダーコード。Data Virtualization の内部例外の場合は、TEIIDxxxx コードの整数サフィックスになります。
TEIIDCODE	string	完全な Data Virtualization イベントコード。通常は TEIIDxxxx です。
EXCEPTION	object	キャッチされる例外は **TeiidSQLException** のインスタンスになります。
CHAIN	object	現在の例外をチェーンされた例外または原因。

注記

Data Virtualization は、SQL 状態の使用における ANSI SQL 仕様に完全に準拠していません。基礎となる SQLException 原因のない Data Virtualization エラーの場合は、Data Virtualization コードを使用することが推奨されます。

例外グループ名は、より高いレベルの例外グループまたはループのカーソル名と同じではない場合があります。

例外グループ処理の例

```
BEGIN
   DECLARE EXCEPTION e = SQLEXCEPTION 'this is bad' SQLSTATE 'xxxxx';
   RAISE variables.e;
EXCEPTION e
   IF (e.state = 'xxxxx')
      //in this trivial example, we'll always hit this branch and just log the exception
      RAISE SQLWARNING e.exception;
   ELSE
      RAISE e.exception;
END
```

3.8.1.7. IF ステートメント

IF ステートメントは条件を評価し、結果に応じて 2 つのステートメントのいずれかを実行します。IF ステートメントをネストすると、複雑な分岐ロジックを作成できます。依存の ELSE ステートメントは、IF ステートメントが **false** に評価される場合にのみステートメントを実行します。

用途

```
IF (criteria)
  block
[ELSE
  block]
END
```

IF ステートメントの例

```
IF ( var1 = 'North America')
BEGIN
 ...statement...
END ELSE
BEGIN
 ...statement...
END
```

この基準には、行値を参照する有効なブール式または **IS DISTINCT FROM** 述語を使用できます。**IS DISTINCT FROM** 拡張機能は、以下の構文を使用します。

```
rowVal IS [NOT] DISTINCT FROM rowValOther
```

rowVal および **rowValOther** は、行値グループへの参照です。これは通常、行の値が変更されているかどうかを迅速に判断するために、ビューの更新トリガーではなく使用されます。

例： IS DISTINCT FROM IF ステートメント

```
IF ( "new" IS DISTINCT FROM "old")
BEGIN
 ...statement...
END
```

IS DISTINCT FROM は、同等の null 値を考慮し、UNKNOWN 値を生成しません。

ヒント

Null 値は IF ステートメントの条件で考慮される必要があります。**IS NULL** 基準を使用して、null 値の存在を検出することができます。

3.8.1.8. ループステートメント

LOOP ステートメントは、結果セットを経由したカーソルに使用される反復制御コンストラクトです。

用途

```
[label :] LOOP ON <select statement> AS <cursorname>
    statement
```

構文ルール

- ラベルは、このラベルを含むステートメントで使用されるラベルと同じにすることはできません。

3.8.1.9. 一方で、ステートメント

WHILE ステートメントは、指定した条件が満たされるたびにステートメントを繰り返し実行するために使用される反復制御コンストラクトです。

用途

```
[label :] WHILE <criteria>
    statement
```

構文ルール

- ラベルは、このラベルを含むステートメントで使用されるラベルと同じにすることはできません。

3.8.1.10. 継続ステートメント

CONTINUE ステートメントは、**LOOP** または **WHILE** コンストラクト内で使用され、ループの残りのステートメントをスキップして次のループを続行します。これは **LOOP** または **WHILE** ステートメント内で使用する必要があります。

用途

```
CONTINUE [label];
```

構文ルール

- ラベルが指定されている場合は、**LOOP** または **WHILE** ステートメントが含まれる必要があります。
- ラベルが指定されていない場合、ステートメントは **LOOP** または **WHILE** ステートメントを含む最も近い値に影響します。

3.8.1.11. break ステートメント

BREAK ステートメントは、**LOOP** または **WHILE** コンストラクト内で使用され、ループから分離します。これは **LOOP** または **WHILE** ステートメント内で使用する必要があります。

用途

```
BREAK [label];
```

構文ルール

- ラベルが指定されている場合は、**LOOP** または **WHILE** ステートメントが含まれる必要があります。
- ラベルが指定されていない場合、ステートメントは **LOOP** または **WHILE** ステートメントを含む最も近い値に影響します。

3.8.1.12. leave ステートメント

LEAVE ステートメントは、複合、**LOOP**、または **WHILE** コンストラクト内で使用され、指定されたレベルに残します。

用途

```
LEAVE label;
```

構文ルール

- ラベルは、複合ステートメント、**LOOP**、または **WHILE** ステートメントが含まれる必要があります。

3.8.1.13. 戻り値のステートメント

RETURN ステートメントは適切に手順を終了し、オプションで値を返します。

用途

```
RETURN [expression];
```

構文ルール

- 式が指定されている場合、手順には return パラメーターが必要で、値は想定されるタイプに暗黙的に変換する必要があります。
- 手順に return パラメーターがある場合でも、**RETURN** ステートメントで戻り値を指定する必要はありません。戻り値のパラメーターは割り当てで設定することも、null として残すこともできます。

サンプル使用例

```
CREATE VIRTUAL FUNCTION times_two(val integer)
  RETURNS integer AS
  BEGIN
    RETURN val*2;
  END
```

3.8.1.14. エラーステートメント

ERROR ステートメントは、手順がエラー状態を入力したことを宣言します。このステートメントは、現在のトランザクションが存在する場合は、そのトランザクションもロールバックします。任意の有効な式は **ERROR** キーワードの後に指定できます。

用途

```
ERROR message;
```

例： エラーステートメント

```
ERROR 'Invalid input value: ' || nvl(Acct.GetBalance.AcctID, 'null');
```

ERROR ステートメントは以下に相当します。

```
RAISE SQLEXCEPTION message;
```

3.8.1.15. ステートメントを引き上げます。

RAISE ステートメントは、例外または警告を発生させるために使用されます。例外を発生させると、このステートメントは、現在のトランザクションが存在する場合は、現在のトランザクションもロールバックします。

用途

```
RAISE [SQLWARNING] exception;
```

例外は、例外または例外式への変数参照になります。

構文ルール

- **SQLWARNING** が指定されている場合は、例外が警告としてクライアントに送信され、手順の実行が続行されます。
- null の警告は無視されます。警告以外の例外は、必ずしも例外が発生します。

raise ステートメントの例

```
RAISE SQLWARNING SQLEXCEPTION 'invalid' SQLSTATE '05000';
```

3.8.1.16. 例外式

例外式は、発生または警告として使用できる例外を作成します。

用途

```
SQLEXCEPTION message [SQLSTATE state [, code]] CHAIN exception
```

構文ルール

- いずれの値も null にすることができます。
- **メッセージ** と **状態** は、例外メッセージと SQL 状態を指定する文字列式です。Data Virtualization は、SQL 状態の使用時に ANSI SQL 仕様に完全に準拠しませんが、選択する SQL 状態を設定できます。
- **code** は、ベンダーコードを指定する整数の式です。

- ***例外*** *は、例外または例外式の変数参照であり、結果として得られる例外を親としてチェーンします。*

3.8.2. 仮想手順

仮想手順は、Data Virtualization の手順言語を使用して定義されます。詳細は、「手順言語」を参照してください。

仮想手順では、0 つ以上の INPUT、INOUT、OUT パラメーター、オプションの RETURN パラメーター、および任意の結果セットがあります。仮想手順では、クエリーおよびその他の SQL コマンドの実行、一時テーブルの定義、一時テーブルへのデータの追加、結果セットの実行、ループの使用、および条件付きロジックの使用を行うことができます。

仮想手順の定義

詳細は、スキーマオブジェクトの DDL メタデータ での **手順/機能の作成** *について参照してください。*

オプションの result パラメーターは、常に最初のパラメーターとみなされます。

この手順のボディーでは、有効なステートメントを使用できます。プロシージャー言語ステートメントの詳細は、「ステップ言語」を参照してください。???

ステートメントが明示的なカーソルや値を返すことはありません。代わりに、結果セットを返す手順で実行される、最後に名前のないコマンドステートメントが結果として返されます。そのステートメントの出力は、手順の予想される結果セットとパラメーターと一致する必要があります。

仮想手順のパラメーター

仮想手順では、ゼロまたは ***IN OUT*** *パラメーターを取り、任意の数の* ***OUT*** *パラメーターと任意の* ***RETURN*** *パラメーターを指定できます。各入力には、ランタイム処理中に使用される以下の情報があります。*

名前
入力パラメーターの名前。

データタイプ
入力パラメーターの設計時間タイプ。

デフォルト値
入力パラメーターが指定されていない場合のデフォルト値。

Null 許容型
NO_NULLS,NULLABLE,NULLABLE_UNKNOWN*; パラメーターは null 可能である場合にオプションであり、名前付きパラメーター構文を使用する場合は一覧表示する必要はありません。*

完全修飾名（またはあいまいな場合）を使用して仮想手順のパラメーターを参照します。例： ***MySchema.MyProc.Param1***

例： 入力パラメーターを参照し、 ***GetBalance*** *の手順に* ***Out*** *パラメーターを割り当てる*

```
BEGIN
  MySchema.GetBalance.RetVal = UPPER(MySchema.GetBalance.AcctID);
  SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID =
MySchema.GetBalance.AcctID;
END
```

この手順で ***INOUT*** *パラメーターの値が割り当てられていない場合は、入力のために割り当てられた値*

を保持します。値を割り当てられていない **OUT/RETURN** パラメーターは、デフォルトの NULL 値を保持します。**INOUT/OUT/RETURN** 出力値は、パラメーターの **NOT NULL** メタデータに対して検証されます。

仮想手順の例

以下の例は、カーソルテーブルを記述し、**CONTINUE** および **BREAK** を使用するループを示しています。

LOOP、CONTINUE、BREAK を使用した仮想手順

```
BEGIN
  DECLARE double total;
  DECLARE integer transactions;
  LOOP ON (SELECT amt, type FROM CashTxnTable) AS txncursor
  BEGIN
    IF(txncursor.type <> 'Sale')
    BEGIN
      CONTINUE;
    END ELSE
    BEGIN
      total = (total + txncursor.amt);
      transactions = (transactions + 1);
      IF(transactions = 100)
      BEGIN
        BREAK;
      END
    END
  END
  SELECT total, (total / transactions) AS avg_transaction;
END
```

以下の例では、条件付きロジックを使用して、実行する 2 つの SELECT ステートメントを決定します。

条件付き SELECT を使用した仮想手順

```
BEGIN
  DECLARE string VARIABLES.SORTDIRECTION;
  VARIABLES.SORTDIRECTION = PartsVirtual.OrderedQtyProc.SORTMODE;
  IF ( ucase(VARIABLES.SORTDIRECTION) = 'ASC' )
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVirtual.SupplierInfo.PART_ID;
  END ELSE
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY PartsVirtual.SupplierInfo.PART_ID DESC;
  END
END
```

仮想手順の実行

SQL **EXECUTE** コマンドを使用して手順を実行します。詳細は、「DML コマンド での **コマンドの実行」を参照し** てください。

手順に入力が定義されている場合は、連続リストまたは name=value 構文でそれらを指定します。この手順の他の列や変数のコンテキストでパラメーター名が曖昧である場合、完全な手順名によってスコープ指定された入力パラメーターの名前を使用する必要があります。

仮想プロシージャーコールは **SELECT** などの結果セットを返すため、これは **SELECT** を使用できる場所の多くで使用できます。通常、以下の構文を使用します。

```
SELECT * FROM (EXEC ...) AS x
```

仮想手順の制限

仮想手順では、結果セットを 1 つだけ返すことができます。結果セットを渡す必要がある場合や、複数の結果セットを渡す必要がある場合には、代わりにグローバル一時テーブルの使用を検討してください。

3.8.3. トリガー

トリガーの表示

ビューは、物理ソースよりも抽象化です。通常、複数のデータソースまたは他のビューから情報を結合したり、複数のテーブルから情報を結合したりします。Data Virtualization は、ビューに対して更新操作を実行できます。ビュー（**INSERT**、**UPDATE**、または **DELETE**）に対して実行するコマンドを更新するには、ビューが統合したテーブルとビューが各タイプのコマンドに影響を与える方法を定義するロジックが必要です。この変換ロジック（ **トリガー** とも呼ばれる）は、ビューに対して更新コマンドが実行されると呼び出されます。更新手順では、ビューに対して実行する更新コマンドが、基礎となる物理ソースに対して実行する個々のコマンドに記述されるロジックを定義します。仮想手順と同様に、更新手順は、クエリーやその他のコマンドの実行、一時テーブルの定義、一時テーブルへのデータの追加、結果セットの実行、ループの使用、条件付きロジックの使用などを実行できます。仮想手順の詳細は、「仮想手順」を参照して ください。

INSTEAD OF トリガーは、従来のデータベースで使用するのと同じ方法で、ビュー上でトリガーできます。ビューに対して、1 つの **INSERT**、**UPDATE**、または **DELETE** 操作ごとに **FOR EACH ROW** 手順を 1 つだけ指定できます。

用途

```
CREATE TRIGGER ON view_name INSTEAD OF INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

更新手順の処理

1. ユーザーアプリケーションは SQL コマンドを送信します。
2. このコマンドは、実行されたビューを検出します。
3. コマンドタイプに応じて正しい手順が選択されます（**INSERT**、**UPDATE**、または **DELETE**）。
4. 手順が実行されます。この手順には独自の SQL コマンドが含まれる可能性があります。この手順のコマンドは、呼び出したアプリケーションから受け取るコマンドのタイプとは異なります。
5. この手順で説明されているように、コマンドは、個々の物理データソースまたはその他のビューに発行されます。

6. 変更された行数を表す値は、呼び出したアプリケーションに返されます。

ソーストリガー

Data Virtualization は、ソーステーブルで **AFTER** トリガーを使用できます。**AFTER** トリガーは、変更データキャプチャー(CDC)システムからイベントによって呼び出されます。

使用方法

```
CREATE TRIGGER ON source_table AFTER INSERT|UPDATE|DELETE AS
FOR EACH ROW
...
```

FOR EACH ROW トリガー

FOR EACH ROW コンストラクトのみがトリガーハンドラーとして機能します。**FOR EACH ROW** トリガー手順は、**UPDATE** ステートメントの影響を受けるビュー/ソースの各行のブロックを評価します。**UPDATE** および **DELETE** ステートメントでは、**WHERE** 条件を渡すすべての行になります。**INSERT** ステートメントには、**VALUES** またはクエリー式からの値セットごとに新しい行が1 つ含まれます。ビューの場合、基礎となる手順ロジックの影響に関係なく、更新された行がこの数として報告されます。

用途

```
FOR EACH ROW
  BEGIN ATOMIC
    ...
  END
```

BEGIN キーワードおよび **END** キーワードは、ブロック境界を示すために使用されます。この手順の本文内では、有効なステートメントを使用できます。

注記

ATOMIC キーワードの使用は現時点で後方互換性のためにオプションですが、通常のブロックとは異なり、**INSTEAD OF** トリガーのデフォルト値は atomic です。

更新手順の特別な変数

更新手順を定義する際に、いくつかの特殊な変数を使用できます。

NEW 変数

定義する **UPDATE** および **INSERT** 変換を持つ view/table のすべての属性には **NEW.<column_name> という名前の同等の変数があります。**
INSERT または UPDATE コマンドがビューに対して実行されるか、またはイベントを受信すると、これらの変数はそれぞれ **INSERT VALUES** 句または **UPDATE SET** 句の値に初期化されます。

UPDATE 手順では、これらの変数のデフォルト値（コマンドで設定されていない場合）は古い値になります。INSERT 手順では、これらの変数のデフォルト値は仮想テーブル属性のデフォルト値です。渡された値とデフォルト値を区別するには、この一覧の **CHANGING** 変数を参照してください。

OLD 変数

*定義する **UPDATE** および **DELETE** 変換のすべての view/table 属性には、 **OLD.<column_name> という名前の同等の変数があります。***
*ビューに対して **DELETE** または **UPDATE** コマンドを実行すると、イベントを受信すると、これらの変数は削除または更新される行の現在の値にそれぞれ初期化されます。*

CHANGING 変数

*定義する **UPDATE** および **INSERT** 変換を持つ view/table のすべての属性には、CHAN **GING. <column_name> という名前の同等の変数があります。***
***INSERT** または **UPDATE** コマンドがビューに対して実行されるか、またはイベントを受け取ると、**INPUT** 変数がコマンドによって設定されているかによって、これらの変数は **true** または **false** に初期化されます。**CHANGING** 変数は通常、デフォルトの挿入値とユーザークエリーで指定されたものを区別するために使用されます。*

たとえば、A、B、C のコラムのあるビューの場合は以下のようになります。

ユーザー実行時...	then...
VT（A、B）の値（0、1）に挿入します。	CHANGING.A = true, CHANGING.B = true, CHANGING.C = false
UPDATE VT SET C = 2	CHANGING.A = false, CHANGING.B = false, CHANGING.C = true

キー変数

***INSERT** トリガーから生成されたキーを返すには、KEY グループが利用でき、返された値を割り当てることができます。通常、これには **generated_key** システム機能を使用する必要があります。ただし、すべてのソースが生成されたキーを返すわけではないので、すべての挿入が生成されたキーを提供するわけではありません。*

```
create view v1 (i integer, k integer not null auto_increment primary key)
OPTIONS (UPDATABLE true) as
  select x, y from tbl;
create trigger on v1 instead of insert as
  for each row begin atomic
    -- ... some logic
    insert into tbl (x) values (new.i);
    key.k = cast(generated_key('y') as integer);
  end;
```

更新手順の例

たとえば、A、B、C のコラムのあるビューの場合は以下のようになります。

DELETE の手順例

```
FOR EACH ROW
BEGIN
   DELETE FROM X WHERE Y = OLD.A;
   DELETE FROM Z WHERE Y = OLD.A; // cascade the delete
END
```

UPDATE の手順例

```
FOR EACH ROW
BEGIN
   IF (CHANGING.B)
   BEGIN
      UPDATE Z SET Y = NEW.B WHERE Y = OLD.B;
   END
END
```

その他の使用方法

*ビューの **FOR EACH ROW** 更新手順を使用して、固有の更新を実行する機能を維持しながら、各行トリガーをエミュレートすることもできます。**この BEFORE/AFTER トリガー動作は、フォームの更新手順を使用してターゲットビューに追加のアップデータブルビューを作成することで実行できます。***

```
CREATE TRIGGER ON outerVW INSTEAD OF INSERT AS
FOR EACH ROW
   BEGIN ATOMIC
   --before row logic
   ...

   --default insert/update/delete against the target view
   INSERT INTO VW (c1, c2, c3) VALUES (NEW.c1, NEW.c2, NEW.c3);

   --after row logic
   ...
   END
```

3.9. コメント

テキストを / */ で囲むことで、Data Virtualization に複数行の SQL コメントを追加できます。*

```
/* comment
comment
comment... */
```

また、1 行のコメントを追加することもできます。

```
SELECT ... -- comment
```

コメントをネスト化することもできます。

3.10. 説明の説明

EXPLAIN ステートメントを使用してクエリー計画を取得できます。EXPLAIN ステートメントを使用してクエリー実行計画を取得することは、SQL 言語のネイティブ機能であり、pg/ODBC トランスポートを使用する場合に推奨される方法です。Teiid JDBC クライアントを使用している場合は、SET/SHOW ステートメントを使用することもできます。SET および SHOW ステートメントの詳細は、『クライアント開発者ガイド』を参照してください。

用途

```
EXPLAIN [(explainOption [, ...])] statement

explainOption :=
    ANALYZE [TRUE|FALSE]
  | FORMAT {TEXT|YAML|XML}
```

オプションを指定しないと、デフォルトでは、クエリーを実行せずにテキスト形式でプランが提供されます。

ANALYZE または ANALYZE TRUE を指定した場合、クライアントが NOEXEC オプションを設定していない限りステートメントが実行されます。作成されるプランには、完全に実行されたステートメントからのランタイムノード統計が含まれます。更新を含むすべての副次的な影響は引き続き発生します。トランザクションを使用して不要な影響をロールバックする必要がある場合があります。

これは PostgreSQL と同じ構文ですが、さまざまな形式で提供されるプランは、以前のバージョンの Teiid が提供するものと同じです。

結果を解釈する方法は、「クエリープラン」を参照し てください。

例

```
EXPLAIN (analyze) select * from really_complicated_view
```

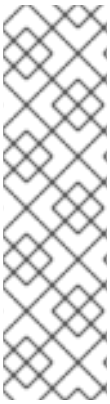
指定されたステートメントの実際の実行からテキストフォーマットのプランを返します。

第4章 データ型

Data Virtualization タイプシステムは Java/JDBC タイプに基づいています。ランタイムオブジェクトは、Long、Integer、boolean、String などの対応する Java クラスによって表されます。詳細は、「ランタイムタイプ」を参照してください。ドメインタイプを使用してタイプシステムを拡張できます。詳細は、「ドメインの DDL メタデータ」を参照してください。

4.1. ランタイムタイプ

Data Virtualization はランタイムタイプのコアセットと連携します。ランタイムタイプは、設計時に type フィールドで定義されるセマンティック型と異なる場合があります。ランタイムタイプは設計時に指定することも、セマンティックタイプに最も近いベースタイプとして自動的に選択されます。



注記

タイプが length、精度、またはスケール引数で宣言されていても、これらの制限はランタイムシステムで事実上無視されますが、OData、ODBC、JDBC によってエッジで強制/報告される可能性があります。地理空間型は同様の方法で動作します。拡張メタデータは、ツール/OData で使用するために SRID、タイプ、ディメンションの数が必要になる場合がありますが、まだ実施されていません。一部のインスタンスでは、SRID が関連付けられたことを確認するには、ST_SETSRID 関数を使用する必要がある場合があります。

表4.1 データ仮想化のランタイムタイプ

タイプ	説明	Java ランタイムクラス	JDBC 型	ODBC タイプ
文字列 または varchar	最大長の 4000 の変数長の文字列。	java.lang.String	VARCHAR	VARCHAR
varbinary	最大期間が 8192 の変数長バイナリー文字列。	byte[] [1]	VARBINARY	VARBINARY

タイプ	説明	Java ランタイムクラス	JDBC 型	ODBC タイプ
char	1 つの 16 ビット文字 - 基本的なマルチコントロールプレーン以外の値を表すことはできません。この制限は、トリミング、textagg、texttable などの単一の文字を想定する関数/式にも適用されます。	java.lang.Character	CHAR	CHAR
boolean	true、false、または null(unknown) できる単一のビットまたはブール値	java.lang.Boolean	BIT	SMALLINT
バイトまたは小さな整数	numeric, integral type, signed 8 ビット	java.lang.Byte	TINYINT	SMALLINT
short または smallint	numeric, integral type, signed 16 ビット	java.lang.Short	SMALLINT	SMALLINT
整数またはシリアル	数値、整数型、署名済み 32 ビット。また、serial 型は null ではないことを意味し、1 から開始する自動増分値があります。シリアル番号タイプは、自動的に UNIQUE ではありません。	java.lang.Integer	INTEGER	INTEGER
long または bigint	numeric, integral type, signed 64bit	java.lang.Long	BIGINT	NUMERIC
biginteger	数値、整数型、最大 1000 桁の任意の精度	java.math.BigInteger	NUMERIC	NUMERIC
浮動小数点または現実	数値、浮動小数点数、32 ビット IEEE 754 浮動小数点番号	java.lang.Float	REAL	FLOAT

タイプ	説明	Java ランタイムクラス	JDBC 型	ODBC タイプ
double	数値、浮動小数点数、64 ビット IEEE 754 浮動小数点番号	java.lang.Double	DOUBLE	DOUBLE
BigDecimal または 10 進数	数値、浮動小数点型、最大 1000 桁の精度（任意の精度）	java.math.BigDecimal	NUMERIC	NUMERIC
date	1 日（年、月、日）を表す日時	java.sql.Date	DATE	DATE
time	日時（時間、分、秒）を表す日時	java.sql.Time	TIME	TIME
timestamp	日時（年、月、日、時間、分、秒）を表す日時。	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	任意の Java オブジェクト。java.lang.336 を実装する必要があります。	すべて	JAVA_OBJECT	VARCHAR
blob	バイトストリームを表すバイナリー大きなオブジェクト。	java.sql.Blob [2]	BLOB	VARCHAR
clob	文字の大きいオブジェクト。文字のストリームを表します。	java.sql.Clob [3]	CLOB	VARCHAR
xml	XML ドキュメント	java.sql.SQLXML[4]	JAVA_OBJECT	VARCHAR
geometry	geospatial オブジェクト	java.sql.Blob [5]	BLOB	BLOB
geography (11.2+)	geospatial オブジェクト	java.sql.Blob [6]	BLOB	BLOB

タイプ	説明	Java ランタイムクラス	JDBC 型	ODBC タイプ
json (11.2+)	JSON 文字のストリームを表す文字大きなオブジェクト。	java.sql.Clob [7]	CLOB	VARCHAR

1. *ランタイムタイプは org.teiid.core.types.BinaryType です。翻訳者は BinaryType 値を明示的に処理する必要があります。代わりに、UDF に byte[] 値が渡されます。*

2. *具体的なタイプは org.teiid.core.types.BlobType となります。*

3. *具体的なタイプは org.teiid.core.types.ClobType となります。*

4. *具体的なタイプは org.teiid.core.types.XMLType となります。*

5. *具体的なタイプは org.teiid.core.types.GeometryType となります。*

6. *具体的なタイプは org.teiid.core.types.GeographyType となります。*

7. *具体的なタイプは org.teiid.core.types.JsonType となります。*

注記

文字、文字列、および文字が大きいオブジェクト(CLOB)タイプは ASCII/extended ASCII 値に制限されません。文字には最大 2^16-1 と String/CLOB のコードを保持することができます。

配列

type 宣言に各アレイディメンションに [] を追加することで、任意のタイプのアレイが指定されます。

例： アレイタイプ

string[]

integer[][]

注記

アレイの処理は通常メモリーにあります。大型のアレイ値の使用には依存しないことを推奨します。通常、大規模なオブジェクト(LOB)の配列はシリアライズ時に正しく処理されません。

4.2. 型変換

データ型は、明示的または暗黙的に別のフォームに変換できます。暗黙的な変換は、開発を容易にするための基準と式で自動的に行われます。明示的なデータタイプ変換では、CONVERT 関数または CAST キーワードを使用する必要があります。

型変換に関する考慮事項

- *すべてのタイプは、OBJECT タイプに暗黙的に変換できます。*
- *OBJECT 型は、他のタイプに明示的に変換できます。*
- *NULL 値は任意のタイプに変換できます。*
- *有効な暗黙的な変換も有効な明示的な変換です。*
- *リテラル値は通常明示的な変換を必要とするシナリオでは、情報が失われない場合に暗黙的な変換を適用できます。*
- *widenComparisonToString が false（デフォルト）の場合、明示的な変換を基準に暗黙的に適用することができないことを検知すると、Data Virtualization は例外を発生させます。*

- *widenComparisonToString が true の場合、比較に応じて幅広い変換が適用されるか、基準が false として扱われます。widenComparisonToString の詳細は、『管理者ガイド』の「システムプロパティー」を参照してください。*

  例

  ```
  SELECT * FROM my.table WHERE created_by = 'not a date'
  ```

  widenComparisonToString が false で、created_by が日付である場合、日付は 日付の値に変換できず、例外結果となります。

- *2 つの型間で許可されない明示的な変換を行うと、実行前に例外が発生します。ランタイム値が実際に変換されないと、処理中に許可される明示的な変換が失敗する可能性があります。*

警告

文字列への float/double/bigdecimal/timestamp の Data Virtualization 変換は、JDBC/Java の定義された出力形式に依存します。プッシュダウン動作はこれらの結果を模倣しようとしますが、実際のソースタイプや変換ロジックによって異なる場合があります。基準や、バリエーションが異なる結果になる可能性がある場所の文字列形式を使用することを想定しないことが推奨されます。

表4.2 型変換

ソースタイプ	有効な暗黙的なターゲットタイプ	有効な明示的なターゲットタイプ
string	clob	char、boolean、byte、short、integer、long、biginteger、float、double、largedecimal、xml [a]
char	string	

ソースタイプ	有効な暗黙的なターゲットタイプ	有効な明示的なターゲットタイプ
boolean	string, byte, short, integer, long, biginteger, float, double, bigdecimal	
byte	string, short, integer, long, biginteger, float, double, bigdecimal	boolean
short	string, integer, long, biginteger, float, double, bigdecimal	boolean、バイト
integer	string, long, biginteger, double, bigdecimal	boolean、バイト、短い、浮動小数点
Long	string, biginteger, bigdecimal, float [b], double [b]	boolean、バイト、短い、整数、浮動小数点、二重
biginteger	string, bigdecimal float [b] double [b]	boolean、bytes、short、integer、long、float、double
bigdecimal	string, float [b] double [b]	boolean、byte、short、integer、long、biginteger、float、double
float	string, bigdecimal, double	boolean、bytes、short、integer、long、largeinteger
double	string, bigdecimal, float [b]	boolean、バイト、短い、整数、long、ビック整数、浮動小数点
date	文字列、タイムスタンプ	
time	文字列、タイムスタンプ	
timestamp	string	date, time
clob		string
json	clob	string
xml		string [c]
geography		geometry

ソースタイプ	有効な暗黙的なターゲットタイプ	有効な明示的なターゲットタイプ

[a] xml への文字列は XMLPARSE(DOCUMENT exp)と同じです。詳細は「XML 関数 の **XMLPARSE** 」を参照してください。

[b] float/double への暗黙的な変換は、 リテラル値に対してのみ発生します。

[c] XML to string は XMLSERIALIZE(exp AS STRING)と同じです。詳細は、XML 関数 の **XMLSERIALIZE** を参照してください。

4.3. 特別な変換ケース

文字列リテラルの変換

Data Virtualization は、SQL ステートメント内の文字列リテラルをそれらの暗黙的な型に自動的に変換します。これは通常、異なるデータタイプを持つ式がリテラル文字列と比較される基準比較で実行されます。以下に例を示します。

```
SELECT * FROM my.table WHERE created_by = '2016-01-02'
```

上記の例では、created_by 列にデータタイプが date の場合、Data Virtualization は自動的に文字列リテラルのデータ型を date に変換します。

ブール値への変換

Data Virtualization は、 リテラル文字列および数値型の値を、以下の表の shwon としてブール値に変換できます。

表4.3 ブール値の変換

タイプ	リテラル値	ブール値
文字列	'false'	false
	'unknown'	null
	その他	true
数値	0	false
	その他	true

日付と時刻への変換

Data Virtualization は、以下の表のように、適切にフォーマットされたリテラル文字列を関連する日付関連のデータ型に変換することができます。

表4.4 日付と時刻への変換

文字列リテラル形式	暗黙的な変換タイプの可能性
yyyy-mm-dd	DATE
hh:mm:ss	TIME
yyyy-mm-dd[hh:mm:ss.[fff...]]	TIMESTAMP

前述の形式は、JDBC の日付タイプによって想定される形式です。他の形式を使用するには、Date 関数および time 関数 の PARSEDATE、PARSETIME、および PARSETIMESTAMP を参照してください。

4.4. エスケープされたリテラル構文

暗黙的な変換に依存せずに、エスケープ構文を使用してデータ型の値を SQL で直接定義できます。指定する文字列値は想定される形式に完全に一致する必要があり、そうでない場合は例外が発生します。

データタイプ	エスケープされた構文	標準のリテラル
BOOLEAN	{b 'true'}	TRUE
DATE	{d 'yyyy-mm-dd'}	DATE 'yyyy-mm-dd'
TIME	{t 'hh-mm-ss'}	TIME 'hh-mm-ss'
TIMESTAMP	{ts 'yyyy-mm-dd[hh:mm:ss.[fff...]]'}	TIMESTAMP 'yyyy-mm-dd[hh:mm:ss.[fff...]]'

第5章 更新可能なビュー

ビューは、updatable とマークできます。多くの場合、ビュー定義を使用すると、INSERT/UPDATE/DELETE 操作を処理するトリガーを手動で定義しなくても、ビューを基本的にデータブルにすることができます。

本質的には、以下のようなクエリーで定義できません。

- セット操作（INTERSECT、EXCEPT、UNION）
- 個別の を選択します。
- 集約（集約関数、GROUP BY、HAVING）。
- LIMIT 句。

UNION ALL は、各 UNION ブランチ自体が本質的にアップデータブルな場合にのみ、本質的にアップデータブルビューを定義できます。UNION ALL で定義されるビューは、パーティション化されたユニオンの場合は、固有の INSERT ステートメントに対応し、INSERT は単一のパーティションに属する値を指定します。詳細は、『Federated optimizations』の「partitioned union 」を参照してください。

列に直接マップされていない view 列は更新できないので、UPDATE set 句でターゲットまたは INSERT 列に対象にすることはできません。

ビューが結合クエリーによって定義されている場合や、WITH 句がある場合、依然としてキャッシュ可能なものである可能性があります。ただし、このような状況ではさらに制限があり、その結果のクエリー計画では複数のステートメントが実行される可能性があります。非単純なクエリーをデータブルにするには、以下の基準が適用されます。

- INSERT/UPDATE は、単一の key-preserved テーブルしか変更できません。
- DELETE 操作を許可するには、1 つの key-preserved テーブルのみが必要です。

key-preserved テーブルの詳細は、「Key-preserved tables」を参照してください。

デフォルトの処理が利用できない場合や、INSERT/UPDATE/DELETE の代替実装が必要な場合は、更新手順またはトリガーを使用して各操作を処理する手順を定義することができます。詳細は、「更新手順(Triggers)」を参照してください。

以下で、本質的に updatable denormalized ビューの例を見てみましょう。

```
create foreign table parent_table (pk_col integer primary key, name string) options (updatable true);

create foreign table child_table (pk_col integer primary key, name string, fk_col integer, foreign key (fk_col) references parent_table (pk_col)) options (updatable true);

create view denormalized options (updatable true) as select c.fk_col, c.name as child_name, p.name from parent_table as p, child_table as c where p.pk_col = c.fk_col;
```

単一の key-preserved テーブル である child_table をターゲットとするため、このビューに対して挿入（fk_col、child_name） の挿入などのクエリーは成功します。ただし、各 parent_ table キーに複数の行を持つ parent_table にマップされるので、正規化（名前）値('a')に挿入 すると失敗します。つまり、これは key-preserved ではありません。

また、子エンティティーが関連付けられている可能性があるため、parent_table に対する INSERT がビューに表示されない可能性があります。

すべてのシナリオが機能する訳ではありません。上記の例では、p. pk_col を使用して定義されるビューと共に denormalized（pk_col、child_name）値(<a')に挿入 すると失敗します。ロジックはまだキーの値と同じであることを考慮しないためです。

5.1. KEY-PRESERVED テーブル

key-preserved テーブルには、クエリーの結果に展開される際に一意のままになるプライマリーキーまたは一意のキーがあります。SELECT 句のキー列を参照するのに実際に必要はないことに注意してください。クエリーエンジンは結合構造を分析して key-preserved テーブルを検出できます。エンジンは、key-preserved テーブルへの参加を外部キーのいずれかに対して行うようにします。

第6章 トランザクション

Data Virtualization は、グローバルトランザクションに参加するために XA トランザクションを使用し、ローカルおよびコマンドスコープのトランザクションをデルメント化します。

Narayana コミュニティープロジェクトで Data Virtualization 用に提供される高度なトランザクション技術に関する情報は、Narayana の ドキュメント を参照し てください。

表6.1 Data Virtualization トランザクションスコープ

スコープ	説明
コマンド	user コマンドは、すべてのソースコマンドが同じトランザクションの範囲内で実行されるかのように処理します。AutoCommitTxn 実行プロパティーは、コマンドレベルのトランザクションの動作を制御します。
ローカル	トランザクション境界は、1つのクライアントセッションによってローカルで定義されます。
グローバル	データ仮想化は、グローバルトランザクションに XA リソースとして参加します。

Data Virtualization のデフォルトのトランザクション分離レベルは READ_COMMITTED です。

6.1. AUTOCOMMITTXN 実行プロパティー

ユーザーレベルのコマンドは、複数のソースコマンドを実行できます。ローカルまたはグローバルトランザクションにない場合のユーザーコマンドのトランザクション動作を制御するには、AutoCommitTxn 実行プロパティーを指定できます。

表6.2 AutoCommitTxn Settings

設定	説明
OFF	各コマンドをトランザクションでラップしないでください。個々のソースコマンドは、全体的なコマンドの成功または失敗に関係なく、コミットまたはロールバックできます。

設定	説明
ON	各コマンドをトランザクションでラップします。このモードは最も安全なものですが、パフォーマンスのオーバーヘッドが発生する場合があります。
検出	これはデフォルト設定です。トランザクションでコマンドを自動的にラップしますが、コマンドがトランザクション安全でないように見える場合のみ、コマンドを自動的にラップします。

トランザクションに関するコマンド安全性の概念は、コマンドタイプ、トランザクション分離レベル、および利用可能なメタデータに基づいて Data Virtualization によって決定されます。以下の基準が当てはまる場合は、ラッピングトランザクションは必要ありません。

- *user コマンドはソースに完全にプッシュされます。*
- *user コマンドは SELECT（XML を含む）で、トランザクション分離は REPEATABLE_READ や SERIALIABLE ではありません。*
- *user コマンドはストアドプロシージャーで、トランザクション分離は REPEATABLE_READ や SERIALIABLE ではなく、更新モデル数はゼロです。詳細は、「モデル数の更新」を参照してください。*

更新数は、モデルの手順メタデータの一部としてすべての手順に設定できます。

6.2. モデル数の更新

「Refresh model count」という用語は、コマンドの実行中にモデルが更新される回数を指します。これは、コマンドを安全に実行するためにトランザクション（任意のスコープ）が必要であるかどうかを判断するために使用されます。

表6.3 モデル数設定の更新

数	説明
0	このコマンドで更新は実行されません。

数	説明
1	このコマンド（およびそのサブコマンド）によって1つのモデルのみが更新されることを示します。その更新の成功または失敗は、コマンドの成功または失敗に対応します。コマンドが失敗した場合は、更新が正常に実行されることができません。実行はトランザクション的安全でないと見なされません。
*	1を超える数値は、実行がトランザクション的に安全でないことを示し、XA トランザクションが必要であることを示します。

6.3. JDBC およびトランザクション

JDBC API の機能

トランザクションのトランザクションスコープは 、 以下の JDBC モードにマッピングします。

コマンド

connection autoCommit プロパティーを true に設定します。

ローカル

Connection autoCommit プロパティーを false に設定します。このトランザクションは、autoCommit を true に設定するか、java.sql.Connection.commit を呼び出すことでコミットされます。java.sql.Connection.rollbackへの呼び出しによってトランザクションをロールバックできます。

グローバル

XAConnection が提供する XAResource インターフェースは、トランザクションを制御するために使用されます。XAConnections は、Data Virtualization が XADataSource、org.teiid.jdbc.TeiidDataSource を介して消費される場合にのみ利用できることに注意してください。JEE コンテナーまたはデータアクセス API は通常、アプリケーションコードの代わりに XA トランザクションを制御します。

J2EE の使用モデル

J2EE では、Bean のトランザクションを管理する以下の方法を利用できます。

クライアント制御

Bean のクライアントはトランザクションを開始および終了します。

bean-managed

Bean 自体はトランザクションを明示的に開始および終了します。

コンテナー管理

アプリケーションサーバーコンテナーは自動的にトランザクションを開始し、終了します。

前述のいずれの場合も、コードおよび記述子の書き込み方法に応じて、トランザクションはローカルまたは XA トランザクションのいずれかになります。XA 仕様では、トランザクション以外のソースと連携するのに、一部のタイプの Bean（ステートフルセッション Bean やエンティティー Bean など）は必要ありません。ただし、仕様に応じて、アプリケーションサーバーはトランザクション以外のソースでこれらの Bean を使用できるようにします。ただし、このような使用方法は移植可能でも予測できない点に注意してください。通常、ほとんどのタイプの EJB アクティビティーに対して移植可能な方法で提供するために、アプリケーションはトランザクションを管理するメカニズムを必要とします。

6.4. 制限事項

- *トランザクション分離レベルのクライアント設定は JDBC コネクターのみに伝播されます。設定は他のコネクタータイプに伝播されません。デフォルトのトランザクション分離レベルは各 XA コネクターに設定できます。ただし、分離レベルは修正され、特定の接続またはコマンドではランタイム時に変更することはできません。*

第7章 データロール

エンタイトルメントとも呼ばれるデータロールは、データアクセスパーミッションを指定する仮想データベースごとに定義されるパーミッションのセットです（作成、読み取り、更新、削除）。データロールは、Data Virtualization がランタイム時に実施し、アクセス違反の監査ログエントリーを提供する粒度の細かいパーミッションシステムを使用します。

データロールを適用する前に、仮想データベースの基本的な設計でソースシステムアクセスを制限したい場合があります。ほとんどの場合、Data Virtualization はインポートされたメタデータで表されるソースエントリーのみにアクセスできます。インポートされたメタデータは、仮想データベースで使用するために必要なものだけに絞り込む必要があります。

データロール検証が有効になり、データロールが仮想データベースに定義されている場合、アクセスパーミッションは Data Virtualization サーバーによって適用されます。データロールの使用は、teiid サブシステム policy-decider-module の設定を削除することで、システム全体で無効にできます。データロールには、行ベースおよびその他の承認チェックに使用できる組み込み セキュリティー機能 もあります。

警告

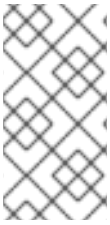
データロールなしでデプロイされた仮想データベースには、任意の認証ユーザーがアクセスできます。

ヒント

デフォルトでは、非表示でないスキーマメタデータは、ユーザーが指定されたオブジェクトに対して何らかのパーミッションがある場合にのみ、JDBC/pg に表示されます。OData アクセスは、デフォルトで非表示ではないメタデータをすべて提供します。JDBC/pg を設定して、全認証ユーザーに非表示でないスキーマメタデータも表示されるようにするには、environment/system プロパティー org.teiid.metadataRequiresPermission を false に設定します。

7.1. パーミッション

複数の方法でデータへのアクセスを制御するか、または付与します。SELECT、UPDATE などの単純なアクセス制限が列レベルまでありました。



注記

少なくとも 1 列を読み取る権限がない限り、列またはテーブルメタデータが JDBC/ODBC ユーザーに表示されません。

パーミッションを使用して結果をフィルターおよびマスクし、更新値を制限/チェックすることもできます。

ユーザークエリーのパーミッション

CREATE、READ、UPDATE、DELETE(CRUD)パーミッションは、VDB 内のリソースパスに設定できます。リソースパスは、列の完全修飾名または一般レベルのモデル（スキーマ）名として固有にできます。特定のパスに付与されたパーミッションは、そのパスと、同じ部分的な名前を共有するリソースパスに適用されます。たとえば、選択を「model」に付与すると、その選択も「model.table」、「model.table.column」などに付与されます。特定のアクションを許可または拒否するには、最も具体的なリソースパスからパーミッションを検索することで決定されます。特定の allow または deny で見つかった最初のパーミッションが使用されます。そのため、高レベルのリソースパス名で非常に一般的なパーミッションを設定し、より具体的なリソースパスに必要な場合にのみ上書きできます。

パーミッション付与は、ロールがアクセスする必要のあるリソースにのみ必要です。パーミッションは、表示および手順の定義で推移的にアクセスされるすべてのリソースではなく、ユーザークエリーの列/テーブル/プロセスのみにも適用されます。そのため、パーミッション付与が同じリソースにアクセスするモデル間で一貫して適用されるようにすることが重要です。

警告

非表示ではないモデルは、ユーザークエリーでアクセスできます。ユーザーアクセスをモデルレベルで制限するには、データロールの確認を有効にするために少なくとも 1 つのデータロールを作成する必要があります。その後、このロールは任意の認証ユーザーにマッピングでき、アクセスできないモデルにパーミッションを付与することはできません。

パーミッションは、SYS および pg_catalog スキーマには適用されません。これらのメタデータレポートスキーマは、ユーザーに関係なく常にアクセスできます。ただし、SYSADMIN スキーマでは、必要に応じてパーミッションが必要になる場合があります。

パーミッションの割り当て

SELECT ステートメントまたはストアドプロシージャーの実行を処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- SELECT- アクセスされるテーブル、または呼び出される手順。
- SELECT- 参照されたすべての列で。

INSERT ステートメントを処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- INSERT: 挿入されるテーブル。
- INSERT: テーブルに挿入されるすべての列。

UPDATE ステートメントを処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- UPDATE- 更新されるテーブルで。
- UPDATE- そのテーブルで更新されるすべての列で。
- SELECT- 条件で参照されるすべての列で。

DELETE ステートメントを処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- DELETE- 削除中のテーブルで
- SELECT- 条件で参照されるすべての列で。

EXEC/CALL ステートメントを処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- **EXECUTE（または SELECT）- *実行される手順。***

任意の機能を処理するには、ユーザーアカウントに以下のアクセス権限が必要です。

- **EXECUTE（または SELECT）- *呼び出される機能。***

***ALTER** または **CREATE TRIGGER** ステートメントを処理するには、ユーザーアカウントに以下のアクセス権限が必要です。*

- *代替方法として、有効なビューまたは手順。**INSTEAD OF Trigger**（更新手順）はまだ完全なスキーマオブジェクトとして処理されず、代わりにビューの属性として処理されます。*

***OBJECTTABLE** 機能を処理するには、ユーザーアカウントに以下のアクセス権限が必要です。*

- **gitopsUAGE - *許可される言語名を指定します。***

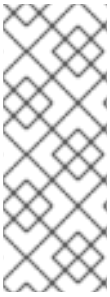
***Data Virtualization** 一時テーブルに対してステートメントを処理するには、以下のアクセス権限が必要です。*

- *任意のロールで **allow-create-temporary-tables** 属性*

- *必要に応じて、**FOREIGN** 一時テーブルに対する操作に必要なターゲットモデル/スキーマに対して* **SELECT、INSERT、UPDATE、DELETE。**

行ベースのセキュリティー

*ユーザークエリーの **CRUD** パーミッションと同様の方法で指定されますが、行ベースおよび列ベースのパーミッションは、より粒度の細かい一貫性のあるレベルでユーザーへ返されるデータを制御するために一緒にまたは別々に使用できます。*



行ベースのセキュリティー

行ベースのセキュリティーに対する GRANT での条件の指定は非推奨になりました。GRANT での条件の指定は、「CREATE POLICY policyName ON schemaName.tblName TO role USING(condition);）」を指定することと同じで、条件がすべての操作に適用されます。

完全修飾テーブル/ビュー/手順に対する POLICY は、指定のロールにより満たされる条件を指定できます。条件は、テーブル/ビュー/手順の列を参照する有効なブール式であればどれでも構いません。手順の結果セット列は proc.col として参照できます。条件は行ベースのフィルターとして機能し、挿入/更新操作のチェックされた制約として機能します。

行ベースの条件の適用

条件は、影響を受けるリソースに対して WHERE 句を更新/削除/選択するために制約が適用されます。そのため、これらのクエリーは、条件を渡す行のサブセット（"SELECT * FROM TBL WHERE something AND 条件 ）に対してのみ有効です。条件は、統合、結合、またはその他の操作のいずれかによって、クエリーでテーブル/ビューがどのように使用されるかに関係なく表示されます。

条件例

```
CREATE POLICY policyName ON schemaName.tblName TO superUser USING ('foo=bar');
```

条件が影響を受ける物理テーブルに対して挿入と更新がさらに検証されるため、挿入/更新の値は、正常な SQL 制約と同じ挿入/更新に条件（evaluate から true）を渡す必要があります。これは、クエリー式、一括挿入/更新などを含むすべてのスタイルで行われます。ビューに対する挿入/更新は、制約に関してチェックされません。

POLICY が適用される操作を制限することで、挿入/更新の制約チェックを無効にできます。

DDL 以外の制約条件の例

```
CREATE POLICY readPolicyName ON schemaName.tblName FOR SELECT,DELETE TO superUser USING ('col>10');
```

当然ながら別の POLICY を追加して、INSERT および UPDATE 操作に対応するには、異なる条件が必要です。

複数の POLICY が同じリソースに適用されると、その条件は OR 経由で誤って累積されます（"(condition1) OR (condition2)...）。そのため、条件「true」で POLICY を作成すると、そのロールのユーザーは指定の操作に対する指定のリソースの行をすべて表示できます。

条件使用時の考慮事項

プッシュされない状況は、パフォーマンスに悪影響を及ぼす可能性があることに注意してください。プッシュされていない条件と同じリソースに対して複数の条件を使用しないと、OR ステートメント全体がプッシュされません。パーミッション条件の挿入が必要な場合は、インラインビューを追加する際には注意してください。これは、ソースと互換性のない場合にパフォーマンスの問題が発生する可能性があるためです。

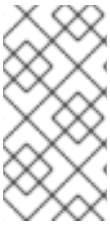
各行の条件をチェックする必要があるため、複数行を挿入/更新操作をプッシュすると抑制されます。

パーミッション条件はロールごとに管理できますが、別の方法として、認証されたロールに条件パーミッションを追加します。この方法でパーミッションを追加すると、この条件は hasRole、user、およびその他のセキュリティー機能を使用するすべてのユーザー が一般的に使用されます。後者アプローチの利点は、静的な行ベースのポリシーを提供することです。その結果、クエリー計画の範囲全体がユーザー間で共有されます。

null 値の処理方法は以下のとおりです。ISNULL チェックを実装し、列が null 可能である場合に null 値が許可されるようにすることが でき ます。

条件を使用する際の制限

- *チェック制約として機能するソーステーブルの条件には、現在相関サブクィジーを含めることはできません。*
- *条件には集約関数やウィンドウ機能が含まれない場合があります。*
- *subqueries で参照されるテーブルと手順には、行ベースのフィルターと列マスクが適用されます。*



注記

行ベースのフィルター条件は、マテリアル化されたビューの負荷に対しても適用されます。

マテリアル化されたビューの生成に使用されるテーブルに、マテリアル化されたビューの結果に影響を与える可能性のある行ベースのフィルター条件がないことを確認する必要があります。

列マスク

完全修飾テーブル/ビュー/手順列に対するパーミッションでも、マスクを指定でき、任意で条件を指定できます。クエリーが送信されると、ロールが参照され、関連するマスク/条件情報が組み合わされ、アクセスによって返された値をマスクするために検索されたケース式を形成します。CRUD が上記で定義されたアクションを許可するのとは異なり、生成されるマスク効果は、ユーザーのクエリーレベルだけでなく、常にユーザークエリーレベルに適用されます。条件と式には、テーブル/ビュー/手順の列を参照する有効な SQL を使用できます。手順の結果セット列は proc.col として参照できます。

列マスクの適用

列のマスクは SELECT にのみ適用されます。列のマスクは、行ベースのセキュリティーに影響を与えた後に論理的に適用されます。ただし、ビューとソーステーブルの両方が行ベースと列ベースのセキュリティーを持つ可能性があるため、実際のビューレベルのマスクはソースレベルのマスクの上に実行できます。条件をマスクと共に指定した場合、有効なマスク式は行のサブセット（CASE WHEN 条件 THEN mask ELSE 列）にのみ影響します。そうでない場合は、条件が TRUE であると想定されます。つまり、マスクがすべての行に適用されます。

複数のロールが列に対してマスクを指定する場合、マスク順序の引数は、検索されたケース式の中で、上から順に優先順位を決定します。たとえば、デフォルトの順序が 0 で、順序が 1 のマスクは「CASE WHEN condition1 THEN mask1 WHEN condition0 THEN mask0 ELSE column」と組み合わされます。

列マスクに関する考慮事項

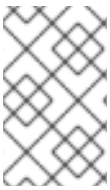
非プッシュマスクの条件/式は、影響を受けるリソースの上にクエリー構成が無害になる可能性があるため、パフォーマンスに悪影響を及ぼす可能性があります。場合によっては、マスクの挿入で、インラインビューの追加でプランを変更する必要がある場合があります。これにより、ソースがインラインビューの使用と互換性がない場合にはパフォーマンスが低下する可能性があります。

order の値を使用してロールごとにマスクを管理する方法に加え、条件/式が hasRole、ユーザー、およびその他のセキュリティー機能を使用して、認証されたユーザー /ロールごとにマスクを指定する

方法です。後者のアプローチの利点は、すべてのクエリー計画をユーザー間で共有できるように、実質的に静的なマスクポリシーがあることです。

列マスクの制限

- 2 つのマスクの order 値が同じ場合、適用される順序は明確に定義されません。
- マスクまたはその条件に集約またはウインドウ関数を含めることはできません。
- subqueries で参照されるテーブルと手順には、行ベースのフィルターと列マスクが適用されます。



注記

マスクは、マテリアル化されたビューの負荷に対しても適用されます。

マテリアル化されたビューの生成に使用されるテーブルにマスクがなくなり、マテリアル化されたビューの結果に影響を与える可能性があることを確認する必要があります。

7.2. ロールマッピング

各 Data Virtualization データロールは、任意の数のコンテナーロールまたは認証済みユーザーにマップできます。

ユーザーは任意の数のコンテナーロールを持つことができ、これにより Data Virtualization データロールのサブセットが暗黙的になります。適用可能な各 Data Virtualization データロールは、ユーザーのパーミッションを累積的に提供します。1 つのロールが他のデータロールのパーミッションを上書きするか、または制御されません。

第8章 システムスキーマ

組み込み SYS および SYSADMIN スキーマは、現在の仮想データベースに対してメタデータテーブルと手順を提供します。

デフォルトでは、ODBC メタデータ pg_catalog のシステムスキーマも公開されます。これは一般的な用途として考慮する必要があります。

メタデータの可視性

SYS システムスキーマテーブルと手順は常に表示され、アクセス可能です。

データロール が使用されている場合、ユーザーはアクセス権限を持つテーブル、ビュー、および手順メタデータエントリーのみを表示できます。キーのすべての列は、エントリーを表示できるようにアクセスする必要があります。

注記

すべてのメタデータを認証ユーザーに表示できるようにするには、環境/システムプロパティー org.teiid.metadataRequiresPermission を false に設定します。

注記

データロールを使用する場合、エントリーの可視性はシステムメタデータのキャッシュの影響を受ける可能性があります。

8.1. SYS スキーマ

公開情報およびアクション用のシステムスキーマ

SYS.Columns

この表は、仮想データベースのすべての要素（列、タグ、属性など）に関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名

コラム名	タイプ	説明
SchemaName	string	スキーマ名
TableName	string	テーブル名
名前	string	要素名（修飾ではない）
Position	integer	グループの位置（1 ベース）
NameInSource	string	ソースの要素の名前
DataType	string	データ仮想化のランタイムデータ型名
スケーリング	integer	10 進数の後の数字の数
ElementLength	integer	要素長（文字列に最も使用される）
sLengthFixed	boolean	長さが固定または変数であるかどうか
SupportsSelect	boolean	要素は SELECT で使用できます。
SupportsUpdates	boolean	要素に値を挿入または更新できる
IsCaseSensitive	boolean	要素では大文字と小文字が区別される
IsSigned	boolean	要素は署名済みの数値です
IsCurrency	boolean	要素は単調値を表します。
IsAutoIncremented	boolean	要素がソースで自動増分される
NullType	string	Null 可能性： "Nullable"、"No Nulls"、"Unknown"
MinRange	string	最小値
MaxRange	string	最大値
DistinctCount	integer	一意の値数。-1 は不明なことを示します。
NullCount	integer	Null 値数。-1 は不明なことを示すことができます。

コラム名	タイプ	説明
SearchType	string	Searchability: "Searchable"、"All Except Like"、"Like Only"、Unsearchable"
形式	string	文字列値の形式
DefaultValue	string	デフォルト値
JavaClass	string	返される Java クラス
精度	integer	数値の数字の数
CharOctetLength	integer	戻り値サイズの測定
radix	integer	数値値の radix
GroupUpperName	string	大文字のフルグループ名
UpperName	string	大文字の要素名
UID	string	要素固有の ID
説明	string	説明
TableUID	string	親テーブルの一意の ID
TypeName	string	タイプ名（ドメイン名である場合があります）
TypeCode	integer	JDBC SQL タイプのコード
ColumnSize	string	数値、精度、文字、長さ、および日付/時刻の場合はリテラル値の文字列長。

SYS.DataTypes

この表はデータタイプの情報を提供します。

コラム名	タイプ	説明
名前	string	データ仮想化のタイプまたはドメイン名
IsStandard	boolean	タイプが基本の場合は True
タイプ	文字列	基本的なユーザー定義、結果セット、ドメインの 1 つ
TypeName	string	設計時間型名（名前と同じ）
JavaClass	string	このタイプの Java クラスが返されました
スケーリング	integer	このタイプの最大スケール
TypeLength	integer	このタイプの最大長
NullType	string	Null 可能性： "Nullable"、"No Nulls"、"Unknown"
IsSigned	boolean	数字が署名されているか？
IsAutoIncremented	boolean	自動インクリメント化か？
IsCaseSensitive	boolean	大文字と小文字が区別されているか？
精度	integer	このタイプの最大精度
radix	integer	このタイプの radix
SearchType	string	Searchability: "Searchable"、"All Except Like"、"Like Only"、"Unsearchable"
UID	string	データ型固有の ID
RuntimeType	string	データ仮想化のランタイムデータ型名
BaseType	string	ベースタイプ
説明	string	タイプの説明
TypeCode	integer	JDBC SQL タイプのコード

コラム名	タイプ	説明
Literal_Prefix	string	リテラル接頭辞
Literal_Prefix	string	リテラルサフィックス

SYS.KeyColumns

この表は、キーによって参照される列に関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
TableName	string	テーブル名
名前	string	要素名
KeyName	string	キー名
KeyType	string	キータイプ： "Primary"、"Foreign"、"Unique" など
RefKeyUID	string	参照されるキー UID
UID	string	キー UID
Position	integer	キーの位置
TableUID	string	親テーブルの一意の ID

SYS.Keys

この表は、プライマリー、外部、および一意キーに関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名

コラム名	タイプ	説明
テーブル名	string	テーブル名
名前	string	キー名
説明	string	説明
NameInSource	string	ソースシステムのキーの名前
タイプ	string	キーのタイプ： "Primary"、"Foreign"、"Unique" など
IsIndexed	boolean	キーがインデックス化されている場合は True
RefKeyUID	string	参照キー UID（外部キーの場合）
RefTableUID	string	参照キーテーブル UID（外部キーの場合）
RefSchemaUID	string	参照キーテーブルスキーマ UID（外部キーの場合）
UID	string	キー一意 ID
TableUID	string	Key Table unique ID
SchemaUID	string	Key Table Schema unique ID
ColPositions	short[]	キーテーブル内の列の位置の配列

SYS.ProcedureParams

これにより、手順パラメーターに関する情報が提供されます。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
ProcedureName	string	手順名
名前	string	パラメーター名

コラム名	タイプ	説明
DataType	string	データ仮想化のランタイムデータ型名
Position	integer	手順引数の位置
タイプ	string	パラメーター方向： "In"、Out、InOut、ResultSet、"ReturnValue"
オプション	boolean	パラメーターは任意です。
精度	integer	パラメーターの精度
TypeLength	integer	パラメーターの値の長さ
スケーリング	integer	パラメーターのスケール
radix	integer	パラメーターの radix
NullType	string	Null 可能性： "Nullable"、"No Nulls"、"Unknown"
説明	string	パラメーターの説明
TypeName	string	タイプ名（ドメイン名である場合があります）
TypeCode	integer	JDBC SQL タイプのコード
ColumnSize	string	数値、精度、文字、長さ、および日付/時刻の場合はリテラル値の文字列長。
DefaultValue	string	デフォルト値

SYS.Procedures

この表は、仮想データベースの手順に関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名

コラム名	タイプ	説明
名前	string	手順名
NameInSource	string	ソースシステムの手順名
ReturnsResults	boolean	結果セットを返します。
UID	string	手順 UID
説明	string	説明
SchemaUID	string	親スキーマー意 ID

SYS.FunctionParams

これにより、関数パラメーターの情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
FunctionName	string	関数名
FunctionUID	string	関数 UID
名前	string	パラメーター名
DataType	string	データ仮想化のランタイムデータ型名
Position	integer	手順引数の位置
タイプ	string	パラメーター方向： "In"、Out、InOut、ResultSet、"ReturnValue"
精度	integer	パラメーターの精度
TypeLength	integer	パラメーターの値の長さ
スケーリング	integer	パラメーターのスケール
radix	integer	パラメーターの radix

コラム名	タイプ	説明
NullType	string	Null 可能性： "Nullable"、"No Nulls"、"Unknown"
説明	string	パラメーターの説明
TypeName	string	タイプ名（ドメイン名である場合があります）
TypeCode	integer	JDBC SQL タイプのコード
ColumnSize	string	数値、精度、文字、長さ、および日付/時刻の場合はリテラル値の文字列長。

SYS.Functions

この表は、仮想データベースの機能に関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
名前	string	関数名
NameInSource	string	ソースシステムの関数名
UID	string	関数 UID
説明	string	説明
IsVarArgs	boolean	関数が変数引数を許可すること

SYS.Properties

この表は、メタモデルエクステンションに基づくすべてのオブジェクトにユーザー定義のプロパティーを提供します。通常、metamodel 拡張機能を使用しない場合は、このテーブルは空になります。

コラム名	タイプ	説明
名前	string	エクステンションプロパティー名
値	string	エクステンションプロパティーの値
UID	string	キーー意 ID
ClobValue	clob	CLOB 値

SYS.ReferenceKeyColumns

この表は、列のキー参照に関する形式を提供します。

コラム名	タイプ	説明
PKTABLE_CAT	string	VDB 名
PKTABLE_SCHEM	string	スキーマ名
PKTABLE_NAME	string	テーブル/表示名
PKCOLUMN_NAME	string	コラム名
FKTABLE_CAT	string	VDB 名
FKTABLE_SCHEM	string	スキーマ名
FKTABLE_NAME	string	テーブル/表示名
FKCOLUMN_NAME	string	コラム名
KEY_SEQ	short	キーシーケンス
UPDATE_RULE	integer	ルールの更新
DELETE_RULE	integer	ルールの削除
FK_NAME	string	FK 名
PK_NAME	string	PK Nmae
DEFERRABILITY	integer	

SYS.Schemas

この表は、システムスキーマ自体（システム）を含む、仮想データベースのすべてのスキーマに関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
名前	string	スキーマ名
IsPhysical	boolean	これがソースを表す場合は True
UID	string	一意の ID
説明	string	説明
PrimaryMetamodelURI	string	このスキーマに使用されるモデルを記述するプライマリーメタモデルの URI

SYS.Tables

この表は、仮想データベースの全グループ（テーブル、ビュー、ドキュメントなど）に関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
名前	string	短いグループ名
タイプ	string	テーブルタイプ（Table、View、Document、...）
NameInSource	string	ソースのこのグループの名前
IsPhysical	boolean	これがソーステーブルの場合は True
SupportsUpdates	boolean	グループを更新できる場合は True

コラム名	タイプ	説明
UID	string	グループ固有の ID
Cardinality	integer	グループの行の概算数
説明	string	説明
IsSystem	boolean	システムテーブルで true の場合は True
SchemaUID	string	親スキーマー意 ID

SYS.VirtualDatabases

この表は、現在接続されている仮想データベースに関する情報を提供します。それらのデータベースには、常に 1 つ（接続のコンテキスト）があります。

コラム名	タイプ	説明
名前	string	VDB の名前
Version	string	VDB のバージョン
説明	string	VDB の説明
LoadingTimestamp	timestamp	タイムスタンプの読み込みが開始されました。
ActiveTimestamp	timestamp	vdb がアクティブになった時点のタイムスタンプ。

SYS.spatial_sys_ref

PostGIS ドキュメントも参照してください。

コラム名	タイプ	説明
srid	integer	空間参照識別子
auth_name	string	標準または標準ボディーの名前
auth_srid	integer	auth_name 認証局の SRID

コラム名	タイプ	説明
srtext	string	よく知られたテキスト表示
proj4text	string	Proj4 ライブラリーでの使用

SYS.GEOMETRY_COLUMNS

PostGIS ドキュメントも参照してください。

コラム名	タイプ	説明
F_TABLE_CATALOG	string	catalog name
F_TABLE_SCHEMA	string	スキーマ名
F_TABLE_NAME	string	テーブル名
F_GEOMETRY_COLUMN	string	列名
COORD_DIMENSION	integer	コーディネートディメンションの数
SRID	integer	空間参照識別子
タイプ	string	ジオメトリータイプ名

注記： coord_dimension および srid プロパティー は、列の {http://www.teiid.org/translator/spatial/2015}coord_dimension および {http://www.teiid.org/translator/spatial/2015}srid 拡張プロパティーに基づいて決定されます。可能な場合、これらの値は関連するインポーターによって自動的に設定されます。値が設定されていない場合、それらはそれぞれ 2 および 0 として報告されます。クライアントロジックが GeoServer との統合などの実際の値を想定している場合には、これらの値を手動で設定できます。

SYS.ArrayIterate

アレイ内の各値に対して、1 つの列が含まれる結果セットを返します。

```
SYS.ArrayIterate(IN val object[]) RETURNS TABLE (col object)
```

例： アレイ

```
select array_get(cast(x.col as string[]), 2) from (exec arrayiterate((('a', 'b'),('c','d')))) x
```

これにより、'b' と 'd' の 2 つの行が生成されます。

8.2. SYSADMIN スキーマ

管理情報およびアクション用のシステムスキーマ

SYSADMIN.Usage

以下の表は、ビューおよび手順の定義方法について説明します。

コラム名	タイプ	説明
VDBName	string	VDB 名
UID	string	オブジェクト UID
object_type	string	オブジェクトのタイプ（StoredProcedure、ForeignProcedure、Table、View、Column など）
名前	string	オブジェクト名または親名
ElementName	string	列またはパラメーターの名前。テーブル/手順を示す null を指定できます。パラメーターレベルの依存関係は実装されていません。
Uses_UID	string	使用されるオブジェクト UID
Uses_object_type	string	使用されるオブジェクトタイプ
Uses_SchemaName	string	使用されるオブジェクトスキーマ
Uses_Name	string	使用されるオブジェクト名または親名

コラム名	タイプ	説明
Uses_ElementName	string	使用されている列またはパラメーター名。テーブル/手順レベルの依存関係を示すために null にすることができます。

手順またはビュー定義で参照されるすべての列、パラメーター、テーブル、または手順が使用済みとして表示されます。同様に、ビュー列を定義する式で参照されるすべての列、パラメーター、テーブル、または手順は、その列で使用されるように表示されます。手順パラメーターの依存関係情報は表示されません。列レベルの依存関係は、一時テーブルまたは共通のテーブルではまだ推測されません。

例： SYSADMIN.Usage

```
SELECT * FROM SYSADMIN.Usage
```

再帰的な共通テーブルクエリーを使用して、推移的な関係を判断できます。

例： すべての着信使用の検索

```
with im_using as (
   select 0 as level, uid, Uses_UID, Uses_Name, Uses_Object_Type, Uses_ElementName
     from usage where uid = (select uid from sys.tables where name='table name' and
schemaName='schema name')
   union all
   select level + 1, usage.uid, usage.Uses_UID, usage.Uses_Name, usage.Uses_Object_Type,
usage.Uses_ElementName
     from usage, im_using where level < 10 and usage.uid = im_using.Uses_UID) select * from
im_using
```

例： 送信使用をすべて検索

```
with uses_me as (
```

```
   select 0 as level, uid, Uses_UID, Name, Object_Type, ElementName
     from usage where uses_uid = (select uid from sys.tables where name='table name' and
schemaName='schema name')
   union all
   select level + 1, usage.uid, usage.Uses_UID, usage.Name, usage.Object_Type,
usage.ElementName
     from usage, uses_me where level < 10 and usage.uses_uid = uses_me.UID) select * from
uses_me
```

SYSADMIN.MatViews

以下の表は、仮想データベース内のすべての優れたビューに関する情報を提供します。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
名前	string	短いグループ名
TargetSchemaName	string	マテリアル化されたテーブルスキーマの名前。内部資料用に null になります。
TargetName	string	マテリアル化されたテーブルの名前
有効	boolean	マテリアル化されたテーブルが現在有効な場合は True。外部マテリアル化の null になります。
LoadState	boolean	ロード状態 - **NEEDS_LOADING**、**LOADING**、**LOADED**、**FAILED_LOAD** のいずれかを使用できます。外部マテリアル化の null になります。
更新済み	timestamp	最後の完全更新のタイムスタンプ。外部マテリアル化の null になります。
Cardinality	integer	マテリアル化されたビューテーブルの行数。外部マテリアル化の null になります。

有効な LoadState、Updated および Cardinality は、SYSADMIN.matViewStatus の手順を使用して外部のマテリアル化されたビューについてチェックできます。

例： SYSADMIN.MatViews

```
SELECT * FROM SYSADMIN.MatViews
```

SYSADMIN.VDBResources

以下の表は、現在の VDB コンテンツを示しています。

列名	タイプ	説明
resourcePath	string	コンテンツへのパス。
コンテンツ	blob	Blob としての内容。

例： SYSADMIN.VDBResources

```
SELECT * FROM SYSADMIN.VDBResources
```

SYSADMIN.Triggers

以下の表は、仮想データベースでのトリガーを示しています。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
TableName	string	テーブル名

コラム名	タイプ	説明
名前	string	トリガー名
TriggerType	string	トリガータイプ
TriggerEvent	string	イベントのトリガー
ステータス	string	有効
本文	clob	アクションのトリガー（約 EACH ROW ...）
TableUID	string	テーブル固有の ID

例： SYSADMIN.Triggers

```
SELECT * FROM SYSADMIN.Triggers
```

SYSADMIN.Views

以下の表は、仮想データベースのビューを示しています。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
名前	string	名前の表示
本文	clob	定義ボディーの表示（選択...）
UID	string	テーブル固有の ID

例： SYSADMIN.Views

```
SELECT * FROM SYSADMIN.Views
```

SYSADMIN.StoredProcedures

以下の表は、仮想データベースの StoredProcedures を示しています。

コラム名	タイプ	説明
VDBName	string	VDB 名
SchemaName	string	スキーマ名
名前	string	手順名
本文	clob	手順定義ボディー(BEGIN ...)
UID	string	一意の ID

例： SYSADMIN.StoredProcedures

```
SELECT * FROM SYSADMIN.StoredProcedures
```

SYSADMIN.Requests

以下の表は、仮想データベースに対するアクティブなリクエストを示しています。

```
VDBName string(255) NOT NULL,
```

コラム名	タイプ	説明
VDBName	string	VDB 名
SessionId	string	セッション識別子
ExecutionId	Long	実行識別子

コラム名	タイプ	説明
コマンド	clob	実行されるクエリー
StartTimestamp	timestamp	開始タイムスタンプ
TransactionId	string	トランザクションマネージャーが報告したトランザクション識別子
ProcessingState	string	処理状態。PROCESSING、DONE、CANCELED のいずれかを使用できます。
ThreadState	string	スレッド状態。RUNNING、QUEUED、IDLE のいずれかを使用できます。

SYSADMIN.Sessions

以下の表は、仮想データベースに対してアクティブなセッションを示しています。

コラム名	タイプ	説明
VDBName	string	VDB 名
SessionId	string	セッション識別子
UserName	string	username
CreatedTime	timestamp	セッションが作成された時点のタイムスタンプ
ApplicationName	string	クライアントが報告したアプリケーション名
IPAddress	string	クライアントが報告した IP アドレス

SYSADMIN.Transactions

以下の表は、アクティブなトランザクションを示しています。

コラム名	タイプ	説明
TransactionId	string	トランザクションマネージャーが報告したトランザクション識別子
SessionId	string	セッションが現在トランザクションに関連付けられている場合のセッション識別子
StartTimestamp	timestamp	トランザクションの開始時間
スコープ	string	トランザクションのスコープはGLOBAL、LOCAL、REQUEST、INHERITED のいずれかです。INHERITED は、トランザクションが呼び出しスレッド（組み込み使用）にすでに関連付けられていることを意味します。

注記： 特定のセッションに関連付けられていないトランザクションは常に表示されます。セッションに関連するトランザクションは、現在の VDB のセッション用である必要があります。

SYSADMIN.isLoggable

ロギングが指定のレベルおよびコンテキストで有効になっているかどうかをテストします。

```
SYSADMIN.isLoggable(OUT loggable boolean NOT NULL RESULT, IN level string NOT NULL
DEFAULT 'DEBUG', IN context string NOT NULL DEFAULT 'org.teiid.PROCESSOR')
```

ロギングが有効な場合に true を返します。レベルは log4j レベルのいずれかになります。レベルは OFF、FATAL、ERROR、WARN、INFO、DEBUG、TRACE です。レベルはデフォルトで 'DEBUG' であり、コンテキストのデフォルトは 'org.teiid.PROCESSOR' です。

例： isLoggable

```
IF ((CALL SYSADMIN.isLoggable(context=>'org.something'))
BEGIN
  DECLARE STRING msg;
  // logic to build the message ...
  CALL SYSADMIN.logMsg(msg=>msg, context=>'org.something')
END
```

SYSADMIN.logMsg

基礎となるロギングシステムにメッセージをログに記録します。

```
SYSADMIN.logMsg(OUT logged boolean NOT NULL RESULT, IN level string NOT NULL
DEFAULT 'DEBUG', IN context string NOT NULL DEFAULT 'org.teiid.PROCESSOR', IN msg
object)
```

メッセージがログに記録された場合は true を返します。レベルは log4j レベルのいずれかになります（OFF、FATAL、ERROR、WARN、INFO、DEBUG、TRACE）。レベルはデフォルトで 'DEBUG' で、コンテキストのデフォルトは 'org.teiid.PROCESSOR' に設定されます。null msg オブジェクトは文字列「null」としてログに記録されます。

例： logMsg

```
CALL SYSADMIN.logMsg(msg=>'some debug', context=>'org.something')
```

上記の例では、デフォルトのレベル DEBUG のメッセージ 'some debug' がコンテキスト org.something に記録されます。

8.2.1. SYSADMIN.refreshMatView

内部マテリアル化されたビューの完全更新/負荷。整数 RowsUpdated を返します。-1 は負荷が進行中であることを示します。それ以外の場合は、テーブルのカーディナリティーが返されます。詳細は『キャッシングガイド』 を参照してください。

SYSADMIN.loadMatView も参照してください。

```
SYSADMIN.refreshMatView(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName
string NOT NULL, IN Invalidate boolean NOT NULL DEFAULT 'false')
```

8.2.2. SYSADMIN.refreshMatViewRow

内部マテリアル化されたビューの行を更新します。

整数 RowsUpdated を返します。-1 はマテリアル化されたテーブルが現在無効であることを示します。0 は、指定した行がライブデータクエリーまたはマテリアル化されたテーブルに存在しなかったことを示します。詳細は『キャッシングガイド』を参照してください。

```
SYSADMIN.CREATE FOREIGN PROCEDURE refreshMatViewRow(OUT RowsUpdated integer NOT NULL RESULT, IN ViewName string NOT NULL, IN Key object NOT NULL, VARIADIC KeyOther object)
```

例： SYSADMIN.refreshMatViewRow

マテリアル化されたビューの SAMPLEMATVIEW には、以下のように TestMat Model の下に 3 つの行があります。

id	a	b	c
100	a0	b0	c0
101	a1	b1	c1
102	a2	b2	c2

プライマリーキーに 1 つの列、id しか含まれていないと仮定して、2 番目の行を更新します。

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101')
```

プライマリーキーに複数の列、a および b が含まれている場合は、2 番目の行を更新します。

```
EXEC SYSADMIN.refreshMatViewRow('TestMat.SAMPLEMATVIEW', '101', 'a1', 'b1')
```

8.2.3. SYSADMIN.refreshMatViewRows

内部マテリアルビューで行を更新します。

整数 RowsUpdated を返します。-1 はマテリアル化されたテーブルが現在無効であることを示します。ライブデータクエリーまたはマテリアル化されたテーブルに存在しない行は、RowsUpdated をカウントしません。詳細は、Teiid Caching Guide を参照してください。

```
SYSADMIN.refreshMatViewRows(OUT RowsUpdated integer NOT NULL RESULT, IN
ViewName string NOT NULL, VARIADIC Key object[] NOT NULL)
```

例： SYSADMIN.refreshMatViewRows

SYSADMIN.refreshMatViewRow の例で、SAMPLEMATVIEW を引き続き使用します。プライマリーキーに 1 つの列 id のみが含まれる場合、すべての行を更新します。

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100',), ('101',), ('102',))
```

プライマリーキーにさらに列、id、プライマリーキーが含まれる場合には、すべての行を更新します。

```
EXEC SYSADMIN.refreshMatViewRows('TestMat.SAMPLEMATVIEW', ('100', 'a0', 'b0'), ('101',
'a1', 'b1'), ('102', 'a2', 'b2'))
```

8.2.4. SYSADMIN.setColumnStats

指定の列の統計を設定します。

```
SYSADMIN.setColumnStats(IN tableName string NOT NULL, IN columnName string NOT
NULL, IN distinctCount long, IN nullCount long, IN max string, IN min string)
```

すべての統計値は null 可能です。null stat 値を渡すと、対応するメタデータ値は変更されません。

8.2.5. SYSADMIN.setProperty

指定のレコードのエクステンションメタデータプロパティーを設定します。拡張メタデータは通常、Translators により使用されます。

```
SYSADMIN.setProperty(OUT OldValue clob NOT NULL RESULT, IN UID string NOT NULL, IN
Name string NOT NULL, IN "Value" clob)
```

値を null に設定するとプロパティーが削除されます。

例： プロパティーセット

```
CALL SYSADMIN.setProperty(uid=>(SELECT uid FROM TABLES WHERE name='tab'),
name=>'some name', value=>'some value')
```

上記の例では、テーブルタブにプロパティー 'some name'='some value' を設定します。

注記

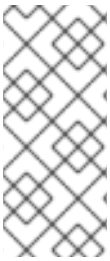
この手順を使用しても、関連付けられた準備済みプランの再計画は発生しません。

ビルトインの teiid_* namespace からのプロパティーは、短い形式 - namespace:key フォームを使用して設定できます。

8.2.6. SYSADMIN.setTableStats

指定のテーブルの統計を設定します。

```
SYSADMIN.setTableStats(IN tableName string NOT NULL, IN cardinality long NOT NULL)
```



注記

SYSADMIN.setColumnStats、SYSADMIN.setProperty、SYSADMIN.setTableStats はメタデータの手順です。

SYSADMIN.matViewStatus

matViewStatus は、schemaName および viewName 経由でマテリアル化されたビューのステータスを取得するために使用されます。

TargetSchemaName、TargetName、Valid、LoadState、Updateed、Cardinality、LoadNumber、OnErrorAction が含まれるテーブルを返します。

```
SYSADMIN.matViewStatus(IN schemaName string NOT NULL, IN viewName string NOT NULL)
RETURNS TABLE (TargetSchemaName varchar(50), TargetName varchar(50), Valid boolean,
LoadState varchar(25), Updated timestamp, Cardinality long, LoadNumber long,
OnErrorAction varchar(25))
```

SYSADMIN.loadMatView

loadMatView は、内部または外部のマテリアル化されたテーブルの完全な更新を実行するために使用されます。

整数の RowsInserted を返します。-1 は、マテリアル化されたテーブルが現在読み込み中であることを示します。そして -3 は、負荷の実行時に例外が発生したことを示します。詳細は『キャッシングガイド』を参照してください。

```
SYSADMIN.loadMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL, IN
invalidate boolean NOT NULL DEFAULT 'false') RETURNS integer
```

例： loadMatView

```
exec SYSADMIN.loadMatView(schemaName=>'TestMat',viewname=>'SAMPLEMATVIEW',
invalidate=>'true')
```

SYSADMIN.updateMatView

updateMatView 手順は、更新基準に基づいて内部または外部のマテリアル化されたテーブルのサブセットを更新するのに使用します。

更新基準は、適格名でビュー列を参照することがあり ます が、ビュー名では . のすべてのインスタンスは _ に置き換えられます。エイリアスが実際に使用されているためです。

整数 RowsUpdated を返します。-1 はマテリアル化されたテーブルが現在無効であることを示します。および-3 は、更新の実行時に例外が発生したことを示します。詳細は『キャッシングガイド』を参照してください。

```
SYSADMIN.updateMatView(IN schemaName string NOT NULL, IN viewName string NOT NULL,
IN refreshCriteria string) RETURNS integer
```

SYSADMIN.updateMatView

SYSADMIN.refreshMatViewRow の例で、SAMPLEMATVIEW を引き続き使用します。ビュー行を更新します。

```
EXEC SYSADMIN.updateMatView('TestMat', 'SAMPLEMATVIEW', 'id = ''101'' AND a = ''a1''')
```

SYSADMIN.cancelRequest

指定のセッションの実行 ID で識別されたユーザー要求をキャンセルします。

SYSADMIN.REQUESTS も参照してください。

```
SYSADMIN.cancelRequest(OUT cancelled boolean NOT NULL RESULT, IN SessionId string
NOT NULL, IN executionId long NOT NULL)
```

例： Cancel

```
CALL SYSADMIN.cancelRequest('session id', 1)
```

SYSADMIN.terminateSession

指定の識別子でセッションを終了します。

SYSADMIN.SESSIONS も参照してください。

```
SYSADMIN.terminateSession(OUT terminated boolean NOT NULL RESULT, IN SessionId
string NOT NULL)
```

例： Termination

```
CALL SYSADMIN.terminateSession('session id')
```

SYSADMIN.terminateTransaction

トランザクションをロールバックとしてマーク付けし、セッションに関連付けられたトランザクションを終了します。

SYSADMIN.TRANSACTIONS も参照してください。

```
SYSADMIN.terminateTransaction(IN sessionid string NOT NULL)
```

注記

セッションに関連するトランザクションのみをキャンセルすることはできません。

例：終了

```
CALL SYSADMIN.terminateTransaction('session id')
```

第9章 翻訳者

Data Virtualization は Teiid Connector Architecture(TCA)を使用します。これは、外部システムと統合するための強力なメカニズムを提供します。TCA は、プッシュダウンに使用できる SQL コンストラクトや外部システムからメタデータをインポートする機能として、Data Virtualization と外部システム間の共通のクライアントインターフェースを定義します。

Translator は TCA の中核で、Data Virtualization と外部システム間のブリッジロジックとして機能します。

トランスフォーマーには、設定可能なさまざまなプロパティーを含めることができます。これらは、データの取得方法を決定する実行プロパティーと、インポート用に読み取るメタデータを決定するインポート設定に分割されます。

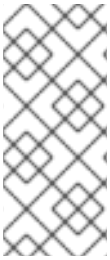
通常、トランスレーターの実行プロパティーには適切なデフォルト値があります。Derby translator などの特定のトランスレータータイプの場合、ベース実行プロパティーはすでにソースと一致するように調整されています。ほとんどの場合、ユーザーは値を調整する必要はありません。

表9.1 ベース実行プロパティー - すべての翻訳者によって共有される

名前	説明	デフォルト
Immutable	ソースが変更されないことを示すには、**true** に設定します。トランザクション機能は NONE として報告され、更新コマンドは失敗します。	false
RequiresCriteria	**true** に設定して、ソース SELECT/UPDATE/DELETE クエリーに where 句が必要であることを示します。	false
SupportsOrderBy	ORDER BY 句を使用できることを示すには、**true** に設定します。	false
SupportsOuterJoins	**true** に設定して、OUTER JOINs が使用できることを示します。	false
SupportsFullOuterJoins	**SupportsOuterJoins** が true に設定されている場合、 **true** は FULL OUTER JOINs を使用できることを示します。	false

名前	説明	デフォルト
SupportsInnerJoins	**true** に設定して INNER JOINs を使用できることを示します。	false
SupportedJoinCriteria	結合機能が有効な場合は、結合条件として使用できる基準を定義します。可能性は、ANY、WiveTA、EQUI、または KEY のいずれかです。	任意
MaxInCriteriaSize	**IN** 条件の使用が有効な場合は、述語ごとの **IN** エントリーの最大数を指定します。**-1 は無** 制限を意味します。	-1
MaxDependentInPredicates	**IN** 条件の使用が有効にされている場合、依存する結合に使用できる述語の最大数を定義します。1 未満の値は、プッシュされる依存値ごとに 1 つの **IN** 述語（pre-7.4 の動作と一致する）のみを使用します。	-1
DirectQueryProcedureName	トランスレーターの **SupportsDirectQueryProcedure** が **true** に設定されている場合、このプロパティーは手順の名前を示します。	native
SupportsDirectQueryProcedure	トランスレーターでコマンドを直接実行できるようにするには、**true** に設定します。	false
ThreadBound	トランスレーターの実行が単一のスレッドによってのみ処理される場合は **true** に設定します。	false
CopyLobs	**true** の場合、大きなオブジェクト(LOB)データ（clob、Blob、sql/xml）が返され、メモリーを安全にエンジンによってコピーされます。ソースがメモリーセーフ LOBS を提供しない場合や、ソース接続から LOBS を切断する場合は、このオプションを使用します。	false

名前	説明	デフォルト
TransactionSupport	最高レベルのトランザクション機能。**autoCommitTxn=DETECT** モードにトランザクションが必要であるかどうかを判断するためのヒントとしてエンジンによって使用されます。XA、NONE、または LOCAL のいずれかを使用できます。XA または LOCAL の場合は、トランザクション下でアクセスがシリアライズされます。	XA



注記

ベース ExecutionFactory の実行プロパティーを使用すると、利用可能なメタデータのサブセットのみを設定できます。すべてのメソッドは BaseDelegatingExecutionFactory で利用できます。

ベースインポーターの設定はありません。

実行プロパティーの上書き

すべての変換を行う場合は、メインの vdb ファイルの Execution Properties を上書きできます。

例： トランスレータープロパティーの上書き

.

```
CREATE FOREIGN DATA WRAPPER "oracle-override" TYPE oracle OPTIONS
(RequiresCriteria 'true');

CREATE SERVER ora FOREIGN DATA WRAPPER "oracle-override" OPTIONS ("resource-
name" 'java:/oracle');

CREATE SCHEMA ora SERVER ora;

SET SCHEMA ora;

IMPORT FROM SERVER ora INTO ora;
```

上記の例では、oracle トランスレーターを上書きし、RequiresCriteria プロパティーを true に設定

します。変更されたトランスレーターは、この VDB のスコープでのみ利用可能です。必要な多くのプロパティーはオーバーライドできます。

VDB 定義も参照してください。

パラメーター化可能なネイティブクエリー

場合によっては、teiid_rel:native-query property とネイティブ手順では、位置的に参照できるパラメーター化可能な文字列を受け入れます。パラメーター参照の形式は '$integer です（例： $1）。1 ベースのインデックスが使用され、IN パラメーターのみが参照される可能性があることに注意してください。このため、ドル記号は予約されていますが、$$ 1 など、別の $' でエスケープできます。値は準備済み値としてバインドされるか、またはリテラルがソース固有の方法にバインドされます。ネイティブクエリーは、呼び出し手順の想定と一致する結果セットを返す必要があります。

たとえば、native-query は g から c を選択します。c1 = $1 and c2 = '$$ 1 ' を選択すると、選択した c の JDBC ソースクエリー（c1 = ? および c2 = '$ 1'）になります。?' は、パラメーター 1 にバインドされている実際の値に置き換えられます。

一般的なインポートプロパティー

複数のインポートプロパティーはすべての翻訳者によって共有されます。

インポータープロパティーを指定する場合は、importer のプレフィックスを指定する必要があります。たとえば、importer.tableTypes です。

名前	説明	デフォルト
autoCorrectColumnNames	Data Virtualization 列名ではピリオド文字が有効ではないため、列名の . の使用を _ に置き換えます。	true
renameDuplicateColumns	true の場合、大文字と小文字の競合または **autoCorrectColumnNames** が原因となる重複列の名前を _ に置き換えます。 接尾辞 **_n**。**n** は整数を追加して、名前を一意にします。	false

名前	説明	デフォルト
renameDuplicateTables	true の場合、大文字と小文字の競合によって生じる重複テーブルの名前を変更します。接尾辞 **_n**。**n** は整数を追加して、名前を一意にします。	false
renameAllDuplicates	true の場合、ケースの競合が混在する重複したテーブル、列、手順、およびパラメーターの名前を変更します。接尾辞 **_n**。**n** は整数を追加して、名前を一意にします。それぞれの名前が重複しているオプションに置き換わります。	false
nameFormat	インポート時にテーブルおよび手順名を修正するには、Java 文字列の形式に設定します。唯一の引数は、元の名前 Data Virtualization 名になります。たとえば、すべての名前に **prod_** のプレフィックスを付けるには、**prod_%s** を使用します。	

9.1. AMAZON S3 TRANSLATOR

*タイプ名 **amazon-s3** によって認識される Amazon Simple Storage Service(S3)トランスレーターは、Amazon S3 オブジェクトリソースを利用するためのストアドプロシージャーを公開します。*

このトランスレーターは通常 TEXTTABLE 関数または XMLTABLE 関数と共に使用され、CSV または XML 形式のデータを消費し、Amazon S3 に保存されている Microsoft Excel ファイルまたはその他のオブジェクトファイルを読み取ります。S3 トランスレーターは、AWS アクセスキー ID およびシークレットアクセスキーを使用して Amazon S3 にアクセスできます。

用途

以下の例では、仮想データベースは、teiidbucket という Amazon S3 バケット から g2.txt という名前の CSV ファイルを読み取ります。

```
e1,e2,e3
5,'five',5.0
6,'six',6.0
7,'seven',7.0
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="example" version="1">
    <model name="s3">
        <source name="web-connector" translator-name="user-s3" connection-jndi-
name="java:/amazon-s3"/>
    </model>
    <model name="Stocks" type="VIRTUAL">
        <metadata type="DDL"><![CDATA[
        CREATE VIEW G2 (e1 integer, e2 string, e3 double,PRIMARY KEY (e1))
            AS  SELECT SP.e1, SP.e2,SP.e3
                FROM (EXEC s3.getTextFile(name=>'g2.txt')) AS f,
                TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER) AS SP;
        ]]> </metadata>
    </model>
    <translator name="user-s3" type="amazon-s3">
      <property name="accesskey" value="xxxx"/>
      <property name="secretkey" value="xxxx"/>
      <property name="region" value="us-east-1"/>
      <property name="bucket" value="teiidbucket"/>
    </translator>
</vdb>
```

実行プロパティー

トランスレーターオーバーライドメカニズムを使用して以下のプロパティーを提供します。

名前	説明	デフォルト
エンコーディング	getTextFiles の手順によって返される CLOB に使用するエンコーディング。値は JRE に認識されるエンコーディングと一致する必要があります。	システムのデフォルトのエンコーディング。
accessKey	Amazon セキュリティーアクセスキー。Amazon コンソールにログインして、セキュリティーアクセスキーを見つけます。これを指定すると、デフォルトのアクセスキーになります。	該当なし
secretKey	Amazon セキュリティーシークレットキー。Amazon コンソールにログインして、セキュリティーシークレットキーを検索します。指定されると、これはデフォルトの秘密鍵になります。	該当なし

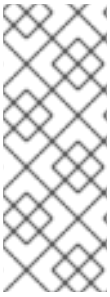
名前	説明	デフォルト
リージョン	要求で使用する Amazon リージョン。これが指定されている場合は、デフォルトのリージョンが使用されます。	該当なし
Bucket	Amazon S3 バケット名。指定された場合、これはすべての要求に使用されるデフォルトバケットとして機能します。	該当なし
暗号化	顧客提供の暗号化キー(SSE-C)を使用したサーバー側の暗号化が使用される場合、キーは使用される暗号化アルゴリズムの「タイプ」を定義するために使用されます。トランスレーターが AES-256 または AWS-KMS 暗号化アルゴリズムを使用するように設定できます。これが指定されている場合、すべての「get」ベースの呼び出しのデフォルトアルゴリズムとして使用されます。	該当なし
encryptionKey	SSE-C タイプの暗号化が使用され、お客様が暗号鍵を提供する場合、このキーは「暗号化キー」を定義するために使用されます。これが指定されている場合、これはすべての「get」ベースの呼び出しのデフォルトキーとして使用されます。	該当なし

ヒント

*プロパティーの設定に関する詳細は、「Translators」の「**Override execution property**」を参照し、以降のセクションの例を確認してください。*

トランスレーターによって公開される手順

*例のようにモデル（スキーマ）を追加すると、以下の手順コールが **Amazon S3** に対して実行できます。*



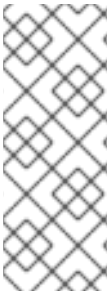
注記

バケット、リージョン、アクセスキー、シークレットキー 、 暗号化および 暗号 化キー は、提供されるメソッドの多くでオプションまたは null 可能なパラメーターです。前述の例に示すように、Translator オーバーライドプロパティーを使用してまだ設定されていない場合にのみ、それらを提供します。

getTextFile(...)

提供されるセキュリティー認証情報を CLOB として使用して、指定されたバケットとリージョンから指定された名前付きオブジェクトをテキストファイルとして取得します。

```
getTextFile(string name NOT NULL, string bucket, string region,
  string endpoint, string accesskey, string secretkey,string encryption, string encryptionkey,
boolean stream default false)
  returns TABLE(file blob, endpoint string, lastModified string, etag string, size long);
```



注記

エンドポイント はオプションです。指定されている場合、提供されたプロパティーで構築されるエンドポイント URL の代わりに使用されます。暗号化および 暗号 化キー は、顧客が提供する鍵(SSE-C)によるサーバー側のセキュリティーを強制する場合にのみ使用します。

stream の値が true の場合、返される LOB は一度だけ読み取られ、通常はディスクにバッファーされません。

例

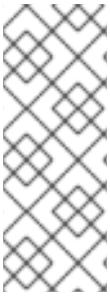
```
exec getTextFile(name=>'myfile.txt');

SELECT SP.e1, SP.e2,SP.e3, f.lastmodified
  FROM (EXEC getTextFile(name=>'myfile.txt')) AS f,
  TEXTTABLE(f.file COLUMNS e1 integer, e2 string, e3 double HEADER) AS SP;
```

getFile(...)

指定されたセキュリティー認証情報を BLOB として使用して、指定されたバケットとリージョンから指定された名前付きオブジェクトを取得します。

```
getFile(string name NOT NULL, string bucket, string region,
  string endpoint, string accesskey, string secretkey, string encryption, string encryptionkey,
boolean stream default false)
  returns TABLE(file blob, endpoint string, lastModified string, etag string, size long)
```



注記

エンドポイント はオプションです。指定されている場合、提供されたプロパティーで構築されるエンドポイント URL の代わりに使用されます。暗号化および 暗号 化キーは、顧客が提供する鍵(SSE-C)によるサーバー側のセキュリティーを強制する場合にのみ使用します。

stream の値が true の場合、IOB は 1 回読み取られ、通常はディスクにバッファーされません。

例

```
exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1', accesskey=>'xxxx',
secretkey=>'xxxx');

select b.* from (exec getFile(name=>'myfile.xml', bucket=>'mybucket', region=>'us-east-1',
accesskey=>'xxxx', secretkey=>'xxxx')) as a,
XMLTABLE('/contents' PASSING XMLPARSE(CONTENT a.result WELLFORMED) COLUMNS e1
integer, e2 string, e3 double) as b;
```

saveFile(…)

CLOB、BLOB、または XML の値を指定された名前およびバケットに保存します。以下の手順では、content パラメーター は LOB タイプのいずれかになります。

```
call saveFile(string name NOT NULL, string bucket, string region, string endpoint,
  string accesskey, string secretkey, contents object)
```

注記

saveFile を使用してファイルの内容のストリームまたはチャンクのアップロードを行うことはできません。非常に大きなオブジェクトのロードを試みると、メモリー不足の問題が発生することがあります。SSE-C 暗号化を使用するように saveFile を設定することはできません。

例

```
exec saveFile(name=>'g4.txt', contents=>'e1,e2,e3\n1,one,1.0\n2,two,2.0');
```

deleteFile(…)

バケットから名前付きオブジェクトを削除します。

```
call deleteFile(string name NOT NULL, string bucket, string region, string endpoint, string accesskey, string secretkey)
```

例

```
exec deleteFile(name=>'myfile.txt');
```

list(…)

バケットの内容を一覧表示します。

```
call list(string bucket, string region, string accesskey, string secretkey, nexttoken string)
  returns Table(result clob)
```

結果として、Amazon S3 が提供する XML ファイルが以下の形式で提供されます。

```
<?xml version="1.0" encoding="UTF-8"?>/n
<ListBucketResult
```

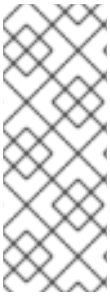
```
    xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>teiidbucket</Name>
    <Prefix></Prefix>
    <KeyCount>1</KeyCount>
    <MaxKeys>1000</MaxKeys>
    <IsTruncated>false</IsTruncated>
    <Contents>
        <Key>g2.txt</Key>
        <LastModified>2017-08-08T16:53:19.000Z</LastModified>
        <ETag>&amp;quot;fa44a7893b1735905bfcce59d9d9ae2e&amp;quot;</ETag>
        <Size>48</Size>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

以下の例のようなクエリーを使用すると、ビューに解析できます。

```
select b.* from (exec list(bucket=>'mybucket', region=>'us-east-1')) as a,
 XMLTABLE(XMLNAMESPACES(DEFAULT 'http://s3.amazonaws.com/doc/2006-03-01/'),
'/ListBucketResult/Contents'
 PASSING XMLPARSE(CONTENT a.result WELLFORMED) COLUMNS Key string, LastModified
string, ETag string, Size string,
 StorageClass string, NextContinuationToken string PATH '../NextContinuationToken') as b;
```

すべてのプロパティー（バケット、リージョン、アクセスキー、およびシークレットキー）がトランスレーターオーバーライドプロパティーとして定義されている場合、以下の単純なクエリーを実行できます。

```
SELECT * FROM Bucket
```



注記

バケットに 1000 以上のオブジェクトがある場合、オブジェクトの次のバッチを取得するために 'NextContinuationToken' の値を リスト 呼び出しに 'nexttoken' として指定する必要があります。これは、拡張リクエストのある Data Virtualization で自動化できます。

9.2. 委譲トランストラクター

コアの Data Virtualization インストールで利用可能な 委譲 トランスレーターを使用して、既存のトランスレーターの機能を変更できます。デバッグの目的で多くの場合、または特別な状況では、トランスレーターの特定の機能をオンまたはオフにしたい場合があります。たとえば、最新バージョンの Hive データベースは ORDER BY コンストラクトをサポートしますが、Hive translator の現在の Data Virtualization バージョンはサポートされません。委譲トランスレーターを使用して、実際にコードを作成せずに ORDER BY 互換性を有効にできます。同様に、逆順を実行し、特定の機能をオフにして、より良いプランを作り出すことができます。

委譲トランスレーターを使用するには、DDL で定義する必要があります。以下の例は、"hive" translator を上書きし、ORDER BY 機能を無効にする方法を示しています。

```
CREATE DATABASE myvdb;
USE DATABASE myvdb;
CREATE FOREIGN DATA WRAPPER "hive-delegator" TYPE delegator OPTIONS
(delegateName 'hive', supportsOrderBy 'false');
CREATE SERVER source FOREIGN DATA WRAPPER "hive-delegator" OPTIONS ("resource-
name" 'java:hive-ds');
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

実行プロパティーを使用して上書きできるトランスレーター機能の詳細は、『Translator Development Guide』の「Translator_Capabilities」を参照してください。上記の例は、Hive translator のデフォルトの ORDER BY 互換性を変更する方法を示しています。

9.2.1. 委譲トランスレーターの拡張

org.teiid.translator.BaseDelegatingExecutionFactory を拡張してトランスレーターを作成できます。クラスがカスタムトランスレーターとしてパッケージ化された後に、別のトランスレーターインスタンスを実行時に委譲して、委譲へのすべての呼び出しをインターセプトできます。このベースクラスは委譲以外の独自の機能を提供しません。トランスレーターは、DDL 設定の一部として定義するのではなく、トランスレーターにハードコードすることができます。代替の動作を提供するためにメソッドを上書きすることもできます。

表9.2 実行プロパティー

名前	説明	デフォルト
delegateName	委譲先となる Translator インスタンス名。	該当なし
cachePattern	トランスレーターキャッシュ API を使用してキャッシュされる必要があるクエリーの正規表現パターン。	該当なし
cacheTtl	キャッシュパターンに一致するクエリーの有効期間（ミリ秒単位）。	該当なし

たとえば、仮想データベースで oracle トランスレーターを使用し、トランスレーターを通過する呼び出しをインターセプトする場合は、以下の例のようにカスタム委譲のトランスレーターを作成できます。

```
@Translator(name="interceptor", description="interceptor")
public class InterceptorExecutionFactory extends
org.teiid.translator.BaseDelegatingExecutionFactory{
   @Override
   public void getMetadata(MetadataFactory metadataFactory, C conn) throws
TranslatorException {
       // do intercepting code here..

       // If you want call the original delegate, do not call if do not need to.
       // but if you did not call the delegate fullfill the method contract
       super.getMetadata(metadataFactory, conn);

       // do more intercepting code here..
   }
}
```

その後、このトランスレーターを Data Virtualization エンジンにデプロイできます。次に DDL ファイルで、以下の例のようにインターセプタートランスレーターを定義します。

```
CREATE DATABASE myvdb VERSION '1';
USE DATABASE myvdb VERSION '1';
CREATE FOREIGN DATA WRAPPER "oracle-interceptor" TYPE interceptor OPTIONS
(delegateName 'oracle');
CREATE SERVER source FOREIGN DATA WRAPPER "oracle-interceptor" OPTIONS
("resource-name" 'java:oracle-ds');
CREATE SCHEMA mymodel SERVER source;
SET SCHEMA mymodel;
IMPORT FROM SERVER source INTO mymodel;
```

上記からのカスタムトランスレーター「インターセプター」に基づく「translator」オーバーライドを定義し、委譲名として「oracle」に委譲する必要があるトランスレーターを提供しました。次に、VDB でこのオーバーライド translator oracle-interceptor を使用しています。この VDB モデルのトランスレーターに今後の呼び出しは、コードによって傍受され、どのような操作を行うことができます。

9.3. ファイルトランスレーター

タイプ名ファイルにより認識されるファイルトランスレーターは、ファイル リソースを利用するためのストアドプロシージャーを公開します。トランスレーターは通常 TEXTTABLE 関数または XMLTABLE 関数と共に使用され、CSV または XML 形式のデータを消費します。

表9.3 実行プロパティー

名前	説明	デフォルト
エンコーディング	getTextFiles の手順によって返される CLOB に使用するエンコーディング。この値は、Data Virtualization に認識されるエンコーディングと一致する必要があります。詳細は、String 関数の **TO_CHARS** および **TO_BYTES** を参照してください。	システムのデフォルトのエンコーディング。
ExceptionIfFileNotFound	指定されたファイル/ディレクトリーが存在しない場合は、getFiles または getTextFiles で例外をスローします。	true

ヒント

プロパティーの設定方法に関する情報は、以下の例を参照してください。また、Translators で実行プロパティーを上書き します。

例： Virtual datbase DDL override

```
CREATE SERVER "file-override"
   FOREIGN DATA WRAPPER file
   OPTIONS(
      Encoding 'ISO-8859-1', "ExceptionIfFileNotFound" false
   );

CREATE SCHEMA file SERVER "file-override";
```

getFiles

```
getFiles(String pathAndPattern) returns
TABLE(file blob, filePath string, lastModified timestamp, created timestamp, size long)
```

指定されたパスとパターンに一致する BLOB としてすべてのファイルを取得します。

```
call getFiles('path/*.ext')
```

パスがディレクトリーである場合は、ディレクトリー内の全ファイルが返されます。パスが 1 つのファイルと一致する場合は、ファイルが返されます。

* 文字は、パス名内の任意の数の文字に一致するワイルドカードとして処理されます。ゼロまたは一致するファイルが返されます。

*' が使用されておらず、パスが存在しない場合に 'ExceptionIfFileNotFound が true の場合、例外が発生します。

getTextFiles

```
getTextFiles(String pathAndPattern) returns
TABLE(file clob, filePath string, lastModified timestamp, created timestamp, size long)
```

注記

サイズはバイト数を報告します。

指定されたパスとパターンに一致する CLOB としてすべてのファイルを取得します。

```
call getTextFiles('path/*.ext')
```

getFiles ファイルを取得しますが、結果が文字セットとしてエンコーディングの実行プロパティーを使用する CLOB 値であることが異なります。

saveFile

CLOB、BLOB、または XML の値を指定されたパスに保存します。

```
call saveFile('path', value)
```

deleteFile

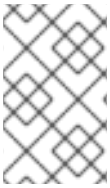
指定されたパスでファイルを削除します。

```
call deleteFile('path')
```

パスは既存のファイルを参照する必要があります。ファイルが存在しないで *ExceptionIfFileNotFound* が *true* の場合、例外が発生します。ファイルを削除できない場合も例外が発生します。

ネイティブクエリー

この機能は、*File translator* には該当しません。



直接クエリーの手順

この機能は、*File translator* には該当しません。

9.4. GOOGLE スプレッドシートトランスレーター

google-spreadsheet *translator* は、*Google* スプレッドシートへの接続に使用されます。

クエリー方法は、ワークシート内のデータに以下の特徴があることを想定します。

- データを含むすべての列はクエリーできます。
- セルが空の場合には、値が null として取得されます。ただし、null 文字列と空の文字列の値を区別すると、Google が相互に置き換え可能なものになるとは限りません。可能な場合は、null 文字列と空の文字列を区別できない場合は、トランスレーターが警告を提供したり、例外をスローする可能性があります。
- 最初の行が存在し、文字列の値が含まれる場合には、行は列ラベルを表すことが想定されます。

デフォルトのネイティブメタデータインポートを使用している場合、トランスレーターの起動時に Google アカウントのメタデータ（ワークシートおよびワークシートの列に関する情報）が読み込まれます。データ型に変更を加えた場合には、仮想データベースを再起動することが推奨されます。

トランスレーターは単一のシートに対してのみクエリーを送信できます。これは、スプレッドシートクエリー言語で利用可能な順序、集計、基本的な述語、およびほとんどの機能を提供します。

google-spreadsheet translator はインポーターの設定を提供しませんが、VDB のメタデータを提供することができます。

警告

Data Virtualization に定義されているヘッダーのあるシートからすべてのデータ行を削除する場合は、Data Virtualization からシートにアクセスできなくなります。Google API はヘッダーをその時点でデータ行として扱い、そのクエリーは有効ではなくなります。

警告

文字列以外のフィールドは、正規の Data Virtualization SQL 値を使用して更新されます。スプレッドシートで非補正ロケールが使用されている場合は、更新を許可しないことを検討してください。詳細は、「TEIID-4854」 および「allTypesUpdatable import プロパティー」を参照してください。

インポーターのプロパティー

- allTypesUpdatable: すべての列を updatable とマークするには、true に設定します。文字列または TEIID-4854 の影響を受けません。デフォルトは true です。

ネイティブクエリー

Google スプレッドシートソース手順は、teiid_rel:native-query エクステンションを使用して作成できます。詳細は「Translators でパラメーター化可能なネイティブクエリー 」を参照してください。この手順では、ネイティブプロシージャーコールと同様の native-query を呼び出します。また、

ARRAYTABLE や同様の機能の使用を必要とするのではなく、クエリーが事前に決定され、結果列タイプが認識されているという利点があります。詳細は、以下の Select *セクションを参照してください。*

直接クエリーの手順

*データソースに対してコマンドを実行できるセキュリティーリスクがあるため、この機能はデフォルトではオフになっています。この機能を有効にするには、*SupportsDirectQueryProcedure *プロパティーを* true *に設定します。詳細は「Translators での* 実行プロパティーの上書き *」を参照してください。*

ヒント

デフォルトでは、クエリーを直接実行する手順の名前は native *と呼ばれます。名前を変更するには、実行プロパティー* DirectQueryProcedureName *を上書きします。詳細は「Translators での* 実行プロパティーの上書き *」を参照してください。*

Google のスプレッドシートトランスレーターは、Data Virtualization の解析または解決を行わずに、アドホッククエリーをソースに対して直接実行する手順を提供します。この手順の実行結果のメタデータは Data Virtualization には認識されないため、オブジェクト配列として返されます。ARRAYTABLE を使用して、クライアントアプリケーションが使用するテーブル出力を作成できます。詳細は、「ARRAYTABLE」を参照してください。

Data Virtualization は、以下の例のように単純なクエリー構造でこの手順を公開します。

***example* の選択**

```
SELECT x.* FROM (call google_source.native('worksheet=People;query=SELECT A, B, C')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

最初の引数は、以下のプロパティーのセミコロンで区切られた(;)名前と値のペアを取り、手順を実行します。

プロパティー	説明	必須
worksheet	Google スプレッドシート名。	はい

プロパティー	説明	必須
query	スプレッドシートのクエリー。	はい
limit	取得する行数。	いいえ
offset	制限または開始から取得する行のオフセット。	いいえ

9.5. JDBC 翻訳者

JDBC Translators は、SQL セマンティクスと、Data Virtualization とターゲット TEMPLATES のデータ型の違いをブリッジします。Data Virtualization には、最も一般的なオープンソースおよびプロプライエタリーリレーショナルデータベースを対象とする特定のトランストラクターの範囲があります。

用途

JDBC ソースの使用は簡単です。Data Virtualization SQL を使用すると、テーブルと手順が Data Virtualization システムのローカルであるかのようにソースをクエリーできます。

リレーショナルデータベースソース、または JDBC ドライバーを持つデータソースを使用している場合で、そのデータソースタイプで利用可能な特定のトランスレーターが見つからない場合は、JDBC ANSI トランスレーター から始めます。JDBC ANSI トランスレーターにより、SQL の基本が実行できるようになります。利用できる特定のデータソース機能がある場合は、必要な操作を行うカスタムトランスレーターを定義できます。詳細は、「Translator Development」を参照してください。

タイプの規則

UUID、GUID、または UNIQUEIDENTIFIER を含む UID タイプは、通常 Data Virtualization 文字列タイプにマッピングされます。JDBC データソースは、UID 文字列を小文字ではないものとして処理しますが、Data Virtualization で大文字と小文字が区別されます。ソースが文字列タイプへの暗黙的な変換をサポートしていない場合、文字列の値を想定する関数の使用がソースで失敗する可能性があります。

以下の表は、すべての JDBC 翻訳者によって共有される実行プロパティーを示しています。

表9.4 すべての JDBC 翻訳者による実行プロパティーファイル

名前	説明	デフォルト

名前	説明	デフォルト
DatabaseTimeZone	データベースのタイムゾーン。date、time、または timestamp の値を取得する場合に使用されます。	システムのデフォルトタイムゾーン
DatabaseVersion	特定のデータベースバージョン。プッシュダウン操作の使用をさらに調整するために使用されます。	ベースと互換性のあるバージョン、または DatabaseMetadata.getDatabaseProductVersion 文字列から派生するバージョン。自動検出には接続が必要です。機能が利用できないために例外が発生する状況（例：接続は利用できません）がある場合は、**DatabaseVersion** プロパティーを設定します。**JDBCExecutionFactory.usesDatabaseVersion（）'** メソッドを使用して、トランスレーターが機能を判断するために接続を必要とするかどうかを制御します。
TrimStrings	**True** は、固定長の文字文字列から末尾の空白をトリミングします。Data Virtualization には文字列または varchar のみがあり、末尾の空白を意味のあるものとして処理することに注意してください。	false
RemovePushdownCharacters	ソースの許可できない文字や望ましくない文字を削除するには、正規表現に設定します。たとえば、**[\u0000] は** null 文字を削除します。これは、PostgreSQL や Oracle などのソースで問題となります。これにより、影響を受ける文字列リテラルの意味とバインド値の意味が事実上変更されます。これらは慎重に考慮する必要があります。	
UseBindVariables	**True** は、PreparedStatements を使用する必要があり、ソースクエリーのリテラル値をバインド変数に置き換える必要があることを示します。**false** の場合、LOB 値のみが PreparedStatements の使用をトリガーします。	true

名前	説明	デフォルト
UseCommentsInSourceQuery	これにより、情報提供の目的で、ソース SQL に session/request id の先頭のコメントが埋め込まれます。CommentFormat プロパティーでカスタマイズできます。	false
CommentFormat	**UseCommentsInSourceQuery** が有効になっている場合に使用される MessageFormat 文字列。形式を以下の値のいずれかに設定できます。 ● 0 - セッション ID 文字列。 ● 1 - 親要求の ID 文字列。 ● 2 - 要求パート ID 文字列。 ● 3 - 実行カウント ID 文字列 ● 4 - ユーザー名の文字列 ● 5 - VDB 名の文字列 ● 6 - VDB バージョン整数 ● 7 - トランザクションのブール値。	**/*Teiid sessionid:\{0}, requestid:\gitops.\{2}*/**
MaxPreparedInsertBatchSize	準備済み挿入バッチの最大サイズ。	2048
StructRetrieval	以下の Struct 取得モードのいずれかを指定します。 ● OBJECT - **getObject** 値が返されます。 ● COPY - **SerialStruct** として返されます。 ● ARRAY - アレイとして返されます。	OBJECT

名前	説明	デフォルト
EnableDependentJoins	一時テーブル（DB2、Derby、H2、HSQL 2.0+、MySQL 5.0+、Oracle、PostgreSQL、SQLServer、SQP IQ、Sybase）を使用するソースに依存のプッシュダウンを許可します。	false

インポータープロパティーですべての JDBC 翻訳者によるインポーターの使用

インポータープロパティーを指定する場合は、importer のプレフィックスを指定する必要があります。例： importer.tableTypes

名前	説明	デフォルト
catalog	See DatabaseMetaData.getTables [1]	null
schemaName	単一のスキーマからインポートするために推奨される設定。スキーマ名は、エスケープされたパターン（設定されている場合は schemaPattern の上書き）に変換されます。	null
schemaPattern	See DatabaseMetaData.getTables [1]	null
tableNamePattern	See DatabaseMetaData.getTables [1]	null
procedureNamePattern	See DatabaseMetaData.getProcedures [1]	null
tableTypes	登録済みのテーブルタイプでスペースを区切られずに、スペースを区切ったリストを使用。See **DatabaseMetaData.getTables** [1]	null

名前	説明	デフォルト
excludeTables	完全修飾テーブル名に対して一致する、大文字と小文字を区別しない正規表現 [2] はインポートから除外します。テーブル名の取得後に適用されます。 負の先読み(?!<inclusion pattern>)を使用して、包含フィルターとして機能します。	null
excludeProcedures	完全修飾手順名に対して一致する、大文字と小文字を区別しない正規表現 [2] はインポートから除外します。 手順名の取得後に適用されます。 負の先読み(?!<inclusion pattern>)を使用して、包含フィルターとして機能します。	null
importKeys	**プライマリー** キーおよび外部キーをインポートする場合はTrue。 注記： インポートされていないテーブルにキーを配置すると無視されます。	true
autoCreateUniqueConstraints	**true**: 外部キーについて 1 つの一意の制約が見つからない場合に一意の制約を作成します。	true
importIndexes	インデックス/一意キー/カーディナリティー情報をインポートする場合は **true**	false
importApproximateIndexes	**true**: 概算インデックス情報をインポートします。DatabaseMetaData.getIndexInfo [1] を参照してください。 警告： **false** に設定すると、インポート時間が長くなることがあります。	true

名前	説明	デフォルト
importProcedures	True **:** 手順と手順列をインポートするには、データベースの制限により、手順の結果の設定列を常にインポートできるわけではないことに注意してください。現在、オーバーロードされた手順をインポートすることはできません。	false
importSequences	シーケンス **を** インポートする場合は True。Db2、Oracle、PostgreSQL、SQL Server、および H2 にのみ互換性があります。一致するシーケンスが 0 引数の Data Virtualization 関数 **name_nextval** にインポートされます。	false
sequenceNamePattern	シーケンスをインポートするときに使用する LIKE パターン文字列。Null または **%** が すべて に一致します。	null
useFullSchemaName	**false** の場合、インポーターに対してオブジェクト名のみを Data Virtualization 名として使用するように指示します。すべてのオブジェクトが同じ外部スキーマから取得されることが想定されています。**true** （推奨）される場合、Data Virtualization 名は **useCatalogName** および **useQualifiedName** プロパティーの指示どおりにカタログおよびスキーマ名を使用して形成されます。複数の外部スキーマからオブジェクトを取得できるようになります。このオプションはソースプロパティーの名前には影響しません。	false（複数の外部スキーマからのインポートする場合のみ変更）。

名前	説明	デフォルト
useQualifiedName	**True** は、**useCatalogName** および **useFullSchemaName** プロパティーによりさらに改良されたように、Data Virtualization 名および名前の両方に名前修飾を使用します。 Data Virtualization 名とソースの名前の両方に対するすべての修飾を無効にするには、**false** に設定します。これにより、**useCatalogName** および **useFullSchemaName** プロパティーを効果的に無視します。 警告： このオプションを **false** に設定すると、複数のスキーマからインポートすると、重複する名前がオブジェクトに含まれる可能性があり、例外が発生します。	True（必須の変更）
useCatalogName	**true** は、null または空でないカタログ名をソースの名前の一部として使用します（例： "catalog"."schema"."table"."column"）、および Data Virtualization ランタイム名（該当する場合）。**false** は、ソースの名前または Data Virtualization ランタイム名のいずれかでカタログ名を使用しません。カタログの概念を使用しない HSQL などのソースには **false** に設定する必要があるが、メタデータに null 以外または空でないカタログ名を返す必要があります。	True（必須の変更）
widenUnsignedTypes	**true**: 署名なしタイプを次の幅広い型に変換します。たとえば、SQL Server は **tinyint** を非署名タイプとして報告します。このオプションを有効にすると、**tinyint** がバイトではなく短い状態でインポートされます。	true
useIntegralTypes	**True**: スケールが 0 の場合に 10 進数ではなく必須型を使用します。	false
quoteNameInSource	**false** はデフォルトを上書きし、データ仮想化に引用なしの識別子を使用してソースクエリーを作成するように指示します。	true

名前	説明	デフォルト
useAnyIndexCardinality	True **は、DatabaseMetaData.getIndexInfo** から返される最大カーディナリティーを使用します。この設定を有効にするには、**importKeys** または **importIndexes** を有効にする必要があります。これにより、統計インデックスを返さないソースから統計収集を改善できます。	false
importStatistics	True **は**、データベース依存ロジックを使用して、何も判断されない場合のカーディナリティーを決定します。Oracle および MySQL でしか利用できないすべてのデータベースタイプでは使用できません。	false
importRowIdAsBinary	**true** は **RowId** 列を varbinary の値としてインポートします。	false
importLargeAsLob	**True** は、Data Virtualization max よりも大きな文字およびバイナリータイプをそれぞれ CLOB または BLOB としてインポートします。プロパティーが有効になっている場合でもメモリーの問題が発生した場合は、**copyLob** 実行プロパティーも使用する必要があります。	false

[1] DatabaseMetaData
[2] 除外の完全修飾名は、トランスレーターの設定とデータベースの特定に基づいています。トランスレーターの設定で使用される適用可能な名前の部分すべて（ useQualifiedName および useCatalogNameを参照）には、カタログ、スキーマ、テーブルを含む catalogName.schemaName.tableName が引用なしの catalogName.schemaName.tableName として組み合わされます。たとえば、Oracle はカタログを報告しないため、比較用のデフォルト設定で使用される名前は schemaName.tableName のみです。

警告

デフォルトのインポート設定により、利用可能なすべてのメタデータが取り消されます。このインポートプロセスには時間がかかるため、ほとんどの状況では完全なメタデータのインポートは必要ありません。最も一般的なものは、少なくとも schemaName または schemaPattern、および tableTypes でインポートを制限します。

例： my-schema からテーブルとビューのみをインポートする Importer 設定。

```
SET SCHEMA ora;

IMPORT FOREIGN SCHEMA "my-schema" FROM SERVER ora INTO ora OPTIONS
("importer.tableTypes" 'TABLE,VIEW');
```

インポーターの設定に関する詳細は、「仮想データベース」を参照して ください。

ネイティブクエリー

物理テーブル、機能、および手順には、任意でネイティブクエリーが関連付けられています。 ネイティブクエリーの検証は実行されません。ソース SQL を生成するためのシンプルな方法で使用されます。 teiid_rel:native-query 拡張メタデータを設定すると、ソースクエリーのインラインビューとしてネイティブクエリーが実行されます。 この機能は、インラインビューを提供するソースでのみ使用する必要があります。 ネイティブクエリーはそのまま使用され、パラメーター化された文字列として処理されません。たとえば、nameInSource="x'" と teiid_rel:native-query="select c from g" の物理テーブル y では、Data Virtualization ソースクエリー "SELECT c FROM y" は SQL クエリー "SELECT c FROM（select c FROM(select c from g)を x" と して生成します。 ネイティブクエリーの列名は、SQL が有効になるように、物理テーブル列の nameInSource と一致する必要があります。

物理手順の場合は、teiid_rel:native-query 拡張メタデータを、位置的参照の IN パラメーターを追加する機能が追加されたクエリー文字列に設定することもできます。詳細は「Translators でパラメーター化 可能なネイティブクエリー 」を参照してください。 teiid_rel:non-prepared 拡張メタデータプロパティーは false に設定して、パラメーターバインディングをオフにすることができます。

適切に検証されていない場合に受信は SQL インジェクション攻撃を許容するため、このオプションの設定は注意して行ってください。 ネイティブクエリーはストアドプロシージャーを呼び出す必要は

ありません。 物理ストアドプロシージャーメタデータによって想定される結果セットに位置的に一致する結果セットを返す SQL はすべて機能します。 たとえば、teiid_rel:native-query="select c from g from g where c1 = $1 and c2 = '$$1'" のあるストアドプロシージャ x の場合、Data Virtualization source query '"CALL x(?)"' は SQL クエリー '"select c from g where c1 = ? and c2 = '$1'"' を生成します。 この例の ? は、パラメーター 1 にバインドされている実際の値に置き換えられます。

直接クエリーの手順

この機能は、ソースに対して任意のコマンドを実行できるようにする固有のセキュリティーリスクがあるため、デフォルトでオフになっています。この機能を有効にするには、SupportsDirectQueryProcedure と呼ばれる実行プロパティーを上書きし、true に設定します。詳細は「Translators での 実行プロパティーの上書き 」を参照してください。

デフォルトでは、クエリーを直接実行する手順の名前は native です。名前を変更するには、実行プロパティー DirectQueryProcedureName を上書きします。

JDBC トランスレーターは、Data Virtualization の解析または解決を行わずに、アドホック SQL クエリーをソースに対して直接実行する手順を提供します。この手順の結果のメタデータは Data Virtualization には認識されないため、オブジェクト配列として返されます。ARRAYTABLE は、クライアントアプリケーションが使用するための表形式出力です。詳細は、arraytable を参照してください。

SELECT の例

```
SELECT x.* FROM (call jdbc_source.native('select * from g1')) w,
 ARRAYTABLE(w.tuple COLUMNS "e1" integer , "e2" string) AS x
```

INSERT の例

```
SELECT x.* FROM (call jdbc_source.native('insert into g1 (e1,e2) values (?, ?)', 112, 'foo')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

UPDATE の例

```
SELECT x.* FROM (call jdbc_source.native('update g1 set e2=? where e1 = ?','blah', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

DELETE の例

```
SELECT x.* FROM (call jdbc_source.native('delete from g1 where e1 = ?', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

9.5.1. Actian Vector translator(actian-vector)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプが actian-vector によって認識されるアクチェンベクタートランスレーターは Actor の Actian Vector 用です。

JDBC ドライバーを http://esd.actian.com/platform からダウンロードします。接続 URL のポート番号は 16967 にマップする AH7 です。

9.5.2. Apache Phoenix Translator(phoenix)

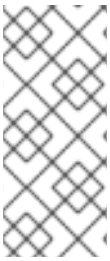
また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 phoenix によって認識される Apache Phoenix translator は、HBase テーブルへのクエリー機能を公開します。Apache Phoenix は、このコマンドを Phoenix SQL にプッシュする際に、このトランスレーターに必要な HBase の JDBC SQL インターフェースです。

トランスレーターは、非推奨の hbase の名前でも知られています。この名前の変更により、トランスレーターは Phoenix 固有のものであり、将来 HBase に接続するため、他の翻訳者が導入される可能性があります。

このトランスレーターでは DatabaseTimezone プロパティーを使用しないでください。

HBase translator は参加コマンドを処理できません。Phoenix は HBase Table Row ID を プライマリー Key として使用します。この Translator は、HBase 0.98.1 以降の Phoenix 4.3 以降で開発されます。

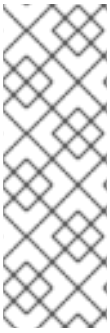


注記

トランスレーターは、Phoenix UPSERT 操作で INSERT/UPDATE を実装します。これは、標準の INSERT/UPDATE とは異なる動作を確認できることを意味します。たとえば、繰り返し挿入してもキーの重複例外は発生せず、代わりに問題の行を更新します。

注記

Phoenix ドライバーの制限により、インポーターは一意の制約を検索しず、デフォルトでは外部キーをインポートしません。



注記

トランスレーターは、SQL OFFSET 引数および Phoenix 4.8 以降の他の機能を処理できます。Phoenix ドライバーは、PhoenixDatabaseMetaData でサーバーバージョンをハードコーディングし、実行時にサーバーバージョンを検出する方法を提供しません。古いサーバーで新しいドライバーを使用する場合は、データベースバージョン translator プロパティーを手動で設定します。

警告

Phoenix ドライバーには、時間の値を堅牢に処理していません。時間の値が 1970-01-01 の日付コンポーネントを使用するように正規化されると、デフォルトの処理が正常に機能します。そうでない場合は、時間列をタイムスタンプとしてモデル化する必要があります。

9.5.3. Cloudera Impala translator(impala)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

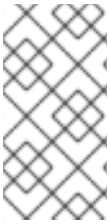
タイプimpala によって認識される Cloudera Impala translator は、Cloudera Impala 1.2.1 以降で使用するためのものです。

Impala のデータ型のサポートは限定的です。time/date/xml または LOB のネイティブサポートはありません。これらの制限は、トランスレーター機能に反映されます。Data Virtualization ビューはこれらのタイプを使用できますが、変換には必要な変換を指定する必要があります。このような状況では、Data Virtualization エンジンで評価が実行されることに注意してください。

Impala トランスレーターでは DatabaseTimeZone translator プロパティーを使用しないでください。

Impala は EQUI join のみをサポートするため、ソーステーブルで他の結合タイプを使用するとクエリーが非効率になります。

パーティション化された列に基づいて基準を記述するには、ソーステーブルで条件をモデル化しますが、選択列に含まれません。



注記

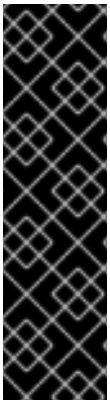
Impala Hive インポーターにはカタログまたはソーススキーマの概念がなく、キー、手順、インデックスなどをインポートします。

アmpala 固有のインポータープロパティー

useDatabaseMetaData

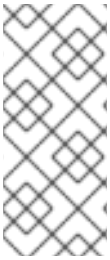
オプションとともに通常のインポートロジックを使用してインデックス情報を無効にするには、true に設定します。デフォルトは false です。

useDatabaseMetaData の値が false の場合、通常の JDBC DatabaseMetaData コールは使用されないので、一般的な JDBC インポータープロパティーすべてが Impala に適用されるわけではありません。excludeTables を使用していても、引き続き excludeTables を使用できます。



重要

Impala のバージョンによっては、ORDER BY の実行時に LIMIT を使用する必要があります。Impala でデフォルトが設定されていない場合、ORDER BY のある Data Virtualization クエリーが発行されていないと例外が発生する可能 性 があります。Impala-wide default を設定するか、新しい接続 SQL 文字列を使用するように SET DEFAULT_ORDER_BY_LIMIT ステートメントを発行するように接続プールを設定する必要があります。制限を超えたタイミングを制御する方法など、Impala 制限オプションの詳細は、Cloudera のドキュメント を参照してください。



注記

Impala JDBC ドライバーで PreparedStatements または parsing ステートメントの処理が一般的な場合は、useBindVariables を無効にしてください。詳細は、https://issues.redhat.com/browse/TEIID-4610 を参照してください。

9.5.4. Db2 Translator(db2)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ name db2 *によって認識される Db2 トランスレーターは、IBM Db2 V8 以降、または i5.4 以降の場合は IBM Db2 で使用します。*

Db2 実行プロパティー

DB2ForI

Db2 インスタンスが i の Db2 であることを示します。デフォルトは false です。

supportsCommonTableExpressions

Db2 インスタンスが共通のテーブル式(CTE)をサポートすることを示します。デフォルトは true です。共通テーブル式は、一部の古いバージョンの Db2 や、変換モードで実行する Db2 のインスタンスでは完全にサポートされていません。これらの環境で CTEs で作業中にエラーが発生した場合は、CTE プロパティーを false に設定します。

9.5.5. Derby translator(derby)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ derby *によって認識される Derby トランスレーターは、Derby 10.1 以降で使用するためのものです。*

9.5.6. Exasol translator(exasol)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ exasol によって認識される Exa sol トランス レーターは、Exasol バージョン 6 以降で使用することです。

用途

Exasol データベースは NULL HIGH のデフォルト順序を持ちますが、Data Virtualization エンジンは NULL LOW モードで動作します。その結果、順序が Exasol にプッシュされたか、エンジンによって実行されるかによって、返される結果の開始または終了のいずれかで NULL を確認できます。一貫性を保つには、org.teiid.pushdownDefaultNullOrder=true で Data Virtualization を実行して NULL LOW の順序を指定できます。NULL LOW 順序付けを適用すると、パフォーマンスが低下する可能性があります。

9.5.7. greenplum Translator(greenplum)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

greenplum 型で認識される Greenplum トランスレーターは、Greenplum データベースで使用することです。このトランスレーターは PostgreSQL トランスレーター の拡張であり、そのオプションを継承します。

9.5.8. H2 Translator(h2)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 h2 によって認識される H2 Translator は、H2 バージョン 1.1 以降で使用するために使用されます。

9.5.9. Hive Translator(hive)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 hive で認識される Hive Translator は、Hive v.10 および SparkSQL v1.0 以降で使用しま

す。

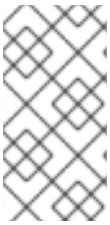
機能

Hive は限定されたデータタイプのセットと互換性があります。time/XML または大規模なオブジェクト(LOB)のネイティブサポートはありません。これらの制限は、トランスレーター機能に反映されます。Data Virtualization ビューはこれらのタイプを使用できますが、変換に必要な変換を指定する必要があります。このような状況では、評価は Data Virtualization エンジンで処理されることに注意してください。

DatabaseTimeZone translator プロパティーは Hive トランスレーターで使用しないでください。

Hive は EQUI join のみをサポートするため、ソーステーブルで他の結合タイプを使用するとクエリーが非効率になります。

パーティション化された列に基づいて基準を記述するには、ソーステーブルで条件をモデル化しますが、選択列に含まれません。



注記

Hive インポーターにはカタログまたはソーススキーマの概念がなく、キー、手順、インデックスなどをインポートします。

プロパティーのインポート

trimColumnNames

Hive 0.11.0 以降では、DESCRIBE コマンドメタデータは、パディングと共に適切に返されます。列名から空白を削除するには、このプロパティーを true に設定します。デフォルトは false です。

useDatabaseMetaData

Hive 0.13.0 以降では、インポートを実行するために通常の JDBC DatabaseMetaData 機能があれば十分です。オプションとともに通常のインポートロジックを使用してインデックス情報を無効にするには、true に設定します。デフォルトは false です。true の場合、trimColumnNames は影響を受けません。false に設定すると、一般的な JDBC DatabaseMetaData 呼び出しが使用されないため、共通の JDBC インポータープロパティーすべてが Hive に適用されるわけではありません。excludeTables を引き続き使用できます。

"Database Name"

Hive で使用されるデータベース名が デフォルト のものと異なる場合、クエリーのメタデータの取得および実行は Data Virtualization で予想通りに機能しません。Hive JDBC ドライバーは暗黙的に接続(< 0.12)で「デフォルト」データベースに接続するように見えるため、接続 URL に記載されているデータベース名は無視されます。コマンドを送信するように接続ソースを設定する場合は、{database-name} を使用 することができます。

これは、バージョン 0.13 以降の Hive JDBC ドライバーで修正されています。詳細は、https://issues.apache.org/jira/browse/HIVE-4256 を参照してください。

制限事項

空のテーブルは、データタイプ情報なしに説明を報告する可能性があります。インポート時にこの問題を回避するには、空のテーブルを除外するか、useDatabaseMetaData オプションを使用します。

9.5.10. HSQL Translator(hsql)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 hsql によって認識される HSQL Translator は、HSQLDB 1.7 以降で使用するために使用されます。

9.5.11. informix Translator(informix)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ name informix によって認識される Informix トランスレーターは、任意の Informix バージョンと使用します。

既知の問題

TEIID-3808

データベースタイムゾーントランスレータープロパティーが設定されている場合でも、Informix ドライバーのタイムゾーン情報の処理に 一貫性がありません。Informix サーバーとアプリケーションサーバーが同じタイムゾーンにあることを確認します。

9.5.12. Ingres Translators(ingres / ingres93)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

Ingres バージョンに応じて、以下の Ingres 翻訳機能のいずれかを使用できます。

Ingres

タイプ名 ingres で認識される Ingres トランスレーターは、Ingresgitops 以降と使用します。

ingres93

タイプ名 ingres93 で認識される Ingres93 トランスレーターは、Ingres 9.3 以降で使用するために使用されます。

9.5.13. InterSystems Cach Warehouse translator(intersystems-cache)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 intersystems-cache によって認識される Intersystem Cach exercise トランスレーターは、Intersystems Cach336 Object データベース（理論的側面のみ）で使用します。

9.5.14. JDBC ANSI トランスレーター(jdbc-ansi)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

jdbc-ansi タイプ名 によって認識される JDBC ANSI トランスレーターは、行 LIMIT/OFFSET および EXCEPT/INTERSECT を除き、Data Virtualization で使用されるほとんどの SQL コンストラクトで機能します。ソース SQL を ANSI 準拠の構文に変換します。別の特定型が利用できない場合は、このトランスレーターを使用する必要があります。互換性のない SQL コンストラクトを使用してソース例外が発生した場合は、JDBC 単純なトランスレーターを使用して機能をさらに制限するか、カスタムのトランスレーター を作成することを検討してください。詳細は、Teiid コミュニティーのカスタムトランスレーターのドキュメントを参照し てください。

9.5.15. JDBC simple translator(jdbc-simple)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

タイプ名 jdbc-simple によって認識される JDBC Simple translator は、jdbc-ansi-translator と同じですが、最大互換性を提供するのを除き、ほとんどのプッシュダウン構成は処理されません。

9.5.16. Microsoft Access 翻訳者

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

access

タイプ名 アクセス によって認識される Microsoft Access トランスレーターは、JDBC-ODBC ブリッジを介して Microsoft Access336 以降で使用することです。

デフォルトのネイティブメタデータインポートまたは Data Virtualization 接続インポーターを使用している場合、インポーターはデフォルトで importKeys=false および excludeTables=.[.]MSysです。 JDBC ODBC ブリッジで提供されるメタデータの問題を回避するために、インポーターはデフォルトの importKeys=false および excludeTables= です。別の JDBC ドライバーを使用する場合は、これらの値の調整が必要になる場合があります。

ucanaccess

タイプ名 ucanaccess によって認識される Microsoft Access トランスレーターは、UCanAccess ドライバー を介して Microsoft Accessで使用するためのものです。

9.5.17. Microsoft SQL Server トランスレーター(sqlserver)

また、「JDBC 翻訳者」の情報 も参照してください。

sqlserver タイプで認識される Microsoft SQL Server のトランスレーターは、SQL Server 2000 以降で使用するために使用されます。SQL Server JDBC ドライバーのバージョン 2.0 以降（または JTDS 1.2 以降などの互換性のあるドライバー）を使用する必要があります。SQL Server DatabaseVersion プロパティーは 2000、2005、2008、または 2012 に設定できますが、それ以外の場合は 10.0 などの標準のバージョン番号を想定し ます。

シーケンス

Data Virtualization 8.5+ では、シーケンス操作を ソース関数 としてモデル化できます。

Data Virtualization 10.0+ では、シーケンスは プロパティーを自動的にインポート できます。

例： ネイティブクエリーのシーケンス

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-query" 'NEXT VALUE FOR seq');
```

実行プロパティー

SQL Server 固有の実行プロパティー：

JtdsDriver

オープンソース JTDS ドライバーの使用を指定します。デフォルトは false です。

9.5.18. MySQL トランスレーター(mysql/mysql5)

また、「JDBC 翻訳者」の情報 も参照してください。

MySQL および MariaDB では、以下の翻訳機能を使用できます。

mysql

タイプ名 mysql によって認識される MySQL トランスレーターは、MySQL バージョン 4.x で使用します。

mysql5

タイプ名 mysql5 によって認識される MySQL5 トランスレーターは、MySQL バージョン 5 以降で使用するために使用されます。トランスレーターは MariaDB などの他の互換性のある MySQL 派生機能でも機能します。

用途

MySQL 翻訳者は、データベースまたはセッションが ANSI モードを使用していることを想定しています。データベースが ANSI モードを使用していない場合は、以下の初期化クエリーを送信することで、プールに ANSI モードを設定できます。

```
set SESSION sql_mode = 'ANSI'
```

データに null タイムスタンプの値が含まれる場合、Data Virtualization は次の変換エラーを生成します。0000-00-00 00:00:00 はタイムスタンプに変換できません。エラーを回避するには、null タイムスタンプ値を持つデータを想定する場合は、接続プロパティー zeroDateTimeBehavior=convertToNull を設定します。

警告

大規模な結果セットを取得する必要がある場合は、接続プロパティー useCursorFetch=true を設定することを検討してください。それ以外の場合は、MySQL は Data Virtualization インスタンスのメモリーに結果セットを完全に取得します。

注記

MySQL は TINYINT(1)列を JDBC BIT タイプとして報告しますが、値の範囲が実際に制限されず、たとえば -1 に true 値として認識される場合などに問題が発生する可能性があります。ネイティブインポーターを使用していない場合は、トランスレーターがブール値変換を適切に処理できるように、影響を受けるソースの BOOLEAN 列を変更して、BIT ではなく「TINYINT(1)」のネイティブ型を持つようにします。

9.5.19. Netezza translator(netezza)

また、「JDBC 翻訳者」の情報 も参照してください。

タイプ netezza で認識される Netezza トランサーは、どのバージョンの IBM Netezza アプライアンスとも使用します。

用途

Netezza の現在のベンダーが提供する JDBC ドライバーは、単一のトランザクション更新では適切に実行されません。可能な限りバッチ化された更新を実行することが推奨されます。

実行プロパティー

Netezza 固有の実行プロパティー：

SqlExtensionsInstalled

Netezza REGEXP_LIKE 関数を処理する機能を含む SQL 拡張がインストールされていることを示します。他のすべての REGEXP 機能はプッシュダウン機能として利用できます。デフォルトは false です。

9.5.20. Oracle translator(oracle)

また、「JDBC 翻訳者」の情報 も参照してください。

タイプ名 oracle で認識される Oracle トランスレーターは、Oracle Database 9i 以降と使用します。

注記

Oracle が提供する JDBC ドライバーは大量のメモリーを使用します。ドライバーはバッファーに大量のデータをキャッシュするため、十分なメモリー割り当てがないコンピューターで問題が発生する可能性があります。

詳しい情報は、以下の資料を参照してください。

- Teiid 問題
- Oracle の取り組み。

インポーターのプロパティー

useGeometryType

SDO_GEOMETRY のソースタイプで列をインポートする場合は、Data Virtualization Geometry タイプを使用します。デフォルトは false です。

注記

Oracle からのメタデータのインポートは時間がかかる可能性があります。スキーマ名フィルターは、最低でも指定することが推奨されます。メタデータクエリーのフェッチサイズを増やすことができる useFetchSizeWithLongColumn=true 接続プロパティー もあります。特にスキーマに多くのテーブルがある場合に、メタデータの負荷プロセスが大幅に改善します。

実行プロパティー

OracleSuppliedDriver

Oracle が提供するドライバー（通常は ojdbc のプレフィックス）が使用されていることを示します。デフォルトは true です。DataDirect またはその他の Oracle JDBC ドライバーを使用する場

合は false に設定します。

Oracle 固有のメタデータ

シーケンス

Oracle トランスレーターでシーケンスを使用できます。シーケンスを DUAL のソースで名前でテーブルとしてモデル化し、ソースの列名を < sequence name>.[nextval|currval]に設定します。

Data Virtualization 10.0+ を使用すると、シーケンスを自動的にインポートできます。

詳細は、「JDBC 翻訳者における Importer プロパティー 」を参照してください。Data Virtualization 8.4 およびそれ以前 Oracle Sequence DDL

```
CREATE FOREIGN TABLE seq (nextval integer OPTIONS (NAMEINSOURCE 'seq.nextval'),
currval integer options (NAMEINSOURCE 'seq.currval') ) OPTIONS (NAMEINSOURCE 'DUAL')
```

Data Virtualization 8.5 では、テーブルの表現と、シーケンスに Oracle 固有の処理に依存する必要がなくなりました。

ソース関数としての通貨と 次 の バルブを示す方法は、「スキーマオブジェクトの DDL メタデータ」を参照してください。

8.5 の例： ネイティブクエリーのシーケンス

```
CREATE FOREIGN FUNCTION seq_nextval () returns integer OPTIONS ("teiid_rel:native-
query" 'seq.nextval');
```

列を autoincrement に設定し、source の名前を < element name>:SEQUENCE=<sequence name>.<sequence value> に設定すると、列の挿入のデフォルト値としてシーケンスを使用することもできます。

Rownum

rownum 列を Oracle 物理テーブルに追加して、rownum 擬似列の使用を有効にすることもできま

す。rownum 列には、 rownum のソースの名前が必要です。これらの rownum 列には Oracle rownum コンストラクトと同じセマンティクスがないため、使用時に注意する必要があります。

パラメーターの結果セット

この手順のパラメーターは、結果セットを返すためにも使用できます。自動インポートで正しく表現されていない場合には、結果セットを手動で作成し、ネイティブタイプ REF CURSOR で output パラメーターを表す必要があります。

out パラメーターの結果セットの DDL

```
create foreign procedure proc (in x integer, out y object options (native_type 'REF CURSOR'))
returns table (a integer, b string)
```

地理的な機能

Oracle のトランスレーターでは、以下の地理的な機能を使用できます。

Relate = sdo_relate

```
CREATE FOREIGN FUNCTION sdo_relate (arg1 string, arg2 string, arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object, arg2 Object, arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 string, arg2 Object, arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_relate (arg1 Object, arg2 string, arg3 string) RETURNS
string;
```

Nearest_Neighbor = sdo_nn

```
CREATE FOREIGN FUNCTION sdo_nn (arg1 string, arg2 Object, arg3 string, arg4 integer)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object, arg2 Object, arg3 string, arg4 integer)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_nn (arg1 Object, arg2 string, arg3 string, arg4 integer)
RETURNS string;
```

Within_Distance = sdo_within_distance

```
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object, arg2 Object, arg3 string)
RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 string, arg2 Object, arg3 string)
```

```
RETURNS string;
CREATE FOREIGN FUNCTION sdo_within_distance (arg1 Object, arg2 string, arg3 string)
RETURNS string;
```

Nearest_Neigher_Distance = sdo_nn_distance

```
CREATE FOREIGN FUNCTION sdo_nn_distance (arg integer) RETURNS integer;
```

filter = sdo_filter

```
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object, arg2 string, arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 Object, arg2 Object, arg3 string) RETURNS
string;
CREATE FOREIGN FUNCTION sdo_filter (arg1 string, arg2 object, arg3 string) RETURNS
string;
```

pushdown 関数

Oracle のバージョンに応じて、Oracle トランスレーターによって、以下の地理的なプッシュダウン機能をエンジンに登録します。

TRUNC

数値バージョンとタイムスタンプバージョンの両方。

LISTAGG

requires the Data Virtualization SQL syntax "LISTAGG(arg [, delim] ORDER BY ...)"（Data Virtualization SQL 構文 "LISTAGG(arg [, delim] ORDER BY ...)" が必要）

SQLXML

Oracle から SQLXML の値を取得し、代わりに oracle.xdb.XMLType または OPAQUE インスタンスを取得する必要がある場合は、以下の変更を加えます。

- クライアントドライバーバージョン 11 以降を使用します。
- xdb.jar および xmlparserv2.jar ファイルをクラスパスに配置します。
- システムプロパティー oracle.jdbc.getObjectReturnsXMLType="false" を設定します。

詳細は、Oracle のドキュメント を参照し てください。

9.5.21. PostgreSQL トランスレーター(postgresql)

また、「JDBC 翻訳者」の情報 も参照してください。

postgresql タイプ名で認識される PostgreSQL トランスレーターは、以下の PostgreSQL クライアントおよびサーバーのバージョンで使用されます。

実行プロパティー

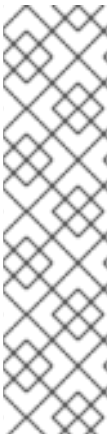
PostgreSQL 固有の実行プロパティー：

PostGisVersion

使用中の PostGIS バージョンを示します。デフォルトは 0 で、PostGIS がインストールされていないことを意味します。データベースのバージョンが設定されていない場合は、自動的に設定されます。

ProjSupported

PostGis バージョンが PROJ コーディネート変換ソフトウェアをサポートするかどうかを示すブール値。データベースのバージョンが設定されていない場合は、自動的に設定されます。



注記

PostgreSQL のドライバーバージョンの中には、列を「INDEX」タイプのテーブルには関連付けません。現在のバージョンの Data Virtualization は、このようなテーブルを自動的に省略します。

古いバージョンの Data Virtualization では、importer.tableType プロパティーまたはその他のフィルタリングセットが必要になる場合があります。

9.5.22. PrestoDB トランスレーター(prestodb)

また、「JDBC 翻訳者」の情報 も参照してください。

タイプ名 prestodb によって認識される PrestoDB トランスレーターは、Presto データソースへのクエリー機能を公開します。データ統合に関しては、PrestoDB には Data Virtualization と同様の機能

がありますが、複数のワーカーノードを使用した分散クエリーの実行についてはそれ以上になります。Data Virtualization の実行モデルは単一の実行ノードに制限され、クエリーをソースにプッシュすることに重点を置いています。Data Virtualization は、より完全なクエリー機能と多くのエンタープライズ機能を提供します。

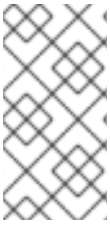
機能

PrestoDB トランスレーターは SELECT ステートメントでのみ使用できます。トランスレーターは、制限された機能セットを提供します。

PrestoDB はリレーショナルデータベースモデルを公開するため、Data Virtualization は Oracle、Db2 などの他のソースと同様に使用できます。PrestoDB の設定に関する情報は、Presto ドキュメントを参照してください。

ヒント

SQL JOIN 操作では、PrestoDB は複数の ORDER BY 列もサポートします。複数の ORDER BY 列を含む JOIN 操作中にエラーが発生した場合は、translator プロパティー supportsOrderBy を設定して、ORDER BY 句の使用を無効にします。



注記

サブキューに null 値を含めると、Presto のバージョンによってはエラーを生成します。

ヒント

PrestoDB はトランザクションをサポートしません。この制限によって生じる問題を解決するには、データソースを非トランザクションとして定義します。

注記

デフォルトでは、PrestoDB のすべてのカタログには information_schema があります。複数のカタログを設定する必要がある場合、テーブルのエラーが重複すると、仮想データベースのデプロイメントが失敗する場合があります。テーブルの重複エラーを防ぐには、インポートオプションを使用してスキーマをフィルタリングします。

複数の Presto カタログを設定する場合は、以下のインポートオプションのいずれかを使用して、ソースのスキーマとテーブルをフィルターします。

- Presto のソース カタログ の名前に一致するように、catalog を特定のカタログ名に設定します。
- schemaName を正規表現に設定し、結果の一致でスキーマをフィルターします。
- excludeTables を正規表現に設定し、結果に一致するテーブルをフィルターします。

9.5.23. Redshift translator(redshift)

また、「JDBC 翻訳者」の情報 も参照してください。

タイプ redshift によって認識される Redshift トランスレーターは、Amazon Redshift データベースと使用します。このトランスレーターは PostgreSQL トランスレーター の拡張であり、そのオプションを継承します。

9.5.24. SAP HANA トランスレーター(hana)

また、「JDBC 翻訳者」の情報 も参照してください。

hana の名前で知られている SAP HANA トランスレーターは、SAP HANA で使用します。

既知の問題

TEIID-3805

SUBSTRING 関数のプッシュダウンは、FROM インデックスが文字列の長さを超える場合に Data Virtualization SUBSTRING 関数と一貫性がありません。SAP HANA は空の文字列を返しますが、Data Virtualization は null 値を生成します。

9.5.25. SAP IQ トランスレーター(sap-iq)

また、「JDBC 翻訳者」の情報 も参照してください。

sap-iq タイプで知られている SAP IQ トランスレーターは、SAP IQ バージョン 15.1 以降と使用します。トランスレーター名 sybaseiq が非推奨になりました。

9.5.26. Sybase トランスレーター(sybase)

また、一般的な JDBC 翻訳の 情報も併せて参照してください。

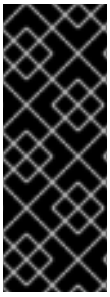
タイプ名 sybase で知られる Sybase トランスレーターは、SAP ASE(Adaptive Server Enterprise) と併用する場合、以前は Sybase SQL Server バージョン 12.5 以降と呼ばれます。

デフォルトのネイティブインポートを使用する場合は、インポートプロパティーを指定する場合、システムテーブル情報の取得中に例外を回避することができます。テーブル情報の取得時にエラーが発生した場合は、schemaName または schemaPattern を指定するか、excludeTables を使用してシステムテーブルを除外します。インポートプロパティー の使用に関する詳細は、「JDBC 翻訳者における Importer プロパティー 」を参照し てください。

ソースメタデータの名前に引用符付きの識別子（予約単語、許可されない文字が含まれる単語など）が含まれており、jConnect Sybase ドライバーを使用している場合は、最初に接続プールを設定して quoted_identifier を有効にする必要があります。

例： SQLINITSTRING を使用したドライバー URL

> jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on



重要

jConnect Sybase ドライバーを使用していて、依存する参加のソースをターゲットにする場合は、JCONNECT_VERSION を 6 以降に設定して、トランスレーターが送信できる値の数を増やします。JCONNECT_VERSION を設定しないと、481 を超えるバインド値を持つステートメントで例外が発生します。

例： JCONNECT_VERSION を使用するドライバー URL

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set quoted_identifier on&JCONNECT_VERSION=6
```

Sybase に固有の実行プロパティー

*JtdsDriver_*

オープンソース JTDS ドライバーが使用されていることを示します。デフォルトは false です。

9.5.27. Data Virtualization translator(teiid)

また、「JDBC 翻訳者」の情報 も参照してください。

Teiid データソースから仮想データベースを作成する場合は、タイプ名 teiid で認識される Teiid トランスレーターを使用します。

9.5.28. Teradata Translator(teradata)

また、「JDBC 翻訳者」の情報 も参照してください。

Teradata Translator は、T eradata Database V2R5.1 以降で使用するためのもの です。

デフォルトでは、Teradata ドライバーバージョン 15 は、Data Virtualization サーバーのタイムゾーンに合わせて日付、時間、およびタイムスタンプの値を調整します。この調整を削除するには、

translator DatabaseTimezone プロパティーを GMT に設定し、Teradata サーバーのデフォルトを指定します。

9.5.29. Vertica トランスレーター(vertica)

また、「JDBC 翻訳者」の情報 も参照してください。

タイプ名 vertica によって認識される Vertica トランスレーターは、Vertica 6 以降で使用します。

9.6. ループバックトランスレーター

タイプ名の ループバック によって認識される Loopback トランスレーターは、迅速なテストソリューションを提供します。これはすべての SQL コンストラクトと連携し、設定可能な動作を指定してデフォルトの結果を返します。

表9.5 実行プロパティー

名前	説明	デフォルト
ThrowError	**True** はエラーを常にスローします。	false
RowCount	更新以外のクエリーに対して返される行。	1
WaitTime	各ソースクエリーで、この期間をミリ秒単位で無作為に待機します。	0
PollIntervalInMilli	正の値の場合は、結果を **非同期** に返さ、(**DataNotAvailableException**) は最初にスローされ、エンジンは結果をポーリングする前にポーリング間隔を待機します。	-1
DelegateName	移動するトランスレーターの名前に設定します。	na

Loopback トランスレーターを使用して、指定のトランスレーターに対して実際のソースクエリーがどのように形成されるかを模倣できます（ループバックは、状況に有用ではないダミーデータを返します）。この動作を有効にするには、DelegateName プロパティーを、マジックするトランスレーターの

名前に設定します。たとえば、すべての機能を無効にするには、DelegateName プロパティーを jdbc-simple に設定します。

9.7. MICROSOFT EXCEL のトランスレーター

タイプ名 excel によって認識される Microsoft Excel Translator は、クエリー機能を Microsoft Excel ドキュメントに公開します。このトランスレーターは、Gracel スプレッドシートの読み取りや、Data Virtualization の他のソースと統合できる表形式でスプレッドシートの内容を提供します。

注記

このトランスレーターは、Windows や Linux を含むすべてのプラットフォームで動作します。Translator は Apache POI ライブラリーを使用して、プラットフォームに依存しない Excel ドキュメントにアクセスします。

翻訳マッピング

以下の表は、Hel Excel ドキュメントのデータをリレーショナルデータベース用語で解釈する方法を示しています。

Excel Term	平易な用語
ワークブック	schema
sheet	テーブル
行	データの行
cell	列の定義またはデータ

Excel のトランスレーターは「ソースメタデータ」機能を提供します。ここでは、特定の Excel ワークブックでは、その中に定義されたワークシートに基づいてスキーマをイントロスペクションおよびビルドできます。ワークシートでヘッダー列とデータ列を検出し、テーブルの正しいメタデータを定義するためのオプションがあります。

DDL の例

以下の例は、仮想データベースに Excel スプレッドシートを公開する方法を示しています。

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
```

```
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name"
'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
IMPORT FROM SERVER connector INTO excel OPTIONS (
    "importer.headerRowNumber" '1',
    "importer.ExcelFileName" 'names.xls');
```

ドキュメントのヘッダー

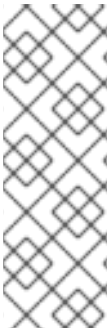
*Excel ドキュメントにヘッダーが含まれる場合は、インポートプロセスに従って、テーブル作成プロセスの列名としてセルヘッダーを選択します。インポートプロパティーの定義に関する情報は、以下の表を参照してください。また、「*Importer Properties *in JDBC translators」を参照してください。*

プロパティーのインポート

Import properties: VDB のデプロイメント時にスキーマ生成の部分をガイドします。これはネイティブインポートで使用できます。

プロパティー名	説明	デフォルト
importer.excelFileName	メタデータをインポートする Excel ドキュメントの名前を定義します。これは、ファイルパターン(*.xls)として定義できますが、パターンとして定義されている場合は、すべてのファイルが同じ形式である必要があり、トランスレーターはメタデータをインポートする任意のファイルを選択します。ファイルパターンを使用して、同じディレクトリー内の複数の Excel ドキュメントからデータを読み取ります。1つのファイルの場合は、絶対名を指定します。	必須
importer.headerRowNumber	列名として使用するセルのヘッダー情報を定義します。	任意。デフォルトはシートの最初のデータ行です。
importer.dataRowNumber	データ行の開始先の行番号を定義します。	任意。デフォルトはシートの最初のデータ行です。

Excel スプレッドシートの情報を正しく解釈できるようにするには、前述のインポータープロパティーをすべて定義することが推奨されます。



注記

列の純粋に数字のセルには、混合型を含む文字列形式が含まれるため、10 進数表現に一致する文字列形式が設定されるため、文法の値には .0 が追加されます。テキスト表現を正確に指定する必要がある場合は、セルは文字列値である必要があります。文字列の値を、引用符(')または 1 つのスペースでセルの数値の前に付けることで強制することができます。

Translator エクステンションプロパティー

- ***Excel 固有の実行プロパティー***

  FormatStrings

  ワークシートの形式に従って、文字列列の文字列以外のセルの値をフォーマットします。デフォルトは false です。

- *メタデータ拡張のプロパティー*

Table、Column、procedure など、スキーマアーティファクトで定義されるプロパティー。これらのプロパティーは、トランスレーターがソースシステムと対話または解釈する方法を説明します。すべてのプロパティーは以下の名前空間で定義されます： "http://www.teiid.org/translator/excel/2014[http://www.teiid.org/translator/excel/2014\]"。これには、認識されたエイリアス teiid_excel もあります。

プロパティー名	スキーマアイテムプロパティーが属する	説明	必須
FILE	テーブル	Excel ドキュメント名または名前パターン(*.xls)を定義します。ファイルパターンを使用すると、複数のファイルからデータを読み取ることができます。	はい
FIRST_DATA_ROW_NUMBER	テーブル	レコードがシートで開始する行番号を定義します（すべてのシートに適用）。	オプション
CELL_NUMBER	テーブルのコラム	特定の列のデータの読み取りに使用するセル番号を定義します。	はい

以下の例は、エクステンションメタデータプロパティーを使用して定義される表を示しています。

```
CREATE DATABASE excelvdb;
USE DATABASE excelvdb;
CREATE SERVER connector FOREIGN DATA WRAPPER excel OPTIONS ("resource-name"
'java:/fileDS');
CREATE SCHEMA excel SERVER connector;
SET SCHEMA excel;
CREATE FOREIGN TABLE Person (
        ROW_ID integer OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_excel:CELL_NUMBER" 'ROW_ID'),
        FirstName string OPTIONS (SEARCHABLE 'Unsearchable',
"teiid_excel:CELL_NUMBER" '1'),
        LastName string OPTIONS (SEARCHABLE 'Unsearchable',
"teiid_excel:CELL_NUMBER" '2'),
        Age integer OPTIONS (SEARCHABLE 'Unsearchable', "teiid_excel:CELL_NUMBER"
'3'),
        CONSTRAINT PK0 PRIMARY KEY(ROW_ID)
      ) OPTIONS ("NAMEINSOURCE" 'Sheet1',"teiid_excel:FILE" 'names.xlsx',
"teiid_excel:FIRST_DATA_ROW_NUMBER" '2')
```

ROW_ID 列を使用した拡張機能

拡張メタデータプロパティー CELL_NUMBER の値が ROW_ID で定義されている場合、その列の値には Excel ドキュメントの行情報が含まれます。この列に プライマリーキー のマークを付けることができます。この列は、比較述語、IN 述語、および LIMIT などの制限された機能セットを持つ SELECT ステートメントで使用できます。その他の列はすべて、クエリーの述語として使用できません。

ヒント

ソースメタデータのインポートは、Ge Excel ドキュメントのスキーマを作成するための唯一の方法ではありません。ソーステーブルを手動で作成してから、完全に機能モデルを作成する必要があるエクステンションプロパティーを追加することもできます。メタデータのインポートは、前述の例と同じスキーマモデルになります。

Excel のトランスレーターは、次の制限があります。

- ROW_ID は直接変更したり、挿入値として使用したりできません。
- UPDATE および INSERT の値はリテラルでなければなりません。
- 更新はトランザクションではありません。つまり、書き込みロックは、ファイルが書き込ま

れている間に保持されますが、更新全体では保持されません。これにより、ある更新が別の更新を上書きする可能性があります。

挿入された行の ROW_ID は、生成されたキーとして返されます。

ネイティブクエリー

この機能は Excel のトランスレーターには適用されません。

直接クエリーの手順

この機能は Excel のトランスレーターには適用されません。

9.8. MONGODB TRANSLATOR

mongodb というタイプによって認識される MongoDB トランスレーターは、MongoDB データベースにあるデータのリレーショナルデータベースビューを提供します。このトランスレーターは、Data Virtualization SQL クエリーを MongoDB ベースのクエリーに変換することができます。これは、すべての SELECT、INSERT、UPDATE、DELETE コールを提供します。

MongoDB は、独自のクエリー言語を持つドキュメントベースの「スキーマレス」データベースです。これは、リレーショナルデータベース概念または SQL クエリー言語と完全にマッピングされません。スケーラビリティーやパフォーマンスを向上させるために、MongoDB などの NOSQL ストアを使用するシステムが増えています。たとえば、監査ログの保存や Web サイトデータの管理などのアプリケーションは MongoDB に適しており、リレーショナルデータベースの構造は必要ありません。MongoDB は JSON ドキュメントをプライマリーストレージユニットとして使用し、それらのドキュメントには親ドキュメント内に追加の組み込みドキュメントを含めることができます。組み込みドキュメントを使用することで、MongoDB は、リレーショナルデータベースでクエリーを行うために重複したデータまたは結合を要求する正規化のために関連情報を同じ場所に配置し、リレーショナルデータベースでクエリーを実行します。

MongoDB が Data Virtualization と連携するには、MongoDB トランスレーターが、リレーショナルデータベースとドキュメントベースのストレージ間のバランスを実現できる MongoDB ストアを設計することです。「スキーマなしの」設計の利点は、開発時に非常に適しています。しかし、「スキーマレス」設計では、アプリケーションのバージョン間の移行時に問題を引き起こす可能性があり、データのクエリー時に、返された情報の効率的な使用が可能になります。

困難であり、既存の MongoDB コレクションに基づいてスキーマを取得することができないため、Data Virtualization は他の変換者と比較して、逆順で問題を順守します。MongoDB を使用する場合は、Data Virtualization メタデータを使用して MongoDB スキーマを定義する必要があります。Data Virtualization は、リレーショナルデータベースをメタデータとしてのみ許可するため、テーブル、手

順、および関数を使用して、リレーショナルデータベース用語で MongoDB スキーマを定義する必要があります。MongoDB の目的で、Data Virtualization メタデータが拡張され、テーブルに定義されたエクステンションプロパティーを提供して MongoDB ベースのドキュメントに変換するようになりました。これらのエクステンションプロパティーを使用すると、MongoDB ドキュメントが構造化され、保存される方法を定義できます。テーブルで定義された関係（primary-key、external-key）と、そのカーディナリティー（ONE-to-ONE、ONE-to-MANY、MANY-to-ONE）に基づいて、関連の情報をコロケーションの親ドキュメントと共に埋め込むことができるようにマッピングされます。そのため、リレーショナルデータベースベースの設計ですが、MongoDB でドキュメントベースのストレージです。

MongoDB トランスレーターの主な対象者は何ですか？

上記は、すべてのユーザーのニーズを満たさない可能性があります。MongoDB のドキュメント構造は、現在定義されている Data Virtualization よりも複雑になります。これは最終的にデータ仮想化の今後のバージョンで追いつくことを期待しています。これは現在以下の目的で設計されています。

- リレーショナルデータベースを使用しているユーザーで、データを MongoDB に移動/移行して、現在実行しているエンドユーザーアプリケーションを変更せずにスケーリングとパフォーマンスを活用します。
- シーケンスされた SQL 開発者であるが、MongoDB の経験がないユーザー。これにより、MongoDB を直接アプリケーション開発者として使用する場合と比較して、エントリーのバリアが低くなります。
- MongoDB ベースのデータを他のエンタープライズデータソースのデータと統合したいユーザー。

用途

仮想データベースの DDL で使用するトランスレーターの名前は「mongodb」です。以下に例を示します。

```
CREATE DATABASE nothwind;
USE DATABASE nothwind;
CREATE SERVER local FOREIGN DATA WRAPPER mongodb OPTIONS ("resource-name"
'java:/mongoDS');
CREATE SCHEMA northwind SERVER local;

SET SCHEMA northwind;
IMPORT FROM SERVER local INTO northwind;
```

MongoDB トランスレーターは、シナリオにおける既存のドキュメントコレクションに基づいてメタデータを派生させることができます。ただし、複雑なドキュメントを使用する場合は、メタデータの解釈が正確になる可能性があります。このような場合には、メタデータを定義する必要があります。たとえば、以下の例のように DDL を使用してスキーマを定義できます。

```
<vdb name="nothwind" version="1">
   <model name="northwind">
      <source name="local" translator-name="mongodb" connection-jndi-name="java:/mongoDS"/>
         <metadata type="DDL"><![CDATA[
            CREATE FOREIGN TABLE  Customer (
               customer_id integer,
               FirstName varchar(25),
               LastName varchar(25)
            ) OPTIONS(UPDATABLE 'TRUE');
         ]]> </metadata>
   </model>
<vdb>
```

Data Virtualization を使用してテーブルに対して以下の INSERT 操作が実行されると、MongoDB トランスレーターは MongoDB にドキュメントを作成します。

```
   INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

```
{
 _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
 customer_id: 1,
 FirstName: "John",
 LastName: "Doe"
}
```

PRIMARY KEY がテーブルに定義されている場合、その列名は MongoDB コレクションで "_id" フィールドとして自動的に使用され、以下の例のようにドキュメント構造は MongoDB に保存されます。

```
   CREATE FOREIGN TABLE  Customer (
      customer_id integer PRIMARY KEY,
      FirstName varchar(25),
      LastName varchar(25)
   ) OPTIONS(UPDATABLE 'TRUE');
```

```
{
 _id: 1,
 FirstName: "John",
 LastName: "Doe"
}
```

Customer テーブルで複合 PRIMARY KEY を定義すると、結果となるドキュメント構造は以下の例のようになります。

```
   CREATE FOREIGN TABLE  Customer (
      customer_id integer,
      FirstName varchar(25),
```

```
        LastName varchar(25),
        PRIMARY KEY (FirstName, LastName)
    ) OPTIONS(UPDATABLE 'TRUE');
```

```
{
  _id: {
        FirstName: "John",
        LastName:  "Doe"
       },
  customer_id: 1,
}
```

データ型

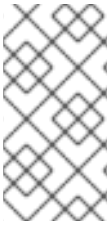
MongoDB トランスレーターは、BLOBS、CLOBS、XML など、Data Virtualization データ型の自動マッピングを MongoDB データ型に提供します。LOB マッピングは MongoDB の GridFS をベースにしています。アレイは以下の形式になります。

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

ユーザーは function array_get を使用してアレイ内の個別のアイテムを取得したり、ARRAYTABLE を使用してアレイを表形式構造に変換することもできます。

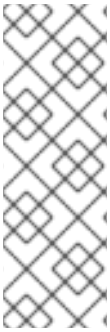
注記

組み込みドキュメントがアレイにある場合でも、埋め込みドキュメントの処理はスカラー値とアレイとは異なることに注意してください。



注記

トランスレーターは通常の式、MongoDB::Code、MongoDB::MinKey、MongoDB::MaxKey、および MongoDB::OID では機能しません。



注記

同じキーの混合タイプの値を含むドキュメントでは、列を検索不可としてマークする必要があります。そうでないと、MongoDB は列に対して述語と正しく一致しません。キーは、混合タイプとして、1 つのドキュメントの文字列値として表され、別のドキュメントの整数として使用されます。詳細は、以下の表の importer.sampleSize プロパティー を参照してください。

インポーターのプロパティー

インポータープロパティーは、物理ソースからメタデータをインポートする際にトランスレーターの動作を定義します。

インポーターのプロパティー

名前	説明	デフォルト
excludeTables	インポートから除外する正規表現。	null
includeTables	インポートからのテーブルを含む正規表現。	null
sampleSize	構造を判断するためのサンプルドキュメントの数。ドキュメントに別のフィールドがある場合、または異なるタイプのフィールドがある場合は、1を超える値である必要があります。	1
fullEmbeddedNames	埋め込みテーブル名の前に親を付けるかどうか（例：parent_embedded）。false の場合、テーブルの名前はフィールドの名前になり、既存のテーブルまたは他の埋め込みテーブルと競合する可能性があります。	false

複雑なドキュメントを構築するための MongoDB メタデータ拡張プロパティー

上記の DDL またはその他のメタデータ機能を使用して、リレーショナルデータベースのテーブルを MongoDB のドキュメントにマップできます。ただし、MongoDB を効果的に使用するには、単一の MongoDB クエリーでデータをクエリーできるように、関連情報を同じ場所に配置できる複雑なドキュメントを構築する必要があります。リレーショナルデータベースとは異なり、MongoDB で結合操作を実行できません。その結果、複雑なドキュメントをビルドしない限り、複数のクエリーを実行してデータを取得し、手動で参加する必要があります。MongoDB のパワーは、「組み込み」ドキュメント、ア

レイなどの複雑なデータ型のサポート、およびそれらをクエリーするための集約フレームワークの使用をサポートしています。このトランスレーターは、ゴールを達成する方法を提供します。

MongoDB で複雑な埋め込みドキュメントを定義しない場合、Data Virtualization は参加処理にステップを表示し、その機能を提供します。ただし、MongoDB 自体の力を利用してデータのクエリーを行い、不必要なデータを組み込み、パフォーマンスを改善したい場合は、これらの複雑なドキュメントの構築について確認する必要があります。

MongoDB トランスレーターは、複雑な「組み込み」ドキュメントの構築に役立つ他の Teiid メタデータプロパティーと 2 つの追加のメタデータプロパティーを定義します。Data Virtualization スキーマのメタデータの詳細は、「スキーマオブジェクトの DDL メタデータ」 を参照してください。DDL で以下のメタデータプロパティーを使用できます。

teiid_mongo:EMBEDDABLE

つまり、このテーブルで定義されたデータは、任意 の親ドキュメントの「埋め込み可能な」ドキュメントとして含めることができます。親ドキュメントは外部キー関係で参照されます。このシナリオでは、Data Virtualization は MongoDB ストアに複数のデータのコピーと、独自のコレクションにあるデータのコピーと、このテーブルと関係のある各親テーブルのコピーを保持します。組み込み可能な別のテーブル内に埋め込み可能なテーブルをネスト化することもできますが、一部制限があります。テーブルでこのプロパティーを使用します。テーブルが存在すると、その関係がすべて含まれます。たとえば、「製品」のカテゴリーを定義する「カテゴリー」テーブルは製品とは独立しており、「製品」テーブルに埋め込むことができます。

teiid_mongo:MERGE

これは、このテーブルのデータが定義された親テーブルにマージされることを意味します。親ドキュメントに埋め込まれたデータのコピーは 1 つだけです。親ドキュメントは、外部キー関係を使用して定義されます。

上記のプロパティーと FOREIGN KEY 関係を使用して、MongoDB で複雑なドキュメントを構築する方法を説明します。

用途

指定されたテーブルには、MongoDB でネスト化のタイプを定義する teiid_mongo:EMBEDDABLE プロパティーまたは teiid_mongo:MERGE プロパティーのいずれかを含めることができます。1 つのテーブル内で両方のプロパティーを使用することはできません。

ONE-2-ONE マッピング

ONE-2-ONE 関係を表す現在の DDL 構造は次のようになります。

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Address (
    CustomerId integer,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE');
```

デフォルトでは、サンプルデータと同様に MongoDB で 2 つの異なるコレクションを生成します。

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Address
{
  _id: ObjectID("..."),
   CustomerId: 1,
   Street: "123 Lane"
   City: "New York",
   State: "NY"
   Zipcode: "12345"
}
```

テーブルの OPTIONS 句で teiid_mongo:MERGE 拡張プロパティーを使用することで、MongoDB のストレージを単一コレクションに強化できます。

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Address (
    CustomerId integer PRIMARY KEY,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

これにより、以下のように MongoDB で単一のコレクションが生成されます。

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Address:
    {
      Street: "123 Lane",
      City: "New York",
      State: "NY",
      Zipcode: "12345"
    }
}
```

上記の両方のテーブルは、SQL コマンドで JOIN 句を使用してクエリーできる単一のコレクションにマージされます。この場合、子/追加レコードが存在すると、「teiid_mongo:MERGE" extension property is right choice」という親テーブルからは意味がありません。

注記

子テーブルに定義されている Foreign キーは、親テーブルと子テーブルの両方で Primary Keys を参照し、1-2-One 関係を形成する必要があります。

ONE-2-MANY マッピング。

通常、この関係には 2 つ以上のテーブルが含まれる可能性があります。MANY side が 単一 のテーブルのみに関連付けられている場合は、テーブルで teiid_mongo:MERGE プロパティーを使用し、ONE を親として定義します。複数のテーブルに関連付けられている場合は、teiid_mongo:EMBEDDABLE を使用します。

たとえば、仮想データベースを以下の DDL のように定義すると、以下のようになります。

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
```

```
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');
```

その後、1 人の顧客には MANY Orders を指定できます。MongoDB ドキュメントの保存方法を定義するオプションは 2 つあります。スキーマの場合、カスタマーテーブルの CustomerId は Order テーブル（順不同の目的で のみ 使用されるカスタマー情報）でのみ参照され、使用できます。

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

これは、以下のようなカスタマーテーブルの単一のドキュメントを生成します。

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Order:
  [
    {
      _id: 100,
      OrderDate: ISODate("2000-01-01T06:00:00Z")
      Status: 2
    },
    {
      _id: 101,
      OrderDate: ISODate("2001-03-06T06:00:00Z")
      Status: 5
    }
    ...
  ]
}
```

Customer テーブルが Order テーブル以外のテーブルで参照される場合は、「teiid_mongo:EMBEDDABLE」プロパティーを使用します。

```
CREATE FOREIGN TABLE Customer (
    CustomerId integer PRIMARY KEY,
```

```
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:EMBEDDABLE" 'TRUE');

CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Comments (
    CommentID integer PRIMARY KEY,
    CustomerId integer,
    Comment varchar(140),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');
```

これにより、MongoDB に 3 つの異なるコレクションが作成されます。

```
Customer
{
 _id: 1,
 FirstName: "John",
 LastName: "Doe"
}

Order
{
 _id: 100,
 CustomerId: 1,
 OrderDate: ISODate("2000-01-01T06:00:00Z")
 Status: 2
 Customer:
  {
   FirstName: "John",
   LastName: "Doe"
  }
}

Comment
{
 _id: 12,
 CustomerId: 1,
 Comment: "This works!!!"
 Customer:
  {
   FirstName: "John",
   LastName: "Doe"
  }
}
```

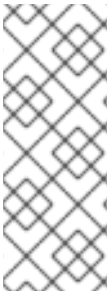
ここで見ると、カスタマーテーブルのコンテンツは参照先の他のテーブルのデータと共に組み込まれています。これにより、これらの埋め込みドキュメントが複数 MongoDB トランスレーターで自動的に管理される重複したデータが作成されます。

注記

すべてのコピーを更新する操作が複数あるため、「teiid_mongo:EMBEDDABLE」プロパティーでテーブルに対して生成される SELECT、INSERT、DELETE 操作はすべてアトミックです。MongoDB にはトランザクションがないため、Data Virtualization は今後のリリースで TEIID-2957 でトランザクションフレームワークを自動的に補正する予定です。

MANY-2-ONE マッピング

これは ONE-2-MANY と同じです。関係を定義するため上記を参照してください。



注記

親テーブルには、複数の「組み込み」と、その中に「マージ」ドキュメントを含めることができ、1 つ以上のドキュメントに限定されません。ただし、MongoDB ではドキュメントサイズが制限されていると 16 MB を超える可能性があることに注意してください。

MANY-2-MANY マッピング。

これは、「teiid_mongo:MERGE」および「teiid_mongo:EMBEDDABLE」プロパティーの組み合わせでマッピングすることもできます（特に）。たとえば DDL は次のようになります。

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE');
```

以下のような DDL を変更します。

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Order');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE',  "teiid_mongo:EMBEDDABLE" 'TRUE');
```

これにより、以下のようなドキュメントが生成されます。

```
{
  _id : 10248,
  OrderDate : ISODate("1996-07-04T05:00:00Z"),
  Status : 5
  OrderDetails : [
   {
    _id : {
          OrderID : 10248,
          ProductID : 11
          Products : {
            ProductID: 11
            ProductName: "Hammer"
          }
    }
   },
   {
    _id : {
      OrderID : 10248,
      ProductID : 14
      Products : {
        ProductID: 14
        ProductName: "Screw Driver"
      }
    }
   }
  ]
}
```

```
Products
{
   {
     ProductID: 11
     ProductName: "Hammer"
   }
   {
     ProductID: 14
     ProductName: "Screw Driver"
   }
}
```

制限事項

- MongoDB では、ネストされたアレイを処理する機能が制限されているため、ドキュメントのネストされた埋め込みは制限されています。複数のレベルで「EMBEDDABLE」プロパティーをネストすることは OK ですが、MERGE を持つ 2 つ以上のレベルは推奨されません。また、1 行にドキュメントサイズ 16 MB を超過しないように注意してください。そのため、ディープネストは推奨されません。

- 関連するテーブル間で JOINS（「EMBEDDABLE」または「MERGE」プロパティーのいずれかを使用する必要があります）。それ以外の場合は、クエリーがエラーが発生します。Data Virtualization が正しく計画および JOINS と連携するようにするには、2 つのテーブルが相互に埋め込まれてい ない 場合は、関係を表す Foreign キーで allow-joins=false プロパティーを使用します。以下に例を示します。

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId) OPTIONS (allow-join
'FALSE')
) OPTIONS(UPDATABLE 'TRUE');
```

上記の例では、Data Virtualization は 2 つのコレクションを作成しますが、ユーザーにとって以下のようなクエリーが発行されると、

```
SELECT OrderID, LastName FROM Order JOIN Customer ON Order.CustomerId =
Customer.CustomerId;
```

上記のプロパティーがないと、JOIN 処理がエラーになる代わりに、Data Virtualization エンジンで

エラーが発生します。

上記のプロパティーを使用し、MongoDB ドキュメント構造を慎重に設計する場合、Data Virtualization トランスレーターは、コロケーションに基づいてデータをインテリジェントな場所に照合し、クエリー中にデータを活用できます。

地理空間関数

MongoDB トランスレーターを使用すると、データが MongoDB ドキュメント の GeoJSon 形式に保存される際に、「WHERE」句で地理空間クエリー演算子を使用できます。以下の関数を利用できます。

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  type string, coordinates double[][]) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string,  coordinates double[], maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, coordinates double[], maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, north double, east double, west double, south double) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, north double, east double, west double, south double) RETURNS boolean;
```

サンプルクエリーは以下のようになります。

```
SELECT loc FROM maps where mongo.geoWithin(loc, 'LineString', ((cast(1.0 as double), cast(2.0 as double)), (cast(1.0 as double), cast(2.0 as double))))
```

ビルトインの Geometry タイプを使用する同じ関数（前述の一覧の関数バージョンは非推奨となり、今後のバージョンで削除されます）。

```
CREATE FOREIGN FUNCTION geoIntersects (columnRef string,  geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION geoWithin (ccolumnRef string,  geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION near (ccolumnRef string, geo geometry, maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION nearSphere (ccolumnRef string, geo geometry, maxdistance integer) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonIntersects (ref string, geo geometry) RETURNS boolean;
CREATE FOREIGN FUNCTION geoPolygonWithin (ref string, geo geometry) RETURNS boolean;
```

サンプルクエリーは以下のようになります。

```
SELECT loc FROM maps where mongo.geoWithin(loc,
ST_GeomFromGeoJSON('{"coordinates":[[1,2],[3,4]],"type":"Polygon"}'))
```

Data Virtualization の Geo Spatial 関数ライブラリーでは、「st_geom..」メソッドが複数あります。

機能

MongoDB トランスレーターは、MongoDB 集約フレームワーク上に設計されています。集約フレームワークの MongoDB バージョンを使用する必要があります。SELECT クエリーとは別に、MongoDB トランスレーターは INSERT、UPDATE、DELETE クエリーでも機能します。

MongoDB トランスレーターは以下の機能で使用できます。

- Grouping.
- マッチング。
- 並び替え。
- フィルタリング。
- 制限。
- GridFS に保存されている LOB の使用。
- 複合プライマリーおよび外部キー。

ネイティブクエリー

MongoDB ソース手順は、teiid_rel:native-query エクステンションを使用して作成できます。詳細は「Translators でパラメーター化 可能なネイティブクエリー 」を参照してください。この手順で

は、ARRAYTABLE や同様の機能を使用する必要なく、クエリーが事前に決定され、結果列タイプが認識されているという利点があります。

直接クエリーの手順

セキュリティーリスクにより、ソースに対してコマンドを実行できるセキュリティーリスクがあるため、この機能はデフォルトではオフになっています。直接クエリー手順を有効にするには、SupportsDirectQueryProcedure と呼ばれる実行プロパティーを true に設定します。詳細は、9章翻訳者 の「実行プロパティーの上書き」を 参照してください。

デフォルトでは、クエリーを直接実行する手順の名前は native と呼ばれます。デフォルト名を変更する方法は、9章翻訳者 の「実行プロパティーの上書き」を 参照してください。

MongoDB トランスレーターは、Data Virtualization の解析または解決を行わずに、アドホックの集約クエリーをソースに対して直接実行する手順を提供します。この手順の結果のメタデータは Data Virtualization には認識されないため、アレイの場所 1(1)に単一の Blob を含むオブジェクトアレイとして返されます。この Blob には JSON ドキュメントが含まれます。XMLTABLE は、クライアントアプリケーションが使用できるように表形式の出力を使用できます。

MongoDB Direct Query の例

```
select x.* from TABLE(call native('city;{$match:{"city":"FREEDOM"}}')) t,
    xmltable('/city' PASSING JSONTOXML('city', cast(array_get(t.tuple, 1) as BLOB))
COLUMNS city string, state string) x
```

上記の例では、「city」と呼ばれるコレクションは、「FREEDOM」に一致するフィルターで検索され、「ネイティブ」の手順を使用して、ネストされたテーブル機能を使用して出力が XMLTABLE コンストラクトに渡され、この手順からの出力が JSONTOXML 関数に送信され、XML の結果が表形式で公開されます。

直接クエリーは形式でなければなりません。

```
"collectionName;{$pipeline instr}+"
```

Data Virtualization 8.10 から、MongoDB トランスレーターは remove、drop、createIndex などの Shell タイプの java スクリプトコマンドを実行することもできます。この場合は、コマンドの形式にす

る必要があります。

```
"$ShellCmd;collectionName;operationName;{$instr}+"
```

たとえば、以下のようになります。

```
"$ShellCmd;MyTable;remove;{ qty: { $gt: 20 }}"
```

9.9. ODATA トランスレーター

*タイプ名「*odata*」によって認識される OData トランスレーターは OData V2 および V3 データソースを公開し、Data Virtualization Web サービスリソースアダプターを使用して Web サービス呼び出しを作成します。このトランスレーターは、Web サービストランスレーター の拡張です。*

OData とは

Open Data Protocol(OData) の Web プロトコルは、データのロックを解除し、現在のアプリケーションに存在する silo から解放する方法を提供するデータのクエリーおよび更新を行います。OData は、さまざまなアプリケーション、サービス、ストアから情報にアクセスするために、HTTP、Atom Publishing Protocol(AtomPub)、JSON などの Web テクノロジーに基づいて適用および構築することでこれを行います。OData は、リレーショナルデータベース、ファイルシステム、コンテンツ管理システム、従来の Web サイトなど、さまざまなソースから情報を公開およびアクセスするために使用されます。

OData4J フレームワークのヘルプとともに OASIS グループからこの仕様を使用することで、Data Virtualization は OData エンティティーをリレーショナルデータベースにマップします。Data Virtualization は、提供された OData エンドポイントから CSDL(Conceual Schema Definition Language)を読み取り、OData スキーマをリレーショナルデータベースに変換することができます。以下の表は、CSDL ドキュメントの OData トランスレーターのマッピングの選択を示しています。

OData	リレーショナルデータベースにマッピング
EntitySet	テーブル
FunctionImport	手順
AssociationSet	テーブル上の外部キー*
ComplexType	ignored**

- *多対多の関連付けにより、リンクテーブルが選択できませんが、結合の目的で使用することができます。*

- *関数で使用されると、暗黙的なテーブルが公開されます。埋め込みテーブルを定義するために使用されると、すべての列がインラインになります。*

すべての CRUD 操作は、OData トランスレーターに送信された SQL に基づいて生成されるエンティティーに適切にマッピングされます。

1. *用途*

OData ソースの使用状況は JDBC トランスレーターと類似しています。メタデータのインポートはトランスレーターによって提供され、メタデータがソースシステムからインポートされ、リレーショナルデータベース用語で公開されると、このソースは EntitySets および Function Imports が Data Virtualization システムにローカルでいるかのようにクエリーできます。

表9.6 実行プロパティー

名前	説明	デフォルト
DatabaseTimeZone	データベースのタイムゾーン。date、time、または timestamp の値を取得する場合に使用されます。	システムのデフォルトタイムゾーン
SupportsOdataCount	システムクエリーでの **$count** オプションの使用を有効にします。	true
SupportsOdataFilter	システムクエリーでの **$filter** オプションの使用を有効にします。	true
SupportsOdataOrderBy	システムクエリーで **$orderby** オプションの使用を有効にします。	true
SupportsOdataSkip	システムクエリーでの **$skip** オプションの使用を有効にします。	true
SupportsOdataTop	システムクエリーでの **$top** オプションの使用を有効にします。	true

表9.7 インポーターのプロパティー

名前	説明	デフォルト

名前	説明	デフォルト
schemaNamespace	インポートするスキーマの namespace。	null
entityContainer	インポートするエンティティーコンテナー名。	デフォルトのコンテナー

例： NetflixCatalog からテーブルおよびビューのみをインポートする Importer 設定

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.entityContainer" value="NetflixCatalog"/>
```

ODATA サーバーが完全に互換性がない

接続する OData サーバーは OData 仕様全体を完全に実装しない可能性があります。サーバーの OData 実装が機能をサポートしていない場合は、「execution properties」を設定して対応する機能をオフにし、Data Virtualization が無効なクエリーをトランスレーターにプッシュしないようにします。

たとえば、$filter をオフにするには、以下のステートメントを仮想データベースの DDL に追加します。

```
CREATE SERVER odata FOREIGN DATA WRAPPER "odata-override" OPTIONS
("SupportOdataFilter" 'false');
```

ネイティブクエリー

OData のトランスレーターはネイティブまたは直接クエリーの実行を実行できません。ただし、Web サービストランスレーターの invokehttp メソッドを使用して REST ベースの呼び出しを実行し、SQLXML を使用して結果を解析できます。

ODATA をサーバーとして使用

Data Virtualization は OData ベースのデータソースのみを使用できますが、すべてのデータソースを OData ベースの Web サービスとして公開することもできます。

OData サーバーの設定に関する詳細は、『Client Developer's Guide』の「OData support」を参照してください。

9.10. ODATA V4 TRANSLATOR

タイプ名「odata4」によって認識される OData V4 トランスレーターは OData Version 4 データソースを公開し、Data Virtualization Web サービスリソースアダプターを使用して Web サービス呼び出しを作成します。このトランスレーターは Web Services Translator の拡張機能です。OData V4 translator は、古い OData V1-3 ソースとは使用しないでください。古い OData ソースには OData トランスレーター("odata")を使用します。

OData とは

Open Data Protocol(OData) Web プロトコルは、データのロックを解除し、現在のアプリケーションに存在する silo から解放する方法を提供するデータのクエリーおよび更新用です。OData は、HTTP、Atom Publishing Protocol(AtomPub)、JSON などの Web テクノロジーに基づいて適用および構築することで、さまざまなアプリケーション、サービス、ストアからの情報へのアクセスを提供します。OData は、リレーショナルデータベース、ファイルシステム、コンテンツ管理システム、従来の Web サイトなど、さまざまなソースから情報を公開およびアクセスするために使用されます。

Olingo フレームワークからこの仕様を使用し、Data Virtualization は提供された OData エンドポイントから OData V4 CSDL(Conceptual Schema Definition Language)ドキュメントをマッピングし、OData メタデータを Data Virtualization のリレーショナルデータベースに変換します。以下の表は、CSDL ドキュメントの OData V4 translator マッピングの選択を示しています。

ODATA をサーバーとして使用

Data Virtualization は OData ベースのデータソースのみを使用できますが、すべてのデータソースを OData ベースの Web サービスとして公開できます。詳細は、『クライアント開発者ガイド』の「OData サポート」を参照してください。

OData	リレーショナルデータベースにマッピング
EntitySet	テーブル
EntityType	表はこちらを参照してください [1]
ComplexType	表は [2]
FunctionImport	手順 2:
ActionImport	手順 2:

OData	リレーショナルデータベースにマッピング
NavigationProperties	table [4]

[1] EntityType が EntitySet で EntitySet として公開される場合のみです。[2] 公開された EntitySet で複雑なタイプがプロパティーとして使用される場合のみです。この表は、外部キー [1-to-1] または [1-to-many] の関係のある子テーブルとして設計されています。
return type が EntityType または ComplexType の場合は、手順がテーブルを返すように設計されています。[4] ナビゲーションプロパティーはテーブルとして公開されます。この表は、親と外部キーの関係で作成されます。

すべての CRUD 操作は、OData トランスレーターに送信された SQL に基づいて生成されるエンティティーに適切にマッピングされます。

用途

OData ソースの使用は JDBC トランスレーターに似ています。メタデータのインポートはトランスレーターによってサポートされ、メタデータがソースシステムからインポートされ、リレーショナルデータベース用語で公開されると、EntitySets、Feature Imports、および Action Imports および Action Imports が Data Virtualization システムにローカルにあるかのようにこのソースをクエリーできます。

複雑なサービス用に Data Virtualization DDL を使用して独自のメタデータを定義することは推奨されません。適切な機能を有効にするには、いくつかのエクステンションメタデータプロパティーが必要です。文字列以外のプロパティーでは NATIVE_TYPE プロパティーが想定されており、完全な EDM タイプ名(Edm.xxx)を指定する必要があります。

以下は、http://odata.org サイトの TripPin サービスからメタデータサービスを読み取ることができる VDB のサンプルです。

```
<vdb name="trippin" version="1">
    <model name="trippin">
        <source name="odata4" translator-name="odata4" connection-jndi-name="java:/tripDS"/>
    </model>
</vdb>
```

Data Virtualization JDBC ドライバーを使用してデプロイした VDB に接続し、以下のように SQL ステートメントを発行します。

```
SELECT * FROM trippin.People;
SELECT * FROM trippin.People WHERE UserName = 'russelwhyte';
SELECT * FROM trippin.People p INNER JOIN trippin.People_Friends pf ON p.UserName =
```

> ***pf.People_UserName; (note that People_UserName is implicitly added by Data Virtualization metadata)***
> ***EXEC GetNearestAirport(lat, lon) ;***

実行プロパティー

デフォルトのプロパティーは、トランスレーターの適切な実行のために調整する必要があります。以下の実行プロパティーは、物理ソース機能に基づいてトランスレーターの機能を拡張するか、制限します。

名前	説明	デフォルト
SupportsOdataCount	$count に対応	true
SupportsOdataFilter	$filter に対応	true
SupportsOdataOrderBy	$orderby に対応	true
SupportsOdataSkip	$skip に対応	true
SupportsOdataTop	$top に対応	true
SupportsUpdates	INSERT/UPDATE/DELETE のサポート	true

接続する OData サーバーは OData 仕様全体を完全に実装しない可能性があります。サーバーの OData 実装が機能をサポートしていない場合は、「execution properties」を設定して対応する機能をオフにし、Data Virtualization が無効なクエリーをトランスレーターにプッシュしないようにします。

```
<translator name="odata-override" type="odata">
    <property name="SupportsOdataFilter" value="false"/>
</translator>
```

次に、ソースモデルでトランスレーター名として「odata-override」を使用します。

インポーターのプロパティー

以下の表は、物理ソースからのメタデータのインポート時にトランスレーターの動作を定義するインポータープロパティーを示しています。

名前	説明	デフォルト

名前	説明	デフォルト
schemaNamespace	インポートするスキーマの namespace	null

odata.org で公開される Trippin サービスからテーブルおよびビューのみをインポートするインポーターの設定例

```
<property name="importer.schemaNamespace"
value="Microsoft.OData.SampleService.Models.TripPin"/>
```

このプロパティーは未定義のままにすることができます。トランスレーターがプロパティーに設定されたインスタンスを検出しない場合は、EntityContainer のデフォルト名を指定します。

ヒント

ネイティブクエリー: ネイティブまたは直接クエリーの実行は、OData トランスレーターを介してはサポートされません。ただし、Web サービストランスレーターの invokehttp メソッドを使用して REST ベースの呼び出しを実行し、SQLXML を使用して結果を解析できます。

9.11. OPENAPI トランスレーター

タイプ名「openapi」によって認識される OpenAPI トランスレーターは、リレーショナルデータベースの概念を介して OpenAPI データソースを公開し、Data Virtualization WS リソースアダプターを使用して Web サービス呼び出しを作成します。

OpenAPI とは

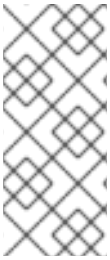
[OpenAPI は RESTful API のシンプルで強力な表現です。コントロールプレーン上の API ツールの最大エコシステムにより、数千の開発者は最新のプログラミング言語およびデプロイ環境で OpenAPI をサポートします。OpenAPI 対応の API を使用すると、インタラクティブドキュメント、クライアント SDK 生成、検出性が得られます。

このトランスレーターは OpenAPI/Swagger v2 および OpenAPI v3 と互換性があります。

用途

OpenAPI ソースの使用は、Data Virtualization の他のトランスレーターに似ています。トランス

レーターはメタデータのインポートを有効にします。メタデータは、ソースシステムのメタデータファイルからインポートされ、Data Virtualization のストアドプロシージャーとして公開されます。ソースシステムは、Data Virtualization システムでこれらのストアドプロシージャーを実行するとクエリーできます。



注記

パラメーターの順序は Swagger ライブラリーにより保証されますが、ネイティブインポートに依存する場合は、位置パラメーターではなく named を使用した手順を呼び出すのが最適です。

以下は、http://petstore.swagger.io/ サイトの Petstore 参照サービスからメタデータを読み取ることができる VDB のサンプルです。

```
<vdb name="petstore" version="1">
    <model visible="true" name="m">
        <property name="importer.metadataUrl" value="/swagger.json"/>
        <source name="s" translator-name="openapi" connection-jndi-name="java:/openapi"/>
    </model>
</vdb>
```

必要な resource-adapter 設定は以下のようになります。

```
<resource-adapter id="openapi">
    <module slot="main" id="org.jboss.teiid.resource-adapter.webservice"/>
    <transaction-support>NoTransaction</transaction-support>
    <connection-definitions>
        <connection-definition class-
name="org.teiid.resource.adapter.ws.WSManagedConnectionFactory" jndi-name="java:/openapi"
enabled="true" use-java-context="true" pool-name="teiid-openapi-ds">
            <config-property name="EndPoint">
                http://petstore.swagger.io/v2
            </config-property>
        </connection-definition>
    </connection-definitions>
</resource-adapter>
```

前述の resource-adapter を設定し、VDB を正常にデプロイした後に、Data Virtualization JDBC ドライバーを使用してデプロイされた VDB に接続でき、以下のような SQL ステートメントを発行します。

```
EXEC findPetsByStatus(('sold',))
EXEC getPetById(1461159803)
EXEC deletePet('', 1461159803)
```

実行プロパティー

実行プロパティーは、物理ソース機能に基づいてトランスレーターの機能を拡張/制限します。デフォルトのプロパティーは、トランスレーターの適切な実行のために調整する必要があります。

実行プロパティー

なし。

インポーターのプロパティー

以下の表は、物理ソースからのインポート時にトランスレーターの動作を定義するインポータープロパティーを示しています。

名前	説明	デフォルト
metadataUrl	OpenAPI メタデータの取得元となる URL。file: URL を使用するローカルファイルである場合があります。	true
server	使用するサーバー。それ以外の場合は、最初にリストされたサーバーが使用されます。	null
preferredProduces	推奨 Accept MIME タイプヘッダー。これは OpenAPI の 'produces' タイプのいずれかになります。	application/json
preferredConsumes	推奨 Content-Type MIME タイプヘッダー。これは OpenAPI の 'consumer' タイプの 1 つである必要があります。	application/json

ヒント

ネイティブクエリー: OpenAPI トランスレーターはネイティブまたは直接クエリーの実行を実行できません。ただし、Web サービストランスレーターの invokehttp メソッドを使用して REST ベースの呼び出しを実行し、SQLXML を使用して結果を解析できます。

制限事項

OpenAPI トランスレーターは OpenAPI のすべての機能を完全に実装しません。以下の制限が適用されます。

- Accept ヘッダーまたは Content-Type ヘッダーのいずれかで MIME タイプを application/xml に設定することはできません。
- ファイルおよびマッププロパティーは使用できません。そのため、複数パートペイロードはサポートされません。
- トランスレーターはセキュリティーメタデータを処理しません。
- トランスレーターは x- で始まるカスタムプロパティーを処理しません。
- トランスレーターは、以下の JSON スキーマキーワードでは機能しません。
  - allOf
  - multipleOf
  - items

9.12. SALESFORCE 翻訳者

Salesforce のトランスレーターを使用して Salesforce.com アカウントに対して SELECT、DELETE、INSERT 、UPSERT、 および UPDATE 操作を実行します。

salesforce

タイプ name salesforce によって認識されるトランスレーターは Salesforce API 37.0 以降と連携します。

表9.8 実行プロパティー

名前	説明	デフォルト
MaxBulkInsertBatchSize	一括挿入の挿入に使用するバッチサイズ。	2048
SupportsGroupBy	**GROUP BY** Pushdown を有効にします。SOQL でエラーが発生する 2000 行を超える行を返すなど、集計によって Data Virtualization プロセスグループを設定する場合は false に設定します。	true

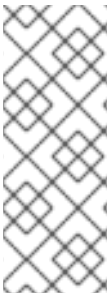
Salesforce トランスレーターはメタデータをインポートできます。

表9.9 プロパティーのインポート

プロパティー名	説明	必須	デフォルト
NormalizeNames	インポーターが引用解除を使用できるようにオブジェクト/フィールド名の変更を試みる場合。	false	true
excludeTables	テーブル名に一致する大文字と小文字を区別しない正規表現は、インポートから除外します。テーブル名の取得後に適用されます。負の先読み(?!<inclusion pattern>)を使用して、包含フィルターとして機能します。	false	該当なし
includeTables	テーブル名に対して一致する大文字と小文字を区別しない正規表現は、インポート時に含まれます。テーブル名がソースから取得された後に適用されます。	false	該当なし
importStatstics	REST API explain plan 機能を使用して、インポート中にカードを取得します。	false	false

プロパティー名	説明	必須	デフォルト
ModelAuditFields	Audit フィールドをモデルに追加します。これには、CreatedXXX、LastModifiedXXX、および SystemModstamp フィールドが含まれます。	false	false

注記：インポート中に **includeTables** *パターンと* **excludeTables** *パターンの両方が存在する場合、* **includeTables** *パターンが最初に一致すると、* **excludePatterns** *が適用されます。*



注記

構築したもの以外の **API** *バージョンへの接続が必要な場合は、既存の接続ペアの使用を試みることがありますが、場合によっては（とくに古い* **Java API** *から後のリモート api にアクセスする場合など）、これは不可能であり、接続がハングする内容が発生します。*

エクステンションメタデータプロパティー

***Salesforce** はリレーショナルデータベースではありませんが、**Data Virtualization** は **Saleforce** データをテーブルや手順などのリレーショナルデータベースにマッピングする方法を提供します。**Salesforce** の **SObject** にマップする **Data Virtualization VDB** の **DDL** を使用して外部テーブルを定義できます。ランタイム時に、このテーブルを **SObject** に解釈するには、**Data Virtualization** は追加のメタデータでこのテーブル定義を分離またはタグ付けします。たとえば、テーブルは以下の例のように定義されます。*

```
CREATE FOREIGN TABLE Pricebook2 (
 Id string,
 Name string,
 IsActive boolean,
 IsStandard boolean,
 Description string,
 IsDeleted boolean)
 OPTIONS (
   UPDATABLE 'TRUE',
   "teiid_sf:Supports Query" 'TRUE');
```

*上記の例では、プロパティー **"teiid_sf:Supports Query"** が **TRUE** に設定されている **OPTIONS** 句のプロパティーは、このテーブルに対して **SELECT** コマンドを実行できることを示しています。以下の表は、**Salesforce** スキーマで使用できるメタデータエクステンションプロパティーを示しています。*

プロパティー名	説明	必須	デフォルト	適用先
Query のサポート	テーブルに対して **SELECT** コマンドを実行できます。	false	true	テーブル
Retrieve をサポートします。	テーブルに対して実行される **SELECT** コマンドの結果を取得できます。	false	true	テーブル

SQL の処理

Salesforce はリレーショナルデータベースと同じ機能セットを提供しません。たとえば、Salesforce はテーブル間の任意の参加をサポートしません。ただし、Data Virtualization Query Planner と組み合わせて、Salesforce コネクターは Data Virtualization のほとんどの SQL 構文機能を使用できます。Salesforce Connector は、使用可能な機能に応じて、可能な限りコマンドを Salesforce に「プッシュ」して SQL コマンドを実行します。Salesforce Connector が指定の SQL コンストラクトを明示的に使用できないと、Data Virtualization は自動的に追加のデータベース機能を提供します。特定の SQL 機能を Salesforce にプッシュできない場合、Data Virtualization は実行できる機能をプッシュし、Salesforce から一連のデータを取得します。その後、Data Virtualization は追加機能を評価し、元のデータセットのサブセットを作成します。最後に、Data Virtualization は結果をクライアントに渡します。

GROUP BY 句でクエリーを発行し、queryMore がサポートされていないことを示す Salesforce エラーを受信し、制限を追加するか、または実行プロパティー SupportsGroupBy を false に設定できます。

```
SELECT array_agg(Reports) FROM Supervisor where Division = 'customer support';
```

Salesforce や Salesforce Connector は、array_agg（） スカラーをサポートしていません。ただし、どちらも CompareCriteriaEquals クエリーと互換性があるため、コネクターはこのクエリーを受信するクエリーを Salesforce に変換します。

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

array_agg（）関数は、Data Virtualization Query Engine によってコネクターによって返される結果セットに適用されます。

コネクターに渡される SQL を処理するため、Salesforce アプリケーションへの複数の呼び出しが行われることがあります。

```
DELETE From Case WHERE Status = 'Closed';
```

Salesforce の API からオブジェクトの削除は、オブジェクト ID でのみ削除できます。そのために、Salesforce コネクターは最初にクエリーを実行して正しいオブジェクトの ID を取得し、次にこれらのオブジェクトを削除します。そのため、上記の DELETE コマンドは以下のコマンド 2 つになります。

```
SELECT ID From Case WHERE Status = 'Closed';
DELETE From Case where ID IN (<result of query>);
```

注記： Salesforce API DELETE 呼び出しは SQL で表現されていませんが、上記は同等の SQL 式です。

Salesforce から大規模なデータセットを取得し、可能な限りクエリーを行うのを避けるために、互換性のない機能を認識しておくと便利です。Salesforce にプッシュできる SQL コンストラクトに関する詳細は、互換性のある SQL 機能 を参照してください。

複数選択の選択リストからの選択

マルチ選択リストとは、1 つのフィールドに複数の値を含むことができる Salesforce のフィールドタイプです。SOQL のこのタイプのフィールドのクエリー基準演算子は、EQ、NE、include および exclude に制限されます。複数選択 の選択リストから選択する Salesforce ドキュメントは、「Multi-select Picklistsのクエリー」を参照してください。

Data Virtualization SQL は includes 演算子または excludes 演算子をサポートしませんが、Salesforce コネクターはこれらの Operator のユーザー定義関数定義を提供し、タイプがマルチ選択のフィールドに同等の機能を提供します。関数の定義は以下のとおりです。

```
boolean includes(Column column, String param)
boolean excludes(Column column, String param)
```

たとえば、これらの値すべてが含まれる Status という名前のマルチ選択のコラムを 1 つ取ります。

- current
- working
- critical

この列については、以下のすべてが有効なクエリーになります。

```
SELECT * FROM Issue WHERE true = includes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = excludes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = includes (Status, 'current;working, critical' );
```

EQ および NE の基準は、指定されるとおり Salesforce に渡されます。たとえば、これらのクエリーはコネクターによって変更されません。

```
SELECT * FROM Issue WHERE Status = 'current';
SELECT * FROM Issue WHERE Status = 'current;critical';
SELECT * FROM Issue WHERE Status != 'current;working';
```

すべてのオブジェクトの選択

Salesforce コネクターを使用して、Salesforce API から queryAll 操作を呼び出すことができます。queryAll 操作は、システム内の現在および削除された全オブジェクトに関するデータを返す例外によるクエリー操作と同等です。

コネクターは、各 Salesforce オブジェクトにある isDeleted プロパティーへの参照を介して query または queryAll 操作を呼び出すかどうかを決定します。また、インポーターによって生成された各テーブルの列としてモデル化されます。デフォルトでは、モデルが生成され、コネクターがクエリーを呼び出すと、この値は false に設定されます。ユーザーはモデルの値を true に自由に変更し、コネクターのデフォルト動作を queryAll に変更します。

isDeleted がクエリーのパラメーターとして使用されている場合の動作は異なります。isDeleted 列がクエリーのパラメーターとして使用され、値が true の場合、コネクターは queryAll を呼び出します。

```
select * from Contact where isDeleted = true;
```

isDeleted 列がクエリーのパラメーターとして使用され、値が false の場合、デフォルトの動作を実行するコネクターがクエリーを呼び出します。

```
select * from Contact where isDeleted = false;
```

更新されたオブジェクトの選択

Salesforce からメタデータをインポートする際にオプションが選択されている場合は、以下の構造でモデルに GetUpdated 手順が生成されます。

```
GetUpdated (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    LatestDateCovered OUT datetime)
returns
    ID string
```

使用方法は、Salesforce ドキュメントの GetUpdated 操作の説明を参照してください。

削除されたオブジェクトの選択

Salesforce からメタデータをインポートする際にオプションが選択されている場合は、以下の構造でモデルに GetDeleted 手順が生成されます。

```
GetDeleted (ObjectName IN string,
    StartDate IN datetime,
    EndDate IN datetime,
    EarliestDateAvailable OUT datetime,
    LatestDateCovered OUT datetime)
returns
    ID string,
    DeletedDate datetime
```

使用方法は、Salesforce ドキュメントの GetDeleted 操作の説明を参照してください。

関係クエリー

リレーショナルデータベースとは異なり、Salesforce は結合操作をサポートしませんが、オブジェクト間の子または子間の関係を含むクエリーをサポートします。これらは、概念的なクエリーです。Outer Join 構文を使用すると、SalesForce コネクターで以下のクエリーを実行できます。

```
SELECT Account.name, Contact.Name from Contact LEFT OUTER JOIN Account
on Contact.Accountid = Account.id
```

このクエリーは、子から親への関係クエリーを生成するために SalesForce モデルをクエリーするための正しい構文を示しています。SalesForce に対して以下のクエリーを解決します。

```
SELECT Contact.Account.Name, Contact.Name FROM Contact
```

```
select Contact.Name, Account.Name from Account Left outer Join Contact
on Contact.Accountid = Account.id
```

このクエリーは、親から子への関係クエリーを生成するために SalesForce モデルをクエリーするための正しい構文を示しています。SalesForce に対して以下のクエリーを解決します。

```
SELECT Account.Name, (SELECT Contact.Name FROM
Account.Contacts) FROM Account
```

制限については、SalesForce ドキュメントの「 Queries 」操作の説明を参照してください。

クエリーの一括挿入

また、JDBC バッチセマンティクスまたは SELECT INTO セマンティクスを使用して、SalesForce トランスレーターで一括挿入ステートメントを使用することもできます。バッチサイズは実行プロパティー MaxBulkInsertBatchSize で決定され、vdb ファイルで上書きできます。バッチのデフォルト値は 2048 です。一括挿入機能は、Salesforce が公開する非同期 REST ベースの API を使用して、パフォーマンスを強化します。

一括選択

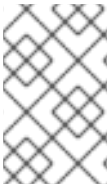
10,000,000 を超えるレコードを持つテーブルをクエリーする場合、または結果のバッチ処理でタイムアウトが発生した場合、Data Virtualization は一括 API を使用して Salesforce へのクエリーを発行できます。一括選択を使用する場合は、クエリーと互換性のあるプライマリーキー(PK)のチャンクが有効になります。

一括 API を使用するには、クエリーに source ヒントが必要です。

```
SELECT /*+ sh salesforce:'bulk' */ Name ... FROM Account
```

salesforce はターゲットソースのソース名に置き換えます。

デフォルトのチャンクサイズ 100,000 レコードが使用されます。



注記

この機能は Salesforce API バージョン 28 以降でのみサポートされます。

互換性のある SQL 機能

以下の SQL 機能を Salesforce コネクターで使用します。これらの SQL コンストラクトは Salesforce にプッシュされます。

- *SELECT コマンド*
- *INSERT コマンド*
- *UPDATE コマンド*
- *DELETE コマンド*
- *NotCriteria*
- *OrCriteria*
- *CompareCriteriaEquals*
- *CompareCriteriaOrdered*
- *IsNullCritiera*
- *InCriteria*
- *LikeCriteria: String フィールドのみに使用できます。*
- *RowLimit*
- *基本的な集約値*
- *結合基準 KEY で参加*

ネイティブクエリー

Salesforce の手順には、任意でネイティブクエリーが関連付けられる場合があります。詳細は「Translators でパラメーター化 可能なネイティブクエリー 」を参照してください。操作接頭辞（select;、insert、update、delete、delete）は native-query に存在する必要がありますが、ソースへのクエリーの一部として発行されません。

Salesforce ネイティブ手順の DDL の例

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS ("teiid_rel:native-query" 'search;SELECT ... complex SOQL ... WHERE col1 = $1 and col2 = $2')
returns (col1 string, col2 string, col3 timestamp);
```

直接クエリーの手順

セキュリティーリスクにより、ソースに対してコマンドを実行できるセキュリティーリスクがあるため、この機能はデフォルトではオフになっています。直接クエリー手順を有効にするには、SupportsDirectQueryProcedure と呼ばれる実行プロパティーを true に設定します。詳細は、9章翻訳者 の「実行プロパティーの上書き」を 参照してください。

ヒント

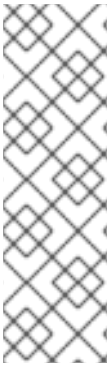
デフォルトでは、クエリーを直接実行する手順の名前は native と呼ばれます。デフォルト名を変更する方法は、9章翻訳者 の「実行プロパティーの上書き」を 参照してください。

Salesforce Translator は、Data Virtualization の解析または解決を行わずに、アドホック SOQL クエリーをソースに対して直接実行する手順を提供します。この手順の結果のメタデータは Data Virtualization には認識されないため、オブジェクト配列として返されます。ARRAYTABLE は、クライアントアプリケーションが使用するための表形式出力です。Data Virtualization は、以下の手順で以下のように単純なクエリー構造で公開します。

example の選択

```
SELECT x.* FROM (call sf_source.native('search;SELECT Account.Id, Account.Type, Account.Name FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS x
```

上記のコードから、「search」キーワードの後にクエリーステートメントが続きます。



注記

SOQL は、パラメーター値をクエリー文字列に適切に挿入できるように、パラメーター化されたネイティブクエリーとして扱われます。詳細は「Translators でパラメーター化 可能なネイティブクエリー 」を参照してください。検索によって返される結果には、選択されたかどうかにかかわらず、最初の列の値としてオブジェクト ID が含まれる場合があります。また、複数のオブジェクトタイプから列を選択するクエリーは適切ではありません。

例の削除

```
SELECT x.* FROM (call sf_source.native('delete;', 'id1', 'id2')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

上記のコードを形成し、「delete;" キーワードの後に varargs から削除する ids」が続きます。

Create example

```
SELECT x.* FROM
 (call sf_source.native('create;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

上記のコードを形成するには、「create」または「update」キーワードの後に以下のプロパティーを指定する必要があります。これらの属性は、手順変数によって位置で一致する必要があります。そのため、例の属性 2 は 2 に設定されます。

プロパティー名	説明	必須
type	テーブル名	はい
attributes	列の名前のコンマ区切りリスト	いいえ

各属性の値は、「ネイティブ」手順に個別の引数として指定されます。

update は、レコードの識別子を定義する、さらに 1 つの追加プロパティー「id」で create と似ています。

更新の例

```
SELECT x.* FROM
 (call sf_source.native('update;id=pk;type=table;attributes=one,two,three', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

ヒント

デフォルトでは、クエリーを直接実行する手順の名前は native と呼ばれますが、DDL ファイルにオーバーライドの実行プロパティーを設定して変更することができます。

9.13. REST トランスレーター

タイプ名 rest によって認識される Rest translator は、REST サービスを呼び出すためのストアドプロシージャーを公開します。 このトランスレーターの結果は通常、CSV、JSON、または XML 形式のデータを使用する TEXTTABLE、JSONTABLE、または XMLTABLE テーブル関数で使用されます。

実行プロパティー

残り のインポーター設定はありませんが、VDB のメタデータを提供できます。

用途

rest translator は、Web サービスへのアクセスに関する低レベルの手順を公開します。

InvokeHTTP 手順

invokeHttp は HTTP(S)呼び出しのバイトコンテンツを返すことができます。

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in
BOOLEAN, contentType out STRING, headers in CLOB) returns BLOB
```

action は、デフォルトで POST を指定する HTTP メソッド（GET、POST など）を示します。

エンドポイントの null 値はデフォルト値を使用します。デフォルトのエンドポイントは rest ソース設定で指定されます。エンドポイント URL は絶対または相対できます。相対的の場合は、デフォルトのエンドポイントと組み合わされます。

複数のパラメーターに値を含める必要はないため、多くの場合、名前付きパラメーター構文で invokeHttp の手順を呼び出すことがより明確です。

```
call invokeHttp(action=>'GET')
```

リクエストは SQLXML、STRING、BLOB、または CLOB のいずれかです。リクエストは、バイト形式で POST ペイロードとして送信されます。STRING/CLOB 値の場合、デフォルトは UTF-8 エンコーディングになります。バイトエンコーディングを制御するには、to_bytes 関数を参照してください。

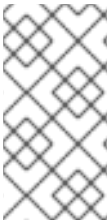
オプションの headers パラメーターを使用して、要求ヘッダーの値を JSON 値として指定できます。JSON 値は、プリミティブまたはプリミティブ値のリストを持つ JSON オブジェクトである必要があります。

```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip',
'deflate') as "Accept-Encoding"))
```

headers パラメーター設定の推奨事項：

- HTTP POST/PUT メソッドが呼び出される場合は、Content-Type が必要になる場合があります。

- *メディア タイプを返す場合は、accept が必要です。*



ネイティブクエリー

Web サービストランスレーターでは、ネイティブクエリーや直接クエリーの実行手順は使用できません。

ストリーミングに関する考慮事項

stream パラメーターが true に設定されている場合、生成される LOB 値は 1 回のみ使用できます。ストリームが null または false の場合、エンジンは結果のコピーを保存して繰り返し使用する必要がある場合があります。キャストや XMLPARSE など、一部の操作として使用すると、ストリームが消費される結果の検証が実行される可能性があります。

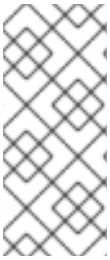
9.14. WEB サービストランスレーター

タイプ名 soap または ws によって認識される Web Services Translator は、web/ SOAP サービスを呼び出すためのストアドプロシージャーを公開します。 このトランスレーターの結果は通常、CSV または XML 形式のデータを使用する TEXTTABLE または XMLTABLE テーブル関数で使用されます。

実行プロパティー

名前	説明	使用時	デフォルト
DefaultBinding	指定がない場合は使用すべきバインディングです。HTTP、SOAP11、または SOAP12 のいずれかを使用できます。	invoke*	SOAP12
DefaultServiceMode	デフォルトのサービスモード。SOAP の場合、MESSAGE モードは、リクエストに SOAP ボディーのコンテンツだけでなく、SOAP エンベロープ全体が含まれることを示します。MESSAGE または PAYLOAD のいずれかを使用できます。	invoke* または WSDL コール	PAYLOAD

名前	説明	使用時	デフォルト
XMLParamName	リクエストドキュメントがクエリー文字列の一部であることを示すために、HTTP バインディング（通常は GET メソッド）で使用されます。	invoke*	null - 未使用



注記

翻訳者に適切なバインディング値を設定することは、呼び出し元が明示的な値を渡す必要がなくなるため、推奨されています。サービスが実際には SOAP11 を使用しますが、バインディングが SOAP12 を使用した場合は、実行に失敗します。

インポーターの設定はありませんが、VDB のメタデータを提供できます。コネクションが特定の WSDL を参照するよう設定されている場合、トランスレーターは指定の service および port ですべての SOAP 操作をインポートします。

インポーターのプロパティー

インポータープロパティーを指定する場合は、「importer」のプレフィックスを指定する必要があります。例： importer.tableTypes

名前	説明	デフォルト
importWSDL	resource-adapter に設定された WSDL URL からメタデータをインポートします。	true

用途

トランスレーターは、Web サービスへのアクセスに関する低レベルの手順を公開します。

プロシージャーの実行

invoke は、複数のバインディングまたはプロトコルモード（HTTP、SOAP11、SOAP12 など）を許可します。

> *Procedure invoke(binding in STRING, action in STRING, request in XML, endpoint in STRING, stream in BOOLEAN) returns XML*

バインディングは、null（デフォルト）の HTTP、SOAP11、または SOAP12 のいずれかにすることができます。SOAP バインディングのアクション は、SOAPAction の値を示します。HTTP バインディングを持つアクションは、デフォルトで POST に設定されている HTTP メソッド（GET、POST など）を示します。

バインディングまたはエンドポイントの null 値はデフォルト値を使用します。デフォルトのエンドポイントはソース設定に指定されます。エンドポイント URL は絶対または相対できます。相対的の場合は、デフォルトのエンドポイントと組み合わされます。

複数のパラメーターに値を含める必要がないため、named パラメーター構文で呼び出しの手順を呼び出すことがより明確になります。

```
call invoke(binding=>'HTTP', action=>'GET')
```

リクエスト XML は有効な XML ドキュメントまたはルート要素である必要があります。

InvokeHTTP 手順

invokeHttp は HTTP(S)呼び出しのバイトコンテンツを返すことができます。

> *Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, stream in BOOLEAN, contentType out STRING, headers in CLOB) returns BLOB*

action は、デフォルトで POST を指定する HTTP メソッド（GET、POST など）を示します。

エンドポイントの null 値はデフォルト値を使用します。デフォルトのエンドポイントはソース設定に指定されます。エンドポイント URL は絶対または相対できます。相対的の場合は、デフォルトのエンドポイントと組み合わされます。

複数のパラメーターに値を含める必要はないため、多くの場合、名前付きパラメーター構文で invokeHttp の手順を呼び出すことがより明確です。

```
call invokeHttp(action=>'GET')
```

リクエストは SQLXML、STRING、BLOB、または CLOB のいずれかです。リクエストは、バイト

形式で POST ペイロードとして送信されます。STRING/CLOB 値の場合、デフォルトは UTF-8 エンコーディングになります。バイトエンコーディングを制御するには、to_bytes 関数を参照してください。

オプションの headers パラメーターを使用して、要求ヘッダーの値を JSON 値として指定できます。JSON 値は、プリミティブまたはプリミティブ値のリストを持つ JSON オブジェクトである必要があります。

```
call invokeHttp(... headers=>jsonObject('application/json' as "Content-Type", jsonArray('gzip', 'deflate') as "Accept-Encoding"))
```

headers パラメーター設定の推奨事項：

- HTTP POST/PUT メソッドが呼び出される場合は、Content-Type が必要になる場合があります。
- メディア タイプを返す場合は、accept が必要です。

WSDL ベースの手順

上記の手順では、エンドポイントを提供して Web サービスメソッドを実行する匿名の方法を提供します。このメカニズムを使用すると、WSDL で定義されたエンドポイントを別のエンドポイントで変更できます。しかし、WSDL にアクセスできる場合は、web-service resource-adapter の接続設定で WSDL URL を設定できます。デフォルトのネイティブメタデータインポートを使用している場合は、Web サービスのソースモデルに手順が表示されます。

ネイティブクエリー

Web サービストランスレーターでは、ネイティブクエリーや直接クエリーの実行手順は使用できません。

ストリーミングに関する考慮事項

stream パラメーターが true に設定されている場合、生成される LOB 値は 1 回のみ使用できます。ストリームが null または false の場合、エンジンは結果のコピーを保存して繰り返し使用する必要がある場合があります。キャストや XMLPARSE など、一部の操作として使用すると、ストリームが消費される結果の検証が実行される可能性があります。

第10章 フェデレーションされたプランニング

コアでは、Data Virtualization はフェデレーションされたリレーショナルデータベースエンジンです。このクエリーエンジンでは、すべてのデータソースを 1 つの仮想データベースとして扱い、単一の SQL クエリーを介してそのデータソースにアクセスすることができます。その結果、手動でコーディングする代わりに、アプリケーションのビルドやデータソース間の他のリレーショナルデータベース操作を実行することにフォーカスできます。

10.1. プランニングの概要

クエリーエンジンが受信 SQL クエリーを受け取ると、以下の操作を実行します。

1. pidgin-gitopsValidates 構文を 解析 し、内部フォームに変換します。
2. すべての識別子を関数ライブラリーにメタデータと関数に 解決 します。
3. メタデータの参照とタイプの署名に基づいた SQL セマンティクスの 検証
4. 式と基準を簡素化するために SQL を 書き換え ます。
5. 論理計画の最適化 で、通常の SQL を、詳細な最適化のための論理プランに反転します。Data Virtualization オプティマイザーは、主にルールベースです。クエリー構造とヒントに基づいて、特定のルールセットが適用されます。これらのルールは、より多くのルールの実行をトリガーする可能性があります。複数のルール内で、Data Virtualization はコスト情報を活用します。論理計画の最適化手順は、『クライアント開発ガイド』 で説明されているように、'SET SHOWPLAN DEBUG' クラッシュを使用して確認できます。手順の例は、『Query Planner』 の「Reading a debug plan 」を参照してください。論理プランノードおよびルールベースの最適化の詳細については、「Query Planner 」を参照してください。
6. 処理計画変換 gitops-gitopsConvert は、ノードが基本的な処理操作を表す実行可能なフォームにロジック計画を反転します。最後の処理計画がクエリープランとして表示されます。詳細は、「プランのクエリー」を参照 してください。

論理クエリー計画は、ソーステーブルのデータを予想される結果セットに変換するために使用される操作のツリーです。ツリーの下部（テーブル）から上部（出力）にデータフロー。主要な 論理操作は、条件に基づいて行を 選択 （選択またはフィルタリング）、プロジェクト （プロジェクトまたはコン

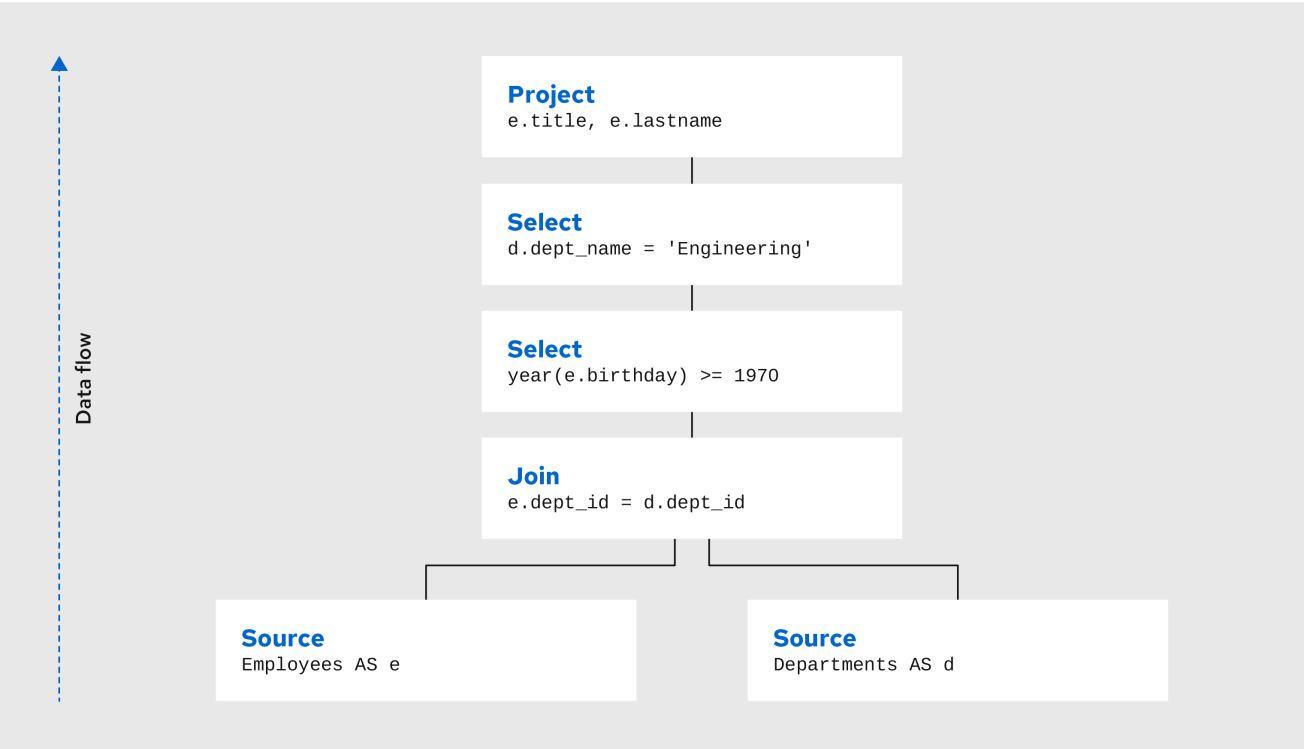
ピュート列値）、結合 （テーブルからのデータの取得）、ソート (ORDER BY)、重複削除 (SELECT DISTINCT)、グループ (GROUP BY)、およびユニオン (UNION)です。

たとえば、1970 からすべてのエンジニアリング従業員を取得する以下のクエリーについて考えてみましょう。

サンプルクエリー

```
SELECT e.title, e.lastname FROM Employees AS e JOIN Departments AS d ON e.dept_id =
d.dept_id WHERE year(e.birthday) >= 1970 AND d.dept_name = 'Engineering'
```

論理的には、Employees テーブルと Departments テーブルからのデータが取得され、指定したとおりに結合されてフィルターされ、最後に出力列が展開されます。そのため、正規のクエリー計画は以下のようになります。



63_RHI_0220

結合から下層のテーブルからデータフローが選択され、最終的にプロジェクトが最終結果を生成します。各ノード間で渡されるデータは、列と行を持つ結果セットの論理的です。

当然ながら、これは、プランが実際に実行される方法ではありません。この初期計画から、クエリープランナーはクエリー計画ツリーで変換を実行し、同じ結果を取得する同等のプランを迅速に生成します。フェデレーションされたクエリープランナーとリレーショナルデータベースプランナーの両方が、同じ概念と多くの同じプラン変換を処理します。この例では、**Departments** と **Employees** テーブルの基準がツリーにプッシュされ、結果をできるだけ早くフィルターします。

いずれの場合も、目的はクエリーの結果をできるだけ早く取得することです。ただし、リレーショナルデータベースプランナーは、主にストレージからデータをプルする際にアクセスパスを最適化することで、これを実現します。

これとは対照的に、フェデレーションされたクエリープランナーはデータソースに負担を課すため、ストレージアクセスに関する懸念はありません。フェデレーションされたクエリープランナーの最も重要な考慮事項は、データ転送を最小限に抑えることです。

10.2. クエリープランナー

user コマンドの各サブコマンドについて、適切なサブプランナー（辞書、**XML**、手順など）が使用されます。

各プランナーには、**3** つの主要なフェーズがあります。

1. 正規プランの生成
2. 最適化
3. コンバーターでコンバーターで処理をプランニングする場合は、データ構造を処理フォームにプランニングします。

リレーショナルデータベースプランナー

論理計画が一連のルールで操作した後に、オーソナイザーによってリレーショナルデータベース処理計画が作成されます。ルールの適用は、クエリー構造とルール自体によって決定されます。デバッグ

プランのノード構造は処理計画のようになりますが、ノードタイプはより論理的に SQL 操作を表します。

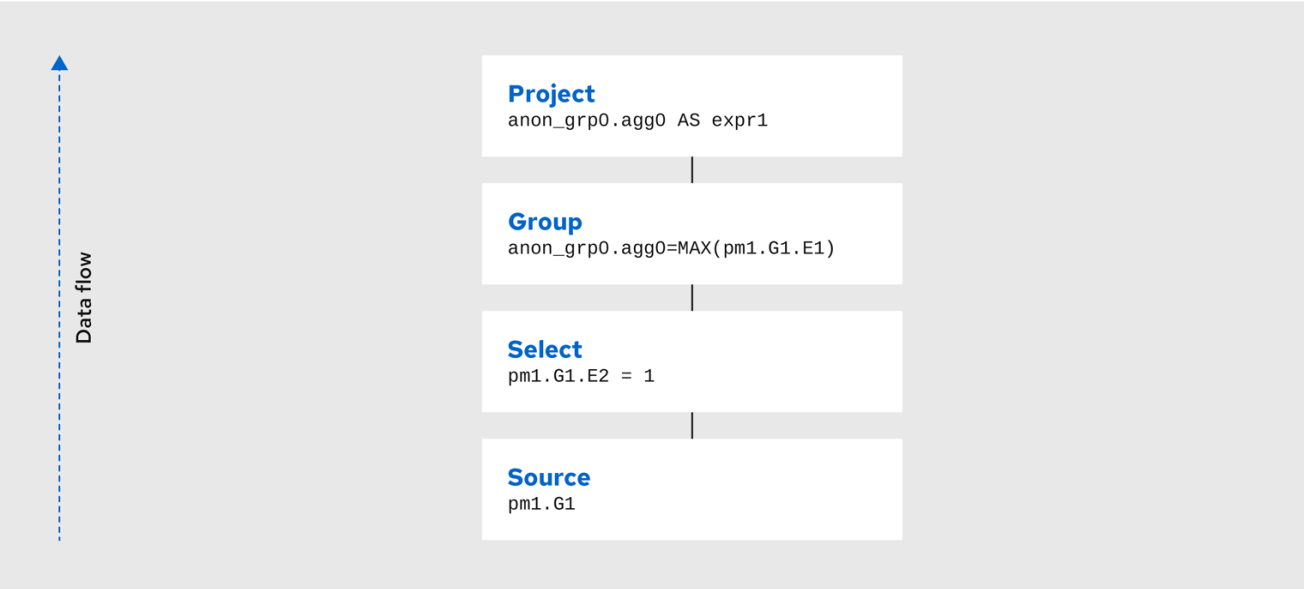
正規計画およびすべてのノード

Planning overview で説明されているように、クエリーエンジンに送信された SQL ステートメントは、正規プランフォームに変換される前に解析、解決、検証、および書き換えられます。正規計画のフォームは、最初の SQL 構造とほぼ同じです。SQL 選択クエリーには、以下の使用可能な句があります(All but SELECT are optional): WITH, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMITこれらの句は、以下の順序で論理的に実行されます。

1. WITH（共通のテーブル式を作成）specifically PROJECT NODE が行います（共通のテーブル式を作成）。
2. FROM（テーブルからすべてのデータを読み取りおよび結合））、from 句項目ごとに SOURCE ノード、または Join ノード（>1 テーブルの場合）が SOURCE ノードによって結合されます。
3. SELECT ノードによる WHERE（フィルター行）autoMember-gitopsProcessed。
4. GROUP BY（グループ行を折りたたんだ行に分類）GROUP ノードにより GROUP-2Processed。
5. HAVING（フィルターグループ化された行）で SELECT ノードを使用。
6. SELECT（evaluate 式および要求された行のみを返します）は、PROJECT ノードおよび DUP_REMOVE ノード（SELECT DISTINCT 用）で で使用された行のみを返します。
7. INTOで出所： SOURCE 子を持つ特別な PROJECT を処理します。
8. ORDER BY（ソート行）で、SORT ノードによって付けられます。
9. LIMIT（結果を特定の範囲に制限）LIMIT - LIMIT ノードによって満たされます。

たとえば、SELECT max(pm1.g1.e1)FROM pm1.g1 WHERE e2 = 1 などの SQL ステートメントが

論理プランを作成します。




```
Project(groups=[anon_grp0], props={PROJECT_COLS=[anon_grp0.agg0 AS expr1]})
  Group(groups=[anon_grp0], props={SYMBOL_MAP={anon_grp0.agg0=MAX(pm1.G1.E1)}})
    Select(groups=[pm1.G1], props={SELECT_CRITERIA=pm1.G1.E2 = 1})
      Source(groups=[pm1.G1])
```

ここでは、Source は FROM 句に対応し、Select は WHERE 句に対応し、Group は最大アグリゲートを作成する暗黙的なグループ化に対応し、Project は SELECT 句に対応します。

注記

グループ化により、グループによって作成される値の展開を処理するインラインビューである anon_grp0 が生成されます。

表10.1 ノードタイプ

タイプ名	説明
ACCESS	ソースアクセスまたはプランの実行。
DUP_REMOVE	重複行を削除します。

タイプ名	説明
JOIN	参加（LEFT OUTER、FULL OUTER、INNER、CROSS、SEMI など）。
PROJECT	タプル値の展開
SELECT	タプルのフィルタリング
SORT	joins などの他の操作を処理するために挿入できる順序付け操作。
ソース	インラインビュー、ソースアクセス、XMLTABLE などを含むタプルの論理ソース。
グループ	グループ化操作。
SET_OP	セット操作(UNION/INTERSECT/EXCEPT)。
NULL	タプルのないソース。
TUPLE_LIMIT	行オフセット / 制限

ノードのプロパティー

各ノードには、通常ノードに表示される適用可能なプロパティーのセットがあります。

表10.2 アクセスプロパティー

プロパティー名	説明
ATOMIC_REQUEST	ソース要求の最終フォーム。
MODEL_ID	ターゲットモデル/スキーマのメタデータオブジェクト。
PROCEDURE_CRITERIA/PROCEDURE_INPUTS/PROCEDURE_DEFAULTS	プランニング手順のリレーショナルデータベースクエリーで使用されます。
IS_MULTI_SOURCE	ノードがマルチソースアクセスを表す場合は true に設定します。
SOURCE_NAME	複数ソース名を追跡するのに使用します。
CONFORMED_SOURCES	準拠した拡張メタデータが使用される場合に、準拠したソースのセットを追跡します。

プロパティー名	説明
SUB_PLAN/SUB_PLANS	マルチソースのプランニングで使用されます。

表10.3 操作プロパティーの設定

プロパティー名	説明
SET_OPERATION/USE_ALL	set 操作(UNION/INTERSECT/EXCEPT)およびすべての行または異なる行が使用される場合を定義します。

表10.4 結合プロパティー

プロパティー名	説明
JOIN_CRITERIA	すべての結合述語。
JOIN_TYPE	join のタイプ（INNER、LEFT OUTER など）。
JOIN_STRATEGY	使用するアルゴリズム（入れ子ループ、マージなど）
LEFT_EXPRESSIONS	結合の左側から発信される equi-join 述語の式。
RIGHT_EXPRESSIONS	結合の右側の equi-join 述語の式。
DEPENDENT_VALUE_SOURCE	依存する結合が使用されている場合に設定します。
NON_EQUI_JOIN_CRITERIA	同等でない結合述語。
SORT_LEFT	結合処理用に左側でソートする必要がある場合。
SORT_RIGHT	参加処理に右側のソートが必要な場合は、以下を行います。
IS_OPTIONAL	join が任意の場合。
IS_LEFT_DISTINCT	左側が、equi 結合述語に関して異なる場合。
IS_RIGHT_DISTINCT	右側が、equi 結合述語に関して異なる場合。
IS_SEMI_DEP	依存する結合が半結合を表す場合。
保存	preserve ヒントが結合順序を保持する場合。

表10.5 プロジェクトのプロパティー

プロパティー名	説明
PROJECT_COLS	Projected 式。
INTO_GROUP	このグループが、クエリー式で選択または挿入する場合にターゲットになります。
HAS_WINDOW_FUNCTIONS	ウィンドウ関数を使用している場合は True。
制約	値がグループに展開される場合に満たさなければならない制約。
UPSERT	挿入が upsert の場合。

表10.6 プロパティーの選択

プロパティー名	説明
SELECT_CRITERIA	フィルター。
IS_HAVING	フィルターがグループ化後に適用される場合。
IS_PHANTOM	ノードが削除対象とマークされているが、プランに一時的に残されている場合は True。
IS_TEMPORARY	最終的なプランで使用されていない可能性のある推論された基準。
IS_COPIED	条件がルールのコピー基準ですでに処理されている場合は、
IS_PUSHED	条件をできるだけ早くプッシュする場合は、
IS_DEPENDENT_SET	基準が依存する結合のフィルターである場合。

表10.7 ソートプロパティー

プロパティー名	説明
SORT_ORDER	ソートを定義する順序。
UNRELATED_SORT	順序に展開されていない値が含まれる場合。
IS_DUP_REMOVAL	ソートが展開全体に対して重複削除を実行する必要がある場合。

表10.8 ソースプロパティー

プロパティー名	説明
SYMBOL_MAP	ソース上のコラムから展開された式へのマッピング。グループノードにも存在します。
PARTITION_INFO	ユニオンブランチのパーティション設定。
VIRTUAL_COMMAND	ソースがビューまたはインラインビューを表す場合は、ビューで定義したクエリーです。
MAKE_DEP	ヒント情報。
PROCESSOR_PLAN	（通常は NESTED_COMMAND からの）非リレーショナルデータベースソースのプロセッサープラン。
NESTED_COMMAND	非リレーショナルデータベースコマンド。
TABLE_FUNCTION	ソースを定義するテーブル関数（XMLTABLE、OBJECTTABLE など）。
CORRELATED_REFERENCES	ソースの下にあるノードの相関参照。
MAKE_NOT_DEP	make not dep が設定されている場合。
INLINE_VIEW	ソースノードがインラインビューを表示する場合。
NO_UNNEST	no_unnest ヒントが設定されている場合。
MAKE_IND	make が hint が設定されている場合。
SOURCE_HINT	ソースヒント「Federated optimizations」を参照してください。
ACCESS_PATTERNS	アクセスパターンがまだ満たされている必要があります。
ACCESS_PATTERN_USED	一貫性のあるアクセスパターン。
REQUIRED_ACCESS_PATTERN_GROUPS	アクセスパターンを満たすために必要なグループ。結合計画で使用されます。

注記

関連付けられたアクセスノードに多くのソースプロパティーが存在します。

表10.9 グループプロパティー

プロパティー名	説明
GROUP_COLS	グループ化列。
ROLLUP	グループにロールが含まれる場合。

表10.10 タプルの制限プロパティー

プロパティー名	説明
MAX_TUPLE_LIMIT	生成されたタプルの最大数を評価する式。
OFFSET_TUPLE_COUNT	開始タプルのタプルオフセットを評価する式。
IS_IMPLICIT_LIMIT	制限がリライトャーによってサブクエリーの最適化の一環として作成される場合。
IS_NON_STRICT	順序のない制限を厳格に強制することはできません。

表10.11 一般および費用のプロパティー

プロパティー名	説明
OUTPUT_COLS	ノードの出力コラム。通常、ルールは出力要素を割り当てた後に設定されます。
EST_SET_SIZE	依存する結合シナリオにおいて、このノードが独立したノードとして生成する、予想されるセットサイズを表します。
EST_DEP_CARDINALITY	依存する結合シナリオの依存するノードとして、このノードで生成した予測されたカーディナリティー（行をマウントする）を表す値。
EST_DEP_JOIN_COST	依存する参加の推定コスト（この参加ストラテジーは Nested Loop または Merge）を表す値。
EST_JOIN_COST	マージ参加の推定コスト（この参加ストラテジーは Nested Loop または Merge）を表す値。
EST_CARDINALITY	このノードで生成した予測されたカーディナリティー（行のマウント）を表します。
EST_COL_STATS	null 値の数、一意の値数などを含む列の統計。

プロパティー名	説明
EST_SELECTIVITY	条件ノードの選択を表します。

ルール

リレーショナルデータベースの最適化は、最初のプランを実行プランに進化させるルール実行に基づいています。すべてのクエリーについてルールスタックに動的にアセンブルされる事前定義済みのルールのセットがあります。 ルールスタックは、ユーザーのクエリーの内容と、アクセスされたビュー/手順に基づいてアセンブルされます。 たとえば、ビュー層がない場合、ビュー層をマージするルール Merge Virtual は必要なく、スタックに追加されません。 これにより、ルールスタックはクエリーの複雑さを反映します。

プランノードのデータ構造は、リーフノード（通常は最終プランのノードへのアクセス）からソースデータが起動するノードのツリーを表します。これは、ツリーを通過してユーザーの結果を生成します。 プラン構造のノードには双方向のリンク、動的プロパティーを持たせることができ、任意の数の子ノードが許可されます。 通常、処理計画には固定プロパティーがあります。

プランルールは、プランツリーを操作して他のルールを実行し、最適化プロセスをアクティブ化します。各ルールは、限定されたタスクセットを実行するように設計されています。一部のルールは複数回実行できます。ルールによっては、正しく実行するのに特定のプリcursors のセットが必要です。

- *アクセスパターンに、すべてのアクセスパターンが満たされていることを確認できます。*
- *Securitygitops-gitopsApplies 行および列レベルのセキュリティーを適用します。*
- *このルールは、すべてのノードの上方向にスクロールして、各ノードの出力コラムを計算します。 不要な列は、すべてのプロジェクトでドロップされます。これは、展開が最小限に抑えられます。 これは、親ノードのフィードに必要な列の両方を追跡し、特定ノードで「作成された」列を追跡することで行われます。*
- *コストの計算： 計画へのコスト情報の追加*
- *このルールは各結合ノードを確認し、結合が依存すべきかどうか、またどの方向にするかを決定します。カーディナリティー、個別の値の数、およびプライマリーキー情報は、複数の式で使用され、依存する参加が悪いかどうかを判断します。依存する結合は、依存する側から返される値の数が少ないため、パフォーマンスに異なります。*

  また、独立した値から依存関係にある値の数も考慮する必要があります。セットが依存側の

IN 条件の値の最大数よりも大きい場合は、クエリーをクエリーセットに分割し、それらの結果を組み合わせる必要があります。 コネクターの各クエリーの実行にはオーバーヘッドがあり、考慮されます。情報を縮小しないと、指定された基準が一意でない（ただし非常に制限あり）上にある多くの一般的なケースが見なされます。

結合は、以下の条件に依存できます。

- *tablea.col = tableb.colなど、最低 1 つの結合条件がある。*
- *結合は完全な外部参加ではなく、参加に依存するのは参加の内側の側にあります。*

結合は、以下の条件のいずれかが優先順位に一覧表示されているかどうかによって変わります。

- *依存する参加条件で満たすことができる、満たされていないアクセスパターンがあります。*
- *結合の潜在的な依存関係には、makedep オプションが付いています。*
- *(4.3.2)コストが有効な場合には、依存する参加の推定コスト（内部参加の場合の各方向の推定コスト）が計算され、依存する参加が行われないことと比較されます。コストがすべて決定されると（関連するすべてのテーブルカーディナリティ、列のアドビ、および場合によってはnnv の値）最も低いものが選択されます。*
- *キーのメタデータ情報が、潜在的な依存側が「small」ではなく、もう一方の側が「小さすぎる」、または(5.0.1)場合、依存するサイドが左側で出入りすることを意味します。*

依存する結合は、マルチソース参加を効率的に処理するために使用する主要な最適化です。ソース A およびすべてのソース B を読み取り、A.x = B.x に参加するのではなく、すべての A を読み込んでから、B のクエリー時に条件として渡される A.x のセットを構築します。A が小型の、B が大きい場合、これは B から取得したデータを大幅に減らすことができるため、全体的なクエリーを大幅にスピードアップします。

- *Join Strategyで参加ストラテジーの選択を、参加コストおよび属性に基づいて選択してください。*

- *clean criteriagitops-TEMPLATESRemoves phantom 条件。*

- *Sourcegitops-gitops を折りたたむと、アクセスノードの下のすべてのノードが取得され、SQL クエリー表現が作成されます。*

- *このルールは、結合の基準に存在する等価基準よりも条件をコピーします。 等価性は同等のものを定義するため、参加の一方の側で結果を制限する新しい基準を作成するのに有効な方法です（特に、マルチソース参加の場合）。*

- *このルールは、パーティションが分割された結合の結合に対して、パーティション単位で最適化を実行します。詳細は、『Federated optimizations』の「Partitioned unions」を参照してください。補正する決定は、結合の各側がパーティション化していることを検知することに基づいています（ANSI 以外の結合により、2 つ以上のテーブルが結合すると最適化が適切な参加が検出されない可能性があります）。現在、ルールは、各側から最大 1 つのパーティションが一致する状況のみを検索します。*

- *選択した結合ストラテジーを処理するために必要なソートおよびその他のノードの実装*

- *Select nodes（Lerginer ctCombines が選択したノードをマージ）*

- *仮想マシンのビュー層とインラインビュー層のマージ*

- *ソースノード下のノードにアクセスするノードへのアクセス先へのアクセスを配置します。アクセスノードは、アクセスノード下のすべてがソースにプッシュされるか、またはプランの呼び出しである時点を表します。それ以降のルールは、アクセス下にプッシュするか、またはツリーを介してアクセスノードをプルして、ソースにより多くの作業を下に移動します。 このルールは、アクセスパターンの配置も行います。詳細は、「Federated optimizations」の「Access pattern in Federated optimizations」 を参照してください。*

- *参加の計画を立て、このメソッドは、アクセスパターンの依存関係を満たしながら、プランで実行される結合の最適な順序を見つけようとします。このルールには、3 つの主要な手順があります。*

  1. *アクセスパターンの条件を満たす参加の順序を決定する必要があります。*

  2. *ソースにプッシュできる結合をヒューリスティックに作成します（結合セットがソー*

スにプッシュされている場合は、そのセット内で最適な順序を作成できなくなります）。 ANSI 以外のマルチ結合構文でソースに送信され、データベースによって最適化される可能性が高くなります。

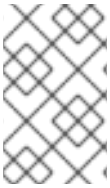
3. *コスト情報を使用し、処理エンジンで実行される結合の最適な順序を決定します。この 3 番目のステップは、結合ソース 7 以下の完全な検索を実行し、8 以上のソースについてオブジェクリティーに参加することでヒューリスティックに実行されます。*

- *プッシュダウンを改善できるように許可されているように、Outer Joinsで結合の計画を立てます。*
- *プランニング手順： 段階的なリレーショナルデータベースに表示される手順*
- *ソート操作や移動プロジェクトの組み合わせなど、ソート関連の並べ替えを計画する。*
- *Data Virtualization 12 用に SubqueriesTEMPLATES-gitopsNew を計画します。Merge criteria で実行されたサブクエリーの最適化を一般化し、展開とフィルタリングの両方でサブクィジションから参加計画を作成できるようにします。*
- *プランの Unions Experience-gitopsReorders Unorders union children for more pushdown.*
- *Plan Aggregatesgitops-gitopsPerforms aggregate decomposition over a join or union.*
- *制限制限をプッシュ - プランにさらに制限ノードの影響を軽減します。*
- *参加以外の条件をプッシュすると、このルールで結合の正確性が必要なければ、on 句から述語がプッシュされます。*
- *Select Violation-gitopsPush をプッシュすると、アクセスノードへの移動、参加、およびビューの層を使用して、できるだけ多くのノードを選択します。 ほとんどの場合、ツリーを下方向に移動すると、プランの前の行がフィルタリングされるためです。現在、プッシュ選択基準により行われる決定は取り消せません。 ただし、基準をソースで評価できない場合には、サブ最適なプランが発生する可能性があります。*
-

トランスレーターよりも大きな述語を Large IN TEMPLATES-TEMPLATESPush IN 述語をプッシュすると、直接依存セットとして処理できます。

マージ基準に関連する最も重要な最適化の 1 つは、条件を結合してプッシュする方法です。 クエリーのプランのサブツリーを表す次のプランツリーについて考えてみましょう。B. y = 3 は A inner join b on(A.x = B.x)から選択してください。

```
SELECT (B.y = 3)
    |
  JOIN - Inner Join on (A.x = B.x)
  /   \
SRC (A)  SRC (B)
```



注記

SELECT ノードは基準を表し、SRC は SOURCE を表します。

これは常に内部参加で有効であり、結合の上層にある結合（単一ソース）の条件を結合します。 これにより、ユーザークエリーの発信基準が最終的に結合の下にあるソースクエリーに存在することができます。 この結果は以下のように視覚的に表示できます。

```
JOIN - Inner Join on (A.x = B.x)
   /   \
  /  SELECT (B.y = 3)
 |       |
SRC (A)  SRC (B)
```

外側の出所に対して指定された基準に同じ最適化が有効です。 以下に例を示します。

```
SELECT (B.y = 3)
    |
  JOIN - Right Outer Join on (A.x = B.x)
  /   \
SRC (A)  SRC (B)
```

become

```
  JOIN - Right Outer Join on (A.x = B.x)
  /   \
 /  SELECT (B.y = 3)
|       |
SRC (A)  SRC (B)
```

ただし、外部参加の内側の内側に指定された基準は、特別な考慮が必要です。 上記のシナリオは、左または完全の外部参加が同じではありません。以下に例を示します。

```
SELECT (B.y = 3)
  |
 JOIN - Left Outer Join on (A.x = B.x)
/   \
SRC (A)  SRC (B)
```

become 可能（5.0.2 の直後に利用可能）。

```
JOIN - Inner Join on (A.x = B.x)
   /   \
  /  SELECT (B.y = 3)
 |      |
SRC (A)  SRC (B)
```

条件には、結合の内側の側から生成される可能性のある null 値に依存していないため、結合タイプも内部参加に変更されている場合のみ、結合結合のすぐ下にプッシュできます。一方、CAN not be move の null 値の有無に依存する基準。以下に例を示します。

```
SELECT (B.y is null)
   |
  JOIN - Left Outer Join on (A.x = B.x)
 /   \
SRC (A)  SRC (B)
```

前述のプランツリーの条件は結合の上に留まる必要があり、外側の結合は null 値自体を導入している可能性があります。

- プランが少なくなると、アクセスノードの取得が試行され、アクセスが試行されます。これは主に、ソースの機能を確認し、操作がソースで達成できるかどうかを判断することで行われます。

- Nullgitops-gitopsRaises null ノードを発生させます。null ノードを設定すると、null ノードよりも古いプランの一部を考慮する必要があります。

- オプション参加 - JoinsTEMPLATESRemoves は、オプションとしてマークが付けられるか、またはオプションとして決定されている参加を結合します。

- 関数ベースのインデックスが存在する場合にのみ、Expressions xmvn-TEMPLATESUsed を置き換えます。

- ソースで必要な場合に Where Allgitops-TEMPLATESEnsures 条件が使用されることを確認します。

コスト計算

ノード操作のコストは、主に、処理する行数（カンディナリティーとも呼ばれる）の数によって決定されます。オプティマイザーは通常、計画の中で最大数点（またはサブプラン）からカードを計算し、通常、ルールの計算コストを計算し、1 つのルールで計画計画やその他の決定を行う場合にとくに参加します。コストの計算は、主に物理テーブルに設定された統計（カーディナリティー、NNV、NDV など）によって転送され、制約の存在（unique、プライマリーキー、インデックスなど）にも影響を与えます。サブ最適なプランが選択されているような状況が選択された場合は、最初に、少なくとも代表的なテーブルのカーディナリティーが関係する物理テーブルに設定されていることを確認してください。

デバッグプランの読み取り

各リレーショナルデータベースサブプランが最適化されると、プランで最適化されている内容と正規の形式が表示されます。

```
OPTIMIZE:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

------------------------------------------------------------------------
GENERATE CANONICAL:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

CANONICAL PLAN:
Project(groups=[x], props={PROJECT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP=
{x.e1=e1}})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1]})
      Source(groups=[pm1.g1])
```

手順呼び出しやサブキューを含むクエリーなど、より複雑なユーザークエリーでは、サブ計画が全体的なプラン内で入れ子になる可能性があります。各プランは、最終的な処理計画を表示して終了します。

```
------------------------------------------------------------------------
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
AccessNode(0) output=[e1] SELECT g_0.e1 FROM pm1.g1 AS g_0
```

ルールの影響は、ルールが実行される前後にプランツリーの状態で確認できます。たとえば、以下のデバッグログは、ルールのマージ virtual の適用を示しています。これにより、"x" inline view レイヤーが削除されます。

```
EXECUTING AssignOutputElements

AFTER:
Project(groups=[x], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1, SYMBOL_MAP=
{x.e1=e1}, OUTPUT_COLS=[e1]})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
      Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
        Source(groups=[pm1.g1], props={OUTPUT_COLS=[e1]})


============================================================================
EXECUTING MergeVirtual

AFTER:
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
    Source(groups=[pm1.g1])
```

いくつかの重要な計画の決定が、アノテーションとして実施される際にプランに表示されます。たとえば、以下のコードスニペットは、親 SELECT ノードにサポートされないサブクエリーが含まれるため、アクセスノードが表示されないことを示しています。

```
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=null})
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1
FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1], props={OUTPUT_COLS=null})


============================================================================
EXECUTING RaiseAccess
LOW Relational Planner SubqueryIn is not supported by source pm1 - e1 IN /*+ NO_UNNEST */
(SELECT e1 FROM pm2.g1) was not pushed

AFTER:
Project(groups=[pm1.g1])
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */ (SELECT e1
FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema name=pm1,
nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1])
```

手順プランナー

手順プランナーはかなりシンプルです。この手順のステートメントを、処理中に実行されるプログラム内の命令に変換します。これはほとんど 1 対 1 のマッピングで、最適化はほとんど行われません。

XQuery

XQuery は、特定の最適化の対象となります。詳細は「XQuery optimization」を参照してください。ドキュメントの展開は、最も一般的な最適化です。これは、デバッグプランにアノテーションとして表示されます。たとえば、ドキュメント列 x 文字列パス '@x' を渡して "xmltable（'/a/b' が含まれるユーザークエリーでは、有効な文字列パス '.'）" を指定すると、デバッグ計画には、コンテキストおよびパス XQuery によって効果的に使用されるドキュメントのツリーが表示されます。

```
MEDIUM XQuery Planning Projection conditions met for /a/b - Document projection will be
used
child element(Q{}a)
  child element(Q{}b)
    attribute attribute(Q{}x)
      child text()
    child text()
```

10.3. クエリープラン

フェデレーションされたクエリープランナーを使用して情報を統合する際に、クエリー計画を表示して情報にアクセスおよび処理される方法をさらに理解し、問題のトラブルシューティングを行うと便利です。

クエリー計画（実行または処理計画としても知られる）は、ユーザーまたはアプリケーションが送信するコマンドを実行するためにクエリーエンジンによって作成される一連の命令です。クエリー計画の目的は、可能な限り効率的な方法でユーザーのクエリーを実行することです。

クエリー計画の取得

コマンドの実行時には、クエリー計画を取得できます。以下の SQL オプションを使用できます。

SET SHOWPLAN [ON|DEBUG]:処理計画またはプランおよび完全なプランナーのデバッグログを返します。詳細は、『Client Developer's Guide』の「Query planner and SET statement」を参照してください。上記のオプションを使用すると、org.teiid.jdbc.TeiidStatement インターフェースにキャストするか、または SHOW PLAN ステートメントを使用して、クエリー計画が Statement オブジェクトから利用できます。詳細は、『Client Developer's Guide』の「SHOW Statement」を参照してください。または、EXPLAIN ステートメントを使用できます。詳細は、「Explain statement」を参照してください。

Data Virtualization 拡張機能を使用したクエリー計画の取得

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
Data VirtualizationStatement tstatement = statement.unwrap(TeiidStatement.class);
PlanNode queryPlan = tstatement.getPlanDescription();
System.out.println(queryPlan);
```

ステートメントを使用したクエリー計画の取得

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
...
ResultSet planRs = statement.executeQuery("show plan");
planRs.next();
System.out.println(planRs.getString("PLAN_XML"));
```

explain を使用したクエリー計画の取得

```
ResultSet rs = statement.executeQuery("explain select ...");
System.out.println(rs.getString("QUERY PLAN"));
```

クエリー計画は、複数の Data Virtualization のツールで自動的に利用可能になります。

クエリー計画の分析

クエリー計画を取得したら、以下の項目について確認できます。

- source pushdown xmvn-gitops 各ソースにプッシュされたクエリーの部分

- *特にインデックスに対して述語がプッシュされていることを確認します。*

- *joins-1.6.0--で合成された結合は、非常にコストのかかる可能性があります。*
  - *join orderingfaillock-gitopsTypical impacting by costing*
  - *条件タイプの不一致に参加します。*
  - *join アルゴリズムを使用して、マージやネストされたループの強化などを行いました。*

- *依存する参加などのフェデレーションされた最適化があること。*

- *ヒントに、必要な影響があることを確認します。ヒントの使用に関する詳細は、以下の追加リソースを参照してください。*
  - キャッシングガイド のヒントとオプション
  - FROM 句の FROM 句 ヒント
  - *サブクエリーの最適化。*
  - *フェデレーション最適化*

処理計画から前述のリスト内のすべての情報を確認できます。通常、最終処理計画のテキスト形式の分析に関心があります。デバッグまたはサポートに対して特定の決定が行われる理由を理解するには、中間計画ステップを含む完全なデバッグログと、特定のプッシュダウン決定が行われる理由へのアノテーションを取得します。

クエリー計画は、ツリー構造で整理されたノードのセットで構成されます。手順を実行している場合、クエリー計画全体には、周りの手順の実行に関連する追加情報が含まれます。

手順のコンテキストでは、子ノードの順序は実行の順序を意味します。ほとんどの場合、子ノードは並行しても任意の順序で実行できます。依存結合などの特定の最適化では、結合の子が順次実行されます。

リレーショナルデータベース計画

リレーショナルデータベースは、論理関係操作のビルディングブロックを表すノードで構成される処理計画を表します。リレーショナルデータベース処理計画には、追加の操作とオプティマイザーで選択した実行の詳細が含まれるという点で、論理デバッグのリレーショナルデータベースとは異なります。

リレーショナルデータベースクエリー計画のノードは以下のとおりです。

- accessで source にアクセスします。ソースクエリーは、ソースに関連付けられた接続ファクトリーに送信されます。（依存する結合の場合、このノードは Dependent Access と呼ばれます。）
- 依存する手順は、複数の入力値のセットを使用してソースのストアドプロシージャーにアクセスします。
- バッチの更新で、更新のセットをバッチとして処理します。
- projectgitops-で、ノードから返された列をDefinines にします。返されたレコード数は変更しません。
- プロジェクトを通常のプロジェクトと同様ですが、ターゲットテーブルに行を出力します。
- プロジェクトにプランの実行で完了し、ソースクエリーではなくプランを実行します。通常、クエリー式を使用してビューに挿入するときに作成されます。
- window function project336-TEMPLATESLike は通常のプロジェクトで使用されますが、ウィンドウ機能が含まれます。
- pidgin-TEMPLATESSelect は、基準評価フィルターノード(WHERE / HAVING)です。

- *jointypes - join タイプ、結合基準、結合ストラテジー（merge または nested loop）に参加します。*

- *このノードのプロパティーをすべて選択せずに、その子から行を渡します。トランザクションまたはソースクエリーの同時実行が許可されているかどうかなど、他の要素によっては、すべての結合子が並行して実行されるわけではありません。*

- *ソートする列を並べ替える - ソートする列、各列のソート方向、重複を削除するかどうかを指定します。*

- *dup removegitops-gitopsRemoves の重複行。処理はツリー構造を使用して重複を検出し、結果を IO 操作のコストが効率的にストリーミングできるようにします。*

- *一連の行のセットをグループにグループ化し、集約関数を評価します。*

- *行が生成されない null rhncfg-で、AnbileA ノード。通常は、基準が常に false（および下のツリー）である Select ノードを置き換えます。このノードのプロパティーはありません。*

- *プランの実行 - 別のサブプランを実行します。通常、サブ計画は関係しないプランです。*

- *依存する手順の実行 - 複数の入力値セットを使用してサブプランを実行します。*

- *制限制限 - 指定した数行数を書き込んでから、処理を停止します。また、存在する場合はオフセットも処理します。*

- *XML table PROVISIONING-gitopsEvaluates XMLTABLE.デバッグ計画には、最適化に関して XQuery/XPath に関する詳細情報が含まれます。詳細は「XQuery optimization」を参照してください。*

- *テキストテーブル - Evaluates TEXTTABLE*

- *array table - Evaluates ARRAYTABLE*

- ***Object table - Evaluates OBJECTTABLE***

ノードの統計

すべてのノードには、出力される統計セットがあります。これらは、ノードを通過するデータの量を決定するために使用できます。プロセッサープランを実行する前には、ノードの統計は含まれません。また、統計はプランの処理時に更新されるので、通常はすべての行がクライアントによって処理された後に最終的な統計が必要です。

統計	説明	ユニット
ノード出力行	ノードからの出力のレコード数。	count
ノードの次のバッチプロセス時間	このノードの時間処理。	millisec
ノードの累積次のバッチプロセス時間	このノード + 子ノードにおける時間処理。	millisec
ノードの累積プロセス時間	処理の開始から終了までの経過時間。	millisec
ノードの次のバッチ呼び出し	ノードが処理のために呼び出された回数。	count
ノードブロック	このノードまたは子によってブロックされた例外がスローされた回数。	count

ノードの統計以外にも、一部のノードの表示コストにより、ノードで計算される予測が予測されます。

Cost Estimates	説明	ユニット
推定ノードのカーディナリティー	ノードから出力されるレコードの推定数。不明な場合は -1。	count

ルートノードが追加情報を表示します。

最上位の統計	説明	ユニット

最上位の統計	説明	ユニット
Data Bytes Sent	クライアントに送信されるシリアライズされたデータ結果（行と緩やかの値）のサイズ。	bytes

プロセッサープランの読み取り

クエリープロセッサープランは、プレーンテキストまたは XML 形式で取得できます。通常、プランのテキスト形式は読みやすくなりますが、XML 形式はツールによって処理が簡単です。ツリー構造は詳細にネストできるため、ツールを使ってプランを調べる必要があります。

ツリーのリーフからルートへのデータフロー。手順実行のサブプランはインラインで示され、インデントが異なります。pm1 .g1.e1=pm1.g3.e1、pm1.g2.e2、pm1.g3.e3 のユーザークエリーが pm1.g1 の内部参加（pm1.g2 left outer は pm1.g2.e1=pm1.g3.e1）の pm1.g1.e1=pm1.g3.e1 で、結合をプッシュしないプロセッサープランのテキストは次のようになります。

```
ProjectNode
  + Output Columns:
    0: e1 (string)
    1: e2 (integer)
    2: e3 (boolean)
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    JoinNode
      + Output Columns:
        0: e1 (string)
        1: e2 (integer)
        2: e3 (boolean)
      + Cost Estimates:Estimated Node Cardinality: -1.0
      + Child 0:
        JoinNode
          + Output Columns:
            0: e1 (string)
            1: e1 (string)
            2: e3 (boolean)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Child 0:
            AccessNode
              + Output Columns:e1 (string)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Child 1:
            AccessNode
              + Output Columns:
                0: e1 (string)
                1: e3 (boolean)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER BY c_0
```

```
            + Model Name:pm1
          + Join Strategy:MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
          + Join Type:INNER JOIN
          + Join Criteria:pm1.g1.e1=pm1.g3.e1
        + Child 1:
          AccessNode
          + Output Columns:
            0: e1 (string)
            1: e2 (integer)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Query:SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY c_0
          + Model Name:pm1
        + Join Strategy:ENHANCED SORT JOIN (SORT/ALREADY_SORTED)
        + Join Type:INNER JOIN
        + Join Criteria:pm1.g3.e1=pm1.g2.e1
      + Select Columns:
        0: pm1.g1.e1
        1: pm1.g2.e2
        2: pm1.g3.e3
```

ネストされた結合ノードはマージ結合を使用しており、各サイドのソースクエリーが結合の予想される順序を生成することを予想することに注意してください。親結合は強化されたソート結合で、受信行に基づいてソートの実行を遅らせることができます。null 内部値がクエリー結果に存在しないため、ユーザークエリーからの外部参加は内部参加に変更されています。

前述のプランは、以下の例のように XML 形式で表現することもできます。

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="ProjectNode">
   <property name="Output Columns">
      <value>e1 (string)</value>
      <value>e2 (integer)</value>
      <value>e3 (boolean)</value>
   </property>
   <property name="Cost Estimates">
      <value>Estimated Node Cardinality: -1.0</value>
   </property>
   <property name="Child 0">
      <node name="JoinNode">
         <property name="Output Columns">
            <value>e1 (string)</value>
            <value>e2 (integer)</value>
            <value>e3 (boolean)</value>
         </property>
         <property name="Cost Estimates">
            <value>Estimated Node Cardinality: -1.0</value>
         </property>
         <property name="Child 0">
            <node name="JoinNode">
               <property name="Output Columns">
                  <value>e1 (string)</value>
                  <value>e1 (string)</value>
```

```
            <value>e3 (boolean)</value>
        </property>
        <property name="Cost Estimates">
            <value>Estimated Node Cardinality: -1.0</value>
        </property>
        <property name="Child 0">
            <node name="AccessNode">
                <property name="Output Columns">
                    <value>e1 (string)</value>
                </property>
                <property name="Cost Estimates">
                    <value>Estimated Node Cardinality: -1.0</value>
                </property>
                <property name="Query">
                    <value>SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0</value>
                </property>
                <property name="Model Name">
                    <value>pm1</value>
                </property>
            </node>
        </property>
        <property name="Child 1">
            <node name="AccessNode">
                <property name="Output Columns">
                    <value>e1 (string)</value>
                    <value>e3 (boolean)</value>
                </property>
                <property name="Cost Estimates">
                    <value>Estimated Node Cardinality: -1.0</value>
                </property>
                <property name="Query">
                    <value>SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS g_0
                        ORDER BY c_0</value>
                </property>
                <property name="Model Name">
                    <value>pm1</value>
                </property>
            </node>
        </property>
        <property name="Join Strategy">
            <value>MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)</value>
        </property>
        <property name="Join Type">
            <value>INNER JOIN</value>
        </property>
        <property name="Join Criteria">
            <value>pm1.g1.e1=pm1.g3.e1</value>
        </property>
    </node>
</property>
<property name="Child 1">
    <node name="AccessNode">
        <property name="Output Columns">
            <value>e1 (string)</value>
            <value>e2 (integer)</value>
        </property>
```

```
            <property name="Cost Estimates">
               <value>Estimated Node Cardinality: -1.0</value>
            </property>
            <property name="Query">
               <value>SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0
                  ORDER BY c_0</value>
            </property>
            <property name="Model Name">
               <value>pm1</value>
            </property>
         </node>
      </property>
      <property name="Join Strategy">
         <value>ENHANCED SORT JOIN (SORT/ALREADY_SORTED)</value>
      </property>
      <property name="Join Type">
         <value>INNER JOIN</value>
      </property>
      <property name="Join Criteria">
         <value>pm1.g3.e1=pm1.g2.e1</value>
      </property>
   </node>
</property>
<property name="Select Columns">
   <value>pm1.g1.e1</value>
   <value>pm1.g2.e2</value>
   <value>pm1.g3.e3</value>
</property>
</node>
```

各プランのフォームに同じ情報が表示されます。場合によっては、デバッグ計画の最終プロセッサープランの簡略化された形式に従うのが簡単になります。以下の出力は、デバッグログが前述の XML 例のプランを表している方法を示しています。

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
ProjectNode(0) output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3] [pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
  JoinNode(1) [ENHANCED SORT JOIN (SORT/ALREADY_SORTED)] [INNER JOIN] criteria=
[pm1.g3.e1=pm1.g2.e1] output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
   JoinNode(2) [MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)] [INNER JOIN]
criteria=[pm1.g1.e1=pm1.g3.e1] output=[pm1.g3.e1, pm1.g1.e1, pm1.g3.e3]
     AccessNode(3) output=[pm1.g1.e1] SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER
BY c_0
     AccessNode(4) output=[pm1.g3.e1, pm1.g3.e3] SELECT g_0.e1 AS c_0, g_0.e3 AS c_1
FROM pm1.g3 AS g_0 ORDER BY c_0
   AccessNode(5) output=[pm1.g2.e1, pm1.g2.e2] SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM
pm1.g2 AS g_0 ORDER BY c_0
```

共通

- Output columns: このノードによって返されるタプルを構成する列。

- *送信されたデータバイト - メッセージングオーバーヘッドを含まないデータバイトの数がこのクエリーによって送信されました。*

- *プランニング時間： クエリーの計画に費やした時間（ミリ秒単位）。*

リレーショナル

- *リレーショナルノード ID336-TEMPLATES は、デバッグログ Node(id)に表示されるノード ID と一致します。*

- *基準 2: フィルタリングに使用するブール式です。*

- *Projection を定義する columns xmvn-gitopshe 列を選択します。*

- *グループ化に使用する列のグルーピング - グループ化に使用する列。*

- *マッピングのグルーピングにより、集約およびグルーピング列の内部名から式フォームへのマッピングが表示されます。*

- *querygitops-gitopsThe source query.*

- *モデル名で、モデル名を使用しない場合は、このモデル名が必要です。*

- *同じソース結果を共有する ID Warehouse を共有すると、同じ共有 ID が設定されます。*

- *依存結合結合が使用されている場合、依存する join が使用されています。*

- *参加ストラテジーにジョインストラテジー（Nested Loop、Sort Merge、Enhanced Sort など）に参加します。*

- *参加するタイプにジョインするタイプ（左の参加、Inner Join、Cross Join）*

- *参加する条件(join predicates)に参加します。*
- *実行プランでネストされた実行計画です。*
- *挿入ターゲットにターゲットを使用。*
- *挿入が upsert の場合は Upsert-1.6.1+If を挿入します。*
- *並び替え列で、ソート用の列を並び替えます。*
- *並べ替えモードにソートすると、ソートも、個別の削除のように別の機能を実行します。*
- *ロールアップオプションにより、グループに rollup オプションがあります。*
- ***statisticsfaillock-gitopsThe processing statistics.***
- *コスト予測 - 依存する結合コスト予測を含むコスト/カーディナリティー予測。*
- *行のオフセット式に、行のオフセット式があります。*
- *行の上限 - 行制限式。*
- *with 節で、with 節を使用します。*
- *ウィンドウ関数で計算されるウィンドウ関数。*
- *テーブル関数（XMLTABLE、OBJECTTABLE、TEXTTABLE など）*

- *streaming で XMLTABLE がストリーム処理を使用していること。*

手順

- *式*
- *結果セット*
- *program*
- *変数*
- *次に、*
- *その他*

その他のプラン

手順実行（トリガーを含む）は、リレーショナルデータベースを含む中間および最終計画のフォームを使用します。通常、XML/手順計画の構造は論理フォームに密接にマッチします。パフォーマンスの問題を分析する際に関連するネストされたリレーショナルデータベースプランです。

10.4. フェデレーション最適化

アクセスパターン

アクセスパターンは、物理テーブルとビューの両方で使用され、一連の列に対して基準の必要性を指定します。条件を指定できないと、run-away ソースクエリーではなく、プランニングエラーが発生します。アクセスパターンは、アクセスパターンの 1 つだけが満たされるようにセットに適用できます。

現在、影響を受ける列を参照する条件の形式が、アクセスパターンを満たしている場合があります。

pushdown

フェデレーションされたデータベースシステムのプッシュダウンとは、ユーザーレベルのクエリー

を各ソースシステムでできるだけ多くの作業を実行するソースクエリーに分割することを指します。プッシュダウン分析には、コネクター API の Data Virtualization で提供されるソースシステム機能に関する知識が必要です。ソースで実行されていない作業は、Federate のリレーショナルデータベースエンジンで処理されます。

機能に基づいて、Data Virtualization はクエリー計画を操作して、各ソースが可能な限り多くの参加、フィルタリング、グループ化などを実行するようにします。多くの場合、結合順序など、プランニングは 標準のリレーショナルデータベース技術 と費用情報、プッシュダウン最適化のためのヒューリスティックを組み合わせたものです。

通常、基準および結合プッシュダウンは、パフォーマンスが懸念される場合にプッシュするクエリーの最も重要な要素です。できるだけ効率的なソースクエリーを確保するためのプランの読み取り方法は、「 プランのクエリー」を参照し てください。

依存する参加

依存結合と呼ばれる特別な最適化は、マルチソース参加に関連する 2 つの関係のいずれかから返される行を減らすために使用されます。依存する結合では、クエリーは並列ではなく、各ソースに順番に発行されます。また、1 つ目のソースから取得された結果で、2 番目から返されたレコードを制限します。2 つ目のソースから取得されるデータ量と、実行する必要がある結合比較数を大幅に減らすことで、依存する結合結合をより速く実行できます。

依存する結合が使用される条件は、アクセスパターン、ヒント、および費用情報に基づいてクエリープランナーによって決定されます。以下のタイプの依存結合を Data Virtualization で使用できます。

等価性または非等価性をもとに結合

エンジンは、トランスレーター機能に基づいて大規模なクエリーを分割する方法を決定します。

キーのプッシュダウン

トランスレーターはキー値の完全なセットにアクセスし、送信するクエリーを決定します。

完全なプッシュダウン

トランスレーターは、独立した側からすべてのデータをトランスレーターに提供します。コストにより自動的に使用することも、ヒントのオプションとして指定できます。

Data Virtualization で以下のタイプのヒントを使用して、依存する結合動作を制御できます。

MAKEIND

句が依存する結合の独立側である必要があることを示します。

MAKEDEP

句が参加に依存することを示します。非ヒントとして、任意の MAX 引数および JOIN 引数を指定して MAKEDEP を使用できます。

MAKEDEP(JOIN)

つまり、結合全体をプッシュする必要があります。

MAKEDEP(MAX:5000)

つまり、独立した結合は、独立した側からの値の最大数より少ない場合にのみ実行する必要があります。

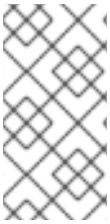
MAKENOTDEP

句が参加に依存することを防ぎます。

これらは OPTION 句 または FROM 句に直接配置できます。すべてのアクセスパターンが 満たされている限り、MAKEIND、MAKEDEP、および MAKENOTDEP のヒントは、コスト情報の使用を上書きします。MAKENOTDEP は、他のヒントの代わりに使用します。

ヒント

MAKEDEP または MAKEIND のヒントは、適切なクエリー計画がデフォルトで選択されていない場合にのみ使用してください。コスト情報が実際のソースカーディナリティーを表すことを確認する必要があります。



注記

不適切な MAKEDEP または MAKEIND ヒントにより、非効率的な結合構造を強制し、多くのソースクエリーが発生する可能性があります。

ヒント

これらのヒントをビューに適用することはできますが、オプティマイザーは可能な場合にデフォルトでビューを削除します。これにより、ヒント配置が元の意図と大きく異なる場合があります。オプティマイザーがこれらのケースでビューを削除しないようにするには、NO_UNNEST ヒントの使用を検討してください。

最も単純なシナリオでは、エンジンは IN 句（または等価述語）を使用して、依存する側からの値をフィルタリングします。独立した値の数が、Translators MaxInCriteriaSize を超える場合、値は MaxDependentPredicates までの複数の IN 述語に分割されます。独立した値の数が MaxInCriteriaSize*MaxDependentPredicates を超える場合、複数の依存クエリーが並行して発行されます。

トランスレーターが support DependentJoins に対して true を返す場合、エンジンは 独立したキーの値のセット全体を提供できます。これは、独立した値の数が MaxInCriteriaSize*MaxDependentPredicates を超える場合に生じ、トランスレーターは単純なシナリオで発生するように特定のロジックを使用して複数のクエリーを実行しないようにします。

トランスレーターがDependentJoins を サポートし、FullDependentJoins をサポート する場合、最適化機能によって完全なプッシュダウンを選択できます。これは、Data-ship のプッシュダウンとも呼ばれることもあります。ここでは、結合の独立した側からの全データが依存側に送信されます。これにより、結合の上にプランもプッシュダウンできるという利点があります。このため、オプティマイザーは、独立した値の数が MaxInCriteriaSize*MaxDependentPredicates を超えていない場合でも完全なプッシュダウンの実行を選択できます。MAKEDEP(JOIN) ヒントを使用して完全なプッシュダウンを強制的に実行することもできます。トランスレーターは通常、独立側を表す一時的なテーブルを作成し、設定し、削除します。依存関係、キー、および完全なプッシュダウンでカスタム変換を使用する方法の詳細は、「Translator Development の Dependent join pushdown」を参照してください。注記： デフォルトでは、Key/Full Pushdown は JDBC 翻訳者のサブセットとのみ互換性があります。これを使用するには、translator のオーバーライドプロパティー enableDependentJoins を true に設定します。JDBC ソースは、通常 Hibernate 方言を必要とする一時テーブルの作成を許可する必要があります。以下の変換は、DB2、Derby、H2、Hana、HSQL、MySQL、Oracle、PostgreSQL、SQL Server、SAP IQ、Sybase、Teiid、および Teradata と互換性があります。

条件のコピー

コピー基準は、結合基準と where 句の条件を組み合わせて、追加の述語を作成する最適化です。たとえば、equi-join 述語 (source1.table.column = source2.table.column) は、source2.table.column に source1.table.column の置き換えによって新規述語を作成するために使用されます。ソース間のシナリオでは、結合の単一部分に適用される基準を両方のソースクエリーに適用することができます。

Projection minimizeation

Data Virtualization は、各プッシュダウンクエリーがユーザークエリーの処理に必要なシンボルのみを提供するようにします。これは、大規模な中間ビュー層でクエリーを行う場合に役立ちます。

部分的な集約プッシュダウン

部分的な集約プッシュダウンにより、複数のソース結合および意図上のグループ化操作が可能になり、グループ化および集約関数の一部をソースにプッシュできます。

任意の参加

任意または冗長な結合は、オプティマイザーによって削除できるものです。オプティマイザーは外部キーに基づいて内部参加を自動的に削除し、外部の結果が一意であれば、出発元の結合を自動的に削除します。

オプションの結合ヒントは自動ケース以外で、ユーザークエリーの出力で列がない場合に結合されたテーブルを省略するか、ユーザークエリーの結果を構築する意味のある方法で省略する必要があることを示します。このヒントは通常、マルチソース参加が含まれるビュー層で使用されます。

オプションの結合ヒントは join 句のコメントとして適用されます。これは、ANSI と非ANSI の結合の両方で適用できます。非ANSI が結合したテーブル全体に参加する場合には、オプションとしてマークされます。

例： オプションの結合ヒント

```
select a.column1, b.column2 from a, /*+ optional */ b WHERE a.key = b.key
```

この例では、ビューレイヤー X を定義しています。X が b.column2 を必要としないようにクエリーされる場合、オプションの結合ヒントはクエリー計画から b が省略されます。X が以下のように定義された場合と同じ結果になります。

例： オプションの結合ヒント

```
select a.column1 from a
```

例： ANSI オプションの結合ヒント

```
select a.column1, b.column2, c.column3 from /*+ optional */ (a inner join b ON a.key = b.key)
INNER JOIN c ON a.key = c.key
```

この例では、ANSI join 構文により、ab と b の結合はオプションとしてマークされます。この例では、ビューレイヤー X を定義しています。列 a.column1 と b.column2 の両方が必要ない場合のみです（例： SELECT column3 FROM X は結合を削除します）。

オプションの結合ヒントは、必要なブリッジテーブルを削除しません。

例： ブランディングテーブル

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c WHERE ON a.key = b.key
AND a.key = c.key
```

この例では、ビューレイヤー X を定義しています。b.column2 または c.column3 が X へのクエリーによってのみ必要とされる場合、結合は削除されます。ただし、a.column1 または b.column2 と c.column3 の両方が必要な場合は、オプションの join ヒントは反映されません。

任意の結合ヒントを使用して join 句を省略すると、関連する基準は適用されません。そのため、クエリー結果に結合が完全に適用されたときと同じカーディナリティーや、同じ行の値を使用できないことがあります。

内部サイドの値が使用されておらず、別の操作を行う行が自動的に任意の結合として扱われ、ヒントは必要ありません。

例： 不要なオプションの結合ヒント

```
select distinct a.column1 from a LEFT OUTER JOIN /*+optional*/ b ON a.key = b.key
```

注記

すべての結合テーブルがオプションとしてマークされているビューに対する単純な "SELECT COUNT(*)FROM VIEW" は意味のある結果を返しません。

ソースヒント

Data Virtualization ユーザーと変換クエリーには、ソースクエリーに追加情報を提供できるメタソースのヒントを含めることができます。ソースヒントの形式は以下のとおりです。

```
/*+ sh[[ KEEP ALIASES]:'arg'] source-name[ KEEP ALIASES]:'arg1' ... */
```

- ソースヒントは、クエリー（SELECT、INSERT、UPDATE、DELETE）のキーワードの後に表示されることが想定されます。
- ソースヒントは、サブクエリーまたはビューに表示できます。指定のソースクエリーに該当するヒントはすべて収集され、一覧としてプッシュされます。ヒントの順序は保証されません。
- sh arg はオプションで、ExecutionContext.getGeneralHints メソッドを使用してすべてのソースクエリーに渡されます。追加の引数には、VDB のトランスレーターに割り当てられたソース名と一致する source-name がなければなりません。source-name が一致すると、ヒントの値は ExecutionContext.getSourceHints メソッド経由で指定されます。ExecutionContext の使用方法は、「Translator Development」を参照してください。
- 引数の値はそれぞれ文字列リテラルの形式を持ちます。これは一重引用符で囲む必要があり、一重引用符は別の一重引用符でエスケープできます。Oracle トランスレーターのみがソースヒントを使用します。Oracle translator は、ソースヒントと一般的なヒント（この順番で）Oracle ヒントを /*+ ... */ で囲まれた Oracle ヒントを形成するのに使用できます。
- KEEP ALIASES オプションが一般的なヒントまたは適用可能なソース固有のヒントのいずれかに使用される場合、ユーザークエリーの表/ビューのエイリアスとネストされたビューはプッシュダウンクエリーで保持されます。これは、ソースヒントがエイリアスを参照する必要があり、ユーザーが生成されたエイリアスに依存したくない場合に便利です（上記のソースクエリーでクエリー計画で確認可能）。ただし、場合によっては、保持されたエイリアス名がソースに対して有効ではない場合や、名前が競合してしまうと、無効なソースクエリーが発生する可能性があります。KEEP ALIASES の使用でエラーが発生した場合には、NO_UNNEST ヒント、エイリアスの変更、または KEEP ALIASES オプションを使用したビューの削除を防ぎ、生成されたエイリアス名を決定するために使用するクエリー計画によってクエリーを変更することができます。

ソースヒントのサンプル

```
SELECT /*+ sh:'general hint' */ ...

SELECT /*+ sh KEEP ALIASES:'general hint' my-oracle:'oracle hint' */ ...
```

パーティション設定解除

変換/インラインビューからは、ユニオンパーティションが推測されます。UNION 列の 1 つ（または複数）が定数によって定義され、または各ブランチを相互に排他的にする定数のみを含む WHERE 句 IN 述語がある場合、UNION はパーティション化されたものとみなされます。UNION ALL を使用し、UNION には LIMIT、WITH、または ORDER BY 句を含めることはできません（個別のブランチは LIMIT、WITH、または ORDER BY を使用できます）。パーティショニング値は null にしないでください。

例： パーティション設定解除

```
create view part as select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)
```

このビューは列 x でパーティション分割されます。1 番目のブランチは値 1 のみで、2 番目のブランチは 2 または 3 の値のみになります。

注記

今後のリリースでは、より高度なパーティションまたは明示的なパーティション設定が考慮されます。

パーティションが分割解除の概念は、パーティション単位で参加、アップデータブルビュー、および Partial Aggregate Pushdown に使用されます。これらの最適化は、マルチソース機能を使用する場合も適用され、明示的なパーティション設定列が導入されました。

パーティション単位で結合すると、計画が結合し、一致するパーティションのみが相互に結合する状態になるように、プランを結合します。パーティション単位で分割された列の明示的な結合を必要とせずに暗黙の結合を行うには、モデル/スキーマプロパティー implicit_partition.columnName を、モデル/スキーマの各パーティション設定されたビューで使用するパーティション列の名前に設定します。

```
CREATE VIRTUAL SCHEMA all_customers SERVER server OPTIONS
("implicit_partition.columnName" 'theColumn');
```

標準のリレーショナルデータベース技術

また、データ仮想化には、効率的なクエリー計画を確保するために、多くの標準的なリレーショナルデータベース技術が組み込まれています。

- 関数の簡素化および評価のための書き換え分析。
- 基本的な基準の簡素化のためのブール値の最適化。
- 不要なビュー層の削除。
- 不要なソート操作の削除。
- 結合ツリーの左線空間を使用した高度な検索手法。
- 実行時のソースアクセスの反復。
- サブクエリーの最適化。

join compensation

ソースシステムによっては、「relationship」クエリーは論理的に除外された結合の結果のみを許可します。内部の参加でクエリーした場合でも、Data Virtualization は適切なオ外側参加の形成を試みます。これらのソースは、キー参加での使用に制限されます。状況によっては、同じソースの外部キー関係を、結合の外側上または参照されたテーブルでトラバートしないようにしてください。拡張プロパティー teiid_rel:allow-join を外部キーで使用することで、プッシュダウンの動作をさらに制限することができます。値が「false」の場合には、結合のプッシュダウンは許可されません。値が「inner」の場合には、参照テーブルは参加の内部側にある必要があります。結合のプッシュダウンが阻止されると、結合はフェデレーションされた結合として処理されます。

10.5. サブクエリーの最適化

- EXISTS サブクラシングは通常、SELECT 式の不要な評価を防ぐために「SELECT 1 FROM ...」に書き換えられます。
- 定量化された比較 SOME サブクナップは常に同等の IN 述語または集約値と比較になります。たとえば、col > SOME（テーブルから col1 を選択）は照合されます（表から min(col1)

を選択します）。

- ソースにプッシュされていない Uncorrelated EXISTs および scalar サブクエリーは、ソースコマンドの形式よりも前に事前評価できます。

- 対応する DELETE/UPDATE の一部としてプッシュされていない DETELE または UPDATE で使用される相関サブキューにより、Data Virtualization が行ごとに補正処理を実行します。

- マージ結合(MJ)のヒントは、可能な限り従来の半結合またはアンチフォール結合結合を使用するようにオプティマイザーに指示します。依存結合(DJ)は MJ ヒントと同じですが、さらにオプティマイザーに対して、依存する結合の独立側としてサブクエリーを使用するようにオプティマイザーに指示します。これは、影響を受けるテーブルにプライマリーキーがある場合にのみ発生します。そうでない場合は、例外が発生します。

- WHERE または HAVING 句 IN、Qantified Comparison、Scalar Subquery Compare、および EXIST の述語は、サブクエリーの直前に表示される MJ、DJ、または NO_UNNEST（nest なし）のヒントを取ることができます。NO_UNNEST ヒントは他のヒントに優先して、サブクエリーを残すようにオプティマイザーに指示します。

- SELECT スカラーサブキューは、サブクエリーの直前に表示される MJ または NO_UNNEST のヒントを取ることができます。MJ ヒントでは、可能な場合は従来の結合または半結合結合を使用するようにオプティマイザーに指示します。NO_UNNEST ヒントは他のヒントに優先して、サブクエリーを残すようにオプティマイザーに指示します。

結合のヒントの使用をマージ

```
SELECT col1 from tbl where col2 IN /*+ MJ*/ (SELECT col1 FROM tbl2)
```

依存する結合ヒントの使用

```
SELECT col1 from tbl where col2 IN /*+ DJ */ (SELECT col1 FROM tbl2)
```

最も簡単なヒントはありません。

```
SELECT col1 from tbl where col2 IN /*+ NO_UNNEST */ (SELECT col1 FROM tbl2)
```

- システムプロパティー org.teiid.subqueryUnnestDefault は、書き換え中にオプティマイザーがデフォルトでフォレスト外のサブクィジーを設定するかどうかを制御します。true の場合、多くの否定されていない WHERE または HAVING 句 EXISTS または IN サブクエリー述語を従来の結合に変換できます。
- プランナーは、費用が優先される場合、常にアンチ参加または半結合のバリアントに変換されます。ヒント を使用して、必要な動作を上書きします。
- プッシュされず、マージ結合に変換されないスケーラーサブクリーピングが存在すると、暗黙的に 1 と 2 の行にはそれぞれ制限で制限されます。
- ネストされたループ参加へのサブクエリー述語の変換はまだ利用できません。

10.6. XQUERY の最適化

ドキュメント展開と呼ばれる手法は、コンテキスト項目ドキュメントのメモリーフットプリントを削減するために使用されます。ドキュメントの展開は、関連する XQuery およびパス式で必要なドキュメントの一部のみを読み込みます。ドキュメントプロジェクト分析は関連するすべてのパス式を使用するため、/a/ b/x ではなく //x など、多数のノードを使用できる可能性のある 1 つの式でもメモリーフットプリントが大きくなります。関連するコンテンツが削除された場合でも、処理のためにドキュメント全体がメモリーに読み込まれます。ドキュメントの展開は、XMLTABLE/XMLQUERY に渡されるコンテキストアイテム（名前のない PASSING 句項目）がある場合にのみ使用されます。名前付き変数には、ドキュメントの展開は実行されません。オプティマイザーがドキュメントの展開を使用するために、使用する式が複雑すぎることがあります。適切な最適化が実行されたかどうかを確認するには、SHOWPLAN DEBUG の完全なプランの出力を確認してください。

追加の制限により、Simple コンテキストパス式を使用すると、メモリーの完全なドキュメントをロードせずに、プロセッサーがドキュメントサブツリーを個別に評価できます。単純なコンテキストパス式の形式は [/][ns:]root/[ns1:]elem/...' の形式にすることができます。ここで、namespace プレフィックスまたは要素名は *ワイルドカードにすることもできます。XQuery 式で名前空間接頭辞が使用されている場合の通常の XQuery 処理と同様に、XMLNAMESPACES 句を使用して宣言する必要があります。

適格な XMLQUERY のストリーミング

> *XMLQUERY('/*:root/*:child' PASSING doc)*

ドキュメントインメモリー全体を DOM ツリーとしてロードするのではなく、各子要素は結果を別個に追加されます。

受信可能な XMLQUERY のストリーミング

> *XMLQUERY('//child' PASSING doc)*

子軸を使用するとストリーミングの最適化は回避されますが、ドキュメントの展開は引き続き実行できます。

XMLTABLE を使用する場合は、COLUMN PATH 式に追加の制限があります。これらはコンテキスト式によって形成された要素サブツリーの任意の部分を参照することができ、直接親から任意の属性値を使用できます。現在のコンテキストアイテムの ancestor またはシブリングを参照できるパス式により、ストリーミングが使用できなくなります。

適格な XMLTABLE のストリーミング

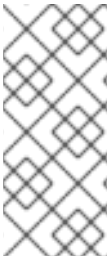
> *XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.', parent_attr string PATH '../@attr', child_val integer)*

コンテキスト XQuery と列パス式では、ドキュメント全体のインメモリーを DOM ツリーとしてロードするのではなく、ストリーミングの最適化が可能になります。各子要素は、結果に個別に追加されます。

streaming ineligible XMLTABLE

> *XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH '../other_child/@attr')*

sibling_attr パスの子サブツリー外の要素の参照により、ストリーミングの最適化は使用されませんが、ドキュメントの展開は依然として実行できます。



注記

パフォーマンスの問題を回避するために、列パスはできるだけターゲットである必要があります。..//child などの一般的なパスにより、各出力行でコンテキストアイテムのサブツリー全体が検索されます。

10.7. フェデレーションされた障害モード

Data Virtualization は、データソースが利用できないか、または失敗した場合に 部分的な結果 を取得する機能を提供します。これは、複数のソースから情報を取り除く場合や、マスターレコードに列を追加する停止した結合を実行する場合などに特に便利です。ただし、追加の情報が利用できない場合でも記録する必要があります。

ソースに関連する接続ファクトリーがクエリーへの応答として例外を生成する場合、ソースは使用不可と見なされます。例外はクエリープロセッサーに伝播され、ステートメントに関する警告になります。部分的な結果および SQLWarning オブジェクトの詳細は、『クライアント開発者ガイド』の「部分結果モード 」を参照してください。

10.8. 準拠したテーブル

準拠テーブルは、複数の物理ソースで同じソーステーブルです。

通常、これは複数のソースにデータを参照する場合に使用されますが、テーブルを表すために単一のメタデータエントリーのみが必要になります。

準拠したテーブルは、以下のエクステンションメタデータプロパティーを適切なソーステーブルに追

加して定義します。

> {http://www.teiid.org/ext/relational/2012}conformed-sources

完全な DDL メタデータを使用するか、ステートメントを変更するか、setProperty システムの手順を使用して、DDL ファイルでエクステンションプロパティーを設定できます。このプロパティーは、物理モデル/スキーマ名のコンマ区切りリストになります。

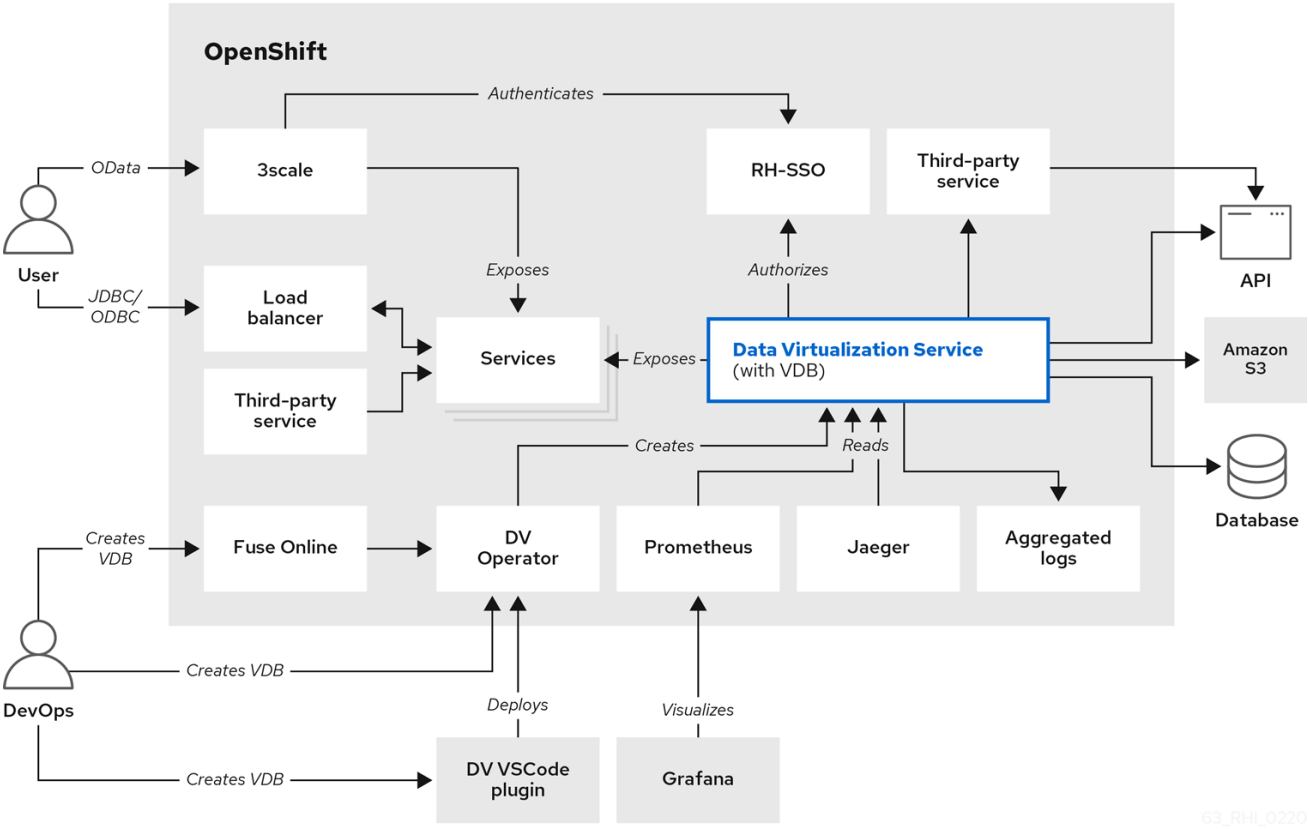
DDL の変更例

```
ALTER FOREIGN TABLE "reference_data" OPTIONS (ADD "teiid_rel:conformed-sources"
'source2,source3');
```

メタデータエントリーが他のスキーマに存在することが予想されます。

エンジンは、準拠したソースの一覧を取得し、モデルメタデータ ID のセットを対応するアクセスノードに関連付けます。また、結合を検討するロジックは、プッシュダウンの決定時に準拠しているセットも考慮します。サブクエリー処理は、親内のサブクエリーに従属するソースのみを確認します。したがって、サブクエリーに準拠しているテーブルがあると、期待通りにプッシュされますが、その逆も同様です。

第11章 DATA VIRTUALIZATION のアーキテクチャー



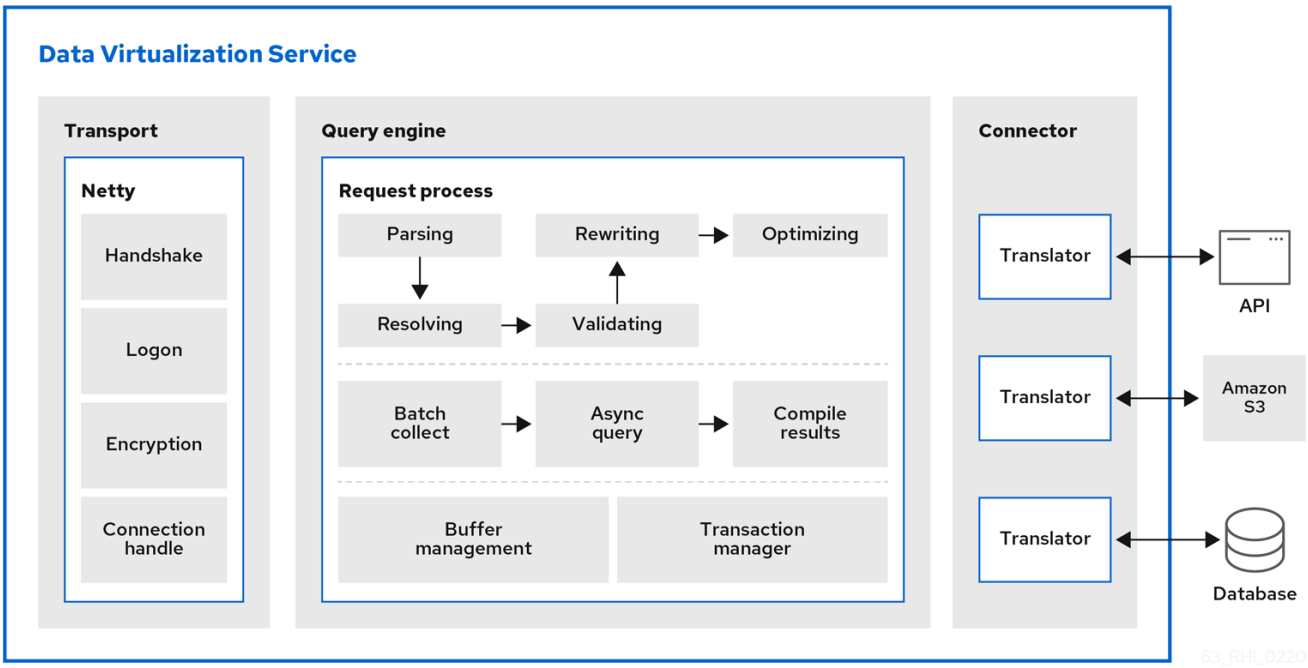
トランスポート

トランスポートサービスはクライアント接続を管理します。セキュリティー認証、暗号化など。

クエリーエンジン

クエリーエンジンには複数のレイヤーとコンポーネントがあります。ハイレベルで、リクエストの処理は以下のように構成されています。

以下の図は、データ仮想化サービスを構成するコンポーネントの詳細を示しています。



1. *SQL はプロセッサープランに変換されます。エンジンは受信 SQL クエリーを受け取ります。内部コマンドへ解析されます。次に、コマンドは解決、検証、および書き換えにより論理プランに変換されます。最後に、ルールとコストベースの最適化は、論理計画を最終的なプロセッサープランに変換します。詳細は、「Federated planning」を参照してください。*

2. *バッチ処理。クエリー処理のソースおよびその他の要素は、結果を処理スレッドに非同期的に返す場合があります。できるだけ早く、クライアントの結果をバッチ処理できるようになります。*

3. *バッファー管理は、Data Virtualization が使用しているオン/オフヒープメモリーの大部分を制御します。メモリーを過剰に消費しないようにし、仮想マシンサイズを超える可能性があります。*

4. *トランザクション管理はトランザクションが必要なタイミングを決定し、XA トランザクションを調整するために TransactionManager サブシステムと対話します。*

ソースクエリーは、クエリーエンジンとインターフェースするデータ層と、トランスレーターを使用してソースと直接対話するコネクターレイヤーによって処理されます。接続は、データベースやデータ機関、NoSQL、Hadoop、Data grid/cache、SaaS など、異種データストア用に提供されます。

トランスレーター

Data Virtualization は一連の変換を開発しました。詳細は、「Translators」を参照し

てください。

11.1. 用語

仮想マシンまたはプロセス

Data Virtualization の Spring Boot インスタンス。

Host

1 つ以上の仮想マシンを実行するマシン。

サービス

仮想マシンで実行されるサブシステム（多くの仮想マシン）は、関連する機能セットを提供します。サービスプラットフォームでは、これらの主なコンポーネントに加えて、サービス上に構築されるアプリケーションで以下のコアサービスセットを利用できます。

Session

セッションサービスは、アクティブなセッション情報を管理します。

バッファーマネージャー

Buffer Manager サービスは、中間結果のデータ管理へのアクセスを提供します。詳細は、「データ管理におけるバッファー管理 」を参照してください。

Transaction

トランザクションサービスは、グローバル、ローカル、およびリクエストスコープ付きトランザクションを管理します。詳細は、「トランザクション 」を参照してください。

11.2. データ管理

Cursoring and batching

Data Virtualization は、1 つのソースまたは多くのソースからのどれかに関係なく、結果に対してどのタイプの処理が行われたかに関係なく、すべての結果のカーソルを行います。

Data Virtualization プロセスにより、バッチ処理が行われます。バッチは、単にレコードのセットです。バッチの行数は、バッファーシステムプロパティー processor-batch-size によって決定され、バッチの推定メモリーフットプリントにスケーリングされます。

クライアントアプリケーションはバッチやバッチサイズに関する直接知識はなく、フェッチサイズを指定します。ただし、フェッチサイズに関係なく、最初のバッチは常に同期クライアントに事前に返されます。後続のバッチは、データのクライアント要求に基づいて返されます。クライアントレベルとコネクターレベルの両方で、事前フェッチが使用されます。

バッファー管理

バッファーマネージャーは、クエリーエンジンで使用されるすべての結果セットのメモリーを管理します。これには、接続ファクトリーから読み取られた結果セット、処理中に一時的に使用される結果セット、およびユーザーに準備される結果セットが含まれます。各結果セットはバッファーマネージャーでタプルソースとして参照されます。

バッファーマネージャーからバッチを取得する場合、バッチのサイズ（バイト単位）が推定され、最大限に対して割り当てられます。

メモリーの管理

バッファーマネージャーには、メモリーマネージャーとディスクマネージャーの 2 つのストレージマネージャーがあります。バッファーマネージャーはすべてのバッチの状態を維持し、バッチをメモリーからディスクに移動する必要があるタイミングを決定します。

ディスク管理

各タプルソースには、ディスク上の専用ファイル（ID によって名前）があります。このファイルは、タプルソースに対して少なくとも 1 つのバッチがディスクにスワップする必要がある場合にのみ作成されます。ファイルはランダムなアクセスです。processor バッチサイズプロパティーは、データが 2048 ビット以上であると仮定して、バッチに乗算的に存在すべき行数を定義します。行がそのターゲットより大きいか、または小さい場合、エンジンはそれらのタプルのバッチサイズを適宜調整します。バッチは、常にストレージマネージャー全体で読み取り、書き込みされます。

ディスクストレージマネージャーはオープンファイルの最大数を制限し、ファイルハンドルが不足しないようにします。バッファーが大きい場合、ファイルハンドルが利用可能になるまで待機している間に待機する可能性があります（デフォルトのオープンファイルは 64 です）。

Cleanup

タプルソースが必要なくなった場合は、バッファーマネージャーから削除されます。バッファーマネージャーはメモリーストレージマネージャーとディスクストレージマネージャーの両方から削除します。ディスクストレージマネージャーはファイルを削除します。さらに、すべてのタプルソースは、通常クライアントのセッション ID である「グループ名」でタグ付けされます。（接続、クライアントのシャットダウンまたは管理終了など）クライアントのセッションが終了すると、呼び出しがバッファーマネージャーに送信され、セッションのタプルソースすべてが削除されます。

さらに、クエリーエンジンがシャットダウンすると、バッファーマネージャーはシャットダウンさ

れ、ディスクストレージマネージャーからすべての状態が削除され、すべてのファイルが閉じられます。クエリーエンジンが停止した場合に、クエリーエンジンの再起動で使用されていないため、バッファーディレクトリー内のファイルは安全に削除でき、バッファーファイルがクリーンアップされないシステムクラッシュが原因で実行する必要があります。

11.3. クエリーの終了

クエリーのキャンセル

クエリーがキャンセルされると、クエリーエンジンとクエリーに関連するすべてのコネクターで処理が停止します。キャンセルコマンドに応答するコネクターのセマンティクスは、コネクターの実装によって異なります。たとえば、JDBC コネクターは基礎となる JDBC ドライバーで非同期的に呼び出します。

ユーザークエリーのタイムアウト

Data Virtualization でのユーザークエリーのタイムアウトはクライアント側またはサーバー側で管理できます。タイムアウトは、返される最初のレコードにのみ関連します。指定したタイムアウト期間内に最初のレコードがクライアントが受け取った場合には、cancel コマンドがリクエストのためにサーバーに発行され、クライアントに結果が返されません。cancel コマンドは、クライアントの介入なしに非同期的に実行されます。

JDBC API は、java.sql.Statement.setQueryTimeout メソッドで設定されたクエリータイムアウトを使用します。connection プロパティー "QUERYTIMEOUT" を使用してデフォルトのステートメントタイムアウトを設定することもできます。ODBC クライアントは set ステートメントを介して実行プロパティーとして QUERYTIMEOUT を使用してデフォルトのタイムアウト設定を制御することもできます。接続/実行プロパティーおよび set ステートメントの詳細は、『クライアント開発者ガイド』を参照してください。

エンジンがクエリーを受信すると、サーバー側のタイムアウトが開始されます。ネットワークレイテンシーまたはサーバーの負荷は、クライアントがクエリーを発行した後に I/O 作業の処理を遅らせることができます。タイムアウトが終了する前に最初の結果が送信されると、タイムアウトはキャンセルされます。仮想データベースの query-timeout プロパティーの設定に関する詳細は、「仮想データベースプロパティー」を参照 してください。すべてのクエリーにデフォルトのクエリータイムアウトを設定するシステムプロパティーの変更に関する詳細は、『管理者ガイド』の「システム プロパティー 」を参照してください。

11.4. 処理

結合アルゴリズム

ネストされたループは最も明らかな処理を行います。外元のソースのすべての行については、内部ソースのすべての行と比較されます。ネストされたループは、結合基準に equi-join 述語がない場合のみ使用されます。

マージ結合は最初に結合した列で入力ソースをソートします。各サイドを並行して実施できます（ソートされたソースごとに 1 つを渡し、一致する場合は行を生成します）。 通常、マージ参加は、ネストされたループでは n*m ではなく n+m の順序になります。 マージ結合はデフォルトのアルゴリズムです。

エンジンはコスト情報を使用することで、完全なソートマージの結合が意思決定を遅らせる可能性があります。エンジンは、実際に関連する行数に基づいて、小さい側のインデックスを構築する（ハッシュ参加と同様のもの）か、関係の大きいものだけを部分的に並べ替えたりを選択できます。

equi-join 述語を伴う結合は、依存する結合に変換することができます。詳細は、Federated optimizations の「Dependent joins」を参照してください。

ソートベースのアルゴリズム

並べ替えは、Soritch(ORDER BY)、Grouping(GROUP BY)、および DupRemoval(SELECT DISTINCT)操作のベースとして使用されます。 ソートアルゴリズムは、すべての結果セットをメモリーに入れる必要がないマルチパスマージ手段で、バッファーマネージャーで許可される最大メモリー容量を使用します。

並べ替えは 2 つのフェーズで構成されます。 最初のフェーズ("sort")では、アルゴリズムはソートされていない入力ストリームを処理し、1 つ以上のソートされた入力ストリームを生成します。 それぞれのパスでは、分類されていないストリームをできる限り読み取り、ソートし、新しいストリームとして書き直します。分類されていないストリームが処理されると、生成されるソートストリームがメモリーに存在するために大きすぎる可能性があります。ソートされたストリームのサイズが利用可能なメモリーを超える場合、これは複数のソートされたストリームに書き込まれます。

ソートアルゴリズム（マージ）の 2 番目のフェーズは、ソートされた入力ストリームの数から次のバッチを取り除く一連のフェーズで構成されます。 その後、各ストリームからソート順に次のタプルを繰り返し取得し、ソートされたバッチを新しいソートストリームにマージします。 フェーズが完了すると、すべての入力ストリームがドロップされます。 これにより、各フェーズはソートされたストリームの数を減らします。 1 つのストリームのみが残っている場合は、最終出力になります。

第12章 SQL GRAMMAR 用の BNF

- *メインエントリーポイント*
  - *呼び出し可能なステートメント*
  - *ddl statement*
  - *explain*
  - *直接実行可能なステートメント*
- *予約されたキーワード*
- *予約されていないキーワード*
- *今後使用するために予約されたキーワード*
- *トークン*
- *Production Cross-Reference*
- *実稼働*

12.1. 予約されたキーワード

キーワード	用途
ADD	設定された子オプション を追加し、オプション、ADD 列、ADD 制約を追加

キーワード	用途
ALL	Standard aggregate function, CREATE POLICY , function,GRANT,query expression body,query term,Revoke GRANT,select clause,quantified comparison predicate
ALTER	Modify ,ALTER PROCEDURE , alter Statement,ALTER TABLE , grant type
AND	述語、 ブール値用語、 ウィンドウフレーム
ANY	標準の集約関数 （ロールおよび定 量化された比較述語）
ARRAY	ARRAY 式コンストラクター
ARRAY_AGG	順序付けされた集約関数
AS	、 ALTER PROCEDURE、 ALTER TABLE、 ALTER TRIGGER 、配列テーブル、 手順の作成、 手順 の作成、 オプションの namespace、 オプションの namespace の作成、 トリガー の作成、 ビュー の作成、 削除ステートメント、 派生列 動的データステートメント、 関数、 JSON テーブル、 ループステートメント、 xml namespace 要素、 オブジェクトテーブル、 派生列、 テーブルサブクエリー、 テキストテーブル、 テーブル 名、 アンエスケープ処理、 更新ステートメント、 list 要素 , xml serialize,xml テーブル
ASC	ソート仕様
ATOMIC	各行トリガーアクションの 複合ステートメント。
AUTHENTICATED	ロールあり
BEGIN	各行トリガーアクションの 複合ステートメント。
BETWEEN	述語とウィンドウフレームの間
BIGDECIMAL	単純なデータ型
BIGINT	単純なデータ型
BIGINTEGER	単純なデータ型
BLOB	単純なデータ型、 xml シリアライズ

キーワード	用途
BOOLEAN	単純なデータ型
BOTH	function
BREAK	ブランチステートメント
BY	Group by clause,order by clause,window specification
BYTE	単純なデータ型
CALL	呼び出し 可能なステートメント、呼び出しステートメント
CASE	ケース式、検索ケース式
CAST	function
CHAR	関数、単純なデータタイプ
CLOB	単純なデータ型、xml シリアライズ
COLUMN	ADD 列、DROP 列、ALTER TABLE TABLE、GRANT、Revoke GRANT
COMMIT	一時テーブルの作成
CONSTRAINT	GRANT、テーブル制約
CONTINUE	ブランチステートメント
CONVERT	function
CREATE	手順 の作成、データラッパー の作成、データベース の作成、ドメインまたはタイプのエイリアス の作成、外部一時テーブル の作成、ロールの 作成、スキーマ の作成、テーブルの作成、一時テーブル の作成、トリガーの作成???
CROSS	cross join
CUME_DIST	分析集計関数
CURRENT_DATE	function
CURRENT_TIME	function

キーワード	用途
CURRENT_TIMESTAMP	function
DATE	数値以外のリテラル の単純なデータタイプ
DAY	function
DECIMAL	単純なデータ型
DECLARE	declare statement
DELETE	Modify,ALTER TRIGGER , CREATE POLICY ,create trigger,delete statement,grant type
DESC	ソート仕様
DISTINCT	標準の集約関数,function,は異なる,クエリー式ボディ,クエリー用語,select 句
DOUBLE	単純なデータ型
DROP	DROP 列、ドロップオプション、Drop データラッパー、ドロップオプション、DROP POLICY、ドロップ手順、ドロップ ロール、ドロップ スキーマ、ドロップ サーバー、ドロップ テーブル、ドロップテーブル、ドロップ タイプ、付与タイプ
EACH	行トリガーアクションごとに
ELSE	ケース式、if ステートメント、検索されたケース式
END	ケース式、複合ステートメント、各行トリガーアクション、検索ケース式
ERROR	エラーステートメントを発生させます。
ESCAPE	述語とテキストテーブルに一致 します。
EXCEPT	クエリー式のボディー
EXEC	動的データステートメント、呼び出しステートメント
EXECUTE	動的データステートメント、付与タイプ、呼び出しステートメント
EXISTS	exists predicate

キーワード	用途
FALSE	オプション、json テーブル、数値以外のリテラルを説明 します。
FETCH	fetch 句
FILTER	Filter 句
FLOAT	単純なデータ型
FOR	CREATE POLICY , for each row trigger action,function,json table column,text aggreate function,text table column,xml table column
FOREIGN	ALTER PROCEDURE、ALTER TABLE、手順の作成、外部またはグローバル一時的なテーブルの作成、外部またはグローバルの一時テーブル の作成、外部 temp テーブル の作成、スキーマ の作成、サーバー、Drop データラッパー の作成、ドロップ手順、ドロップスキーマ、ドロップテーブル、外部キー、インポート外部スキーマ (role)???
FROM	delete statement,from 句,function Import foreign schema,is distinct,Revoke GRANT
FULL	修飾テーブル
FUNCTION	手順、ドロップ手順、GRANT、Revoke GRANTを作成
GLOBAL	外部またはグローバルな一時テーブル、ドロップテーブルの作成
GRANT	GRANT
GROUP	function、group by 句
HANDLER	データラッパーの作成
HAVING	having 句
HOUR	function
IF	if ステートメント
IMMEDIATE	動的データステートメント

キーワード	用途
IMPORT	別のデータベースの インポート、外部スキーマのインポート
IN	関数、手順パラメーター、述語
INNER	修飾テーブル
INOUT	手順パラメーター
INSERT	Modify,ALTER TRIGGER , CREATE POLICY , create trigger,function,insert statement,grant type
INTEGER	単純なデータ型
INTERSECT	クエリー用語
INTO	動的データステートメント、外部スキーマのインポート、文の挿入、句
IS	固有で、null 述語です。
JOIN	cross join,make dep options,qualified table
LANGUAGE	GRANT、オブジェクトテーブル、Revoke GRANT
LATERAL	テーブルサブクエリー
LEADING	function
LEAVE	ブランチステートメント
LEFT	関数、修飾テーブル
LIKE	述語の一致
LIKE_REGEX	正規表現述語のように
LIMIT	Limit 句
LOCAL	外部一時テーブル を作成し、一時テーブルを作成 します。
LONG	単純なデータ型
LOOP	ループステートメント

キーワード	用途
MAKEDEP	オプション句、テーブルプライマリー
MAKEIND	オプション句、テーブルプライマリー
MAKENOTDEP	オプション句、テーブルプライマリー
MERGE	ステートメントを挿入
MINUTE	function
MONTH	function
NO	make dep options,xml namespace element,text aggreate function,text table column,text table
NOCACHE	オプション句
NOT	列オプションを変更 し、述語、複合ステートメント、テーブル要素 の作成、ドメインまたはタイプエイリアス の作成、要素の表示、GRANT の表示、固有 の述語、述語、ブール値係数、手順パラメーター、正規表現述語、一時的なテーブル要素で の正規表現述語など、手順の結果列 を変更します。
NULL	列オプションの変更、テーブル要素 の変更、ドメインまたはタイプエイリアスの作成、要素の表示、null 述語、数値以外のリテラル、手順パラメーター、手順結果列、一時的なテーブル要素、xml クエリー
OF	ALTER TRIGGER を 変更 し、トリガーを作成します。
OFFSET	Limit 句
ON	、ALTER TRIGGER を 変更 し、外部一時ディレクトリーテーブル , CREATE POLICY を作成し、一時テーブル の作成、トリガー、DROP POLICY、GRANT、ループステートメント、修飾されたテーブル、Revoke GRANT、xml クエリーの作成
ONLY	fetch 句
OPTION	オプション句

キーワード	用途
OPTIONS	子オプションリスト の変更、オプションリスト、options句 の変更
OR	ブール値式
ORDER	GRANT、order by 句
OUT	手順パラメーター
OUTER	修飾テーブル
OVER	ウィンドウ指定
PARAMETER	手順の変更
PARTITION	ウィンドウ指定
PERCENT_RANK	分析集計関数
PRIMARY	一時的なテーブル、インライン制約、プライマリーキーの作成
PROCEDURE	Modify,ALTER PROCEDURE , create procedure , CREATE POLICY , DROP POLICY ,drop procedure,GRANT,Revoke GRANT
RANGE	ウィンドウフレーム
REAL	単純なデータ型
REFERENCES	外部キー
RETURN	割り当てステートメント、リターンステートメント、データステートメント
RETURNS	手順の作成
REVOKE	GRANT の取り消し
RIGHT	関数、修飾テーブル
ROLLUP	Group by clause

キーワード	用途
ROW	配列テーブル、フェッチ句、行トリガーアクション、limit 句、テキストテーブル、ウィンドウフレームバインド
ROWS	array table,create temporary table,fetch clause,limit clause,window frame
SECOND	function
SELECT	CREATE POLICY , grant type,select clause
SERVER	代替サーバー、スキーマの 作成 、サーバーの作成、サーバー の 削除、外部スキーマのインポート
SET	set child オプション を追加し、セットオプション、オプション namespace、更新ステートメント、スキーマの設定
SHORT	単純なデータ型
SIMILAR	述語の一致
SMALLINT	単純なデータ型
SOME	標準の集約関数、定量化された比較述語
SQLEXCEPTION	sql exception
SQLSTATE	sql exception
SQLWARNING	ステートメントを引き上げます。
STRING	動的データステートメント、単純なデータ型、xml シリアライゼーション
TABLE	代替表 TABLE 、手順の作成、外部またはグローバル一時テーブル の作成、外部 またはグローバルの一時テーブルの作成、一時テーブル、破棄テーブル、ドロップテーブル、GRANT、クエリープライマリー、Revoke GRANT、テーブルサブクエリー
TEMPORARY	外部またはグローバルな一時テーブル を作成し、外部一時テーブル を作成し、一時テーブル、ドロップテーブル 、GRANT、Revoke GRANTを作成します。
THEN	ケース式、検索ケース式

キーワード	用途
TIME	数値以外のリテラル の単純なデータタイプ
TIMESTAMP	数値以外のリテラル の単純なデータタイプ
TINYINT	単純なデータ型
TO	列オプション の名前を変更 ,RENAME Table , CREATE POLICY , DROP POLICY ,GRANT,match predicate
TRAILING	function
TRANSLATE	function
TRIGGER	ALTER TRIGGER を 変更 し、トリガーを作成します。
TRUE	オプション、json テーブル、数値以外のリテラルを説明 します。
UNION	クロス結合、クエリー式のボディー
UNIQUE	その他の制約、インライン制約
UNKNOWN	数値以外のリテラル
UPDATE	Modify,ALTER TRIGGER , CREATE POLICY , create trigger,dynamic data statement,grant type,update statement
USER	function
USING	CREATE POLICY , dynamic data statement
VALUES	クエリープライマリー
VARBINARY	単純なデータ型、xml シリアライズ
VARCHAR	単純なデータ型、xml シリアライズ
VIRTUAL	代替プル要求OCEDURE 、ALTER TABLE、手順の作成、スキーマ の作成、ビュー の作成、破棄手順、スキーマ のドロップ、テーブル
WHEN	ケース式、検索ケース式

キーワード	用途
WHERE	Filter 句、where 句
WHILE	一方で、ステートメント
WITH	割り当てステートメント、ロールの作成、別のデータベースのインポート、クエリー式、データステートメント
WITHIN	function
WITHOUT	割り当てステートメント、データステートメント
WRAPPER	ALTER DATA WRAPPER、データラッパーの作成、サーバー、Drop データラッパーの作成
XML	オプション を説明 し、単純なデータタイプ
XMLAGG	順序付けされた集約関数
XMLATTRIBUTES	XML 属性
XMLCAST	unescapedFunction
XMLCOMMENT	function
XMLCONCAT	function
XMLELEMENT	xml 要素
XMLEXISTS	xml クエリー
XMLFOREST	xml forest
XMLNAMESPACES	XML 名前空間
XMLPARSE	XML パース
XMLPI	function
XMLQUERY	xml クエリー
XMLSERIALIZE	xml serialize
XMLTABLE	XML テーブル

キーワード	用途
XMLTEXT	function
YEAR	function

12.2. 予約されていないキーワード

名前	用途
ACCESS	basicNonReserved、別のデータベースのインポート
ACCESSPATTERN	basicNonReserved、その他の制約
AFTER	、basicNonReserved を変更し、トリガーを作成します。
ANALYZE	basicNonReserved、explain オプション
ARRAYTABLE	アレイテーブル、basicNonReserved
AUTO_INCREMENT	列オプションの変更、basicNonReserved、テーブル要素の表示
AVG	標準の集約関数 (basicNonReserved)
CHAIN	basicNonReserved、sql 例外
COLUMNS	array table,basicNonReserved,json table,object table,text table,xml table
CONDITION	basicNonReserved、GRANT、Revoke GRANT
CONTENT	basicNonReserved,xml parse,xml serialize
CONTROL	basicNonReserved、別のデータベースのインポート
COUNT	標準の集約関数 (basicNonReserved)
COUNT_BIG	標準の集約関数 (basicNonReserved)
CURRENT	basicNonReserved、window frame bound
DATA	ALTER DATA WRAPPER, basicNonReserved,create data wrapper,create server,Drop data wrapper

名前	用途
DATABASE	ALTER DATABASE、basicNonReserved、データベースの作成、別のデータベース のインポート、データベースの使用
DEFAULT	xml namespace 要素、予約されていない識別子、オブジェクトテーブル列、作成後の列、手順パラメーター、xml テーブル列
DELIMITER	basicNonReserved、テキスト aggreate 関数、テキストテーブル
DENSE_RANK	analytic 集約関数 (basicNonReserved)
DISABLED	Modify,ALTER TRIGGER, basicNonReserved
DOCUMENT	basicNonReserved,xml parse,xml serialize
DOMAIN	basicNonReserved:ドメインまたはタイプのエイリアスを作成します。
EMPTY	basicNonReserved,xml クエリー
ENABLED	Modify,ALTER TRIGGER, basicNonReserved
ENCODING	basicNonReserved、text aggreate function、xml serialize
EPOCH	basicNonReserved,function
EVERY	標準の集約関数 (basicNonReserved)
EXCEPTION	複合ステートメント、declaステートメント、予約されていない識別子
EXCLUDING	basicNonReserved、xml serialize
EXPLAIN	basicNonReserved、説明
EXTRACT	basicNonReserved,function
FIRST	basicNonReserved,fetch 句,sort specification
FOLLOWING	basicNonReserved、window frame bound
FORMAT	basicNonReserved、explain オプション

名前	用途
GEOGRAPHY	予約されていない識別子、単純なデータタイプ
GEOMETRY	予約されていない識別子、単純なデータタイプ
HEADER	basicNonReserved、テキスト aggreate 関数、テキストテーブル列、テキストテーブル
INCLUDING	basicNonReserved、xml serialize
INDEX	その他の制約、インライン制約、予約されていない識別子
INSTEAD	Modify,ALTER TRIGGER, basicNonReserved,create trigger
JAAS	basicNonReserved （ロールあり）
JSON	予約されていない識別子、単純なデータタイプ
JSONARRAY_AGG	basicNonReserved、順序付けされた集約関数
JSONOBJECT	basicNonReserved、json オブジェクト
JSONTABLE	basicNonReserved、json テーブル
KEY	basicNonReserved,create temporary table,foreign key,inline constraint,primary key
LAST	basicNonReserved、ソート指定
LISTAGG	basicNonReserved,function
MASK	basicNonReserved、GRANT、Revoke GRANT
MAX	標準の集約関数 （basicNonReserved,make dep オプション）
MIN	標準の集約関数 (basicNonReserved)
NAME	basicNonReserved,function,xml 要素
NAMESPACE	basicNonReserved、オプションの namespace
NEXT	basicNonReserved,fetch 句

名前	用途
NONE	basicNonReserved
NULLS	basicNonReserved、ソート指定
OBJECT	予約されていない識別子、単純なデータタイプ
OBJECTTABLE	basicNonReserved、オブジェクトテーブル
ORDINALITY	basicNonReserved,json table column,テキストテーブル列,xml テーブル列
PASSING	basicNonReserved,object table,xml query,xml query,xml table
PATH	basicNonReserved,json table column,xml table column
POLICY	basicNonReserved , CREATE POLICY , DROP POLICY
POSITION	basicNonReserved,function
PRECEDING	basicNonReserved、window frame bound
PRESERVE	basicNonReserved,create temporary table
PRIVILEGES	basicNonReserved、GRANT、Revoke GRANT
QUARTER	basicNonReserved,function
QUERYSTRING	basicNonReserved、querystring 関数
QUOTE	basicNonReserved、テキスト aggreate 関数、テキストテーブル
RAISE	basicNonReserved,raise ステートメント
RANK	analytic 集約関数 (basicNonReserved)
RENAME	ALTER PROCEDURE , ALTER TABLE , basicNonReserved
REPOSITORY	basicNonReserved、外部スキーマのインポート
RESULT	basicNonReserved,procedure パラメーター

名前	用途
ROLE	basicNonReserved,create role , drop role , with role
ROW_NUMBER	analytic 集約関数 (basicNonReserved)
SCHEMA	basicNonReserved,Create schema,drop schema,GRANT,Import foreign schema,Revoke GRANT,set schema
SELECTOR	basicNonReserved、テキストテーブル列、テキストテーブル
SERIAL	列オプションの変更、テーブル要素、要素 の表示、予約されていない識別子、一時テーブル要素
SKIP	basicNonReserved （テキストテーブル）
SQL_TSI_DAY	basicNonReserved、時間間隔
SQL_TSI_FRAC_SECOND	basicNonReserved、時間間隔
SQL_TSI_HOUR	basicNonReserved、時間間隔
SQL_TSI_MINUTE	basicNonReserved、時間間隔
SQL_TSI_MONTH	basicNonReserved、時間間隔
SQL_TSI_QUARTER	basicNonReserved、時間間隔
SQL_TSI_SECOND	basicNonReserved、時間間隔
SQL_TSI_WEEK	basicNonReserved、時間間隔
SQL_TSI_YEAR	basicNonReserved、時間間隔
STDDEV_POP	標準の集約関数 (basicNonReserved)
STDDEV_SAMP	標準の集約関数 (basicNonReserved)
SUBSTRING	basicNonReserved,function
SUM	標準の集約関数 (basicNonReserved)
TEXT	basicNonReserved、explain オプション

名前	用途
TEXTAGG	basicNonReserved、テキスト aggreate 関数
TEXTTABLE	basicNonReserved （テキストテーブル）
TIMESTAMPADD	basicNonReserved,function
TIMESTAMPDIFF	basicNonReserved,function
TO_BYTES	basicNonReserved,function
TO_CHARS	basicNonReserved,function
TRANSLATOR	ALTER DATA WRAPPER, basicNonReserved,create data wrapper,create server,Drop data wrapper
TRIM	basicNonReserved,function,text table column,テキストテーブル
TYPE	列オプションの変更、basicNonReserved、データラッパーの作成、サーバーの作成
UNBOUNDED	basicNonReserved、window frame bound
UPSERT	basicNonReserved,insert statement
USAGE	basicNonReserved、GRANT、Revoke GRANT
USE	basicNonReserved （データベースを使用）
VARIADIC	basicNonReserved,procedure パラメーター
VAR_POP	標準の集約関数 (basicNonReserved)
VAR_SAMP	標準の集約関数 (basicNonReserved)
VERSION	basicNonReserved、データベースの作成、サーバーの作成、別のデータベースのインポート、データベースの使用、xml シリアライゼーション
VIEW	Modify,ALTER TABLE, basicNonReserved,create view,drop table
WELLFORMED	basicNonReserved,xml parse

名前	用途
WIDTH	basicNonReserved （テキストテーブル列）
XMLDECLARATION	basicNonReserved、xml serialize
YAML	basicNonReserved、explain オプション

12.3. 今後使用するために予約されたキーワード

ALLOCATE	ARE	ASENSITIVE
ASYMETRIC	AUTHORIZATION	BINARY
CALLED	CASCADED	文字
チェック	閉じる	COLLATE
接続	対応する機能	条件
CURRENT_USER	カーソル	CYCLE
DATALINK	DEALLOCATE	DEC
DEREF	DESCRIBE	決定論的
DISCONNECT	DLNEWCOPY	DLPREVIOUSCOPY
DLURLCOMPLETE	DLURLCOMPLETEONLY	DLURLCOMPLETEWRITE
DLURLPATH	DLURLPATHONLY	DLURLPATHWRITE
DLURLSCHEME	DLURLSERVER	DLVALUE
動的	要素	EXTERNAL
無料	GET	HAS
HOLD	アイデンティティー	インディケーター
INPUT	INSENSITIVE	INT
INTERVAL	分離	LARGE
LOCALTIME	LOCALTIMESTAMP	MATCH

メンバー	メソッド	MODIFY
モジュール	マルチセット	NATIONAL
自然	NCHAR	NCLOB
NEW	NUMERIC	OLD
OPEN	OUTPUT	OVERLAPS
精度	PREPARE	READS
RECURSIVE	参照	RELEASE
ROLLBACK	SAVEPOINT	SCROLL
SEARCH	SENSITIVE	SESSION_USER
SPECIFIC	SPECIFICTYPE	SQL
開始	STATIC	SUBMULTILIST
SYMETRIC	システム	SYSTEM_USER
TIMEZONE_HOUR	TIMEZONE_MINUTE	翻訳
TREAT	VALUE	異なる
WHENEVER	ウィンドウ	XMLBINARY
XMLDOCUMENT	XMLITERATE	XMLVALIDATE

12.4. トークン

名前	定義	用途
all in group identifier	<identifier> <period> <star>	すべてのグループ
バイナリー文字列リテラル	"X" | "x" "\'" (<hexit> <hexit>)+ "\'"	数値以外のリテラル
colon	":"	make dep options,statement

名前	定義	用途
comma	","	子オプションリスト の変更、オプションリスト、ARRAY 式のコンストラクター、列リスト、作成手順、入力要素リスト、CREATE POLICY、テーブル本文の作成、一時テーブルの作成、ビューボディー の作成、派生列リスト、sql 例外 リスト named parameter list,explain,expression list,from 句,function,GRANT,identifier list,json table,limit clause,nested expression,object table,option 句 , options 句,options 句, Order by clause ,simple data type,query expression,query primary,querystring function,Revoke GRANT,select clause,set clause list,in predicate,text aggreate function,text table, xml 属性,xml 要素,xml クエリー,xml フォレスト,xml 名前空間,xml クエリー,xml テーブル
concat_op	"||"	共通値式
decimal numeric literal	(<digit>)* <period> <unsigned integer literal >	署名されていない数値リテラル
digit	\["0"\-"9"\]	
dollar	"$"	パラメーターの参照
double_amp_op	"&&"	共通値式
eq	"="	割り当てステートメント、呼び出し可能なステートメント、宣言ステートメント、名前付きパラメーターリスト、比較演算子、セット句リスト
escaped 関数	"{" "fn"	署名なし値式のプライマリー
escaped join	"{" "oj"	テーブルの参照
escaped type	"{" ("d" | "t" | "ts" | "b")	数値以外のリテラル

名前	定義	用途
approximate numeric literal	<digit> <period> <signedinteger literal > \["e","E"\](<plus> | <minus>)? <unsigned integer literal >	署名されていない数値リテラル
ge	">="	比較演算子
gt	">"	名前付きパラメーターリスト,比較演算子
hexit	\["a"\-"f","A"\-"F"\] | <digit>	
identifier	<quoted_id> (<period> <quoted_id>)*	ドメインまたはタイプエイリアス、識別子、データタイプ、非修飾識別 子、署名されていない値式のプライマリーを作成 します。
id_part	("" | "@" | "#" | <letter>) (<letter> | "" | <digit>)*	
lbrace	"{"	呼び出し可能なステートメント、述語と一致
le	"<="	比較演算子
letter	\["a"\-"z","A"\-"Z"\] | \["\u0153"\-"\ufffd"\]	

名前	定義	用途
lparen	"("	標準の集約関数、子オプションリストの変更、オプションリストの変更、分析集計関数、配列テーブル、呼び出し可能な ステートメント、列リスト、その他の制約、手順、CREATE POLICY、および create table body , create table body , create table body , create table body ,create table body, 一時的なテーブル を作成し、ビューボディ の作成、説明、フィルター句、関数、group by 句、if ステートメント、json オブジェクト、JSONテーブル、ループステートメント、デッドオプション、ネストされた式、オブジェクトテーブル、 options 句、順序付けされた集約関数、単純なデータ型、クエリープライマリー、クエリー文字列関数、述語、呼び出しステートメント、サブクエリー、定量化された比較述語、テーブルサブクエリー、表プライマリー、テキストのアグレート関数 テキストテーブル、非エスケープ処理Function、およびステートメント、ウィンドウ仕様、リスト要素、xml 属性、xml 要素、xml クエリー、xml フォレスト、xml 名前空間、xml 解析、xml クエリー、 xml serialize,xml table
lsbrace	"["	ARRAY 式コンストラクター、基本データ型、データタイプ、値式のプライマリー
lt	"<"	比較演算子
minus	"-"	プラスまたはマイナス
ne	"<>"	比較演算子
ne2	"!="	比較演算子
period	"."	
plus	"+"	プラスまたはマイナス
qmark	"?"	呼び出し可能なステートメント、パラメーター参照

名前	定義	用途
quoted_id	<id_part> | "\"" ("\"\"" | ~\["\""\])+ "\""	
rbrace	"}"	呼び出し可能なステートメント、述語、数値以外のリテラル、テーブル参照、署名されていない値式のプライマリーに一致 します。
rparen	")"	標準の集約関数、子オプションリストの変更、オプションリストの変更、分析集計関数、配列テーブル、呼び出し可能な ステートメント、列リスト、その他の制約、手順、CREATE POLICY、および create table body , create table body , create table body , create table body ,create table body, 一時的なテーブル を作成し、ビューボディ の作成、説明、フィルター句、関数、group by 句、if ステートメント、json オブジェクト、JSONテーブル、ループステートメント、デッドオプション、ネストされた式、オブジェクトテーブル、 options 句、順序付けされた集約関数、単純なデータ型、クエリープライマリー、クエリー文字列関数、述語、呼び出しステートメント、サブクエリー、定量化された比較述語、テーブルサブクエリー、表プライマリー、テキストのアグレート関数 テキストテーブル、非エスケープ処理Function、およびステートメント、ウィンドウ仕様、リスト要素、xml 属性、xml 要素、xml クエリー、xml フォレスト、xml 名前空間、xml 解析、xml クエリー、 xml serialize,xml table
rsbrace	"]"	ARRAY 式コンストラクター、基本データ型、データタイプ、値式のプライマリー
semicolon	";"	区切られたステートメント
slash	"/"	star または slash

名前	定義	用途
star	"*"	標準の集約関数、動的データステートメント、選択句、星、またはスラッシュ
string literal	("N" | "E")? "\'" ("\'\'" | ~\["\'"\])* "\'"	string
unsigned integer literal（未署名整数リテラル）	(<digit>)+	署名されていない整数（未署名の数値リテラル）

12.5. PRODUCTION CROSS-REFERENCE

名前	用途
add set child option	子オプション一覧の変更
add set option	オプション一覧の変更
標準の集約機能	unescapedFunction
all in group	サブリストの選択
alter	直接実行可能なステートメント
ADD column	テーブルの変更
ADD constraint	テーブルの変更
alter child option pair	設定した子オプションの追加
alter child options list	列オプションの変更
alter column options	手順の変更、 テーブル の変更
ALTER DATABASE	alterStatement
DROP column	テーブルの変更
alter option pair	セットオプションの追加
alter options list	データベース の 変更、手順の変更 、サーバー の変更 、テーブルの 変更、データラッパーの変更
ALTER PROCEDURE	alterStatement

名前	用途
rename column options	手順の変更、 テーブル の変更
RENAME Table	テーブルの変更
ALTER SERVER	alterStatement
alterStatement	ddl statement
ALTER TABLE	alterStatement
ALTER DATA WRAPPER	alterStatement
ALTER TRIGGER	alterStatement
analytic aggregate function	unescapedFunction
ARRAY expression constructor	署名なし値式のプライマリー
array table	テーブルプライマリー
assignment statement	区切られたステートメント
assignment statement operand	割り当てステートメント、ステートメント の宣言
basicNonReserved	ドメインまたはタイプエイリアス、予約されていない識別子、データタイプを作成 します。
between predicate	boolean primary
boolean primary	CREATE POLICY , filter clause,boolean factor
branching statement	区切られたステートメント
callable statement	
case expression	署名なし値式のプライマリー
character	述語、テキストのアグレート関数、テキストテーブルに一致 します。
column list	その他の制約、一時的なテーブルの作成、外部キー、挿入ステートメント、プライマリーキー、リスト要素あり

名前	用途
common value expression	述語と ブール値のプライマリー,比較述語,sql exception,関数,は異なる , 述語の ,述語の一致、述語の述語、 述語 のテーブル）
comparison predicate	boolean primary
boolean term	ブール値式
boolean value expression	condition
compound statement	ステートメント、直接実行可能なステートメント
other constraints	テーブルの制約
table element	ADD 列、テーブルボディの作成
create procedure	ddl statement
create data wrapper	ddl statement
create database	ddl statement
ドメインまたはタイプエイリアスの作成	ddl statement
typed element list	アレイテーブル、動的データステートメント
create foreign or global temporary table	テーブルの作成
create foreign temp table	直接実行可能なステートメント
option namespace	ddl statement
CREATE POLICY	ddl statement
create role	ddl statement
create schema	ddl statement
create server	ddl statement
create table	ddl statement
create table body	外部またはグローバル一時テーブルを作成し、外部一時テーブル を作成します。

名前	用途
create temporary table	直接実行可能なステートメント
create trigger	DDL ステートメント、直接実行可能なステートメント
create view	テーブルの作成
create view body	ビューの作成
view element	ビューボディーの作成
condition	expression,having 句,if statement, ifqualified table,search case expression,where clause,while statement
cross join	結合されたテーブル
ddl statement	ddl statement
declare statement	区切られたステートメント
delete statement	割り当てステートメントオペランド、直接の実行ステートメント
delimited statement	statement
derived column	派生列リスト、オブジェクトテーブル、クエリー文字列関数、テキストのアッテート関数、xml 属性、xml クエリー、xml クエリー、xml テーブル
derived column list	JSON オブジェクト,xml フォレスト
drop option	子オプション一覧の変更
Drop data wrapper	ddl statement
drop option	オプション一覧の変更
DROP POLICY	ddl statement
drop procedure	ddl statement
drop role	ddl statement
drop schema	ddl statement

名前	用途
drop server	ddl statement
drop table	直接実行可能なステートメント
drop table	ddl statement
dynamic data statement	data statement
raise error statement	区切られたステートメント
sql exception	割り当てステートメントオペランド、例外参照
exception reference	SQL exception,raise statement
named parameter list	呼び出し 可能なステートメント、呼び出しステートメント
exists predicate	boolean primary
explain	
explain option	explain
expression	標準の集約関数、ARRAY 式コンストラクター、割り当てステートメントオペランド、ケース式、派生列、動的データステートメント、エラーステートメント 、名前付きパラメーターリスト、式リスト 、関数、ネストされた式、オブジェクトテーブル列、順序付けされた集約関数、 列の作成後、手順パラメーター、クエリー文字列関数、戻り値文、検索されたケース式、派生列、セット句リスト、ソートキー リスト、定型化された比較述語、アンエスケープ処理 Function、xml テーブル列、 xml 要素,xml parse,xml serialize
expression list	callable statement,other constraints,function,group by clause,query primary,call statement,window specification
fetch clause	Limit 句
filter clause	function、unescapedFunction
for each row trigger action	ALTER TRIGGER を 変更 し、トリガーを作成します。
foreign key	テーブルの制約

名前	用途
from clause	query
function	UnescapedFunction,unsigned value expression primary
GRANT	ddl statement
group by clause	query
having clause	query
identifier	Modify , modifyingchild option pair, alter column options,ALTER DATABASE, DROP column,alter option pair,ALTER PROCEDURE , RENAME TableRENAME Table ALTER SERVER, ALTER TABLE , ALTER DATA WRAPPER ,ALTER DATA WRAPPER , ??? ALTER TRIGGER , array table,assignment statement,branching statement,callable statement,column list,compound statement,table element,create data wrapper,create database,typed element list,,create foreign temp table, オプションの namespace , CREATE POLICY , create schema,create trigger,view element,declare statement,delete statement,derived column,drop option Drop data wrapper,drop option , DROP POLICY , drop procedure,drop role , drop schema,drop server,drop table,drop table,dynamic data statement,exception reference,named parameter list,foreign key,function,GRANT,identifier list, 別のデータベース、 Import foreign schema、 insert statement, insert statement ,into 句,json table column,json table,loop statement,xml namespace element,object table column,object table,option clause,option pair, Procedure parameter,procedure result column,query primary,Revoke GRANT,Select derived column,set clause list,statement,call statement,table subquery,table constraint,temporary table element, テキスト関数、 テキストテーブル列、 テキストテーブル列、 テーブル 名、 更新ステートメント、 データベースの使用、 スキーマの設定、 リスト要素、 xml テーブル列、 xml要素 、 xml テーブル
identifier list	ロールのある スキーマの 作成
if statement	statement
別のデータベースのインポート	ddl statement

名前	用途
外部スキーマのインポート	ddl statement
inline constraint	列の作成後
insert statement	割り当てステートメントオペランド、直接の実行ステートメント
integer parameter	fetch 句、limit 句
unsigned integer	動的データステートメント、関数、GRANT、整数パラメーター、diveオプション、パラメーター参照、単純なデータ型、テキストテーブル列、テキストテーブル、ウィンドウフレームバインド
time interval	function
into clause	query
is distinct	boolean primary
is null predicate	boolean primary
joined table	テーブルプライマリー、テーブル参照
json table column	json table
json object	function
json table	テーブルプライマリー
limit clause	クエリー式のボディー
loop statement	statement
make dep options	オプション句、テーブルプライマリー
match predicate	boolean primary
xml namespace 要素	XML 名前空間
nested expression	署名なし値式のプライマリー
non numeric literal	子オプションのペア を変更し、オプションのペア（オプションペア ）、値式のプライマリーを変更

名前	用途
non-reserved identifier	識別子、非修飾識別子、署名されていない値式のプライマリー
boolean factor	ブール型用語
object table column	オブジェクトテーブル
object table	テーブルプライマリー
comparison operator	比較述語、定量化された比較述語
option clause	callable statement,delete statement,insert statement,query expression body,call statement,update statement
option pair	Options 句
options clause	手順の作成、データラッパー の作成、データベースの作成、スキーマ の作成、サーバー 本文の作成、テーブル本文 の作成、ビュー の作成、ビューボディの作成、外部スキーマのインポート、作成後の列 パラメーター、手順結果列、テーブル制約
order by clause	関数、順序付けされた集約関数、クエリー式のボディー、テキストのアグレッテート関数、ウィンドウの指定
ordered aggregate function	unescapedFunction
parameter reference	署名なし値式のプライマリー
basic data type	typed element list,json table column,object table column,data type,temporary table element,text table column,xml table column
data type	列オプションの変更、テーブル要素 の変更、手順の作成、ドメインまたはタイプのエイリアス の作成、要素の表示、要素 の表示、要素の宣言、関数、手順パラメーター、手順結果列、アンエスケープ処理
simple data type	基本的なデータ型
numeric value expression	Common value expression,value expression primary
plus or minus	子オプションのペア の変更、オプションのペア、オプションのペア、数値値の式、および式 のプライマリーの変更

名前	用途
post create column	テーブル要素、要素 の表示
primary key	テーブルの制約
procedure parameter	手順の作成
procedure result column	手順の作成
qualified table	結合されたテーブル
query	クエリープライマリー
query expression	Change,ALTER TABLE , ARRAY expression constructor,assignment statement operand,create view,insert statement,loop statement,subquery,table subquery,directly executable statement,with list element
query expression body	クエリー式、クエリープライマリー
query primary	クエリー用語
querystring function	function
query term	クエリー式のボディー
raise statement	区切られたステートメント
grant type	GRANT、GRANT の取り消し
with role	ロールの作成
like regex predicate	boolean primary
return statement	区切られたステートメント
Revoke GRANT	ddl statement
searched case expression	署名なし値式のプライマリー
select clause	query
select derived column	サブリストの選択
select sublist	Select 句

名前	用途
set clause list	動的データステートメント、更新ステートメント
in predicate	boolean primary
sort key	ソート仕様
sort specification	Order by clause
data statement	区切られたステートメント
statement	ALTER PROCEDURE、複合ステートメント の 変更、各行のトリガーアクション、if ステートメント、loop ステートメント 、および文の作成
call statement	割り当てステートメント、サブクエリー、テーブルサブクエリー、直接実行可能なステートメント
string	character,create database,option namespace,create server,function,GRANT,Import another Database,json table column,json table,xml namespace element,non numeric literal, object table column,object table,text table column, テキストテーブル、データベース、xml テーブル列、xml クエリー、xml クエリー、xml クエリー、xmlシリアライズ、xml テーブルの使用
subquery	述語、述語、定量化された比較述語、 署名されていない値式のプライマリーに存在 する。
quantified comparison predicate	boolean primary
table subquery	テーブルプライマリー
table constraint	ADD 制約、テーブルボディの作成、ビューボディーの作成
temporary table element	一時テーブルの作成
table primary	結合と結合されたテーブル
table reference	from 句,修飾テーブル
text aggreate function	unescapedFunction
text table column	テキストテーブル

名前	用途
text table	テーブルプライマリー
term	数値式
star or slash	term
table name	テーブルプライマリー
unescapedFunction	署名なし値式のプライマリー
Unqualified identifier	手順の作成、データラッパーの作成、外部またはグローバル一時的なテーブルの作成、外部またはグローバル一時的な テーブルの作成、外部 一時テーブルの作成、ロール の 作成、サーバー の作成、一時テーブル の作成、ビューの作成
unsigned numeric literal	子オプションのペア を変更し、オプションのペア（オプションペア ）、値式のプライマリーを変更
unsigned value expression primary	整数パラメーター,値式 プライマリー
update statement	割り当てステートメントオペランド、直接の実行ステートメント
use database	ddl statement
set schema	ddl statement
directly executable statement	説明、データステートメント
value expression primary	array table,json table,term
where clause	ステートメント、クエリー、更新ステートメントを削除
while statement	statement
window frame	ウィンドウ指定
window frame bound	ウィンドウフレーム
window specification	unescapedFunction
with list element	クエリー式
xml attributes	xml 要素

名前	用途
xml テーブル列	XML テーブル
xml element	function
xml query	boolean primary
xml forest	function
xml namespaces	xml 要素,xml クエリー,xml フォレスト,xml クエリー,xml テーブル
xml parse	function
xml query	function
xml serialize	function
xml table	テーブルプライマリー

12.6. 実稼働

12.6.1. string ::=

- *<string literal >*

文字列リテラル値。'' を使用して、文字列の ' をエスケープします。

例:

```
'a string'
```

```
'it''s a string'
```

12.6.2. non-reserved identifier ::=

- *EXCEPTION*
- *SERIAL*
- *オブジェクト*
- *INDEX*
- *JSON*
- *ジオメトリー*
- *GEOGRAPHY*
- *DEFAULT*
- *<basicNonReserved>*

予約されていないキーワードを識別子として解析できるようにする

例： SELECT COUNT FROM ...

12.6.3. basicNonReserved ::=

- *INSTEAD*
- *VIEW*

- *ENABLED*
- *DISABLED*
- *KEY*
- *TEXTAGG*
- *カウント*
- *COUNT_BIG*
- *ROW_NUMBER*
- *ランク*
- *DENSE_RANK*
- *SUM*
- *AVG*
- *最小*
- *最大*
- *実行頻度*

- *STDDEV_POP*
- *STDDEV_SAMP*
- *VAR_SAMP*
- *VAR_POP*
- *DOCUMENT*
- *コンテンツ*
- *TRIM*
- *空*
- *ORDINALITY*
- *PATH*
- *FIRST*
- *最終*
- *次へ*
- *SUBSTRING*

- *EXTRACT*
- *TO_CHARS*
- *TO_BYTES*
- *TIMESTAMPADD*
- *TIMESTAMPDIFF*
- *QUERYSTRING*
- *NAMESPACE*
- *RESULT*
- *ACCESSPATTERN*
- *AUTO_INCREMENT*
- *WELLFORMED*
- *SQL_TSI_FRAC_SECOND*
- *SQL_TSI_SECOND*
- *SQL_TSI_MINUTE*

- *SQL_TSI_HOUR*
- *SQL_TSI_DAY*
- *SQL_TSI_WEEK*
- *SQL_TSI_MONTH*
- *SQL_TSI_QUARTER*
- *SQL_TSI_YEAR*
- *TEXTTABLE*
- *ARRAYTABLE*
- *JSONTABLE*
- *セレクター*
- *SKIP*
- *WIDTH*
- *合格*
- *名前*

- *ENCODING*
- *カラム*
- *DELIMITER*
- *QUOTE*
- *ヘッダー*
- *NULLS*
- *OBJECTTABLE*
- *バージョン*
- *INCLUDE*
- *除外*
- *XMLDECLARATION*
- *VARIADIC*
- *RAISE*
- *CHAIN*

- *JSONARRAY_AGG*
- *JSONOBJECT*
- *保存*
- *UPSERT*
- *AFTER*
- *タイプ*
- *TRANSLATOR*
- *JAAS*
- *CONDITION*
- *MASK*
- *ACCESS*
- *コントロール*
- *NONE*
- *DATA*

- *DATABASE*
- *権限*
- *ROLE*
- *SCHEMA*
- *USE*
- *リポジトリー*
- *RENAME*
- *DOMAIN*
- *使用方法*
- *POSITION*
- *CURRENT*
- *UNBOUNDED*
- *前述の手順*
- *FOLLOWING*

- *LISTAGG*
- *EXPLAIN*
- *ANALYZE*
- *TEXT*
- *FORMAT*
- *YAML*
- *EPOCH*
- *金*
- *POLICY*

12.6.4. Unqualified identifier ::=

- *<identifier>*
- *<予約されていない識別子>*

単一エンティティーの修飾名。

例:

```
"tbl"
```

12.6.5. identifier ::=

- <identifier>
- <予約されていない識別子>

単一エンティティーのパーシャルまたはフルネーム。

例:

```
tbl.col
```

```
"tbl"."col"
```

12.6.6. create trigger ::=

- CREATE TRIGGER (<identifier>)?ON <identifier> ((INSTEAD OF)| AFTER) (INSERT | UPDATE | DELETE) AS <for each row trigger action>

指定されたターゲットでトリガーアクションを作成します。

例:

```
CREATE TRIGGER ON vw INSTEAD OF INSERT AS FOR EACH ROW BEGIN ATOMIC ... END
```

12.6.7. alter ::=

- 代替者(VIEW <identifier> AS <query expression>)|(PROCEDURE <identifier> AS <statement>)|（ TRIGGER (<identifier>)?ON <identifier> ((INSTEAD OF)| AFTER ）(INSERT | UPDATE | DELETE)（（各行トリガーアクション>）| ENABLED | DISABLED ））???

指定されたターゲットを変更します。

例:

```
ALTER VIEW vw AS SELECT col FROM tbl
```

12.6.8. for each row trigger action ::=

- FOR EACH ROW （（ BEGIN (ATOMIC)?(<statement>)* END ）| < statement >）

各行で実行するアクションを定義します。

例:

```
FOR EACH ROW BEGIN ATOMIC ... END
```

12.6.9. explain ::=

- EXPLAIN (<lparen> <explain option> <comma> <explain option>)* <rparen>）? <directly executable statement>

ステートメントのクエリー計画を返します。

例： EXPLAIN select 1

12.6.10. explain option ::=

- （ *ANALYZE(TRUE | FALSE)?*）
- （ *XML | TEXT | YAML)?*）

explain ステートメントのオプション

例： FORMAT YAML

12.6.11. directly executable statement ::=

- *<query expression>*
- *<call statement>*
- *<insert statement>*
- *<update statement>*
- *<delete statement>*
- *<drop table>*

- *<一時テーブルの作成>*
- *< Create external temp table>*
- *<Changes>*
- *<作成トリガー>*
- *<compound statement>*

ランタイム時に実行できるステートメント。

例:

```
SELECT * FROM tbl
```

12.6.12. drop table ::=

- *DROP TABLE <identifier>*

指定のテーブルを破棄します。

例:

```
DROP TABLE #temp
```

12.6.13. create temporary table ::=

- *CREATE (LOCAL)?TEMPORARY TABLE <Unqualified identifier> <lparen> <temporary table element>(<comma> <temporary table element>)*(<comma> PRIMARY KEY <column list>)? <rparen>(ON COMMIT PRESERVE ROWS)?*

一時テーブルを作成します。

例:

```
CREATE LOCAL TEMPORARY TABLE tmp (col integer)
```

12.6.14. temporary table element ::=

- *<identifier>(<basic data type> | SERIAL) (NULL ではない) ?*

一時的なテーブル列を定義します。

例:

```
col string NOT NULL
```

12.6.15. raise error statement ::=

- *ERROR <expression>*

指定のメッセージでエラーを発生させます。

例:

```
ERROR 'something went wrong'
```

12.6.16. raise statement ::=

- *RAISE* (*SQLWARNING*)? <*exception reference*>

指定のメッセージでエラーまたは警告を発生させます。

例:

```
RAISE SQLEXCEPTION 'something went wrong'
```

12.6.17. exception reference ::=

- <*identifier*>
- <*SQL exception*>

例外への参照

例:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

12.6.18. sql exception ::=

- *SQLEXCEPTION* <*common value expression*> (*SQLSTATE* <*common value*

expression>(<comma> <common value expression>)?）?）(CHAIN <exception reference>)?

指定したメッセージ、状態、およびコードで sql 例外または警告を作成します。

例:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

12.6.19. statement ::=

- *((<identifier> <colon>)?(<loop statement> | <while statement> | <compound statement>)）*
- *<if statement> | <delimited statement>*

手順ステートメント。

例:

```
IF (x = 5) BEGIN ... END
```

12.6.20. delimited statement ::=

- *(<assignment statement> | <data statement> | <raise error statement> | <raise statement> | <declare statement> | <branching statement> | <return statement>) <semicolon>*

; で終了する手順ステートメント。

例:

```
SELECT * FROM tbl;
```

12.6.21. compound statement ::=

- 開始 （ ではない ）?ATOMIC ）?(<statement>)* （ EXCEPTION <identifier> (<statement>)*）?END

BEGIN END に含まれる procedure ステートメントブロック。

例:

```
BEGIN NOT ATOMIC ... END
```

12.6.22. branching statement ::=

- (BREAK | CONTINUE)(<identifier>)?）
- (LEAVE <identifier>)

手順の分岐制御ステートメント。通常、コントロールを返すラベルを指定します。

例:

```
BREAK x
```

12.6.23. return statement ::=

- *RETURN (<expression>)?*

戻り値のステートメント。

例:

```
RETURN 1
```

12.6.24. while statement ::=

- *WHILE <lparen> <condition> <rparen> <statement>*

条件が false になるまで実行されるステートメントの手順。

例:

```
WHILE (var) BEGIN ... END
```

12.6.25. loop statement ::=

- *LOOP ON <lparen> <query expression> <rparen> AS <identifier> <statement>*

指定のカーソルで実行する手順ループステートメント。

例:

```
LOOP ON (SELECT * FROM tbl) AS x BEGIN ... END
```

12.6.26. if statement ::=

- IF <lparen> <condition> <rparen> <statement>(ELSE <statement>)?

指定のカーソルで実行する手順ループステートメント。

例:

```
IF (boolVal) BEGIN variables.x = 1 END ELSE BEGIN variables.x = 2 END
```

12.6.27. declare statement ::=

- DECLARE (<data type> | EXCEPTION)<identifier>(<eq> <assignment statement operand>)?

変数を作成し、任意で値を割り当てる手順宣言ステートメント。

例:

```
DECLARE STRING x = 'a'
```

12.6.28. assignment statement ::=

- <identifier> <eq>（<assignment statement operand> |（<call statement>(WITH | WITHOUT) RETURN ）?）

手順の値に変数を割り当てます。

例:

```
x = 'b'
```

12.6.29. assignment statement operand ::=

- <insert statement>
- <update statement>
- <delete statement>
- <expression>
- <query expression>
- <SQL exception>

割り当てで使用可能な値またはコマンド。式を除くすべての割り当ては非推奨.{note} です。

12.6.30. data statement ::=

- (<directly executable statement> | <dynamic data statement>)（(WITH | WITHOUT) RETURN ）?

SQL ステートメントを実行する手順ステートメント。更新ステートメントでは、ROWCOUNT 変数により更新数にアクセスできます。

12.6.31. dynamic data statement ::=

- (EXECUTE | EXEC)(STRING | IMMEDIATE)? <expression>（ AS <typed element list> (INTO <identifier>)?）?(USING <set clause list>)?（ UPDATE (<signedinteger> | <star>)）?

任意の sql を実行できる手順ステートメント。

例:

```
EXECUTE IMMEDIATE 'SELECT * FROM tbl' AS x STRING INTO #temp
```

12.6.32. set clause list ::=

- <identifier> <eq> <expression>(<comma> <identifier> <eq> <expression>)*

値割り当ての一覧。

例:

```
col1 = 'x', col2 = 'y' ...
```

12.6.33. typed element list ::=

- *<identifier> <basic data type>(<comma> <identifier> <basic data type>)**

typed 要素の一覧。

例:

```
col1 string, col2 integer ...
```

12.6.34. callable statement ::=

- *<lbrace> (<qmark> <eq>)?CALL <identifier>（<lparen>（<named parameter list> | (<expression list>)?）<rparen>）? <rbrace>(<option clause>)?*

JDBC エスケープ構文を使用して定義される callable ステートメント。

例:

```
{? = CALL proc}
```

12.6.35. call statement ::=

- *((EXEC | EXECUTE | CALL)<identifier> <lparen>（<named parameter list> | (<expression list>)?）<rparen>）(<option clause>)?*

指定のパラメーターを使用して手順を実行します。

例:

```
CALL proc('a', 1)
```

-

12.6.36. named parameter list ::=

- （<identifier> <eq>(<gt>)? <expression>（<comma> <identifier> <eq>(<gt>)? <expression>）*）

名前付きパラメーターの一覧。

例:

```
param1 => 'x', param2 => 1
```

12.6.37. insert statement ::=

- (INSERT | MERGE | UPSERT) INTO <identifier>(<column list>)? <query expression> (<option clause>)?

指定のターゲットに値を挿入します。

例:

```
INSERT INTO tbl (col1, col2) VALUES ('a', 1)
```

12.6.38. expression list ::=

- <expression>(<comma> <expression>)*

式のリスト。

例:

```
col1, 'a', ...
```

12.6.39. update statement ::=

- UPDATE <identifier>(AS)? <identifier>）?SET <set clause list>(<where clause>)? (<option clause>)?

指定のターゲットの値を更新します。

例:

```
UPDATE tbl SET (col1 = 'a') WHERE col2 = 1
```

12.6.40. delete statement ::=

- DELETE FROM <identifier>（(AS)? <identifier>）?(<where clause>)?(<option clause>)?(< option clause >)?

指定のターゲットから行を削除します。

例:

```
DELETE FROM tbl WHERE col2 = 1
```

12.6.41. query expression ::=

- （& lt;with list element>(<comma> <with list element>)*）? <query expression body>

データの宣言的クエリー。

例:

```
SELECT * FROM tbl WHERE col2 = 1
```

12.6.42. with list element ::=

- <identifier>(<column list>)?AS <lparen> <query expression> <rparen>

エンクロージングクエリーで使用するクエリー式。

例:

```
X (Y, Z) AS (SELECT 1, 2)
```

12.6.43. query expression body ::=

- <query term>(UNION | EXCEPT)(ALL | DISTINCT)? <query term>）*(<order by clause>)?(<limit clause>)?(<option clause>)?

クエリー式のボディーはオプションで順序付けおよび制限できます。

例:

```
SELECT * FROM tbl ORDER BY col1 LIMIT 1
```

12.6.44. query term ::=

- <query primary>（ INTERSECT (ALL | DISTINCT)? <query primary>）*

INTERSECT の優先順位を確立するために使用されます。

例:

```
SELECT * FROM tbl
```

```
SELECT * FROM tbl1 INTERSECT SELECT * FROM tbl2
```

12.6.45. query primary ::=

- <query>
- (VALUES <lparen> <expression list> <rparen>(<comma> <lparen> <expression list> <rparen>)*)
- (TABLE <identifier>)
- (<lparen> <query expression body> <rparen>)

行の宣言的ソース。

例:

```
TABLE tbl
```

```
SELECT * FROM tbl1
```

12.6.46. query ::=

- <select clause>(<into clause>)?（<from 句>(<where clause>)?(<group by clause>)?(<having clause>)?）?

SELECT クエリー。

例:

```
SELECT col1, max(col2) FROM tbl GROUP BY col1
```

12.6.47. into clause ::=

- INTO <identifier>

クエリーをテーブルに転送するために使用されます。{note} これは非推奨です。代わりにクエリー式を付けて INSERT INTO を使用します。{note}

例:

```
INTO tbl
```

12.6.48. select clause ::=

- SELECT (ALL | DISTINCT)? (<star> | (<select sublist>(<comma> <select sublist>)*))

クエリーによって返される列。オプションで一意にできます。

例:

```
SELECT *
```

```
SELECT DISTINCT a, b, c
```

12.6.49. select sublist ::=

- <派生列の選択>
- <すべてのグループ>

select 句の要素

例:

```
tbl.*
```

```
tbl.col AS x
```

12.6.50. select derived column ::=

- (<expression>(AS)? <identifier>) ?)

単一の列を選択する select 句項目。{note} は AS キーワードは任意です。{note}

例:

```
tbl.col AS x
```

12.6.51. derived column ::=

- *（<expression>(AS <identifier>)?）*

任意の名前付き式。

例:

```
tbl.col AS x
```

12.6.52. all in group ::=

- *<すべてのグループ識別子>*

指定のグループからすべての列を選択できる選択サブリスト。

例:

```
tbl.*
```

12.6.53. ordered aggregate function ::=

- (XMLAGG | ARRAY_AGG | JSONARRAY_AGG)<lparen> <expression>(<order by clause>)? <rparen>

任意で順序付け可能な集約関数。

例:

```
XMLAGG(col1) ORDER BY col2
```

```
ARRAY_AGG(col1)
```

12.6.54. text aggreate function ::=

- TEXTAGG <lparen>(FOR)? <derived column>(<comma> <derived column>)*(DELIMITER <character>)?(QUOTE <character>)|(NO QUOTE))？（ヘッダー）?(ENCODING <identifier>)?(<order by clause>)? <rparen>

別個の値 clob を作成するための集約関数。

例:

```
TEXTAGG (col1 as t1, col2 as t2 DELIMITER ',' HEADER)
```

12.6.55. 標準の集約機能 ::=

- ((COUNT | COUNT_BIG) <lparen> <star> <rparen>)
- （（count | count_BIG | sum | avg | min | max | every | stdDEV_POP | stdDEV_SAMP | var_SAMP | var_POP | some | ANY ）<lparen>(DISTINCT | ALL)? <expression>

<rparen>)

標準の集約機能。

例:

```
COUNT(*)
```

12.6.56. analytic aggregate function ::=

- *(ROW_NUMBER | RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST)<lparen> <rparen>*

分析集計関数。

例:

```
ROW_NUMBER()
```

12.6.57. filter clause ::=

- *FILTER <lparen> WHERE <boolean primary> <rparen>*

値を調整する前に適用される集約フィルター句。

例:

```
FILTER (WHERE col1='a')
```

12.6.58. from clause ::=

- *FROM （<table reference>(<comma> <table reference>)*）*

テーブル参照の一覧が含まれる from 句のクエリー。

例:

```
FROM a, b
```

```
FROM a right outer join b, c, d join e".</p>
```

12.6.59. table reference ::=

- *(<escaped join> < jointable> <rbrace>)*
- *<join table>*

任意でエスケープしたテーブル。

例:

```
a
```

```
a inner join b
```

12.6.60. joined table ::=

- *<table primary>(<cross join> | <qualified table>)**

テーブルまたは結合。

例:

> *a*

> *a inner join b*

12.6.61. cross join ::=

- *(CROSS | UNION) JOIN <table primary>）*

クロス参加。

例:

> *a CROSS JOIN b*

12.6.62. qualified table ::=

- *（（RIGHT （外部）?）|（LEFT(OUTER)?）|（FULL (OUTER)?）| INNER ）?JOIN <table reference> ON <condition>）*

INNER または OUTER の参加。

例:

```
a inner join b
```

12.6.63. table primary ::=

- （<text table> | <array table> | <json table> | <xml table> | <object table> | <table name> | <table subquery> |(<lparen> < jointable> <rparen>)） (MAKEDEP <make dep options>)| MAKENOTDEP ）?(MAKEIND <make dep options>)）?

行の 1 つのソース。

例:

```
a
```

12.6.64. make dep options ::=

- （<lparen>(MAX <colon> <unsigned integer>)?(NO)?JOIN ）? <rparen>）?

make dep ヒントのオプション

例:

```
(min:10000)
```

12.6.65. xml serialize ::=

- XMLSERIALIZE <lparen>(DOCUMENT | CONTENT)? <expression>（ AS (STRING | VARCHAR | CLOB | VARBINARY | BLOB))?(ENCODING <identifier>)?(VERSION

<string>)?（(INCLUDING | EXCLUDING) XMLDECLARATION ）? <rparen>

XML 値をシリアライズします。

例:

```
XMLSERIALIZE(col1 AS CLOB)
```

12.6.66. array table ::=

- *ARRAYTABLE <lparen>(ROW | ROWS)? <value expression primary> COLUMNS <typed element list> <rparen>(AS)? <identifier>*

ARRAYTABLE テーブル関数はアレイから表形式の結果を作成します。ネストされたテーブルの参照として使用できます。

例:

```
ARRAYTABLE (col1 COLUMNS x STRING) AS y
```

12.6.67. json table ::=

- *JSONTABLE <lparen> <value expression primary> <comma> <string>（<comma>(TRUE | FALSE)）?COLUMNS <json table column>(<comma> <json table column>)* <rparen>(AS)? <identifier>*

JSONTABLE テーブル関数は、JSON から表形式の結果を作成します。ネストされたテーブルの参照として使用できます。

例:

```
JSONTABLE (col1, '$..book', false COLUMNS x STRING) AS y
```

12.6.68. json table column ::=

- <identifier>(FOR ORDINALITY)|（<basic data type>(PATH <string>)?）

JSON テーブルの列。

例:

```
col FOR ORDINALITY
```

12.6.69. text table ::=

- TEXTTABLE <lparen> <common value expression>(SELECTOR <string>)?COLUMNS <text table column>(<comma> <text table column>)*(NO ROW DELIMITER)|(ROW DELIMITER <character>)）?(DELIMITER <character>)?(ESCAPE <character>)|(QUOTE <character >)）?（ HEADER （未署名整数>）?）(SKIP <unsigned integer>)（ TRIM なし ）? <rparen>(AS)? <identifier>

TEXTTABLE テーブル関数は、テキストから表形式結果を作成します。ネストされたテーブルの参照として使用できます。

例:

```
TEXTTABLE (file COLUMNS x STRING) AS y
```

12.6.70. text table column ::=

-

<identifier>(FOR ORDINALITY)|（(HEADER <string>)? <basic data type>（ WIDTH <unsigned integer>(NO TRIM)?）?(SELECTOR <string> <unsigned integer>)?）

テキストテーブル列。

例:

```
x INTEGER WIDTH 6
```

12.6.71. xml query ::=

- XMLEXISTS <lparen>(<xml namespaces> <comma>)? <string>(PASSING <derived column>)(<comma> <derived column>)*）? <rparen>

XQuery を実行して XML 結果を返します。

例:

```
XMLQUERY('<a>...</a>' PASSING doc)
```

12.6.72. xml query ::=

- XMLQUERY <lparen>(<xml namespaces> <comma>)? <string>（ PASSING <derived column>(<comma> <derived column>)*）?（(NULL | EMPTY) ON EMPTY ）? <rparen>

XQuery を実行して XML 結果を返します。

例:

```
XMLQUERY('<a>...</a>' PASSING doc)
```

12.6.73. object table ::=

- OBJECTTABLE <lparen>(pid UAGE <string>)? <string>（ PASSING <derived column>(<comma> <derived column>)*）?COLUMNS <object table column>(<comma> <object table column>)* <rparen>(AS)? <identifier>

スクリプトを処理してテーブル結果を返します。

例:

```
OBJECTTABLE('z' PASSING val AS z COLUMNS col OBJECT 'teiid_row') AS X
```

12.6.74. object table column ::=

- <identifier> <basic data type> <string>(DEFAULT <expression>)?

オブジェクトテーブル列。

例:

```
y integer 'teiid_row_number'
```

12.6.75. xml table ::=

- XMLTABLE <lparen>(<xml namespaces> <comma>)? <string>（ PASSING <derived column>(<comma> <derived column>)*）？（ COLUMNS <xml table column>(<comma>

<xml table column>)）? <rparen>(AS)? <identifier>*

XQuery を処理して、テーブルの結果を返します。

例:

```
XMLTABLE('/a/b' PASSING doc COLUMNS col XML PATH '.') AS X
```

12.6.76. xml table column ::=

- *<identifier>(FOR ORDINALITY)|（<basic data type>(DEFAULT <expression>)?(PATH <string>)?）*

XML テーブルの列。

例:

```
y FOR ORDINALITY
```

12.6.77. unsigned integer ::=

- *< 署名されていない整数リテラル >*

署名されていないインタージェリー値。

例:

```
12345
```

12.6.78. table subquery ::=

- (TABLE | LATERAL)? <lparen>(<query expression> | <call statement>)<rparen>(AS)? <identifier>

サブクエリーで定義されるテーブル。

例:

```
(SELECT * FROM tbl) AS x
```

12.6.79. table name ::=

- (<identifier>(AS)? <identifier>) ?)

FROM 句で名前が付けられたテーブル。

例:

```
tbl AS x
```

12.6.80. where clause ::=

- WHERE <condition>

検索条件を指定します。

例:

```
WHERE x = 'a'
```

12.6.81. condition ::=

- *<boolean value expression>*

ブール式。

12.6.82. boolean value expression ::=

- *<boolean term>（ または <boolean term>）**

任意で ORed のブール値式。

12.6.83. boolean term ::=

- *<boolean factor>(AND <boolean factor>)**

オプションの AND ブール値係数。

12.6.84. boolean factor ::=

- *( )? <boolean primary>*

ブール値係数。

例:

```
NOT x = 'a'
```

12.6.85. boolean primary ::=

- *(<common value expression>(<between predicate> | <match predicate> | <like regex predicate> | <in predicate> | <is null predicate> | <quantified comparison predicate> | <comparison predicate> | <is distinct>)?)*

- *<exists predicate>*

- *<xml query>*

ブール値の述語または単純な式。

例:

```
col LIKE 'a%'
```

12.6.86. comparison operator ::=

- *<eq>*

- *<ne>*
- *<ne2>*
- *<lt>*
- *<le>*
- *<gt>*
- *<ge>*

比較演算子。

例:

```
=
```

12.6.87. is distinct *::=*

- *は （ ではない ）?DISTINCT FROM <common value expression>*

Is Distinct Right Hand Side

例:

```
IS DISTINCT FROM expression
```

12.6.88. comparison predicate ::=

- <comparison operator> <common value expression>

値の比較

例:

```
= 'a'
```

12.6.89. subquery ::=

- <lparen>(<query expression> | <call statement>)<rparen>

サブクエリー。

例:

```
(SELECT * FROM tbl)
```

12.6.90. quantified comparison predicate ::=

- <comparison operator>(ANY | SOME | ALL) (<subquery> |(<lparen> <expression> <rparen>))

サブクエリーの比較。

例:

```
= ANY (SELECT col FROM tbl)
```

12.6.91. match predicate ::=

- （は）?（LIKE |(SIMILAR TO)<common value expression>（ESCAPE <character> | (<lbrace> ESCAPE <character> <rbrace>)）?

パターンに基づいて照合します。

例:

```
LIKE 'a_'
```

12.6.92. like regex predicate ::=

- （は）?LIKE_REGEX <共通値式>

正規表現の一致。

例:

```
LIKE_REGEX 'a.*b'
```

12.6.93. character ::=

- *<文字列>*

1 文字

例:

```
'a'
```

12.6.94. between predicate ::=

- *（は）?BETWEEN <common value expression> AND <common value expression>*

2 つの値の比較

例:

```
BETWEEN 1 AND 5
```

12.6.95. is null predicate ::=

- *は （ではない）?NULL*

null テスト。

例:

```
IS NOT NULL
```

12.6.96. in predicate ::=

- (は) ?IN （<subquery> | （<lparen> <common value expression>(<comma> <common value expression>)* <rparen>）)

複数の値の比較。

例:

```
IN (1, 5)
```

12.6.97. exists predicate ::=

- EXISTS <subquery>

行が存在する場合のテスト。

例:

```
EXISTS (SELECT col FROM tbl)
```

12.6.98. group by clause ::=

- GROUP BY (ROLLUP <lparen> <expression list> <rparen> | <expression list>)

グループ化列を定義します。

例:

```
GROUP BY col1, col2
```

12.6.99. having clause ::=

- *HAVING* <*condition*>

グループ化後に適用される検索条件。

例:

```
HAVING max(col1) = 5
```

12.6.100. order by clause ::=

- *ORDER BY* <*sort specification*>(<*comma*> <*sort specification*>)*

行の順序を指定します。

例:

```
ORDER BY x, y DESC
```

12.6.101. sort specification ::=

- <sort key>(ASC | DESC)?（ NULL（ 最初 | 最後 の ））?

特定の式でソートする方法を定義します。

例:

```
col1 NULLS FIRST
```

12.6.102. sort key ::=

- <expression>

ソート式。

例:

```
col1
```

12.6.103. integer parameter ::=

- <署名なし整数>
- <unsigned value expression primary>

整数または整数へのパラメーター参照。

例:

```
?
```

12.6.104. limit clause ::=

- （ LIMIT <integer parameter>(<comma> <integer parameter>)|(OFFSET <integer parameter>)）?）
- （ OFFSET <integer parameter>(ROW | ROWS)(<fetch clause>)?）
- <fetch 句>

結果の行の制限やオフセット。

例:

```
LIMIT 2
```

12.6.105. fetch clause ::=

- FETCH (FIRST | NEXT)(<integer parameter>)?（ 行 | 行 ） のみ

ANSI 制限。

例:

```
FETCH FIRST 1 ROWS ONLY
```

12.6.106. option clause ::=

- オプション（ MAKEDEP <identifier> <make dep options>(<comma> <identifier> <make dep options>)* | MAKEIND <identifier> <make dep options>（<comma> ??? <identifier> <make dep options>）* | MAKENOTDEP <identifier>(<comma> <identifier>)* | NOCACHE（<identifier>(<comma> <identifier>)*）？）*

クエリーオプションを指定します。

例:

```
OPTION MAKEDEP tbl
```

12.6.107. expression ::=

- <condition>

値。

例:

```
col1
```

12.6.108. common value expression ::=

- （<numeric value expression>(<double_amp_op> | <concat_op>)<numeric value expression>）*）

concat の優先順位を確立します。

例:

```
'a' || 'b'
```

12.6.109. numeric value expression ::=

- *(<term>(<plus or minus> <term>)*)*

例:

```
1 + 2
```

12.6.110. plus or minus ::=

- *<plus>*
- *<minus>*

+ または - 演算子。

例:

```
+
```

12.6.111. term ::=

- (<value expression primary>(<star or slash> <value expression primary>)*)

数値用語

例:

```
1 * 2
```

12.6.112. star or slash ::=

- <star>
- <スラッシュ>

* または / 演算子。

例:

```
/
```

12.6.113. value expression primary ::=

- <数値以外のリテラル >
- (<plus or minus>)? (<署名されていない数値リテラル > |(<lsbrace> <numeric value expression> <rsbrace>)*)

単純な値式。

例:

```
+col1
```

12.6.114. parameter reference ::=

- *<qmark>*
- *（<ドル> <署名なし整数>）*

後でバインドするパラメーター参照。

例:

```
?
```

12.6.115. unescapedFunction ::=

- *((<text aggreate function> | <standard aggregate function> | <ordered aggregate function>)(<filter clause>)?(<window specification>)?）|（<analytic aggregate function> (<filter clause>)? <window specification>）|（<function>(<window specification>)?）*
- *(XMLCAST <lparen> <expression> AS <data type> <rparen>)*

12.6.116. nested expression ::=

- （<lparen>（<expression>(<comma> <expression>)*）?(<comma>)? <rparen>）

parens でネストされた式

例:

> (1)

12.6.117. unsigned value expression primary ::=

- <パラメーター参照>
- (<escaped function> <function> <rbrace>)
- <unescapedFunction>
- <identifier> | < <non-reserved identifier>
- <subquery>
- <nested expression>
- <ARRAY expression constructor>
- <search case expression>

- *<case expression>*

署名のない単純な値式。

例:

```
col1
```

12.6.118. ARRAY expression constructor ::=

- *ARRAY （（<lsbrace>(<expression> < expression > <expression>)*）? <rsbrace>）| (<lparen> <query expression> <rparen>)）???*

指定の式の作成および配列を作成します。

例:

```
ARRAY[1,2]
```

12.6.119. window specification ::=

- *OVER <lparen>(PARTITION BY <expression list>)?(<order by clause>)?(<window frame>)? <rparen>*

分析またはウィンドウされた集約関数のウィンドウ仕様。

例:

```
OVER (PARTION BY col1)
```

-

12.6.120. window frame ::=

- (RANGE | ROWS)(BETWEEN <window frame bound> AND <window frame bound>)| <window frame bound>)

ウィンドウフレームのモード、開始、およびオプションで終了する

例:

```
RANGE UNBOUNDED PRECEDING
```

12.6.121. window frame bound ::=

- (UNBOUNDED | <signedinteger>)(FOLLOWING | PRECEDING))
- (現在 の行)

ウィンドウフレームの開始または終了を定義します。

例:

```
CURRENT ROW
```

12.6.122. case expression ::=

- CASE <expression>(WHEN <expression> THEN <expression>)+(ELSE

<expression>)?END

共通の検索の事前使用を使用した if/then/else チェーン。

例:

```
CASE col1 WHEN 'a' THEN 1 ELSE 2
```

12.6.123. searched case expression ::=

- *CASE (WHEN <condition> THEN <expression>)+(ELSE <expression>)?END*

複数の検索条件を使用した if/then/else チェーン。

例:

```
CASE WHEN x = 'a' THEN 1 WHEN y = 'b' THEN 2
```

12.6.124. function ::=

- *(CONVERT <lparen> <expression> <comma> <data type> <rparen>)*
- *(CAST <lparen> <expression> AS <data type> <rparen>)*
- *(SUBSTRING <lparen> <expression> （ （ FROM <expression>(FOR <expression>)?）|(<comma> <expression list>)） <rparen>）*
- *(EXTRACT <lparen>(YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | QUARTER | EPOCH) FROM <expression> <rparen>）*

- (TRIM <lparen> (((LEADING | TRAILING | BOTH)(<expression>)?) | <expression>) FROM) ? <expression> <rparen>)
- ((TO_CHARS | TO_BYTES)<lparen> <expression> <comma> <string>(<comma> <expression>)? <rparen>)
- ((TIMESTAMPADD | TIMESTAMPDIFF)<lparen> <time interval> <comma> <expression> <comma> <expression> <rparen>)
- <queryString function>
- ((LEFT | RIGHT | CHAR | USER | YEAR | MONTH | HOUR | MINUTE | SECOND | XMLCONCAT | XMLCOMMENT | XMLTEXT)<lparen>(<expression list>)? <rparen>)
- ((TRANSLATE | INSERT)<lparen>(<expression list>)? <rparen>)
- <xml parse>
- <xml element>
- (XMLPI <lparen> ((NAME)? <identifier>) (<comma> <expression>)? <rparen>)
- <xml forest>
- <json object>
- <xml serialize>
- <xml query>

- (POSITION <lparen> <common value expression> IN <common value expression> <rparen>)

- (LISTAGG <lparen> <expression>(<comma> <string>)? <rparen> WITHIN GROUP <lparen> <order by clause> <rparen>)

- (<identifier> <lparen>(ALL | DISTINCT)?(<expression list>)?(<order by clause>)? <rparen>(<filter clause>)?)

- (CURRENT_DATE (<lparen> <rparen>)?)

- ((CURRENT_TIMESTAMP | CURRENT_TIME)(<lparen> <unsigned integer> <rparen>)?)

scalar 関数を呼び出します。

例:

```
func('1', col1)
```

12.6.125. xml parse ::=

- XMLPARSE <lparen>(DOCUMENT | CONTENT)<expression>(WELLFORMED)? <rparen>

指定された値を XML として解析します。

例:

```
XMLPARSE(DOCUMENT doc WELLFORMED)
```

12.6.126. querystring function ::=

- *QUERYSTRING <lparen> <expression>(<comma> <derived column>)* <rparen>*

指定の引数から URL クエリー文字列を生成します。

例:

```
QUERYSTRING('path', col1 AS opt, col2 AS val)
```

12.6.127. xml element ::=

- *XMLELEMENT <lparen>（(NAME)? <identifier>）(<comma> <xml namespaces>)? (<comma> <xml attributes>)?(<comma> <expression>)* <rparen>*

XML 要素を作成します。

例:

```
XMLELEMENT(NAME "root", child)
```

12.6.128. xml attributes ::=

- *XMLATTRIBUTES <lparen> <derived column>(<comma> <derived column>)* <rparen>*

含まれる要素の属性を作成します。

例:

```
XMLATTRIBUTES(col1 AS attr1, col2 AS attr2)
```

12.6.129. json object ::=

- *JSONOBJECT <lparen> <derived column list> <rparen>*

名前と値のペアを含む JSON オブジェクトを生成します。

例:

```
JSONOBJECT(col1 AS val1, col2 AS val2)
```

12.6.130. derived column list ::=

- *<derived column>(<comma> <derived column>)**

名前と値のペアの一覧

例:

```
col1 AS val1, col2 AS val2
```

12.6.131. xml forest ::=

- *XMLFOREST <lparen>(<xml namespaces> <comma>)? <derived column list> <rparen>*

派生した各列の要素を生成します。

例:

```
XMLFOREST(col1 AS ELEM1, col2 AS ELEM2)
```

12.6.132. xml namespaces ::=

- *XMLNAMESPACES <lparen> <xml namespace element>(<comma> <xml namespace element>)* <rparen>*

XML 名前空間の URI/プレフィックスの組み合わせを定義します。

例:

```
XMLNAMESPACES('http://foo' AS foo)
```

12.6.133. xml namespace element ::=

- *(<string> AS <identifier>)*
- *(デフォルト なし)*
- *(DEFAULT <string>)*

xml 名前空間

例:

```
NO DEFAULT
```

12.6.134. simple data type ::=

- *(STRING (<lparen> <unsigned integer> <rparen>)?)*
- *(VARCHAR (<lparen> <unsigned integer> <rparen>)?)*
- *BOOLEAN*
- *BYTE*
- *TINYINT*
- *SHORT*
- *SMALLINT*
- *(CHAR (<lparen> <unsigned integer> <rparen>)?)*
- *INTEGER*
- *LONG*

- *BIGINT*
- *(Big INTEGER (<lparen> <unsigned integer> <rparen>)?)*
- *FLOAT*
- *REAL*
- *DOUBLE*
- *(BIGDECIMAL (<lparen> <unsigned integer>(<comma> <unsigned integer>)? <rparen>) ?)*
- *(DECIMAL (<lparen> <unsigned integer>(<comma> <unsigned integer>)? <rparen>) ?)*
- *DATE*
- *TIME*
- *(TIMESTAMP (<lparen> <unsigned integer> <rparen>)?)*
- *(OBJECT (<lparen> <unsigned integer> <rparen>)?)*
- *(BLOB (<lparen> <unsigned integer> <rparen>)?)*
- *(CLOB (<lparen> <unsigned integer> <rparen>)?)*
- *JSON*

- (VARBINARY (<lparen> <unsigned integer> <rparen>)?)
- ジオメトリー
- GEOGRAPHY
- XML

コレクションでないデータ型。

例:

```
STRING
```

12.6.135. basic data type ::=

- <simple data type>(<lsbrace> <rsbrace>)*

データタイプ。

例:

```
STRING[]
```

12.6.136. data type ::=

- <basic data type>

- *((<identifier> | <basicNonReserved>) (<lsbrace> <rsbrace>)*)*

データタイプ。

例:

```
STRING[]
```

12.6.137. time interval ::=

- *SQL_TSI_FRAC_SECOND*
- *SQL_TSI_SECOND*
- *SQL_TSI_MINUTE*
- *SQL_TSI_HOUR*
- *SQL_TSI_DAY*
- *SQL_TSI_WEEK*
- *SQL_TSI_MONTH*
- *SQL_TSI_QUARTER*
- *SQL_TSI_YEAR*

時間間隔のキーワード。

例:

```
SQL_TSI_HOUR
```

12.6.138. non numeric literal ::=

- *<文字列>*
- *<バイナリー文字列リテラル >*
- *FALSE*
- *TRUE*
- *UNKNOWN*
- *NULL*
- *(<escaped type> <string> <rbrace>)*
- *（(DATE | TIME | TIMESTAMP)<string>）*

エスケープまたは単純な数値以外のリテラル。

例:

> *'a'*

12.6.139. unsigned numeric literal ::=

- *< 署名されていない整数リテラル >*
- *<approximate numeric literal >*
- *<decimal numeric literal >*

未署名の数値リテラル値。

例:

> *1.234*

12.6.140. ddl statement ::=

- *<テーブルの作成 >(<create table> | <create procedure>)?*
- *<オプションの namespace>*
- *<alterStatement>*
- *<作成トリガー>*
- *<ドメインまたはタイプエイリアスの作成>*

- <create server>
- < ロールの作成 >
- < ドロップ role >
- <GRANT>
- <revoke GRANT>
- <ポリシーの作成 >
- <DROP POLICY >
- <ドロップ server>
- <drop table>
- <import External schema>
- <別のデータベースのインポート>
- <データベースの作成>
- <use database>
- <ドロップスキーマ>

- *<set schema>*
- *<Create schema>*
- *<create procedure>(<ddl statement>)?*
- *<データラッパーの作成>*
- *<drop data wrapper>*
- *<drop procedure>*

データ定義ステートメント。

例:

```
CREATE FOREIGN TABLE X (Y STRING)
```

12.6.141. option namespace ::=

- *SET NAMESPACE <string> AS <identifier>*

オプションキーの完全な名前を短くするために使用される namespace。

例:

```
SET NAMESPACE 'http://foo' AS foo
```

12.6.142. create database ::=

- *CREATE DATABASE <identifier>(VERSION <string>)?(<options clause>)?*

新規データベースの作成

例:

```
CREATE DATABASE foo OPTIONS(x 'y')
```

12.6.143. use database ::=

- *USE DATABASE <identifier>(VERSION <string>)?*

作業コンテキストへのデータベース

例:

```
USE DATABASE foo
```

12.6.144. create schema ::=

- *CREATE （仮想 | 外部）?SCHEMA <identifier>(SERVER <identifier list>)?（<options 句>）?*

データベースでのスキーマの作成

例:

```
CREATE VIRTUAL SCHEMA foo SERVER (s1,s2,s3);
```

12.6.145. drop schema ::=

- ドロップ （ 仮想 | 外部 ）?SCHEMA <identifier>

データベースのスキーマのドロップ

例:

```
DROP SCHEMA foo
```

12.6.146. set schema ::=

- SET SCHEMA <identifier>

後続の ddl ステートメントにスキーマを設定

例:

```
SET SCHEMA foo
```

12.6.147. create a domain or type alias ::=

- *CREATE DOMAIN (<identifier> | <basicNonReserved>)? <data type>(NOT NULL)?*

オプションの制約を使用して名前付きタイプを作成します。

例:

```
CREATE DOMAIN my_type AS INTEGER NOT NULL
```

12.6.148. create data wrapper ::=

- *CREATE FOREIGN （(DATA WRAPPER)| TRANSLATOR ）<Unqualified identifier>（(TYPE | HANDLER)<identifier>）?(<options clause>)?*

トランスレーターを定義します。オプションを使用してトランスレータープロパティーを上書きします。

例:

```
CREATE FOREIGN DATA WRAPPER wrapper OPTIONS (x true)
```

12.6.149. Drop data wrapper ::=

- *DROP FOREIGN （(DATA WRAPPER)| TRANSLATOR ）<identifier>*

トランスレーターを削除します。

例:

```
DROP FOREIGN DATA WRAPPER wrapper
```

-

12.6.150. create role ::=

- *CREATE ROLE <Unqualified identifier>(WITH <with role>)?*

データベースのデータロールを定義します。

例:

```
CREATE ROLE lowly WITH FOREIGN ROLE "role"
```

12.6.151. with role ::=

- *(認証 済み)*
- *(JAAS | FOREIGN) ROLE <identifier list>)*

12.6.152. drop role ::=

- *DROP ROLE <identifier>*

データベースのデータロールを削除します。

例:

```
DROP ROLE <data-role>
```

12.6.153. CREATE POLICY ::=

- *CREATE POLICY <identifier> ON （<identifier>（ FOR （ ALL |(SELECT | INSERT | UPDATE | DELETE)（<comma>(SELECT | INSERT | UPDATE) | DELETE ））*）））?）|（ PROCEDURE <identifier>(FOR ALL)?））TO <identifier> USING <lparen> <boolean primary> <rparen>*

CREATE row level policy

例:

```
CREATE POLICY pname ON tbl FOR SELECT,INSERT TO role USING col = user();
```

12.6.154. DROP POLICY ::=

- *DROP POLICY <identifier> ON （<identifier> |(PROCEDURE <identifier>))TO <identifier>*

DROP 行レベルのポリシー

例:

```
----DROP POLICY pname ON tbl TO role
----
```

12.6.155. GRANT ::=

- *GRANT （（<grant type>(<comma> <grant type>)*）?ON （ TABLE <identifier>(CONDITION)(NOT)?CONSTRAINT ）? <string>）? | FUNCTION <identifier> |*

PROCEDURE <identifier>（ CONDITION (NOT)?CONSTRAINT ）? <string>）? | SCHEMA <identifier> | COLUMN <identifier>（ MASK (ORDER <unsigned integer>)? <string>）?）|（ ALL PRIVILEGES ）|(TEMPORARY TABLE)|(USAGE ON gitops UAGE <identifier>)） TO <identifier>

ロールの GRANT を定義します。

例:

```
GRANT SELECT ON TABLE x.y TO role
```

12.6.156. Revoke GRANT ::=

- REVOKE （（<grant type>(<comma> <grant type>)*）?ON （ TABLE <identifier>(CONDITION)? | FUNCTION <identifier> | PROCEDURE <identifier>(CONDITION)? | SCHEMA <identifier> | COLUMN <identifier>(MASK)?）） |(ALL PRIVILEGES) |(TEMPORARY TABLE) |(US AGE ON gitops UAGE < identifier >)） FROM <identifier>

ロールの GRANT の取り消し

例:

```
REVOKE SELECT ON TABLE x.y TO role
```

12.6.157. create server ::=

- CREATE SERVER <Unqualified identifier>(TYPE <string>)?(VERSION <string>)?FOREIGN （(DATA WRAPPER)| TRANSLATOR ）<Unqualified identifier> (<options clause>)?

ソースへの接続を定義します。

例:

```
CREATE SERVER "h2-connector" FOREIGN DATA WRAPPER h2 OPTIONS ("resource-name"
'java:/accounts-ds');
```

12.6.158. drop server ::=

- DROP SERVER <identifier>

外部ソースへの接続の削除を定義します。

例:

```
DROP SERVER server_name
```

12.6.159. create procedure ::=

- CREATE （仮想 | 外部 ）?(PROCEDURE | FUNCTION)<Unqualified identifier>（<lparen>(<procedure parameter> <comma> <procedure parameter>)*）? <rparen>(RETURNS) （<options 句>）?（(TABLE)? <lparen> <procedure result column>(<comma> <procedure result column>)* <rparen>）| <data type>）） ? （<options 句>）?(AS <statement>)?）

手順または関数呼び出しを定義します。

例:

```
CREATE FOREIGN PROCEDURE proc (param STRING) RETURNS STRING
```

12.6.160. drop procedure ::=

- *ドロップ （ 仮想 | 外部 ）?(PROCEDURE | FUNCTION)<identifier>*

テーブルまたはビューを破棄します。

例:

```
DROP FOREIGN TABLE table-name
```

12.6.161. procedure parameter ::=

- *(IN | OUT | INOUT | VARIADIC)? <identifier> <data type>(NOT NULL)?（結果 ）?(DEFAULT <expression>)?（<options 句>）?*

手順または関数パラメーター

例:

```
OUT x INTEGER
```

12.6.162. procedure result column ::=

- *<identifier> <data type> （ NULL ではない）？（<options 句>）?*

手順結果の列

例:

-

```
x INTEGER
```

12.6.163. create table ::=

- *CREATE* (<*create view*> | <*create foreign or global temporary table*>)

テーブルまたはビューを定義します。

例:

```
CREATE VIEW vw AS SELECT 1
```

12.6.164. create foreign or global temporary table ::=

- ((*FOREIGN TABLE*)|(*GLOBAL TEMPORARY TABLE*)) <*Unqualified identifier*> <*create table body*>

外部またはグローバルな一時テーブルを定義します。

例:

```
FOREIGN TABLE ft (col integer)
```

12.6.165. create view ::=

- (*仮想*) ?*VIEW* <*Unqualified identifier*> (<*create view body*> |(<*options clause*>)?) *AS* <*query expression*>

ビューを定義します。

例:

```
VIEW vw AS SELECT 1
```

12.6.166. drop table ::=

- ドロップ （ 外部テーブル ） | （ 仮想 ）?VIEW ）|(GLOBAL TEMPORARY TABLE) <identifier>

テーブルまたはビューを破棄します。

例:

```
DROP VIEW name
```

12.6.167. create foreign temp table ::=

- CREATE (LOCAL)?FOREIGN TEMPORARY TABLE <Unqualified identifier> <create table body> ON <identifier>

外部一時テーブルを定義します。

例:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

12.6.168. create table body ::=

- *<lparen> <table element> (<comma>(<table constraint> | <table element>)* <rparen> (<options clause>)?*

テーブルを定義します。

例:

```
(x string) OPTIONS (CARDINALITY 100)
```

12.6.169. create view body ::=

- *<lparen> <view element> (<comma>(<table constraint> | <view element>)* <rparen> (<options clause>)?*

ビューを定義します。

例:

```
(x) OPTIONS (CARDINALITY 100)
```

12.6.170. table constraint ::=

- *(CONSTRAINT <identifier>)?(<primary key> | <other constraints> | <foreign key>) (<options clause>)?*

テーブルまたはビューで制約を定義します。

例:

```
FOREIGN KEY (a, b) REFERENCES tbl (x, y)
```

12.6.171. foreign key ::=

- *FOREIGN KEY <column list> REFERENCES <identifier>(<column list>)?*

外部キーの参照制約を定義します。

例:

```
FOREIGN KEY (a, b) REFERENCES tbl (x, y)
```

12.6.172. primary key ::=

- *PRIMARY KEY <column list>*

プライマリーキーを定義します。

例:

```
PRIMARY KEY (a, b)
```

12.6.173. other constraints ::=

- (UNIQUE | ACCESSPATTERN)<column list>）
- (INDEX <lparen> <expression list> <rparen>)

ACCESSPATTERN および UNIQUE 制約および INDEXes を定義します。

例:

```
UNIQUE (a)
```

12.6.174. column list ::=

- <lparen> <identifier> (<comma> <identifier>)* <rparen>

列名のリスト。

例:

```
(a, b)
```

12.6.175. table element ::=

- <identifier>（ SERIAL |（<data type>（ NULL ではない ）?(AUTO_INCREMENT)?）<post create column>

テーブル列を定義します。

例:

```
x INTEGER NOT NULL
```

12.6.176. view element ::=

- <identifier>（ SERIAL |（<data type>（ NULL ではない ）?(AUTO_INCREMENT)?）? <post create column>

任意のタイプでビュー列を定義します。

例:

```
x INTEGER NOT NULL
```

12.6.177. post create column ::=

- (<inline constraint>)?(DEFAULT <expression>)?（<options 句>）?

列の最後に一般的なオプション

例:

```
PRIMARY KEY
```

12.6.178. inline constraint ::=

- (プライマリー キー)
- UNIQUE
- INDEX

1 つの列で制約を定義します。

例:

```
x INTEGER PRIMARY KEY
```

12.6.179. options clause ::=

- OPTIONS <lparen> <option pair>(<comma> <option pair>)* <rparen>

ステートメントオプションの一覧。

例:

```
OPTIONS ('x' 'y', 'a' 'b')
```

12.6.180. option pair ::=

- <identifier>（数値以外のリテラル > |（<plus または minus>）? <署名されていない数値リテラル >)

オプションのキーと値のペア。

例:

```
'key' 'value'
```

12.6.181. alter option pair ::=

- *<identifier>（数値以外のリテラル > |（<plus または minus>）? <署名されていない数値リテラル >）*

An オプションのキー/値のペアを変更します。

例:

```
'key' 'value'
```

12.6.182. alterStatement ::=

- *MODIFY TABLE > | < CHANGE PROCEDURE > | < ALTER TRIGGER > | < ALTER SERVER > | < ALTER DATA WRAPPER > | < ALTER DATABASE&GT ;）*

12.6.183. ALTER TABLE ::=

- *（（仮想 ）?VIEW <identifier>）|(FOREIGN)?TABLE <identifier>）(AS <query expression>)| <ADD column> | <ADD constraint> | <alter options list> | <DROP column> |（ ALTER COLUMN <alter column options> ）|（ RENAME Table> |(COLUMN <rename column options>)））*

データベースのオプションの変更

例:

```
ALTER TABLE foo ADD COLUMN x xml
```

12.6.184. RENAME Table ::=

- *TO* <*identifier*>

テーブル名の変更

例:

```
ALTER TABLE foo RENAME TO BAR;
```

12.6.185. ADD constraint ::=

- *ADD* <*table constraint*>

テーブルを変更し、制約を追加します。

例:

```
ADD PRIMARY KEY (ID)
```

12.6.186. ADD column ::=

- *ADD COLUMN* <*table element*>

テーブルを変更して列を追加する

例:

```
ADD COLUMN bar type OPTIONS (ADD updatable true)
```

12.6.187. DROP column ::=

- *DROP COLUMN* <*identifier*>

テーブルを変更して列を追加する

例:

```
DROP COLUMN bar
```

12.6.188. alter column options ::=

- <*identifier*> （ *TYPE* （ *SERIAL* | （<*data type*>(*NOT NULL*)?(*AUTO_INCREMENT*)?） | <*alter child options list*>）

列オプションのセットを変更する

例:

-

```
ALTER COLUMN bar OPTIONS (ADD updatable true)
```

12.6.189. rename column options ::=

- <identifier> TO <identifier>

テーブル列または手順のパラメーター名のいずれかの名前を変更します。

例:

```
RENAME COLUMN bar TO foo
```

12.6.190. ALTER PROCEDURE ::=

- （仮想 | 外部）?PROCEDURE <identifier>(AS <statement>)| <alter options list> |(ALTER PARAMETER <alter column options>)|(RENAME PARAMETER <rename column options>))

データベースのオプションの変更

例:

```
ALTER PROCEDURE foo OPTIONS (ADD x y)
```

12.6.191. ALTER TRIGGER ::=

- TRIGGER ON <identifier> INSTEAD OF (INSERT | UPDATE | DELETE)(AS <for each row trigger action> | ENABLED | DISABLED)

テーブルトリガーのオプションの変更

例:

```
ALTER TRIGGER ON vw INSTEAD OF INSERT ENABLED
```

12.6.192. ALTER SERVER ::=

- SERVER <identifier> <alter options list>

データベースのオプションの変更

例:

```
ALTER SERVER foo OPTIONS (ADD x y)
```

12.6.193. ALTER DATA WRAPPER ::=

- (DATA WRAPPER)| TRANSLATOR ） <identifier> <alter options list>

データラッパーのオプションを変更する

例:

```
ALTER DATA WRAPPER foo OPTIONS (ADD x y)
```

12.6.194. ALTER DATABASE ::=

- *DATABASE* <*identifier*> <*alter options list*>

データベースのオプションの変更

例:

```
ALTER DATABASE foo OPTIONS (ADD x y)
```

12.6.195. alter options list ::=

- *OPTIONS* <*lparen*>(<*add set option*> | <*drop option*>)（<*comma*>(<*add set option*> | <*drop option*>)）* <*rparen*>

オプションの変更一覧

例:

```
OPTIONS (ADD updatable true)
```

12.6.196. drop option ::=

- *DROP* <*identifier*>

drop オプション

例:

-

```
DROP updatable
```

12.6.197. add set option ::=

- (ADD | SET)<alter option pair>

オプションペアの追加または設定

例:

```
ADD updatable true
```

12.6.198. alter child options list ::=

- OPTIONS <lparen>(<add set child option> | <drop option>)（<comma>(<add set child option> | <drop option>)）* <rparen>

オプションの変更一覧

例:

```
OPTIONS (ADD updatable true)
```

12.6.199. drop option ::=

- DROP <identifier>

drop オプション

例:

```
DROP updatable
```

12.6.200. add set child option ::=

- (ADD | SET)<alter child option pair>

オプションペアの追加または設定

例:

```
ADD updatable true
```

12.6.201. alter child option pair ::=

- <identifier>（数値以外のリテラル > |（<plus または minus>）? <署名されていない数値リテラル >）

An オプションのキー/値のペアを変更します。

例:

```
'key' 'value'
```

12.6.202. Import foreign schema ::=

- *IMPORT (FOREIGN SCHEMA <identifier>)?FROM (SERVER | REPOSITORY) <identifier> INTO <identifier>(<options clause>)?*

サーバーからスキーマメタデータをインポート

例:

> *IMPORT FOREIGN SCHEMA foo FROM SERVER bar*

12.6.203. Import another Database ::=

- *IMPORT DATABASE <identifier> VERSION <string&gt ; (ACCESS CONTROL)?*

別のデータベースを現在のデータベースにインポート

例:

> *IMPORT DATABASE vdb VERSION '1.2.3' WITH ACCESS CONTROL]*

12.6.204. identifier list ::=

- *<identifier> (<comma> <identifier>)**

12.6.205. grant type ::=

- *SELECT*
- *INSERT*
- *UPDATE*
- *DELETE*
- *実行*
- *ALTER*
- *DROP*