Alibaba Cloud
大数据#算服#

ユーザーガイド

Document Version20200326

目次

1 データ型..... 1
2 データ型..... 2
3 共通コマンド..... 7
3.1 共通コマンド一覧..... 7
3.2 プロジェクトの操作..... 8
3.3 テーブルの操作..... 11
3.4 インスタンス..... 16
3.5 リソース..... 20
3.6 関数..... 22
3.7 Set 操作..... 24
3.8 その他の操作..... 25
4 データのアップロードとダウンロード..... 27
4.1 データのアップロードとダウンロード..... 27
4.2 クラウド データの移行..... 28
4.3 データのアップロードおよびダウンロード ツール..... 28
4.4 Tunnel SDK..... 31
4.4.1 概要..... 31
4.4.2 TableTunnel..... 32
4.4.3 UploadSession..... 34
4.4.4 DownloadSession..... 36
4.4.5 TunnelBufferedWriter..... 38
4.5 大量データチャネル SDK の例..... 39
4.5.1 例..... 39
4.5.2 アップロード例..... 39
4.5.3 ##下#示例..... 41
4.5.4 マルチスレッドアップロードの例..... 43
4.5.5 マルチスレッドダウンロードの例..... 46
4.5.6 BufferedWriter アップロードの例..... 49
4.5.7 BufferedWriter マルチスレッドアップロードの例..... 49
4.6 Data Integration を使用したデータのインポートまたはエクスポート..... 50
5 SQL..... 55
5.1 SQL 概要..... 55
5.2 SQL の制限..... 56
5.3 他の SQL 構文との相違点..... 58
5.4 演算子..... 60
5.5 型変換..... 63
5.6 DDL SQL..... 69
5.6.1 テーブル操作..... 69
5.6.2 テーブルのライフサイクル..... 76
5.6.3 列とパーティションの操作..... 77

5.6.4 ビュー操作 80
5.7 Insert 操作 82
5.7.1 INSERT OVERWRITE と INSERT INTO 82
5.7.2 MULTI INSERT 84
5.7.3 動的パーティション 85
5.7.4 VALUES 87
5.8 側面図 90
5.9 Select 操作 93
5.9.1 SELECT 構文の説明 93
5.9.2 SELECT 実行順序 97
5.9.3 サブクエリ 98
5.9.4 UNION ALL/UNION [DISTINCT] 100
5.9.5 JOIN 操作 101
5.9.6 SEMI JOIN 103
5.9.7 MAPJOIN のヒント 104
5.9.8 HAVING 句 105
5.10 共通テーブル式 (CTE) 105
5.11 Explain 107
5.12 組み込み関数 110
5.12.1 日付関数 110
5.12.2 算術関数 127
5.12.3 ウィンドウ関数 149
5.12.4 集計関数 165
5.12.5 文字列関数 172
5.12.6 その他の関数 195
5.13 UDF 216
5.13.1 UDF の概要 216
5.13.2 Java UDF 218
5.14 UDT 230
5.14.1 UDT 230
5.15 UDJ 241
5.16 補足 252
5.16.1 エスケープ文字 252
5.16.2 LIKE の使用方法 253
5.16.3 正規表現 253
5.16.4 予約語とキーワード 255
5.16.5 Hive データ型の対応表 256
6 MapReduce 258
6.1 概要 258
6.1.1 MapReduce 258
6.1.2 拡張 MapReduce 261
6.1.3 オープンソースの MapReduce 262
6.2 MR の制限事項 267
6.3 機能紹介 270
6.3.1 基本概念 270

6.3.2 コマンド..... 271
6.3.3 入出力..... 273
6.3.4 リソース..... 273
6.3.5 ローカル実行..... 274
6.4 プログラム例..... 276
6.4.1 WordCount サンプル..... 277
6.4.2 MapOnly サンプル..... 279
6.4.3 複数の入出力..... 282
6.4.4 マルチタスクサンプル..... 286
6.4.5 セカンダリソートサンプル..... 288
6.4.6 リソースサンプル..... 291
6.4.7 Counter サンプル..... 293
6.4.8 Grep サンプル..... 296
6.4.9 Join サンプル..... 300
6.4.10 Sleep サンプル..... 303
6.4.11 Unique サンプル..... 304
6.4.12 ソートサンプル..... 307
6.4.13 パーティションサンプル..... 309
6.4.14 Pipeline サンプル..... 311
6.5 Java SDK..... 313
6.5.1 Java SDK..... 313
6.5.2 SDK の互換バージョンの概要..... 320
7 Java サンドボックス..... 341
8 外部テーブル..... 346
8.1 外部テーブルの概要..... 346
8.2 OSS STS モードの許可..... 347
8.3 OSS 非構造化データへのアクセス..... 349
8.4 OSS のオープンソース形式データの処理..... 361
8.5 OSS にエクスポートされた非構造化データ..... 368
8.6 Table Store データへのアクセス..... 377
9 Lightning..... 385
9.1 Lightning の概要..... 385
9.2 制約と制限..... 388
9.3 クイックスタート..... 388
9.3.1 ガイドの説明..... 388
9.3.2 前提条件..... 389
9.3.3 接続用クライアントツールの準備..... 389
9.3.4 サービスへのアクセスと分析の実行..... 389
9.4 アクセスドメイン名..... 390
9.5 JDBC インターフェイスを使用したアクセスサービス..... 392
9.5.1 JDBC ドライバー..... 392
9.5.2 JDBC 接続の設定..... 394
9.5.3 共通ツールを使用したサービスへのアクセス..... 396
9.6 SQL リファレンス..... 401

9.7 タスクの表示..... 402
9.8 Lightning に関する FAQ..... 403
10 Graph..... 406
10.1 概要..... 406
10.2 Aggregator..... 409
10.3 制限..... 417
10.4 機能の概要..... 418
10.5 SDK の概要..... 421
10.6 開発とデバッグ..... 422
10.7 例..... 430
10.7.1 SSSP..... 431
10.7.2 PageRank..... 434
10.7.3 K-means..... 437
10.7.4 2 部マッチング..... 442
10.7.5 強連結成分..... 445
10.7.6 連結成分..... 452
10.7.7 トポロジカルソート..... 455
10.7.8 線形回帰..... 458
10.7.9 三角形カウント..... 462
10.7.10 頂点の入力..... 464
10.7.11 辺の入力..... 471
11 ジョブ実行情報の表示..... 478
11.1 Logview..... 478
11.2 MaxCompute コンパイラ使用時のエラーと警告..... 482

1 データ型

2 データ型

ここでは、**MaxCompute 2.0** で使用可能なデータ型を説明します。基本データ型と複合データ型があります。

基本データ型

MaxCompute2.0 が対応しているデータ型を下表に記載します。 **MaxCompute** のテーブル列は、下表に記載したデータ型のいずれかでなければなりません。 新規データ型は、**Tinynt**、**Smallint**、**Int**、**Float**、**Varchar**、**Timestamp**、および **Binary** です。データ型の詳細は、次のとおりです。

 注 :

MaxCompute SQL と新バージョンの **Mapreduce** で使用可能な **set** コマンドは、次の **2** 通りに分けられます。

- セッションレベル**:** 新規データ型 **(Tinyint**、**Smallint**、**Int**、**Float**、**Varchar**、**Timestamp**、**Binary)** を使用するには、テーブル文の前に、次の **set** 文を追加する必要があります。

  ```
  set odps.sql.type.system.odps2=true;
  ```

  次に、テーブル文と一緒に、実行する **SQL** 文を送信します。**SQL** ツールでは、デフォルトで **MaxCompute Studio** を通して自動的に **set** 文が追加され、実行コマンドが送信されます。
- **Int** データ型の場合、**set** 文に **32** ビットが追加されます。**Set** 文を追加しない場合は、**64** ビットの **Bigint** 型に変換されます。
- **SDK 0.27.2** パブリックバージョン以降、**Client 0.27.0** バージョン以降が新規データ型に対応しています。

新規データ型を `odps.sql.type.system.odps2` でオープンにします。

- 暗黙的な型変換ルールが変わります。

  発生の可能性がある互換性問題**:** 新規データ型システムでは、いくつかの型変換が無効になります。**String** → **Bigint**、**String** → **Datetime**、**Double** → **Bigint**、**Decimal** → **Double**、**Decimal** → **Bigint** はすべて、正しく変換されなかったり、エラーが発生する恐れがあります。 このような状況が発生した場合は、キャスト変換によって解決できます。

- サポート対象の操作、組み込み関数、**UDF** は同じではありません。

  操作や組み込み関数で新規データ型をパラメーターや戻り値として使用する場合、使用する前に新規データ型をオープンにする必要があります。新規データ型をパラメーターや戻り値として複数定義する **UDF** も対象です。 発生の可能性のある互換性問題**:** たとえば、**UDF** に **Bigint** 型と **Int** 型の **2** つが定義されているとします。 旧タイプの **MaxCompute** では **Bigint** 型のオーバーロードでなければなりませんが、新タイプの **MaxCompute** では **Int** 型のオーバーロードに解決されることがあります。
- 定数のデータ型が変わります。

  123 のような単一定数は、旧タイプの **MaxCompute** では **Bigint** 型になり、新タイプの **MaxCompute** では **Int** 型になります。 発生の可能性のある互換性問題**: Int** 型によって、後続操作で呼び出される機能のプロトタイプで不一致が発生します。たとえば、テーブルの削除の後に新データ型のテーブルを作成する (ディスクへの書き込み) などの操作で、周辺ツールやジョブの動作が変わってしまいます。
- **Bigint** 型キーワードの解決が異なります。

  旧タイプのシステムでは、**Int** 型キーワードは **Bigint** 型として処理されますが、新タイプのシステムでは真の **Int** 型として処理されます。 発生の可能性のある互換性問題**: Int** 型によって、後続操作で呼び出される機能のプロトタイプで不一致が発生します。たとえば、テーブルの削除の後に新データ型のテーブルを作成する (ディスクへの書き込み) などの操作で、周辺ツールやジョブの動作が変わってしまいます。

setproject の詳細は、「その他の操作

」をご参照ください。

現在、**MR** タスクは新規データ型に対応していません。

データ型	新規	定数	説明
Tinyint	はい	**1Y**、**-127Y**	**8** ビット符号付き整数で、範囲は **-128**～**127** です。
Smallint	はい	**32767S**、**-100S**	**16** ビット符号付き整数で、範囲は **-32768**～**32767** です。
Int	はい	**1000**、**-15645787** (注 **2**)	**32** ビット符号付き整数で、範囲は -2^{31} から $2^{31}-1$ です。

データ型	新規	定数	説明
Bigint	いいえ	**100000000000L**、**-1L**	**64** ビット符号付整数で、範囲は **$-2^{63}+1$** から **$2^{63}-1$** です。
Float	はい	なし	**32** ビットバイナリ浮動小数点です。
Double	いいえ	**3.1415926 1E+7**	**64** ビットバイナリ浮動小数点です。
Decimal	いいえ	**3.5BD**、**9999999999.9999999BD** (注 **1**)	**10** 進数の数値型で、整数部分の範囲は **$-10^{26}+1$** ～ **$10^{26}-1$** で、小数部分は **10** まで正確です。
Varchar	はい	なし (注 **3**)	可変文字型。**n** は長さで範囲は **1**～**65535** です。
String	いいえ	“**abc**”、’**bcd**’、”**alibaba**” ‘**inc**’ (注 **4**)	単一文字型で、長さは最大 **8M**です。
Binary	はい	なし	バイナリ単一文字型で、長さは最大 **8M**です。
Datetime	いいえ	**DATETIME** ‘**2017-11-11 00:00:00**’	日付型で、**999** 年 **12** 月 **31** 日から **9,0000** 年 **1** 月 **1** 日の範囲であり、ミリ秒まで正確です (注 **5**)。
Timestamp	はい	**TIMESTAMP** ‘**2017-11-11 00:00:00.123456789**’	タイムゾーンに依存し、**0000** 年 **1** 月 **1** 日から **9999** 年 **12** 月 **31**日 **23.59:59.999999999** の範囲であり、ナノ秒レベルまで正確です。
Boolean	いいえ	**TRUE**、**FALSE**	**True** または **False** のブール型です。

上記の表のデータ型は、すべて **NULL** にすることができます。

注：

- 注 1: Decimal 型フィールドに定数を挿入する場合、定数の記述に注意してください。 定数の定義で 3.5BD を挿入する場合の例は、次のとおりです。

```
insert into test_tb(a) values (3.5BD);
```

フィールド a は Decimal 型です。

- 注 2: Int 定数では、Int 型の範囲を超えている場合、Int 型は Bigint 型に変換されます。Bigint 型の範囲を超えている場合は、Double 型に変換されます。

MaxCompute 2.0 より前のバージョンでは、SQL スクリプト内のすべての INT 型は、Bigint 型に変換されます。以下に例を示します。

```
create table a_bigint_table(a int); -- the int here is actually
treated as a bigint
select cast(id as int) from mytable; -- the int here is actually
treated as a bigint
```

MaxCompute 2.0 はMaxCompute の旧バージョンと互換性があるため、 odps.sql.type.system.odps2 を True と設定しなくても、この変換機能は保持されます。 ただし、Int 型が Bigint 型として処理されると、警告が発生します。 この場合、混乱を避けるために Int 型を Bigint 型に変更することを推奨します。

- 注 3: Varchar 型の定数は、暗黙変換の String 型の定数で表すことができます。
- 注 4: String 型の定数では、連結が行われます。たとえば、`abc xyz` は `abcxyz` として解釈されるので、異なる部分が異なる行に書き込まれることがあります。
- 注 5: 現行のクエリで表示される時間値には、ミリ秒は含まれません。 Tunnel コマンドで、`-dfp` を使用すると、ミリ秒まで指定できます。たとえば、`tunnel upload -dfp 'yyyy-MM-dd HH:mm:ss.SSS'` と記述します。Tunnel コマンドの詳細は、「*#unique_4*」をご参照ください。

複合データ型

MaxCompute2.0 は、下表に記載する複合データ型に対応しています。

データ型	定義方法	構築方法
Array	array< int >;array< struct < a:int, b:string >>	array(1, 2, 3); array(array(1, 2); array(3, 4))
Map	map< string, string >;map < smallint, array< string>>	map(“k1” , “v1” , “k2 ” , “v2”); map(1S, array (‘a’ , ‘b’), 2S, array(‘ x’ , ‘y))

データ型	定義方法	構築方法
Struct	**struct< x:int, y:int>;struct < field1:bigint, field2: array< int>, field3:map< int, int>>**	**named_struct(‘x’ , 1, ‘ y’ , 2); named_struct(‘ field1’ , 100L, ‘field2’ , array(1, 2), ‘field3’ , map (1, 100, 2, 200)**

 注 :

現在、 **MaxCompute SQL** および **Mapreduce** の新バージョンでは、次の **2** 通りの方法で **set** コマンドを使用できます。

- セッションレベル: 新しいデータ型 **(Tinyint、Smallint、Int、Float、Varchar、Timestamp Binary)** を使用するには、テーブル文の前に **set** 文を追加します。

  ```
  set odps.sql.type.system.odps2=true;
  ```

  次に **set** 文をテーブル文と一緒に送信します。
- プロジェクトレベル: プロジェクトレベルの **set** 文が使用できるようになりました。 プロジェクトオーナーは、必要に応じてプロジェクトを設定できます。

  ```
  setproject odps.sql.type.system.odps2=true;
  ```

  setproject コマンドの詳細は、「その他の操作」をご参照ください。

3 共通コマンド

3.1 共通コマンド一覧

ここでは、**MaxCompute** について理解を深めていただけるよう、クライアント側で関連コマンドを使用する方法を説明します。

最新の **MaxCompute** サービスでは、一般のコマンドに修正を加えて **Hive** に似たコマンド形式となっています。**Hadoop/Hive** ユーザーもすぐにお使いいただけます。

MaxCompute では、 プロジェクト、テーブル、リソース、インスタンスといったオブジェクトをさまざまに処理することができます。 コンソールコマンドおよび **SDK** でオブジェクトを操作します。

> 注：
>
> - 本モジュールで紹介するコマンドは、主に最新の *Client* を対象としています。
> - クライアントをインストールし、設定する方法の詳細は、「クイックスタート **(Quick Start)**」の 「クライアントのインストールと設定 *(Install and configure a client)*」をご参照ください。
> - **SDK** についての詳細は、「*MaxCompute SDK の概要 (MaxCompute SDK introduction)*」をご参照ください。

共通コマンド一覧

addaliasalter

costcreate

delete/dropdesc/describedownload/get

extended

flag/flagsfunctionfunctions

get

helphistory

instance/instances

jar/mapreduce

kill

lifecyclelist

odpscmd

partition

q/quit

resource

setshowsqlstop/kill

tunnel

upload

waitwho

使用上の制限

- リソースの操作を実行する際、各リソースファイルの容量が **500 M** を超えないように注意してください。また、**1** つの **SQL** タスクまたは **MapReduce** タスクで参照されるリソースの合計容量が、**2048 M** を超えないようにする必要があります。 制限事項についての詳細は、「*MR の制限事項 (MR limits)*」をご参照ください。

3.2 プロジェクトの操作

ここでは、プロジェクトに移動して、プロジェクトスペースのプロパティを設定する方法を紹介します (権限、ホワイトリストなど)。

ワークスペースへの移動

コマンド形式**:**

```
use <project_name>;
```

コマンド動作**:**

- 特定のワークスペースに移動します。 ワークスペースに移動すると、そのワークスペース内にあるすべてのオブジェクトを操作できるようになります。
- ワークスペースが存在しない、あるいはワークスペースに属するユーザーがいない場合は、例外が返されます。

例：

```
odps:my_project>use my_project; --my_project is a workspce the user
has privilege to access.
```



注：

上記例は、**MaxCompute** クライアントで実行しています。 すべての **MaxCompute** コマンド、キーワード、ワークスペース名、テーブル名、列名は、大文字小文字が区別されません。

プロジェクトを作成すると、**MaxCompute** プロジェクトも同時に作成されます。

このコマンドを実行すると、指定したワークスペース内にあるすべてのオブジェクトにアクセスできるようになります。 以下の例では、「**my_project**」 プロジェクトに **test_src** というテーブルが存在するとします。 次のコマンドを実行します。

```
odps:my_project>select * from test_src;
```

MaxCompute では、自動的に **my_project** 内のテーブルが検索されます。 指定したテーブルが存在する場合は、テーブルのデータが返されます。 指定したテーブルが存在しない場合、例外が返されます。 「**my_project**」プロジェクトから「**my_project2**」といった別のプロジェクトのテーブル **test_src** にアクセスするには、次のようにテーブルの前にプロジェクト名を指定する必要があります。

```
odps:my_project>select * from my_project2.test_src;
```

最初の例に出てくる **my_project** プロジェクト内にある **test_src** テーブルのデータではなく、**my_project2** プロジェクト内のデータが返されます。

MaxCompute には、ワークスペースを作成および削除するためのコマンドがありません。必要に応じて、**MaxCompute** コンソールを使用して、多くの設定と操作を実行できます。詳細は、「プロジェクトリスト *(project list)*」をご参照ください。

SetProject コマンド

コマンド形式:

```
setproject [<KEY>=<VALUE>];
```

コマンド動作:

- プロジェクトの属性を設定するには、setproject コマンドを使用します。

  以下の例では、フルテーブルスキャンの方法を設定します。

```
setproject odps.sql.allow.fullscan = true;
```

- <KEY>=<VALUE> の値を指定しない場合、現行ワークスペース属性の設定が表示されます。

  コマンド形式:

```
setproject;  --Display the parameters set by the setproject command.
```

パラメーター

プロパティ名	権限所有者	説明	値の範囲
odps.sql.allow.fullscan	プロジェクトオーナー	フル テーブル スキャンを許可するかどうかを指定します。	**True** (許可) / **false** (禁止)
odps.table.drop.ignorenonexistent	全ユーザー	存在しないテーブルを削除しようとしたとき、エラーを報告するかどうかを指定します。「**True**」を指定すると、エラーは報告されません。	**True** (エラーを報告しない)/ **false**
odps.security.ip.whitelist	プロジェクトオーナー	プロジェクトへのアクセスを許可する **IP** ホワイトリストを指定します。	**IP** リストはカンマ (,) で区切って指定します。
odps.instance.remain.days	プロジェクトオーナー	インスタンス情報の保持期間を指定します。	**3**～**30**
READ_TABLE_MAX_ROW	プロジェクトオーナー	クライアントから **Select** 文が実行されたときに返すデータ数を指定します。	**1**～**10000**

odps.security.ip.**ホワイトリストの例**

MaxCompute では、ワークスペースレベルで **IP** ホワイトリストを設定できます。

注:

- **IP** ホワイトリストを設定すると、ホワイトリスト内の **IP** (コンソール **IP** または **SDK** が格納されている **exit** の **IP**) のみが、対応するワークスペースにアクセスできます。
- **IP** ホワイトリストを設定してから、変更が反映されるまで、少なくとも **5** 分待つ必要があります。
- サポートが必要な場合には、チケットを起票し、サポート センターへお問い合わせください。

同一コマンド内で使用できるホワイトリストの IP 形式は、次の 3 通りあります。 これらのコマンドは、カンマ (,) で区切ります。

- IP アドレス: 例101.132.236.134
- サブネットマスク: 例 100.116.0.0/16
- ネットワークセグメント: 例 101.132.236.134-101.132.236.144

IP ホワイトリストを設定するためのコマンドラインツールの例は、次の通りです。

```
setproject odps.security.ip.whitelist=101.132.236.134,100.116.0.0/16,
101.132.236.134-101.132.236.144;
```

ホワイトリストに IP アドレスが 1 つも追加されていない場合、ホワイトリスト機能は無効になります。

```
setproject odps.security.ip.whitelist=;
```

3.3 テーブルの操作

ここでは、MaxCompute クライアントから共通コマンドを使用して、テーブルを作成、削除、表示する方法を紹介します。

テーブルを操作する際、クライアント側から共通コマンドを使用してください。 また、DataWorks の可視データテーブルマネージャーを使用すると、テーブルの収集、権限の適用、パーティションの表示を簡単に実行できます。 詳細は、「テーブルの詳細 *(Table Details)*」をご参照ください。

テーブルの作成

コマンド形式:

```
CREATE TABLE [IF NOT EXISTS] table_name
 [(col_name data_type [COMMENT col_comment], ...)]
 [COMMENT table_comment]
 [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
 [LIFECYCLE days]
 [As select_statement]
CREATE TABLE [IF NOT EXISTS] table_name
 LIKE existing_table_name
```

コマンド動作:

テーブルを作成します。

 注:

- テーブル名と列名は、共に大文字小文字が区別されません。また、同じ命名規則に従います。 最長 **128** バイトで、 英数字とアンダースコア「_」を使用できます。
- コメントには有効な文字列を使用し、**1024** バイト以内で指定できます。
- **[LIFECYCLE days]:** パラメーターの「**days**」は、テーブル操作のライフサイクルを実行するまでの時間を指します。 正の整数でなければなりません。 単位は「日」です。
- 「**table_name**」が非パーティションテーブルの場合、 最終更新日から計算して **N (日)** が経過してもデータが変更されない場合、ユーザーの介入なしに **MaxCompute** によって自動的にテーブルがリサイクルされます **(**「テーブルの削除」操作と同様**)**。
- 「**table_name**」がパーティションテーブルの場合、 各パーティションの **LastDataModifiedTime** に基づいて、テーブルをリサイクルするかどうか **MaxCompute** によって判断されます。 非パーティションテーブルとは異なり、すべてのパーティションがリサイクルされた後でもパーティションは削除されません。 ライフサイクルは、テーブルに対してのみ設定でき、特定のパーティションに対しては設定できません。

例**:**

```
CREATE TABLE IF NOT EXISTS sale_detail(
 shop_name STRING,
 customer_id STRING,
 total_price DOUBLE)
PARTITIONED BY (sale_date STRING,region STRING); --Create a partition
table sale_detail.
```

テーブルの削除

コマンド形式**:**

```
DROP TABLE [IF EXISTS] table_name; -- Table name to be deleted.
```

コマンド動作**:**

- テーブルを削除します。
- **[IF EXISTS]** を指定すると、テーブルが存在するかどうかに関係なく、コマンドは正常に終了します。 **[IF EXISTS]** を指定せず、テーブルが存在しない場合、例外が返されます。

例**:**

```
DROP TABLE sale_detail; -- If the table exists, success returns.
DROP TABLE IF EXISTS sale_detail; -- No matter whether the table
sale_detail exists or not, success returns.
```

テーブル情報の照会

コマンド形式**:**

```
DESC <table_name>; -- Table name or view name.
```

```
DESC extended <table_name>; -- View the extended table information.
```

コマンド動作:

指定したテーブルの以下の情報が返されます。

- **Owner:** テーブルのオーナー
- **Project:** テーブルが割り当てられているプロジェクト
- **CreateTime:** テーブル作成時間
- **LastDDLTime:** 最後の **DDL** 操作
- **LastModifiedTime:** 最後にテーブルが変更された時間
- **InternalTable:** 照会するオブジェクトがテーブルであることを示します。 デフォルト値は、「**YES**」です。
- **Size:** テーブルデータが占有するストレージ容量です。圧縮率は **5** で、 単位はバイトです。
- **Native Columns:** 列名、データ型、コメントなどの非パーティション列の情報
- **Partition Columns:** パーティション名、データ型、コメントなどのパーティション列の情報
- **Extended Info: StorageHandler** やロケーションなどの拡張テーブルの情報

例:

```
odps@ project_name>DESC sale_detail; -- Describe a partition table.
+------------------------------------------------------------------------------------
+
| Owner: ALIYUN$odpsuser@aliyun.com | Project: test_project |
| TableComment: |
+------------------------------------------------------------------------------------
+
| CreateTime: 2014-01-01 17:32:13 |
| LastDDLTime: 2014-01-01 17:57:38 |
| LastModifiedTime: 1970-01-01 08:00:00 |
+------------------------------------------------------------------------------------
+
| Internaltable: Yes | size: 0 |
+------------------------------------------------------------------------------------
+
| Native Columns: |
+------------------------------------------------------------------------------------
+
| Field | Type | Comment |
+------------------------------------------------------------------------------------
+
| shop_name | string | |
| customer_id | string |
| total_price | double | |
+------------------------------------------------------------------------------------
+
| Partition Columns: |
+------------------------------------------------------------------------------------
+
| sale_date | string | |
| region | string | |
```

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



注：

- 上記例は、 **MaxCompute** クライアントで実行しています。
- パーティションのないテーブルの場合、パーティション列の情報は表示されません。
- 「ビュー」を照会するには「**InternalTable**」オプションは表示されませんが、「**VirtualView**」オプションが表示されます。デフォルト値は、「**YES**」です。 同様に、「**Size**」オプションは、ビューの定義を表す「**View Text**」オプションに置き換えられます。たとえば、`select * from src`と記述します。 詳細は、「ビュー操作」をご参照ください。

パーティションテーブルの表示

コマンド形式:

```
desc table_name partition(pt_spec
```

コマンド動作:

パーティションテーブルの特定パーティションの情報を表示します。

例:

```
odps@ project_name>desc meta.m_security_users partition (ds='20151010
');
+------------------------------------------------------------------------------
+
| PartitionSize: 2109112 |
+------------------------------------------------------------------------------
+
| CreateTime: 2015-10-10 08:48:48 |
| LastDDLTime: 2015-10-10 08:48:48 |
| LastModifiedTime: 2015-10-11 01:33:35 |
+------------------------------------------------------------------------------
+
OK
```

全テーブルまたは一部テーブルの表示

コマンド形式:

```
SHOW TABLES;
SHOW TABLES like 'chart';
```

コマンド動作:

- **SHOW TABLES:** 現行プロジェクトの全テーブルが表示されます。

- **SHOW TABLES like 'chart':** 現行プロジェクトのテーブル名が 「**chart**」と一致するテーブルが表示されます。 正規表現に対応しています。

例：

```
odps@ project_name>show tables;
odps@ project_name>show tables like 'ods_brand*';
ALIYUN$odps_user@aliyun.com:table_name
......
```

 注：

- 上記例は、 **MaxCompute** クライアントで実行しています。
- **ALIYUN** はシステムプロンプトで、コマンド実行者が **Alibaba Cloud** ユーザーであることを示しています。
- この例では、**odps_user@aliyun.com** がテーブルの作成者です。
- この例では、**example,table_name** がテーブル名です。

パーティションの表示

コマンド形式**:**

```
SHOW PARTITIONS ;  -- table_name: Specify the table to be queried. If
the table does not exist or it is not a partition table, an exception
is thrown.
```

コマンド動作**:**

テーブルの全パーティションが表示されます。

例：

```
odps@ project_name>SHOW PARTITIONS table_name;
partition_col1=col1_value1/partition_col2=col2_value1
partition_col1=col1_value2/partition_col2=col2_value2
...
```

 注：

- 上記例は、**MaxCompute** クライアントで実行しています。
- **Partition_col1** と **partition_col2** は、テーブルのパーティション列です。
- **Col1_value1**、**col2_value1**、**col1_value2**、**col2_value2** は、パーティション列に対応する値です。

3.4 インスタンス

インスタンスの表示

コマンド形式:

```
SHOW INSTANCES [FROM startdate TO enddate] [number];
SHOW P [FROM startdate TO enddate] [number];
SHOW INSTANCES [-all];
SHOW P [-all];
```

コマンド動作:

現行ユーザーが作成したインスタンスに関する情報が表示されます。

パラメーター

- `startdate`、`enddate`: **yyyy-mm-dd** の形式で指定した期間 (**startdate** から **enddate** まで) のインスタンス情報が返されます。単位は「日」です。 これは任意のパラメーターです。指定しない場合、デフォルトでは、**3** 日間に送信されたインスタンス情報が返されます。
- `number`: 表示するインスタンス数を指定します。 スケジュールされた時間に基づき、現在時刻に最も近い **N** 個 (数) のインスタンスが返されます。 このパラメーターを指定しない場合、要件を満たすすべてのインスタンスが表示されます。
- `-all`: 要件を満たす全インスタンス情報が返されます。 このコマンドを実行するユーザーは、プロジェクトにおける「**list**」権限が必要です。 デフォルトでは、このコマンドを実行して返されるのは、**50** レコードのみです。 `-limit number` パラメーターを使用すれば、さらに多くのレコードを表示できます。 たとえば、`show p -all -limit number 100` を使用すると、プロジェクト内の **100** インスタンスのレコードが表示されます。
- 出力される項目には、**StartTime** (秒単位)、**RunTime (s)** 、**Status (Waiting**、**Success**、**Failed**、**Running**、**Cancelled**、中断) があります。

インスタンス **ID** と、対応する **SQL** は、次の通りです。

```
StartTime RunTime Status InstanceID Query
2015-04-28 13:57:55 1s Success 20150428xxxxxxxxxxxxxxxxxx ALIYUN$xxxxx
@aliyun-inner.com select * from tab_pack_priv limit 20;
...    ...    ...    ...    ...    ...
...    ...    ...    ...    ...    ......
```

インスタンスのステータスは、次の通りです。

- **Running** (実行中)
- **Success** (成功)
- **Waiting** (待機中)

- **Failed** (失敗)。(ジョブは失敗しても、ターゲット テーブルのデータは変更されています)
- **Suspended** (中断)
- **Canceled** (キャンセル)

 注:
上記例のコマンドは、**MaxCompute** クライアントで実行しています。

インスタンスのステータス表示

コマンド形式:

```
status <instance_id>; -- instance_id: the unique identifier of an
instance, to specify which instance to be queried.
```

コマンド動作:

- 指定したインスタンスのステータスが返されます。ステータスには、**Success**、**Failed**、**Running**、**Cancelled** などがあります。
- 現行ユーザーが作成したインスタンスでない場合は、例外が返されます。

例:

```
odps@ $project_name>status 20131225123xxxxxxxxxxxxxxx;
Success
```

インスタンス **ID** が **20131225123xxxxxxxxxxxxxxx** のステータス照会結果は、「**Success** (成功)」です。

 注:
上記例のコマンドは、**MaxCompute** クライアントで実行しています。

稼働インスタンスの表示

コマンド形式:

```
top instance;top instance -all;
```

コマンド動作:

権限の要件: このコマンドは、プロジェクトオーナーまたは管理者が実行しなければなりません。

`top instance`: プロジェクト内で稼働中の現行アカウントのジョブ情報が表示されます。 表示される項目には、インスタンス **ID**、オーナー、ジョブの種類、進捗度、ステータス、開始時間、

優先度、実行時使用率 (CPU およびメモリ)、合計使用率 (CPU およびメモリ)、キュー情報 (位置および長さ) などがあります。

`top instance-all`：現行プロジェクト内のすべての現行ジョブが返されます。 デフォルトでは、このコマンドで 50 レコードが返されます。 `-limit number` パラメーターを使用すると、さらに多くのレコードを表示できます。

例:

```
odps@ $project_name>top instance;
```



注：
上記例のコマンドは、MaxCompute クライアント (バージョン 0.29.0 以降) で実行されています。

インスタンスの停止

コマンド形式:

```
kill <instance_id>; -- instance_id: The unique identifier of an
instance,  which must be ID of an instance whose status is 'Running',
otherwise, an error is returned.
```

コマンド動作:

指定したインスタンスが停止されます。 指定するインスタンスのステータスは、「Running (実行中)」でなければなりません。

例:

```
odps@ $project_name>kill 20131225123xxxxxxxxxxxxxxx;
```

インスタンス ID 20131225123xxxxxxxxxxxxxxx を停止します。



注：

- 上記例のコマンドは、MaxCompute クライアントで実行しています。
- これは非同期処理です。 システムでリクエストが受け付けられ、結果が返された場合でも、分散タスクが停止しているとは限りません。 `status` コマンドを使用すると、インスタンスが削除されているかを確認できます。

インスタンス情報の表示

コマンド形式:

```
desc instance <instance_id>; -- instance_id: The unique identifier of
an instance.
```

コマンド動作:

インスタンス ID に対応するジョブ情報が表示されます。表示される項目には、SQL、オーナー、開始時間、終了時間、ステータスなどがあります。

例:

```
odps@ $project_name> desc instance 20150715xxxxxxxxxxxxxxx;
ID 20150715xxxxxxxxxxxxxxx
Owner ALIYUN$XXXXXX@alibaba-inc.com
StartTime 2015-07-15 18:34:41
EndTime 2015-07-15 18:34:42
Status Terminated
console_select_query_task_1436956481295 Success
Query select * from mj_test;
```

インスタンス ID 20150715xxxxxxxxxxxxxxxに関連するすべてのジョブ情報が照会されます。

 注:
上記例のコマンドは、MaxCompute クライアントで実行しています。

インスタンスのログ情報の取得

コマンド形式:

```
wait <instance_id>;  -- instance_id: The unique identifier of an
instance.
```

コマンド動作:

実行タスクの情報が取得されます。インスタンス ID に関するログとログビューのリンクが表示されます。 ログビューのリンクにアクセスすると、タスクの詳細が表示されます。

例:

```
wait 201709251611xxxxxxxxxxxxx;
ID = 201709251611xxxxxxxxxxxxx
Log view:
http://logview.odps.aliyun.com/logview/?h=http://service.odps.aliyun.
com/xxxxxxxxxx
Job Queueing...
Summary:
resource cost: cpu 0.05 Core * Min, memory 0.05 GB * Min
inputs:
        alian.bank_data: 41187 (588232 bytes)
```

```
outputs:
        alian.result_table: 8 (640 bytes)
Job run time: 2.000
Job run mode: service job
Job run engine: execution engine
M1:
        instance count: 1
        run time: 1.000
        instance time:
                min: 1.000, max: 1.000, avg: 1.000
        input records:
                TableScan_REL5213301: 41187 (min: 41187, max: 41187,
avg: 41187
)
        output records:
                StreamLineWrite_REL5213305: 8 (min: 8, max: 8, avg: 8)
R2_1:
        instance count: 1
        run time: 2.000
        instance time:
                min: 2.000, max: 2.000, avg: 2.000
        input records:
                StreamLineRead_REL5213306: 8 (min: 8, max: 8, avg: 8)
        output records:
                TableSink_REL5213309: 8 (min: 8, max: 8, avg: 8)
```

3.5 リソース

ここでは、**MaxCompute** クライアントから共通コマンドを使用してリソースを操作する方法を紹介します。

DataWorks に搭載されている可視化オンラインデータ開発ツールを使用して、リソースの検索やアップロードを行うこともできます。 詳細は、「リソース管理 *(Resource management)*」をご参照ください。

リソース名の追加

コマンド形式:

```
add file <local_file> [as alias] [comment 'cmt'][-f];
add archive <local_file> [as alias] [comment 'cmt'][-f];
add table <table_name> [partition <(spec)>] [as alias] [comment 'cmt']
[-f];
add jar <local_file.jar> [comment 'cmt'][-f];
```

パラメーター

- `file/archive/table/jar`: リソースタイプを示します。 詳細は、「リソース *(Resources)*」をご参照ください。
- `local_file`: ローカルファイルのパスを示します。このファイル名がリソース名になります。 リソース名は、リソースの一意の識別子です。

- table_name: MaxCompute のテーブル名を示します。 現在のところ、外部テーブルはリソースに追加できません。
- [PARTITION (spec)]: MaxCompute では、追加するリソースがパーティションテーブルの場合、パーティションテーブル全体ではなく 1 つのパーティションのみをリソースとして追加できます。
- alias: リソース名を指定します。 このパラメーターを指定しない場合、ファイル名がデフォルトでリソース名になります。 Jar および Python リソースは、この機能に対応していません。
- [comment 'cmt']: リソースのコメントを追加します。
- [-f]: 同じ名前のリソースが存在する場合、このパラメーターの名前が元のリソース名の代わりに使用されます。 このパラメーターを追加せず、同じ名前のリソースが存在する場合、操作は失敗します。

例:

```
odps@ odps_public_dev>add table sale_detail partition (ds='20150602')
as sale.res comment 'sale detail on 20150602' -f;
OK: Resource 'sale.res' have been updated.
---Add a resource named sale.res in MaxCompute.
```

 注:

各リソースファイルのサイズは、500 MB 以下にする必要があります。 1 つの SQL または MapReduce タスクで参照されるリソースサイズは、2048 MB 以下にする必要があります。 詳細は、「*MR の制限事項 (MR Restrictions)*」をご参照ください。

リソースの削除

コマンド形式:

```
DROP RESOURCE <resource_name>; --resource_name : a specified resource
name.
```

リソース一覧の表示

コマンド形式:

```
LIST RESOURCES;
```

コマンド動作:

現行プロジェクトの全リソースの一覧が表示されます。

例:

```
odps@ $project_name>list resources;
Resource Name Comment Last Modified Time Type
1234.txt 2014-02-27 07:07:56 file
mapred.jar 2014-02-27 07:07:57 jar
```

リソースのダウンロード

リソースをダウンロードするには、次のコマンド形式を使用します。

```
GET RESOURCE <resource_name> <path>;
```

コマンド動作:

ローカルデバイスにリソースがダウンロードされます。 リソースタイプは、ファイル、 **jar**、アーカイブ、**py** のいずれかでなければなりません。

例:

```
odps@ $project_name>get resource odps-udf-examples.jar d:\;
OK
```

3.6 関数

ここでは、クライアントから共通コマンドを使用して関数を操作する方法を紹介します。

DataWorks に搭載されている可視化オンラインデータ開発ツールを使用して、関数を操作することもできます。

関数の作成

コマンド形式：

```
CREATE FUNCTION <function_name> AS <package_to_class> USING <
resource_list>;
```

パラメーター

- function_name: **SQL** で参照される **UDF** 名です。
- package_to_class: **Java UDF** の場合、この名前が、完全修飾クラス名 (最上位パッケージ名から **UDF** クラス名まで) です。 このパラメーターは二重引用符で囲む必要があります。また、**Python UDF** の場合、この名前が **python script name.class name** です。 **Java UDF** と **Python** スクリプトの場合、このパラメーターを指定する際は二重引用符 (“ “) で囲む必要があります。 名前は引用符で囲む必要があります。

- `resource_list`: UDF で使用するリソースリストを指定します。
  - UDF コードが保存されているリソースをリストに含める必要があります。
  - UDF コードが分散キャッシュインターフェイスを通してリソースファイルを読み取る場合、このリストに UDF で読み取られるリソースリストも含めます。
  - リソースリストは、カンマ (,) で区切られた複数のリソース名で構成されます。 リソースリストは二重引用符 (“”) で囲む必要があります。
  - リソースが割り当てられているプロジェクトを次のように指定します。 `<project_name>/resources/<resource_name>`

例:

- org.alidata.odps.udf.examples.Lower という Java UDF クラスが my_lower.jar にあるとします。関数 my_lower を作成するコマンドは、次の通りです。

```
CREATE FUNCTION test_lower AS 'org.alidata.odps.udf.examples.Lower'
USING 'my_lower.jar';
```

- Python UDF の MyLower を pyudf_test.py プロジェクトで使用するとします。関数 my_lower を作成するコマンドは、次の通りです。

```
create function test_lower as 'pyudf_test.MyLower' using 'test_proje
ct/resources/pyudf_test.py';
```

- Java UDF クラス com.aliyun.odps.examples.udf.UDTFResource が、udtfexample1.jar 内にあり、ファイルリソース file_resource.txt およびテーブルリソース table_resource1 と依存関係にあるとします。関数 test_udtf を作成するコマンドは、次の通りです。

```
create function test_udtf as 'com.aliyun.odps.examples.udf.
UDTFResource' using 'udtfexample1.jar, file_resource.txt, table_reso
urce1,test_archive.zip';
```



注:

- リソースファイルと同様に、同じ名前の UDF は一度しか登録できません。
- 通常、UDF はシステムの組み込み関数を上書きすることはできません。 プロジェクトオーナーのみが、組み込み関数を上書きする権限を持ちます。組み込み関数を上書きする UDF を使用すると、SQL 実行後にサマリーに警告が出力されます。

関数の削除

コマンド形式:

```
DROP FUNCTION <function_name>;
```

例:

```
DROP FUNCTION test_lower;
```

関数一覧の表示

コマンド形式:

```
list functions; --現行プロジェクト内のすべてのユーザー定義関数を表示します。
list functions -p my_project; --プロジェクト 'my_project' 内のすべてのユーザー定義関数を表示します。
```

3.7 Set 操作

ここでは、**set** コマンドを使用して **Maxcompute** やユーザー定義のシステム変数を設定する方法を紹介します。また、**set** コマンドの設定を取り消す方法も紹介します。

Set コマンド

コマンド形式

```
Set <key> = <value>
```

コマンド動作

set コマンドを使用すると、**MaxCompute** や **MaxCompute** 操作を設定できます。

現在、**MaxCompute** で使用できるシステム変数は、次の通りです。

```
-- MaxCompute SQL および Mapreduce（新バージョン）でサポートされる Set コマンド
set odps.sql.allow.fullscan=  -- パーティションテーブルに対してフルテーブルスキャンを許可するかどうかを設定します。 True は許可、false は不許可です。
set odps.stage.mapper.mem=   -- map worker のメモリサイズを設定します。 単位は M で、デフォルト値は 1024M です。
set odps.stage.reducer.mem= -- reduce worker のメモリサイズを設定します。単位は M です。デフォルト値は 1,024M です。
set odps.stage.joiner.mem=   -- join worker のメモリサイズを設定します。 単位は M で、デフォルト値は 1024M です。
set odps.stage.mem =    -- 特定の MaxCompute ジョブ内のすべての worker のメモリサイズを設定します。 前述の 3 つの set key より優先順位は低いです。 単位は M で、デフォルト値はありません。
set odps.stage.mapper.split.size=   -- 各 map worker の入力データ量（入力ファイルのバーストサイズ）を変更します。 つまり、各 map ステージの worker 数を制御します。 単位は M で、デフォルト値は 256M です。
set odps.stage.reducer.num=    -- 各 reduce ステージの worker 数を変更します。デフォルト値はありません。
```

```
set odps.stage.joiner.num=    -- 各 join ステージの worker 数を変更します。
デフォルト値はありません。
set odps.stage.num=    -- 特定の MaxCompute ジョブ内のすべてのステージの同時
 worker 数を変更します。 前述の 3 つの set key より優先順位が低く、デフォルト値
はありません。
set odps.sql.type.system.odps2=    -- デフォルト値は false です。 SQL 文に
新しいデータ型（TINYINT、SMALLINT、INT、FLOAT、VARCHAR、TIMESTAMP、BINARY
 など）が存在する場合、true を設定する必要があります。
```

フラグの表示

コマンド形式：

```
show flags;  -- Set コマンドで設定されたパラメーターを表示します。
```

コマンド動作:

Use Project コマンドを実行すると、**set** コマンドによる設定を消去できます。

3.8 その他の操作

ALIAS コマンド

ALIAS コマンドでは、コードを修正することなく *MapReduce* や *UDF* 内で固定リソース名を使用して、別のリソース (データ) を読み取ることができます。

コマンド形式:

```
ALIAS <alias>=<real>;
```

動作:

リソースの別名を作成します。

例:

```
ADD TABLE src_part PARTITION (ds='20121208') AS res_20121208;
ADD TABLE src_part PARTITION (ds='20121209') AS res_20121209;
ALIAS resName=res_20121208;
jar -resources resName -libjars work.jar -classpath ./work.jar com.
company.MainClass args ... ;//job 1
ALIAS resName=res_20121209;
jar -resources resName -libjars work.jar -classpath ./work.jar com.
company.MainClass args ... ;//job 2
```

上記例では、リソースの別名 **resName** によって、**2** つのジョブ内の異なるリソーステーブルが参照されます。 コードを修正せずに、異なるデータを読み取ることができます。

SQL 料金の表示

コマンド形式:

```
cost sql <SQL Sentence>;
```

コマンド動作:

入力データのサイズ、UDF 数、SQL 複雑度などの SQL 計測値を推計します。

 注:

以下の情報は、参考としてご使用ください。 実際の課金の基準としてご利用にならないでください。

例:

```
odps@ $odps_project >cost sql select distinct project_name, user_name
 from meta.m_security_users distribute by project_name sort by
project_name;
ID = 20150715113033121xxxxxxxx
Input:65727592 Bytes
UDF:0
Complexity:1.0
```

4 データのアップロードとダウンロード

4.1 データのアップロードとダウンロード

ここでは、サービス接続、**SDK**、ツール、クラウドデータの移行など、**MaxCompute** システムデータのアップロードとダウンロード処理の概要を紹介します。

MaxCompute システムにアクセスするには、**DataHub** のリアルタイムデータトンネル機能または **Tunnel** のバッチデータトンネル機能を使用します。

DataHub と **Tunnel** は、どちらも独自の **SDK** を実装しています。 **SDK** および派生データのアップロードとダウンロードツールは、さまざまなシナリオでのデータのアップロードおよびダウンロード要件を満たすことができます。

データのアップロードツールとダウンロードツールには、**DataWorks**、**DTS**、**OGG** プラグイン、**Sqoop**、**Flume** プラグイン、**Logstash** プラグイン、**Fluentd** プラグイン、**Kettle** プラグイン、**MaxCompute** コンソールなどがあります。

これらのツールで使用される基盤のデータトンネルは、次の通りです。

- **DataHub** トンネルツール
  - **OGG**
  - **Flume**
  - **LogStash**
  - **Fluentd**
- **Tunnel** ツール
  - **DataWorks**
  - **DTS**
  - **Sqoop**
  - **Kettle**
  - **MaxCompute** コンソール

幅広いデータのアップロードとダウンロードツールは、ほとんどのクラウドデータ移行シナリオに適用可能です。 各種ツール、**Hadoop** データ移行、データベースデータの同期化、ログ収集などのクラウド移行シナリオについては、他のドキュメントで紹介します。 技術的ソリューションを選択する際に、これらのドキュメントをご参照ください。

4.2 クラウド データの移行

MaxCompute プラットフォームのデータのアップロードツールとダウンロードツール は、さまざまなクラウド データの移行シナリオに適用できます。 本ページでは、いくつかの一般的なシナリオを紹介します。

Hadoop のデータ移行

Hadoopのデータ移行には、**Sqoop**または**DataWorks**を使用できます。

- **Sqoop** では、**MR** ジョブが元の **Hadoop** クラスタ上で実行されるため、効率的な**MaxCompute** への分散データ転送が実現します。 詳細は、「*Sqoop ツールの概要(Sqoop tool introduction)*」をご参照ください。
- **DataWorks** と **DataX** を一緒に使用すると、**Hadoop** データを移行できます。

データベースの同期

MaxCompute にデータベースのデータを同期するには、データベースの種類と同期ルールに基づいて、適切なツールを選択する必要があります。

- オフライン バッチデータを同期化する場合は、**DataWorks** を使用します。 **DataWorks** は、**MySQL**、**SQL Server**、**PostgreSQL** を始めとした幅広いデータベースのタイプに対応しています。 詳細は、「データ統合ノード」をご参照ください。 インスタンスの操作手順については、「同期タスクの作成」をご参照ください。
- リアルタイムで **Oracle** データを同期する場合は、**OGG** プラグイン ツールを使用できます。
- リアルタイムで **RDS** データを同期する場合は、**DTS** を使用します。

ログ収集

ログ収集には、**Flume**、**Fluentd**、**Logstash** ツールを使用します。

4.3 データのアップロードおよびダウンロード ツール

MaxCompute プラットフォームは、データのアップロードとダウンロードを実行するための幅広いツールに対応しています。 ほとんどのツールのソースコードは、オープンソースコミュニティの **GitHub** にあり、データのアップロードとダウンロードが可能です。 シナリオに応じたツールを選択できます。 ツールは、**Alibaba Cloud DTplus** プロダクトとオープンソース プロダクトの **2** 種類に分けられます。 本ページでは、これらのツールについて詳しく説明します。

Alibaba Cloud DTplus **プロダクト**

- **DataWorks** のデータ統合

  DataWorksのデータ統合（データ同期）は、**Alibaba Cloud**が提供する安定した、効率的でスケーラブルなデータ同期プラットフォームです。 これは、**Alibaba Cloud** の異種データストレージ システムで、オフライン同期、リアルタイムの増分データ同期、統合、および交換サービスに対応できるように設計されています。

  データ同期タスクは、**MaxCompute**、**RDS (MySQL**、**SQL Serve**、**PostgreSQL)**、**Oracle**、**FTP**、**AnalyticDB (ADS)**、**OSS**、**Memcache** および **DRDS** のデータ型に対応しています。 詳細は、「*Data Integration の概要*」をご参照ください。

- **MaxCompute** コンソール

  - コンソールのインストールと基本的な使い方については、「クライアント概要 *(Client introduction)*」をご参照ください。
  - クライアントには、データのアップロードとダウンロード用のバッチ データ トンネル **SDK** を基盤とした組み込み **Tunnel** コマンドが実装されています。 詳細は、「*Tunnel* 基本コマンドの使用方法 *(Basic Tunnel command usage)* 」 をご参照ください。

  > 注：
  > これは、オープンソースの *aliyun-odps-console* です。

- **DTS**

  DTS (Data Transmission) は、**Alibaba Cloud** が提供するデータ サービスであり、**RDBMS**、**NoSQL**、**OLAP** などのデータ ソース間のデータ交換に対応しています。 また、データ移行、リアルタイムのデータ サブスクリプション、リアルタイムのデータ同期、およびその他のデータ転送機能を搭載しています。

  DTS は、**ApsaraDB for RDS** と **MySQL**インスタンスから **MaxCompute** テーブルへのデータ同期に対応しています。 現在のところ、その他の種類のデータ ソースには対応していません。

オープンソース プロダクト

- **Sqoop**

  Sqoop は、**Sqoop 1.4.6** コミュニティに基づいて開発されたツールです。**Sqoop** には、**MySQL** やその他のリレーショナル データベースから **MaxCompute** テーブルへのデータ イ

ンポートとエクスポート機能が搭載されています。 **HDFS/Hive** のデータを**MaxCompute** テーブルにインポートすることもできます。


注：
これは、オープンソースの *aliyun-maxcompute-data-collectors* です。

- **Kettle**

  Kettleは、**Windows**、**Unix**、または **Linux** 上で動作する **Java** ベースのオープンソースの **ETL** ツールです。 ドラッグアンドドロップ コンポーネントを使用して、データ送信トポロジを簡単に定義できるグラフィック インターフェイスを実装しています。

  
  注：
  これは、オープンソースの *aliyun-maxcompute-data-collectors* です。

- **Flume**

  Apache Flume は、信頼性の高い分散システムであり、膨大な量のログ データを効率的に収集、集約し、さまざまなデータソースから集中データストレージ システムに移行できます。 数多くのソース プラグインとシンク プラグインに対応しています。

  Apache Flume の**DataHub Sink** プラグインを使用すると、リアルタイムでログ データを **DataHub** にアップロードし、**MaxCompute** テーブルにアーカイブできます。

  
  注：
  これは、オープンソースの *aliyun-maxcompute-data-collectors*です。

- **Fluentd**

  Fluentd は、さまざまなソースからログ (アプリケーション ログ、システム ログ、アクセス ログなど) を収集するために使用されるオープンソースのソフトウェア プロダクトです。 **MySQL**、**Oracle**、**MongoDB**、**Hadoop**、**Treasure Data** など、さまざまなデータ プロセッサにログ データをフィルタリングして格納するためのプラグインを選択できます。

  Fluentd の**DataHub** プラグインを使用すると、リアルタイムでログデータを **DataHub** にアップロードし、**MaxCompute** テーブルにアーカイブできます。

- **LogStash**

  Logstash はオープンソースのログ収集と処理のフレームワークです。 **logstash-output-datahub** プラグインを使用すると、データを**DataHub**にインポートできます。 データの収集と送信を実行するためのツールの設定は簡単です。 **MaxCompute** や **StreamCompute**

と一緒に使用すると、データ収集から分析までが可能なオールインワンのストリーミングデータ ソリューションを簡単に作成できます。

Logstash の**DataHub** プラグインを使用すると、リアルタイムでデータを **DataHub** にアップロードし、**MaxCompute** テーブルにアーカイブできます。

- **OGG**

  OGGの**DataHub** プラグインを使用すると、**Oracle** データベースのデータをリアルタイムで **DataHub** に段階的に同期させ、**MaxCompute** テーブルにデータをアーカイブできます。

  > 注：
  > これは、オープンソースの *aliyun-maxcompute-data-collectors* です。

4.4 Tunnel SDK

4.4.1 概要

MaxCompute Tunnel は、**MaxCompute** のデータトンネルです。 **Tunnel** を 介して **MaxCompute** にデータをアップロードまたはダウンロードできます。 ただし、**Tunnel** は、テーブルデータのアップロードとダウンロードにのみ対応しています。

MaxCompute には、**Tunnel SDK** を基盤とした データ アップロードおよびダウンロードツールが搭載されています。

Maven を使用する際、**Maven** データベースで `odps-sdk-core` を検索すると、**Java SDK** の各種バージョンを検索できます。 設定は次のとおりです。 **SDK (available in different versions).**

```
<dependency>
    <groupId>com.aliyun.odps</groupId>
    <artifactId>odps-sdk-core</artifactId>
    <version>0.24.0-public</version>
</dependency>
```

本ページでは、**Tunnel SDK** の主なインターフェイスを紹介します。ただし、**SDK** のバージョンによっては異なる場合があります。 詳細は、「*SDK Java Doc*」をご参照ください。

インターフェイス	説明
TableTunnel	**MaxCompute** にアクセスするためのポータルクラスのインターフェイス。**Alibaba Cloud** のインターネットまたはイントラネットを通じて **MaxCompute** と **Tunnel** サービスにアクセスできます。 イントラネットを使用し、**MaxCompute Tunnel** を介してデータをダウンロードする場合、トラフィック料金は発生しません。 イントラネットアドレスは、杭州リージョンのクラウドプロダクトに対してのみ有効です。
TableTunnel.UploadSession	**MaxCompute** テーブルへのデータのアップロード処理を示す
TableTunnel.DownloadSession	**MaxCompute** テーブルからのデータのダウンロード処理を示す

注：

- **SDK** に関する詳細は、「*SDK Java Doc*」をご参照ください。
- サービス接続に関する詳細は、「*ドメインとデータセンタへのアクセス (Access Domains and Data Centers)*」をご参照ください。

4.4.2 TableTunnel

TableTunnel は、**MaxCompute Tunnel** サービスにアクセスするための入り口のクラスです。**TableTunnel.UploadSession** インターフェイスは、データを **MaxCompute** テーブルにアップロードするセッションです。 **TableTunnel.DownloadSession** インターフェイスは、**MaxCompute** テーブルにデータをダウンロードするセッションです。

TableTunnel インターフェイスは、次のように定義されています。

```
public class TableTunnel {
 public DownloadSession createDownloadSession(String projectName,
String tableName);
 public DownloadSession createDownloadSession(String projectName,
String tableName, PartitionSpec partitionSpec);
 public UploadSession createUploadSession(String projectName, String
tableName);
 public UploadSession createUploadSession(String projectName, String
tableName, PartitionSpec partitionSpec);
 public DownloadSession getDownloadSession(String projectName, String
tableName, PartitionSpec partitionSpec, String id);
 public DownloadSession getDownloadSession(String projectName, String
tableName, String id);
 public UploadSession getUploadSession(String projectName, String
tableName, PartitionSpec partitionSpec, String id);
```

```
 public UploadSession getUploadSession(String projectName, String
tableName, String id);
 }
```

パラメーター

- ライフサイクル**: TableTunnel** のライフサイクルです。**TableTunnel** インスタンスの作成からプロセスの完了までを指します。
- **PublicClassTableTunnel:** アップロードオブジェクトやダウンロードオブジェクトを作成するメソッドです。
- セッション**:** テーブルまたはパーティションをアップロードおよびダウンロードするプロセスです。 **1** つのセッションは、**Tunnel RESTful API** への **1** つ以上の **HTTP** リクエストで構成されます。
- アップロードセッション**: TableTunnel** のアップロードセッションは、**INSERT INTO** セマンティクスです。つまり、同一テーブルやパーティションへのアップロードが発生しても、互いに干渉しません。 各アップロードセッションは、異なるディレクトリ内にあります。
- ブロック **ID:** 対応するファイル名です。 アップロードセッションでは、各 **RecordWriter** が **1** つの **HTTP** リクエストに対応し、ブロック **ID** によって識別されます。また、サーバー側の **1** ファイルと対応します。
- **RecordWriter: 1** つのセッション内で、同じブロック **ID** を使用して **RecordWriter** を複数回オープンすると、データは上書きされます。 最後の **RecordWriter close()** によってアップロードされたデータが保持されます。 ブロックのアップロードが失敗した場合は、この機能を使用して、データ伝送を再開できます。

TableTunnel の実装プロセスは、次の通りです。

1. **RecordWriter.write()** により、データは一時ディレクトリにアップロードされます。
2. **RecordWriter.close()** により、前述のファイルが一時ディレクトリからデータディレクトリに移動されます。
3. **Session.commit()** により、対応するデータディレクトリ内のすべてのファイルが、対応するテーブルを保有するディレクトリに移動され、テーブルのメタデータが更新されます。 つまり、テーブルに移動されたデータは、他の **MaxCompute** タスク **(SQL** や **MR** など**)** から参照できるようになります。

制限事項**:**

- ブロック **ID** の範囲は、**0** から **20000** です。 **1** つのブロックでアップロード可能なデータサイズは、**100 GB** 以下です。
- セッションのタイムアウトは、**24** 時間です。データ伝送時間が、しきい値である **24** 時間を超えることが予想される場合は、大量データを複数のセッションに分けてください。

- **RecordWriter** の **HTTP** リクエストのタイムアウトは、**120** 秒です。**HTTP** 接続によるデータ伝送が **120** 秒間認識されないと、接続が自動的にクローズされます。

   注：
  デフォルトでは、**HTTP** のバッファは **8 KB** です。 このため、**RecordWriter.write()** 呼び出しのたびに **HTTP** 接続によるデータフローを判断するのは困難です。 また、**TunnelRecordWriter.flush()** により、強制的にバッファ内のデータが消去されることがあります。

- データフローが予測できないため、ログが書き込まれる間にタイムアウトが発生する可能性があります。 注：
  - すべてのタイプのデータに対して **RecordWriter** を使用することは推奨しません。 各 **RecordWriter** は、**1** つのファイルに対応しているので、小さなファイルが多数生成され、**MaxCompute** のパフォーマンスに大きく影響します。
  - コードのキャッシュデータサイズが **64 MB** を超える場合、バッチ処理内で **RecordWriter** を呼び出してデータを書き込むことを推奨します。
- **RecordReader** タイムアウトのしきい値は、**300** 秒です。

4.4.3 UploadSession

ここでは、**UploadSession** インターフェイスを紹介します。

`UploadSession` インターフェイスの定義

UploadSession インターフェイスは、次のように定義されています。

```
public class UploadSession {
        UploadSession(Configuration conf, String projectName, String
tableName,
            String partitionSpec) throws TunnelException;
        UploadSession(Configuration conf, String projectName, String
tableName,
            String partitionSpec, String uploadId) throws TunnelExce
ption;
        public void commit(Long[] blocks);
        public Long[] getBlockList();
        public String getId();
        public TableSchema getSchema();
        public UploadSession.Status getStatus();
        public Record newRecord();
        public RecordWriter openRecordWriter(long blockId);
        public RecordWriter openRecordWriter(long blockId, boolean
compress);
        public RecordWriter openBufferedWriter();
        public RecordWriter openBufferedWriter(boolean compress);
```

```
    }
```

アップロードオブジェクトの説明

- ライフサイクル: アップロードインスタンスの作成から始まり、アップロードプロセスが完了すると終わります。
- アップロードインスタンスの作成: **Constructor** を呼び出すか、TableTunnelを使用すると、インスタンスを作成できます。
  - リクエストモード: 同期
  - このアップロードインスタンスのセッションがサーバーで作成され、一意の UploadId が生成されます。 クライアントから getId を使用して **UploadId** を取得します。
- データのアップロード:
  - リクエストモード: 同期
  - openRecordWriter メソッドを呼び出して、RecordWriter インスタンスを作成します。 ブロック **ID** によって、アップロード対象のデータが識別されます。ブロック **ID** はテーブル内の位置を示し、値の範囲は **0** から**20000** です。 データのアップロードに失敗した場合、ブロック **ID** を使用して再度アップロードしてください。
- アップロードの表示:
  - リクエストモード: 同期
  - 現行のアップロードスターテスを取得するには、getStatus を呼び出します。
  - 正常にアップロードされた blockId リストを取得するには、getBlockList を呼び出します。 アップロードに失敗した blockId を見つけて、再度アップロードするには、アップロードの blockId リストの 結果を比較します。
- アップロードの完了:
  - リクエストモード: 同期
  - commit (Long[] blocks) メソッドを呼び出します。 ブロックリストには、正常にアップロードされたブロックが表示されます。 このリストがサーバーで検証されます。
  - この機能により、データ検証が強化されます。 ブロックリストがサーバー上のブロックリストと一致しない場合、エラーが発生します。
  - **commit** に失敗した場合は、再試行してください。

- 6 種類のステータスの説明は、次の通りです。
  1. **UNKNOWN:** セッションが作成されたときの初期値です。
  2. **NORMAL:** アップロードオブジェクトは、正常に作成されています。
  3. **CLOSING:** `complete` が呼び出されると、ステータスが **CLOSING** に変更されます。
  4. **CLOSED:** アップロードは完了しています。 正確には、データが結果テーブルのあるディレクトリに移動されています。
  5. **EXPIRED:** アップロードセッションはタイムアウトしています。
  6. **CRITICAL:** サービスエラーが発生しています。

 注 :

- 同一 `UploadSession` 内にある `blockId` は、一意でなければなりません。**1** つの `UploadSession`で、`blockId` を使用して `RecordWriter` を開く場合は、データを一括で書き込んでから `close` を呼び出し、その後 **commit** を呼び出します。 別の `RecordWriter` を使用してデータを書き込む場合、同じ `blockID` を使用しないでください。
- ブロックの最大サイズは **100 GB** です。**64 MB** より大きい方が望ましいです。
- サーバー上の各セッションのしきい値は、**24** 時間です。
- データのアップロード中、**Writer** によってデータが **8 KB** 書き込まれるたびにネットワーク動作がトリガーされます。 **120** 秒間、ネットワーク動作がトリガーされない場合、サーバーによって接続が閉じられます。 この場合、**Writer** は使用できなくなり、新しい接続を開く必要があります。
- データのアップロードには `openBufferedWriter` インターフェイスを使用することを推奨します。 このインターフェイスには、`blockId` の詳細は表示されません。アップロード失敗時の自動再試行用の内部データキャッシュが含まれます。 詳細は、アップロード手順および`TunnelBufferedWriter` の例をご参照ください。

4.4.4 DownloadSession

DownloadSession インターフェイスは、次のように定義されています。

```
public class DownloadSession {
        DownloadSession(Configuration conf, String projectName, String
 tableName,
            String partitionSpec) throws TunnelException
        DownloadSession(Configuration conf, String projectName, String
 tableName,
            String partitionSpec, String downloadId) throws TunnelExce
ption
        public String getId()
        public long getRecordCount()
        public TableSchema getSchema()
```

```
        public DownloadSession.Status getStatus()
        public RecordReader openRecordReader(long start, long count)
        public RecordReader openRecordReader(long start, long count,
boolean compress)
    }
```

パラメーター：

- ライフサイクル: ダウンロードインスタンスの作成から始まり、ダウンロードが完了すると終わります。
- ダウンロードインスタンスの作成: インスタンスを作成するには、**Constructor** を呼び出すか、**TableTunnel** を使用します。
  - リクエストモード: 同期
  - このダウンロードインスタンスのセッションがサーバーで作成され、一意のダウンロード **ID** が生成されます。 クライアントから **getId** を実行して、ダウンロード **ID** を取得します。
  - この操作はコストが高くなります。 データファイルのインデックスが、サーバーで作成されます。 大容量ファイルは、通常ダウンロードに時間がかかります。
  - 同時に、サーバーによってレコード総数が返されます。このレコード総数に基づいて、複数の同時ダウンロードが開始されます。
- データのダウンロード:
  - リクエストモード: 非同期
  - **RecordReader** インスタンスを作成するには、`openRecordReader` メソッドを呼び出します。「**start**」は、レコードのダウンロード開始位置を示し、**0** より小さくはできません。「**count**」はダウンロード対象のレコード数を示します。**0** より大きくする必要があります。
- ダウンロード情報の照会:
  - リクエストモード: 同期
  - 現行のダウンロードステータスを取得するには、`getStatus` を呼び出します。
- ステータスは、次の **4** 種類です。
  - **UNKNOWN:** セッションが作成されたときの初期値です。
  - **NORMAL:** ダウンロードオブジェクトは、正常に作成されています。
  - **CLOSED:** ダウンロードは完了しています。
  - **EXPIRED:** ダウンロードセッションはタイムアウトしています。

4.4.5 TunnelBufferedWriter

アップロードプロセスを実行するには、次の手順に従ってください。

1. データを分割します。
2. openRecordWriter (id) を呼び出して、各データブロックのブロック **ID** を指定します。
3. **1** つ以上のスレッドを使用して、ブロックをアップロードします。 **1** つのブロックのアップロードが失敗すると、全ブロックをアップロードし直さなくてはなりません。
4. すべてのブロックのアップロードが完了したら、アップロードした **ID** のリストをサーバーに渡して検証を実行します。 session.commit([1,2,3,…])を呼び出して、この処理を実行します。

   接続タイムアウトなどのサーバーブロックマネージャーの制限事項により、アップロードプロセスのロジックは複雑です。 このようなプロセスを簡便化するため、**SDK** には高度な **RecordWriter—TunnelBufferWriter** インターフェイスが実装されています。

このインターフェイスは、次のように定義されています。

```
public class TunnelBufferedWriter implements RecordWriter {
        public TunnelBufferedWriter(TableTunnel.UploadSession session
, CompressOption option) throws IOException;
        public long getTotalBytes();
        public void setBufferSize(long bufferSize);
        public void setRetryStrategy(RetryStrategy strategy);
        public void write(Record r) throws IOException;
        public void close() throws IOException;
    }
```

パラメーター

- ライフサイクル**: RecordWriter** の作成から始まり、データのアップロードが完了すると終わります。
- **TunnelBufferedWriter** インスタンスの作成**: UploadSession** の openBufferedWriter インターフェイスを呼び出してインスタンスを作成します。
- データのアップロード**: Write** インターフェイスを呼び出します。 データは、まずローカルキャッシュに書き込まれます。 キャッシュがいっぱいになると、接続タイムアウトを避けるため、データは一括でサーバーに送信されます。 アップロードが失敗した場合、自動再試行がサポートされます。
- アップロードの完了**:** Close インターフェイスを呼び出し、**UploadSession** の **Commit** インターフェイスを呼び出してアップロードを完了します。
- バッファ制御**:** バッファが占有するメモリのサイズ (バイト単位) を変更するには、setBufferSize インターフェイスを使用します。**64 MB** を超えると、サーバーのパ

フォーマンスが低下する可能性があります。 通常、追加設定なしで、このパラメーターのデフォルト値が使用されます。

- 再試行ポリシーの設定: **EXPONENTIAL_BACKOFF**、**LINEAR_BACKOFF**、および **CONSTANT_BACKOFF** の 3 つの再試行回避ポリシーを選択できます。 例:次のコードセグメントは、書き込み再試行回数を **6** に設定しています。 不必要な再試行を避けるために、再試行は、**4** 秒、**8** 秒、**16** 秒、**32** 秒、**64** 秒、および **128** 秒 (指数関数的に増加) 後に実行されます。 これはデフォルト設定であり、基本的には変更しないでください。

```
RetryStrategy retry
  = new RetryStrategy(6, 4, RetryStrategy.BackoffStrategy.EXPONENTIA
L_BACKOFF)
writer = (TunnelBufferedWriter) uploadSession.openBufferedWriter();
writer.setRetryStrategy(retry);
```

4.5 大量データチャネル SDK の例

4.5.1 例

ここでは、**Tunnel** サービスアドレスと **Tunnel Buffered Writer** を使用してデータをアップロードする際の推奨事項を紹介します。

- **MaxCompute** には、**2** つのサービスアドレスが用意されています。 **Tunnel** サービスアドレスを選択すると、データアップロードの効率と料金に影響することがあります。詳細については、「*Tunnel SDK 概要 (Tunnel SDK overview)*」をご参照ください。
- データをアップロードする場合、**TunnelBufferedWriter** インターフェイスを使用することを推奨します。 詳細は、「*BufferedWriter*」のサンプルコードをご参照ください。
- 操作は、**SDK** のバージョンによって異なる場合があります。 この例は、参考用にご利用ください。 操作を進める前に、バージョン間の相違点を考慮してください。

4.5.2 アップロード例

```
import java.io.IOException;
 import java.util.Date;
 import com.aliyun.odps.Column;
 import com.aliyun.odps.Odps;
 import com.aliyun.odps.PartitionSpec;
 import com.aliyun.odps.TableSchema;
 import com.aliyun.odps.account.Account;
 import com.aliyun.odps.account.AliyunAccount;
 import com.aliyun.odps.data.Record;
 import com.aliyun.odps.data.RecordWriter;
 import com.aliyun.odps.tunnel.TableTunnel;
 import com.aliyun.odps.tunnel.TunnelException;
 import com.aliyun.odps.tunnel.TableTunnel.UploadSession;
 public class UploadSample {
         private static String accessId = "<your access id>";
         private static String accessKey = "<your access Key>";
```

```
        private static String odpsUrl = "http://service.odps.aliyun.
com/api";
        private static String tunnelUrl = "http://dt.cn-shanghai.
maxcompute.aliyun-inc.com";
                        //The tunnelURL must be set if you need to
connect internal network, otherwise, the system uses public network as
 default. The example shows the Tunnel Endpoint of classical network
in HuaDong 2, for other regions, see Access domain and data centers.
        private static String project = "<your project>";
        private static String table = "<your table name>";
        private static String partition = "<your partition spec>";
        public static void main (String args []) {
                Account account = new AliyunAccount(accessId,
accessKey);
                Odps odps = new Odps(account);
                Odps 。 setEndpoint (odpsUrl);
                odps.setDefaultProject(project);
                try {
                        TableTunnel tunnel = new TableTunnel(odps);
                        tunnel.setEndpoint(tunnelUrl);  //set
tunnelUrl
                        PartitionSpec partitionSpec = new PartitionS
pec(partition);
                        UploadSession uploadSession = tunnel.
createUploadSession(project,
                                        table, partitionSpec);
                        System.out.println("Session Status is : "
                                        + uploadSession.getStatus().
toString());
                        TableSchema schema = uploadSession.getSchema
();
                         // After preparing data, open a Writer to
start writing data. The prepared data is written to one block.
                         // When the data written to individual
 blocks is too small, the system will produce a large number of
small files, seriously degrading computing performance. We strongly
 recommend over 64 MB of data be written each time (up to 100 GB of
data can be written to the same block).
                         // You can use the average data volume and
record count to estimate the total value. For example: 64MB < Average
data size x Record count < 100GB.
                        RecordWriter recordWriter = uploadSession 。
openRecordWriter (0);
                        Record record = uploadSession.newRecord();
                        for (int i = 0; i < schema.getColumns().size
(); i++) {
                                Column column = schema.getColumn(i);
                                switch (column.getType()) {
                                case BIGINT:
                                        record.setBigint(i, 1L);
                                        break;
                                Case Boolean:
                                        record.setBoolean(i, true);
                                        break;
                                case DATETIME:
                                        record.setDatetime(i, new
Date());
                                        break;
                                case DOUBLE:
                                        record.setDouble(i, 0.0);
                                        break;
                                case STRING:
                                        record.setString(i, "sample
");
```

```
                                        break;
                                    default:
                                        throw new RuntimeException("
Unknown column type: "
                                                + column.
getType());
                                    }
                            }
                            for (int i = 0; i < 10; i++) {
                                 // Writes data to the server. Each 8
 KB of data written triggers a network transmission.
                                 // If no network transmission occurs
 for 120 seconds, the server closes the connection. At this time, the
Writer becomes unavailable and you must write data again.
                                recordWriter.write(record);
                            }
                            recordWriter.close();
                            uploadSession.commit(new Long[]{0L});
                            System.out.println("upload success!") ;
                    } catch (TunnelException e) {
                            e.printStackTrace();
                    } catch (IOException e) {
                            e.printStackTrace();
                    }
            }
 }
```

コンストラクタ:

PartitionSpec(String spec):このオブジェクトのクラスの作成に文字列を使用します。

パラメーター

spec: パーティション定義文字列 (例：**pt=’1’,ds=’2’**)。

このプログラムでは、次のように設定する必要があります。

```
private static String partition = “pt=’XXX’,ds=’XXX’”;
```



注：

ここでは、中国東部 2 クラシックネットワークの **Tunnel Endpoint** を指定しています。

他のリージョンの **Tunnel Endpoint** 設定については、「エンドポイントの設定 *(Configure Endpoint)*」をご参照ください。

4.5.3 ##下#示例

```
import java.io.IOException;
 import java.util.Date;
 import com.aliyun.odps.Column;
 import com.aliyun.odps.Odps;
 import com.aliyun.odps.PartitionSpec;
 import com.aliyun.odps.TableSchema;
 import com.aliyun.odps.account.Account;
 import com.aliyun.odps.account.AliyunAccount;
 import com.aliyun.odps.data.Record;
 import com.aliyun.odps.data.RecordReader;
```

```
 import com.aliyun.odps.tunnel.TableTunnel;
 import com.aliyun.odps.tunnel.TableTunnel.DownloadSession;
 import com.aliyun.odps.tunnel.TunnelException;
 public class DownloadSample {
         private static String accessId = "<your access id>";
         private static String accessKey = "<your access Key>";
         private static String odpsUrl = "http://service.odps.aliyun.
com/api";
         private static String tunnelUrl = "http://dt.cn-shanghai.
maxcompute.aliyun-inc.com";
                         //#置tunnelUrl，若需要走内网#必##置，否#默#公网。此
##的是##2#典网#Tunnel Endpoint，其他region可以参考文档《##域名和数据中心》。
         private static String project = "<your project>";
         private static String table = "<your table name>";
         private static String partition = "<your partition spec>";
         public static void main (String args []) {
                 Account account = new AliyunAccount(accessId,
accessKey);
                 Odps odps = new Odps(account);
                 odps.setEndpoint(odpsUrl);
                 odps.setDefaultProject(project);
                 TableTunnel tunnel = new TableTunnel(odps);
                 tunnel.setEndpoint(tunnelUrl);//tunnelUrl#置
                 PartitionSpec partitionSpec = new PartitionSpec(
partition);
                 try {
                         DownloadSession downloadSession = tunnel.
createDownloadSession(project, table,
                                         partitionSpec);
                         System.out.println("Session Status is : "
                                         + downloadSession.getStatus
().toString());
                         long count = downloadSession.getRecordCount
();
                         System.out.println("RecordCount is: " + count
);
                         RecordReader recordReader = downloadSession.
openRecordReader(0,
                                         count);
                         Record record;
                         while ((record = recordReader.read()) != null
) {
                                 consumeRecord(record, downloadSession
.getSchema());
                         }
                         recordReader.close();
                 } catch (TunnelException e) {
                         e.printStackTrace();
                 } catch (IOException e1) {
                         e1.printStackTrace();
                 }
         }
         private static void consumeRecord(Record record, TableSchema
schema) {
                 for (int i = 0; i < schema.getColumns().size(); i++)
 {
                         Column column = schema.getColumn(i);
                         String colValue = null;
                         switch (column.getType()) {
                         case BIGINT: {
                                 Long v = record.getBigint(i);
                                 colValue = v == null ? null : v.
toString();
                                 break;
```

```
                        }
                        case BOOLEAN: {
                                Boolean v = record.getBoolean(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case DATETIME: {
                                Date v = record.getDatetime(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case DOUBLE: {
                                Double v = record.getDouble(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case STRING: {
                                String v = record.getString(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        default:
                                throw new RuntimeException("Unknown
column type: "
                                                + column.getType());
                        }
                        System.out.print(colValue == null ? "null" :
colValue);
                        if (i != schema.getColumns().size())
                                System.out.print("\t");
                }
                System.out.println();
        }
 }
```

上記の例では、テスト用としてデータを **System.out.println** で直接表示していますが、実際に使用する際は、テキストファイルに直接出力するように書き換えることができます。

4.5.4 マルチスレッドアップロードの例

```
import java.io.IOException;
 import java.util.ArrayList;
 import java.util.Date;
 import java.util.concurrent.Callable;
 import java.util.concurrent.ExecutorService;
 import java.util.concurrent.Executors;
 import com.aliyun.odps.Column;
 import com.aliyun.odps.Odps;
 import com.aliyun.odps.PartitionSpec;
 import com.aliyun.odps.TableSchema;
 import com.aliyun.odps.account.Account;
 import com.aliyun.odps.account.AliyunAccount;
 import com.aliyun.odps.data.Record;
 import com.aliyun.odps.data.RecordWriter;
 import com.aliyun.odps.tunnel.TableTunnel;
 import com.aliyun.odps.tunnel.TunnelException;
 import com.aliyun.odps.tunnel.TableTunnel.UploadSession;
```

```
 class UploadThread implements Callable<Boolean> {
         private long id;
         private RecordWriter recordWriter;
         private Record record;
         private TableSchema tableSchema;
         public UploadThread(long id, RecordWriter recordWriter,
Record record,
                         TableSchema tableSchema) {
                 this.id = id;
                 this.recordWriter = recordWriter;
                 this.record = record;
                 this.tableSchema = tableSchema;
         }
         @Override
         public Boolean call() {
                 for (int i = 0; i < tableSchema.getColumns().size();
i++) {
                         Column column = tableSchema.getColumn(i);
                         switch (column.getType()) {
                         Case bigint:
                                 record.setBigint(i, 1L);
                                 Break;
                         Case Boolean:
                                 record.setBoolean(i, true);
                                 break;
                         case DATETIME:
                                 record.setDatetime(i, new Date());
                                 break;
                         case DOUBLE:
                                 record.setDouble(i, 0.0);
                                 break;
                         case STRING:
                                 record.setString(i, "sample");
                                 break;
                         default:
                                 throw new RuntimeException("Unknown
column type: "
                                                 + column.getType());
                         }
                 }
                 for (int i = 0; i < 10; i++) {
                         try {
                                 recordWriter.write(record);
                         } catch (IOException e) {
                                 recordWriter.close();
                                 e.printStackTrace();
                                 return false;
                         }
                 }
                 recordWriter.close();
                 return true;
         }
 }
 public class UploadThreadSample {
         private static String accessId = "<your access id>";
         private static String accessKey = "<your access Key>";
         private static String odpsUrl = "<http://service.odps.aliyun.
com/api>";
         private static String tunnelUrl = "<http://dt.cn-shanghai.
maxcompute.aliyun-inc.com>";
                         //The tunnelURL must be set if you need to
connect internal network, otherwise, the system uses public network as
 default. The example shows the Tunnel Endpoint of classical network
in HuaDong 2, for other regions, see Access domain and data centers.
```

```
        private static String project = "<your project>";
        private static String table = "<your table name>";
        private static String partition = "<your partition spec>";
        private static int threadNum = 10;
        public static void main(String args[]) {
                Account account = new AliyunAccount(accessId,
accessKey);
                Odps odps = new Odps(account);
                odps.setEndpoint(odpsUrl);
                odps.setDefaultProject(project);
                try {
                        TableTunnel tunnel = new TableTunnel(odps);
                        tunnel.setEndpoint(tunnelUrl);  //set
tunnelUrl
                        PartitionSpec partitionSpec = new PartitionS
pec(partition);
                        UploadSession uploadSession = tunnel.
createUploadSession(project,
                                        table, partitionSpec);
                        System.out.println("Session Status is : "
                                        + uploadSession.getStatus().
toString());
                        ExecutorService pool = Executors.newFixedTh
readPool(threadNum);
                        ArrayList<Callable<Boolean>> callers = new
ArrayList<Callable<Boolean>>();
                        for (int i = 0; i < threadNum; i++) {
                                RecordWriter recordWriter =
uploadSession.openRecordWriter(i);
                                Record record = uploadSession.
newRecord();
                                callers.add(new UploadThread(i,
recordWriter, record,
                                                uploadSession.
getSchema()));
                        }
                        pool.invokeAll(callers);
                        pool.shutdown();
                        Long[] blockList = new Long[threadNum];
                        for (int i = 0; i < threadNum; i++)
                                blockList[i] = Long.valueOf(i);
                        uploadSession.commit(blockList);
                        System.out.println("upload success!") ;
                } catch (TunnelException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
 }
```

Tunnel エンドポイントは、指定しても空白でも構いません。

- 指定すると、データのアップロードは、指定したエンドポイントを介して実行されます。
- 指定しない場合、データのアップロードは、パブリックネットワークを介して実行されます。

- この例では、**East China 2** クラシックネットワークを介してアップロードが実行されています。 他のリージョンの **Tunnel** エンドポイントの設定については、「*エンドポイントの設定 (Configure Endpoint)*」をご参照ください。

4.5.5 マルチスレッドダウンロードの例

ここでは、**TableTunnel** インターフェイスを使用してマルチスレッドダウンロードを実行するためのコード例を紹介します。

```
import java.io.IOException;
 import java.util.ArrayList;
 import java.util.Date;
 import java.util.List;
 import java.util.concurrent.Callable;
 import java.util.concurrent.ExecutionException;
 import java.util.concurrent.ExecutorService;
 import java.util.concurrent.Executors;
 import java.util.concurrent.Future;
 import com.aliyun.odps.Column;
 import com.aliyun.odps.Odps;
 import com.aliyun.odps.PartitionSpec;
 import com.aliyun.odps.TableSchema;
 import com.aliyun.odps.account.Account;
 import com.aliyun.odps.account.AliyunAccount;
 import com.aliyun.odps.data.Record;
 import com.aliyun.odps.data.RecordReader;
 import com.aliyun.odps.tunnel.TableTunnel;
 import com.aliyun.odps.tunnel.TableTunnel.DownloadSession;
 import com.aliyun.odps.tunnel.TunnelException;
 class DownloadThread implements Callable<Long> {
         private long id;
         private RecordReader recordReader;
         private TableSchema tableSchema;
         public DownloadThread(int id,
                         RecordReader recordReader, TableSchema
tableSchema) {
                 this.id = id;
                 this.recordReader = recordReader;
                 this.tableSchema = tableSchema;
         }
         @Override
         public Long call() {
                 Long recordNum = 0L;
                 try {
                         Record record;
                         while ((record = recordReader.read()) ! =
null) {
                                 recordNum++;
                                 System.out.print("Thread " + id + "\t
");
                                 consumeRecord(record, tableSchema);
                         }
                         recordReader.close();
                 } catch (IOException e) {
                         e.printStackTrace();
                 }
                 return recordNum;
         }
         private static void consumeRecord(Record record, TableSchema
schema) {
```

```
                for (int i = 0; i < schema.getColumns().size(); i++)
 {
                        Column column = schema.getColumn(i);
                        String colValue = null;
                        switch (column.getType()) {
                        case BIGINT: {
                                Long v = record.getBigint(i);
                                colValue = v == null ? null : v.
toString();
                                Break;
                        }
                        case BOOLEAN: {
                                Boolean v = record.getBoolean(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case DATETIME: {
                                Date v = record.getDatetime(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case DOUBLE: {
                                Double v = record.getDouble(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        case STRING: {
                                String v = record.getString(i);
                                colValue = v == null ? null : v.
toString();
                                break;
                        }
                        Default:
                                throw new RuntimeException("Unknown
column type: "
                                                + column.getType());
                        }
                        System.out.print(colValue == null ? "null" :
colValue);
                        If (I! = schema.getColumns().size())
                                System.out.print("\t");
                }
                System.out.println();
        }
 }
 public class DownloadThreadSample {
         private static String accessId = "<your access id>";
         private static String accessKey = "<your access Key>";
         private static String odpsUrl = "http://service.odps.aliyun.
com/api";
         private static String tunnelUrl = "http://dt.cn-shanghai.
maxcompute.aliyun-inc.com";
                        //The tunnelURL must be set if you need to
connect internal network, otherwise, the system uses public network as
 default. The example shows the Tunnel Endpoint of classical network
in HuaDong 2, for other regions, see Access domain and data centers.
         private static String project = "<your project>";
         private static String table = "<your table name>";
         private static String partition = "<your partition spec>";
         private static int threadNum = 10;
         public static void main(String args[]) {
```

```
                    Account account = new AliyunAccount(accessId,
accessKey);
                    Odps odps = new Odps(account);
                    odps.setEndpoint(odpsUrl);
                    odps.setDefaultProject(project);
                    TableTunnel tunnel = new TableTunnel(odps);
                    tunnel.setEndpoint(tunnelUrl);  //set tunnelUrl
                    PartitionSpec partitionSpec = new PartitionSpec(
partition);
                    DownloadSession downloadSession;
                    try {
                            downloadSession = tunnel.createDownloadSessio
n(project, table,
                                            partitionSpec);
                            System.out.println("Session Status is : "
                                            + downloadSession.getStatus
().toString());
                            long count = downloadSession.getRecordCount
();
                            System.out.println("RecordCount is: " + count
);
                            ExecutorService pool = Executors.newFixedTh
readPool(threadNum);
                            ArrayList<Callable<Long>> callers = new
ArrayList<Callable<Long>>();
                            long start = 0;
                            long step = count / threadNum;
                            for (int i = 0; i < threadNum - 1; i++) {
                                    RecordReader recordReader =
downloadSession.openRecordReader(
                                                    step * i, step);
                                    callers.add(new DownloadThread( i,
recordReader, downloadSession.getSchema()));
                            }
                            RecordReader recordReader = downloadSession.
openRecordReader(step * (threadNum - 1), count
                                            - ((threadNum - 1) * step));
                            callers.add(new DownloadThread( threadNum - 1
, recordReader, downloadSession.getSchema()));
                            Long downloadNum = 0L;
                            List<Future<Long>> recordNum = pool.invokeAll
(callers);
                            for (Future<Long> num : recordNum)
                                    downloadNum += num.get();
                            System.out.println("Record Count is: " +
downloadNum);
                            pool.shutdown();
                    } catch (TunnelException e) {
                            e.printStackTrace();
                    } catch (IOException e) {
                            e.printStackTrace();
                    } catch (InterruptedException e) {
                            e.printStackTrace();
                    } catch (ExecutionException e) {
                            e.printStackTrace();
                    }
            }
     }
```



注：

Tunnel エンドポイントは、指定しても空白でも構いません。

- 指定すると、データのダウンロードは、指定したエンドポイントを介して実行されます。
- 指定しない場合、データのダウンロードはパブリックネットワークを介して実行されます。
- 上記例では、**East China 2** クラシックネットワークの **Tunnel** エンドポイントを介してダウンロードが実行されています。 他のリージョンの **Tunnel** エンドポイントの設定については、「エンドポイントの設定 *(Configure Endpoint)*」をご参照ください。

4.5.6 BufferedWriter アップロードの例

ここでは、**BufferedWriter** インターフェイスを使用してデータをアップロードするためのコード例を紹介します。

```
// Initializes MaxCompute and Tunnel code
RecordWriter writer = null;
TableTunnel.UploadSession uploadSession = tunnel.createUploadSession(
projectName, tableName);
try {
  int i = 0;
  // Generates TunnelBufferedWriter instance
  writer = uploadSession.openBufferedWriter();
  Record product = uploadSession.newRecord();
  for (String item : items) {
    product.setString("name", item);
    product.setBigint("id", i);
    // Calls the Write interface to write data
    writer.write(product);
    i += 1;

} finally {
  if (writer ! = null) {
    // Closes TunnelBufferedWriter
    writer.close();


// Submits data via uploadSession to end the upload process
uploadSession.commit();
```

4.5.7 BufferedWriter マルチスレッドアップロードの例

ここでは、**BufferedWriter** インターフェイスを使用してマルチスレッドアップロードを理解するためのコード例を紹介します。

```
class UploadThread extends Thread {
  private UploadSession session;
  private static int RECORD_COUNT = 1200;
  public UploadThread(UploadSession session) {
    this.session = session;
  }
  @Override
  Public void run (){
    RecordWriter writer = up.openBufferedWriter();
    Record r = up.newRecord();
    for (int i = 0; i < RECORD_COUNT; i++) {
      r.setBigint(0, i);
```

```
      writer.write(r);
    }
    writer.close();
  }
};
public class Example {
  public static void main(String args[]) {
   // Initializes MaxCompute and Tunnel code
   TableTunnel.UploadSession uploadSession = tunnel.createUplo
adSession(projectName, tableName);
   UploadThread t1 = new UploadThread(up);
   UploadThread t2 = new UploadThread(up);
   t1.start();
   t2.start();
   t1.join();
   t2.join();
   uploadSession.commit();
 }
```

4.6 Data Integration を使用したデータのインポートまたはエクスポート

DataWorks の *Data Integration* 機能を使用すると、データの同期タスクを作成し、**MaxCompute** データをインポートおよびエクスポートできます。

前提条件

データをインポートまたはエクスポートする前に、事前準備が完了している必要があります。 詳細は、「*Alibaba Cloud* アカウントの準備」および「プロジェクトの購入と作成」をご参照ください。

MaxCompute データソースの追加

注：

- プロジェクト管理者のみがデータソースを作成できます。 他のロールは、データソースの閲覧のみ可能です。
- 追加するデータソースが現行の **MaxCompute** プロジェクトの場合、この操作を省略してください。 **MaxCompute** プロジェクトを作成して **Data Integration** のデータソースとして表示すると、このプロジェクトはデフォルトで **odps_first** という名前の **MaxCompute** データソースとして追加されます。

手順

1. *DataWorks* コンソールに管理者としてログインし、 **[Project List]** の該当するプロジェクトの操作列で、**[Enter Workspace]** をクリックします。

2. 上部ナビゲーションウィンドウの **[Data Integration]** をクリックします。 左側のナビゲーションウィンドウで **[Data Source]** をクリックします。
3. **[New Source]** をクリックし、**[Large Data Storage]** セクションで **[MaxCompute (ODPS)]** を選択します。
4. データのダイアログボックスで、必要な設定を入力します。

   パラメーター

   - `Name`: 英数字とアンダースコア (_) を使用できます。 文字またはアンダースコアで始まる必要があり、60 文字を超えることはできません。
   - `Data source description`: データソースの簡単な説明を入力します。80 文字を超えることはできません。
   - `Data source type`: 現時点では、**ODPS** のみ選択可能です。
   - `ODPS Endpoint`: デフォルトでは読み取り専用です。 システム構成から自動的に設定値が読み込まれます。
   - `ODPS Item name`: 対応する **MaxCompute** プロジェクトを識別するためのプロジェクト名です。
   - `Access ID`: **MaxCompute** プロジェクトオーナーのアカウントに関連付けられた **Access ID** です。
   - `AccessKey`: **MaxCompute** プロジェクトオーナーのアカウントに関連付けられた **AccessKey** で、**Access ID** とペアで使用します。
5. (任意) 関連フィールドに必要な情報をすべて入力した後、接続をテストするには、**[Test Connectivity]** をクリックします。
6. 接続テストが成功したら、**[Save]** をクリックします。

 注 :

その他のデータソースの設定については、「データソースの設定」をご参照ください。

Data Integration を使用したデータのインポート

MaxCompute に **MySQL** データをインポートする例を紹介します。同期タスクは、**Wizard** モードまたは **Script** モードで設定できます。

Wizard モードで同期タスクを設定

1. **Wizard** モードで同期タスクを作成します。

2. ソースを選択します。

   MySQL データソースとソーステーブル **"mytest"** を選択します。 デフォルトでは、データ表示領域は折り畳まれています。 **[Next]** をクリックします。

3. ターゲットを選択します。

   ターゲットは、作成済みの **MaxCompute** テーブルにする必要があります。 **[Quick Table Creation]** をクリックして、新規テーブルを作成することもできます。

   パラメーター

   - `Partition information:` パーティションのレベルを指定します。 **3** レベルのパーティションを持つテーブルにデータを書き込む場合は、最下位レベルのパーティションを指定する必要があります。たとえば、**pt=20150101, type=1, biz=2** と指定します。 この項目は、非パーティションテーブルでは使用できません。
   - `Data clearing rules:`
     - `Clear existing data before writing:` データをテーブルまたはパーティションにインポートする前に、テーブルまたはパーティション内のすべてのデータがクリアされます。これは、「上書き挿入」と同じです。
     - `Retain existing data before writing:` 新しいデータをインポートする際、既存のデータはクリアされません。 インポート操作のたびに、新しいデータが追加されます。これは、「挿入」と同じです。

4. フィールドをマッピングします。

   フィールド間のマッピングを選択します。 フィールドのマッピング関係を設定します。 左側の **Source Table Fields** は、右側の **Target Table Fields** と **1** 対 **1** で対応しています。

5. チャネルを制御します。

   [Next] をクリックして、最大ジョブレートとダーティデータチェックルールを設定します。

   パラメーター

   - `Maximum job rate`: データ同期ジョブで可能な最大レートを決定します。 ジョブの実際のレートは、ネットワーク環境、データベース構成、およびその他の要因によって異なる場合があります。
   - `Concurrent job count`: 単一の同期ジョブの場合、同時ジョブ数* 個々のジョブ送信レート = ジョブ送信レートの合計になります。

   Maximum job rate が指定されている場合の同時ジョブ数の選択方法

   - データソースがオンラインのビジネスデータベースの場合は、オンラインデータベースとの干渉を避けるために、同時ジョブ数に大きな値を設定しないことを推奨します。
   - 高いデータ同期レートが必要な場合は、最も高いジョブレートと大きな同時ジョブ数を選択することを推奨します。

6. プレビューをした後に保存します。

   タスクの設定が正しいことを確認し、**[Save]** をクリックします。

同期タスクの実行

同期タスクを直接実行

システム変数パラメーターが同期タスクに設定されている場合、タスク操作中に変数パラメーター設定ウィンドウが表示されます。

タスクの保存後、**[Run]** をクリックすると、すぐにタスクが実行されます。**[Submit]** をクリックすると、同期タスクが **DataWorks** のスケジューリングシステムに送信されます。 スケジューリングシステムでは、設定属性に従って、タスクが翌日から自動的かつ定期的に実行されます。スケジュール設定の詳細は、「スケジュール設定」をご参照ください。

Script モードで同期タスクを設定

次のスクリプトを使用して、同期タスクを設定できます。 その他の設定とジョブの操作は **Wizard** モードと同じです。

```
{
  "type": "job",
  "version": "1.0",
  "configuration": {
    "reader": {
      "plugin": "mysql",
      "parameter": {
        "datasource": "mysql",
        "where": "",
```

```
        "splitPk": "id",
        "connection": [
          {
            "table": [
              "person"
            ],
            "datasource": "mysql"
          }
        ],
        "connectionTable": "person",
        "Column ":[
          "id",
          "name"
        ]
      }
    },
    "writer": {
      "plugin": "odps",
      "parameter": {
        "datasource": "odps_first",
        "table": "a1",
        "truncate": true,
        "partition": "pt=${bdp.system.bizdate}",
        "Column ":[
          "id",
          "col1"
        ]
      }
    },
    "Setting ":{
      "speed": {
        "mbps": "1",
        "concurrent": "1"
      }
    }
  }
}
```

参照情報

- **Reader** での各種データソースの設定については、「*Reader プラグインの設定*」をご参照ください。
- **Writer** での各種データソースの設定については、「*Writer プラグインの設定*」をご参照ください。

5 SQL

5.1 SQL 概要

ここでは、**MaxCompute SQL** のキーワード、型変換の手順、パーティションテーブル、**UNION ALL** 操作、および使用上の制限について説明します。

SQL 概要

MaxCompute SQL は、さまざまなシナリオに適しています。 大量データ (**GB**、**TB**、**EB** レベル) は、オフラインのバッチ処理で対応する必要があります。 ジョブの送信後、スケジュールに数秒から数分かかります。 このため、**MaxCompute SQL**は、**1** 秒当たり数万トランザクションを処理するサービスに向いています。

MaxCompute SQL 構文は、**SQL** と似ているので、標準 **SQL** のサブセットとみなすことができます。 しかし、**MaxCompute SQL** とデータベースを混同しないでください。 **MaxCompute SQL** は、トランザクション、主キー制約、インデックスなどのデータベースの特徴を備えていません。 **MaxCompute SQL** の最大サイズは **2 MB**です。

予約語

MaxCompute では、**SQL** 文のキーワードは予約語として認識されます。 テーブル、列、またはパーティションの名前にキーワードを使用する場合、``` `` ``` 記号を使用してキーワードのエスケープ処理を行う必要があります。エスケープ処理を行わないと、エラーが発生します。 予約語は、大文字と小文字が区別されません。以下は、最も一般的に使用される予約語です (予約語のリストは、「*MaxCompute SQL 予約語*」をご参照ください)。

```
% & && ( ) * +
 - . / ; < <= <>
 =    >    >=    ?    ADD ALL ALTER
 AND AS ASC BETWEEN BIGINT BOOLEAN BY
 CASE CAST COLUMN COMMENT CREATE DESC DISTINCT
 DISTRIBUTE DOUBLE DROP ELSE FALSE FROM FULL
 GROUP IF IN INSERT INTO IS JOIN
 LEFT LIFECYCLE LIKE LIMIT MAPJOIN NOT NULL
 ON OR ORDER OUTER OVERWRITE PARTITION RENAME
 REPLACE RIGHT RLIKE SELECT SORT STRING TABLE
 THEN TOUCH TRUE UNION VIEW WHEN WHERE
```

型変換

MaxCompute SQL では、データ型の変換が可能です。 型変換の方法には、明示的な型変換と暗黙的な型変換があります。 詳細は、「型変換」をご参照ください。

- 明示的変換: **cast** 演算子でデータ型の値を変換します。
- 暗黙的変換 : コンテキスト環境と変換規則に従って、**MaxCompute** 稼働中に暗黙的な型変換が自動的に実行されます。 暗黙的変換の範囲には、さまざまな演算子、組み込み関数などがあります。

パーティションテーブル

MaxCompute SQL は、パーティションテーブルに対応しています。 パーティションを指定すると、操作を簡素化できます。 たとえば、**SQL** 実行効率の向上、コストの削減などが実現します。 詳細は、「基本概念」の「パーティション」をご参照ください。

UNION ALL

UNION ALL 操作に関連するすべての列のデータ型、列数、および列名は同じである必要があります。異なる場合は、エラーが発生します。

5.2 SQL の制限

制限事項を考慮し、サービスが中断しないようにする必要があります。 **MaxCompute SQL** の制限は次のとおりです。

名前	最大値/制限	分類	説明
テーブル名の長さ	**128** バイト	長さ制限	テーブル名と列名に特殊文字を含めることはできません。 英字 **(a ～ z、A ～ Z)**、数字、および下線（_）のみを使用できます。
注釈の長さ	**1,024** バイト	長さ制限	注釈には最大 **1,024** バイトの有効な文字列を使用できます。
列定義	**1,200**	数量制限	**1** つのテーブルには最大 **1,200** 列を定義できます。
パーティション	**60,000**	数量制限	**1** つのテーブルには最大 **60,000** のパーティションを指定できます。
テーブルのパーティション階層	**6** 階層	数量制限	テーブルのパーティションを最大 **6** 階層にできます。
統計定義	**100**	数量制限	**1** つのテーブルに最大 **100** 個の統計定義を含めることができます。
統計定義	**64,000**	長さ制限	統計定義には最大 **64,000** バイトを使用できます。

名前	最大値/制限	分類	説明
画面表示	**10,000** 行	数量制限	**SELECT** 文の画面表示には、最大 **10,000** 行が出力されます。
INSERT ターゲット	**256**	数量制限	マルチイン操作では、一度に最大 **256** のターゲットを挿入できます。
UNION ALL	**256**	数量制限	**UNION ALL** 操作は、最大 **256** 個のテーブルに対して実行できます。
MAPJOIN	**8** つの小さなテーブル	数量制限	**MAPJOIN** 操作は、最大 **8** つの小さなテーブルに対して実行できます。
MAPJOIN メモリ制限	**512 MB**	数量制限	**MAPJOIN** 操作が実行されるすべての小さなテーブルのメモリサイズは、**512 MB** を超えることはできません。
Window 関数	**5**	数量制限	**SELECT** 文には、最大 **5** つの **Window** 関数を指定できます。
ptinsubq	**1000** 行	数量制限	**PT IN SUBQUERY** によって返される結果は、**1,000** 行を超えることはできません。
SQL 文	**2 MB**	長さ制限	**SQL** 文の最大長は、**2 MB** です。
where 句の条件数	**256**	数量制限	**where** 句では最大 **256** 個の条件を使用できます。
列レコードの長さ	**8 MB**	数量制限	テーブル内のセルの最大長は **8 MB** です。
in 文のパラメーター数	**1,024**	数量制限	**in** 文のパラメーターの最大数を、**in (1,2,3…,1024)** のように指定します。 **in(…)** のパラメーターが大きすぎると、コンパイルで負荷が発生します。 **1,024** は推奨値であり、制限値ではありません。
jobconf.json	**1 MB**	長さ制限	**jobconf.json** のサイズは **1 MB** です。 テーブルに含まれるパーティションが多すぎると、**'jobconf.json'** が **1 MB** を超えることがあります。

名前	最大値/制限	分類	説明
ビュー	書き込み不可	操作制限	**INSERT** 文を使用して、ビューの書き込みや操作を実行できません。
列データ型	禁止	操作制限	列のデータ型と位置は変更できません。
java udf 関数	抽象型または静的型にすることはできません	操作制限	**Java UDF** を抽象型または静的型にすることはできません。
最大 **10,000** のパーティションを照会できます。	**10,000**	数量制限	最大 **10,000** のパーティションを照会できます。



注：

MaxCompute SQL の制限を手動で変更または設定することはできません。

5.3 他の SQL 構文との相違点

ここでは、**SQL** の観点から、**MaxCompute SQL** と **Hive**、**MySQL**、**Oracle**、**SQL Server** を比較しながら、サポート対象外の **Pant** 構文と **DML** 構文について説明します。

MaxCompute でサポート対象外の Pant 構文

文法	MaxCompute	Hive	MySql	Oracle	SQL Server
CREATE TABLE—PRIMARY KEY	**N**	**N**	**Y**	**Y**	**Y**
CREATE TABLE—NOT NULL	**N**	**N**	**Y**	**Y**	**Y**
Create Table-cluster	**Y**	**Y**	**N**	**Y**	**Y**
Create Table-External table	**Y (OSS** と **OTS** 外部テーブルに対応)	**Y**	**N**	**N**	**N**
Create Table-maid table	**N**	**Y**	**Y**	**Y**	**Y (#** 接頭辞付き **)**
Create Index	**N**	**Y**	**Y**	**Y**	**Y**

文法	MaxCompute	Hive	MySql	Oracle	SQL Server
Virtual Column	N	N (事前定義で 2 つのみ)	N	Y	Y

MaxCompute でサポート対象外の DML構文

文法	MaxCompute	Hive	MySQL	Oracle	SQL Server
Select-recurrent CTE	N	N	N	Y	Y
Select-group by roll up	N	Y	Y	Y	Y
Select-group by cube	N	Y	N	Y	Y
Select-grouping set	N	Y	N	Y	Y
Maid join	Y	Y	N	Y	Y
Select-Fig	N	N	N	Y	Y
Select-correlated subquery	N	Y	Y	Y	Y
Set operator-Union (distinct)	Y	Y	Y	Y	Y
Set operator-intersect	N	N	N	Y	Y
Set operator-minus	N	N	N	Y	Y (キーワード)
Update... Where	N	Y	Y	Y	Y
Update... Order by limit	N	N	Y	N	Y
Delete... Where	N	Y	Y	Y	Y
Delete... Order by limit	N	N	Y	N	N
Analytic-reusable windowing clause	N	Y	N	N	N
Analytic-range	N	Y	N	Y	Y

5.4 演算子

演算子は、プログラムコードで演算を行うために使用します。 ここでは、関係演算子、算術演算子、ビット演算子、論理演算子の **4** 種類の演算子について説明します。

関係演算子

演算子	説明
A=B	**A** または **B** が **NULL** の場合は **NULL** が返されます。 **A** と **B** が等しい場合は **TRUE**、そうでない場合は **FALSE** が返されます。
A<>B	**A** または **B** が **NULL** の場合は **NULL** が返されます。 **A** と **B** が等しくない場合は **TRUE** 、そうでない場合は **FALSE** が返されます。
A<B	**A** または **B** が **NULL** の場合は **NULL** が返されます。 **A** が **B** より小さい場合は **TRUE**、そうでない場合は **FALSE** が返されます。
A<=B	**A** または **B** が **NULL** の場合は **NULL** が返されます。 **A** が **B** 以下の場合は **TRUE**、そうでない場合は **FALSE** が返されます。
A>B	**A** または **B** が **NULL** の場合は **NULL** が返されます。 **A** が **B** より大きい場合は **TRUE**、そうでない場合は **FALSE** が返されます。
A>=B	**A** または **B** が **NULL** の場合は **NULL** が返されます。**A** が **B** 以上の場合は **TRUE**、そうでない場合は **FALSE** が返されます。
A IS NULL	**A** が **NULL** の場合は **TRUE** が返されます。そうでない場合は、**FALSE** が返されます。
A IS NOT NULL	**A** が **NULL** でない場合、**TRUE** が返されます。そうでない場合は、**FALSE** が返されます。
A LIKE B	**A** または **B** が **NULL** の場合は、**NULL** が返されます。 文字列 **A** が **SQL** の単純正規表現 **B** と一致する場合は **TRUE**、そうでない場合は **FALSE** が返されます。 **B** 内の文字列 **%** は、**A** 内の任意の数の文字列と一致します。**B** 内の文字列 **_** は、**A** 内の任意の **1** 文字と一致します。**%** または **_** と一致させるには、エスケープ文字 **'(%')'** と **(_')** を使用します。 `‘aaa’ like‘a_’= TRUE` `‘aaa’ like‘a%’ = TRUE` `‘aaa’ like‘aab’= FALSE` `‘a%b’ like‘a\\%b’= TRUE` `‘axb’ like ‘a\\%b’= FALSE`
A RLIKE B	**A** は文字列で、**B** は文字列定数の正規表現です。 **A** の部分文字列が **Java** の正規表現の **B** と一致する場合は **TRUE**、そうでない場合は **FALSE** が返されます。 式 **B** が空の場合、エラーを報告して終了します。 **A** または **B** が **NULL** の場合は、**NULL** が返されます。

演算子	説明
A IN B	**B** は集合です。 **A** が **NULL** の場合は、**NULL** が返されます。 **A** が **B** の中にある場合、**TRUE** が返されます。そうでない場合は **FALSE** が返されます。式 **B** に **1** つの要素 **NULL** だけがある場合、つまり **A IN (NULL)** の場合は **NULL** が返されます。 式 **B** に **NULL** 要素が含まれている場合は、**B** 集合の他の要素の型として **NULL** をとります。 **B** は定数でなければならず、少なくとも **1** つの要素を持つ必要があります。すべての型は一致していなければなりません。
BETWEEN AND	`A [NOT] BETWEEN B AND C` の式で、**A**、**B**、**C** のいずれかが **Empty** の場合、**Empty** が返されます。 **A** が **B** 以上 **C** 以下の場合は **TRUE**、そうでない場合は **FALSE** が返されます。

一般的な使用方法

```
select * from user where user_id = '0001';
select * from user where user_name <> 'maggie';
select * from user where age > ‘50’;
select * from user where birth_day >= '1980-01-01 00:00:00';
select * from user where is_female is null;
select * from user where is_female is not null;
select * from user where user_id in (0001,0010);
select * from user where user_name like 'M%';
```

MaxCompute では、**Double** 値の精度が異なります。 このため、**2** つの **Double** 型のデータの比較に等号 (=) を使用することは推奨しません。 **2** つの **Double** 型を引き算し、絶対値から判断できます。 絶対値が非常に小さい場合、**2** つの **double** 値は等しいとみなされます。

例

```
abs(0.9999999999 - 1.0000000000) < 0.000000001
 -- 0.9999999999  と 1.0000000000 は、小数点以下 10 桁の精度ですが、0.
000000001 は小数点以下 9 桁の精度です。
 -- 0.9999999999 は 1.0000000000 と等しいとみなされます。
```

 注:

- **ABS** は、絶対値をとるための **MaxCompute** 組み込み関数です。 詳細は、「*ABS*」をご参照ください。
- 一般的に、**MaxCompute** の **double** 型は **14** 桁の数字を保持できます。

算術演算子

演算子	説明
A + B	式 **A** または **B** が **NULL** の場合は **NULL** が返されます。そうでない場合は、**A** と **B** を足した結果が返されます。

演算子	説明
A – B	式 **A** または **B** が **NULL**の場合は **NULL** が返されます。そうでない場合は、**A** から**B** を引いた結果が返されます。
A * B	式 **A** または **B** が **NULL** の場合は **NULL** が返されます。そうでない場合は、**A** と **B** を掛けた結果が返されます。
A / B	式 **A** または **B** が **NULL** の場合は **NULL** が返されます。そうでない場合は、**A** を **B** で割った結果が返されます。 式 **A** と **B** が **bigint** 型の場合、結果は **double** 型になります。
A % B	式 **A** または **B** が **NULL** の場合は **NULL** が返されます。そうでない場合は、**A** を **B** で割った余りが返されます。
+A	**A** が返されます。
-A	式 **A** が **NULL** の場合は **NULL** が返されます。そうでない場合は **-A** が返されます。

一般的な使用方法

```
select age+10, age-10, age%10, -age, age*age, age/10 from user;
```

注：

- 算術演算では **string**、**bigint**、**double** 型のみ使用可能です。 日付型とブール型は使用できません。
- 算術演算の前に、**string** 型は暗黙の型変換によって **double** 型に変換されます。
- **bigint** 型と **double** 型がどちらも算術演算に含まれている場合、**bigint** 型は暗黙の型変換によって **double** 型に変換されます。
- **A** と **B** が **bigint** 型の場合、**A/B** の戻り値は **double** 型になります。 他の算術演算では、戻り値も **bigint** 型です。

ビット演算子

演算子	説明
A & B	**A** と **B** のビット積の結果が返されます。たとえば、**1**&**2**は **0**、**1**&**3**は **1**、**NULL** とその他の値のビット積はすべて **NULL** が返されます。 式 **A** と **B** は **bigint** 型でなければなりません。
A | B	**A** と **B** のビット和の結果が返されます。たとえば、**1 | 2** は **3**、**1 | 3** は **3** が返されます。 **NULL** とその他の値のビット和はすべて **NULL** が返されます。 式 **A** と **B** は **Bigint** 型でなければなりません。


注：
ビット演算子は暗黙の型変換に対応しておらず、**bigint** 型のみを使用できます。

論理演算子

```
演算子 説明
A and B TRUE and TRUE=TRUE
                TRUE and FALSE=FALSE
                FALSE and TRUE=FALSE
                FALSE and NULL=FALSE
                NULL and FALSE=FALSE
                TRUE and NULL=NULL
                NULL and TRUE=NULL
                NULL and NULL=NULL
A or B TRUE or TRUE=TRUE
                TRUE or FALSE=TRUE
                FALSE or TRUE=TRUE
                FALSE or NULL=NULL
                NULL or FALSE=NULL
                TRUE or NULL=TRUE
                NULL or TRUE=TRUE
                NULL or NULL=NULL
NOT A A が NULL の場合、NULL が返されます。
                A  が TRUE の場合、FALSE が返されます。
                A が FALSE の場合、TRUE が返されます。
```


注：
論理演算ではブール型のみ使用できます。暗黙の型変換には対応していません。

5.5 型変換

MaxCompute SQLでは、データ型の変換が可能です。 明示的型変換と暗黙的型変換の **2** 通りの変換方法があります。

明示的変換

明示的な変換では、**CAST** を使用して値の型を別の型に変換します。 次の表に、**MaxCompute SQL** で明示的に変換できる型を示します。

From/To	BIGINT	DOUBLE	STRING	DATETIM	BOOLEA	DECIMA	FLOAT
BIGINT	–	Y	Y	N	Y	Y	Y
DOUBLE	Y	–	Y	N	Y	Y	Y
STRING	Y	Y	–	Y	Y	Y	Y
DATETIM	N	N	Y	–	N	N	N
BOOLEAN	Y	Y	Y	N	–	Y	Y

From/ To	BIGINT	DOUBLE	STRING	DATETIM	BOOLEA	DECIMA	FLOAT
DECIMAL	Y	Y	Y	N	Y	–	Y
FLOAT	Y	Y	Y	N	Y	Y	-

Y: 変換可能 **N:** 変換不可 **-:** 変換不要

例：

```
select cast(user_id as double) as new_id from user;
select cast('2015-10-01 00:00:00' as datetime) as new_date from user;
```

 注：

- **DOUBLE** 型を **BIGINT** 型に変換する場合、小数点以下の桁は削除されます。 たとえば、`cast(1.6 as bigint) = 1`になります。
- **DOUBLE** 型の形式を満たす **STRING** 型を **BIGINT** 型に変換するには、**DOUBLE** 型に変換してから **BIGINT** 型に変換します。 したがって、小数点以下の桁は削除されます。 たとえば、`cast("1.6" as bigint) = 1`です。
- **BIGINT** 型の形式を満たす **STRING** 型は、**DOUBLE** 型に変換でき、常に小数点以下 **1** 桁を保持します。 たとえば、`cast("1" as double) = 1.0` です。
- サポートされていない型の明示的な変換によって、例外が返されることがあります。
- 実行中に変換が失敗した場合、変換は中断され、例外が返されます。
- **DATETIME** 型を変換するには、デフォルトの形式 **yyyy-mm-dd hh:mi:ss** を使用します。詳細については、「*STRING* 型と *DATETIME* 型の間の変換」をご参照ください。
- 一部の型は明示的には変換できませんが、組み込み **SQL** 関数を使用して変換できます。 たとえば、`to_char` 関数を使用すると、**BOOLEAN** 型の値を **STRING** 型に変換できます。 詳細については、「*TO_CHAR*」をご参照ください。 `to_date` 関数を使用すると、**STRING** 型の値を **DATETIME** 型に変換できます。 詳細については、「*TO_DATE*」をご参照ください。
- 詳細については、「*CAST*」をご参照ください。
- **DECIMAL** 値が値の範囲を超えると、**CAST STRING TO DECIMAL** で **MSB** オーバーフローエラーまたは **LSB** オーバーフロー切り捨てが発生することがあります。

暗黙的変換と範囲

MaxCompute では、コンテキスト環境と変換ルールに基づいて、暗黙的な型変換が自動的に実行されます。 次の表に、**MaxCompute** で暗黙的に変換できる型を示します。

	bool	tinyi	smal	int	bigir	float	double	Decimal	strin	varcl	time	binary
boolean to	Y	N	N	N	N	N	N	N	N	N	N	N
tinyint to	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N
smallint to	N	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N
int to	N	N	Y	Y	Y	Y	Y	Y	Y	Y	N	–
bigint to	N	N	N	N	Y	Y	Y	Y	Y	Y	N	N
float to	N	N	N	N	Y	Y	Y	Y	Y	Y	N	–
double to	N	N	N	N	N	N	Y	Y	Y	Y	N	N
decimal to	N	N	N	N	N	N	N	Y	Y	Y	N	N
string to	N	N	N	N	N	N	Y	Y	Y	Y	N	N
varchar to	N	N	N	N	Y	Y	Y	Y	N	N	–	–
timestamp to	N	N	N	N	N	N	N	N	Y	Y	Y	N
binary to	N	N	N	N	N	N	N	N	N	N	N	Y

Y: 変換可能 **N:** 変換不可

注 :

- **DECIMAL** 型および **DATETIME** 定数定義モードが **MaxCompute2.0** に追加されました。 **100BD** は値が **100** である **DECIMAL** を示します。 日時`2017-11-11 00:00:00` は、**DATETIME** 型の定数を示します。定数定義は、**values** 句とテーブルで直接使用できるので便利です。
- 以前のバージョンの **MaxCompute** では、**DOUBLE** 型の値を暗黙的に **BIGINT** 型に変換できます。 何らかの理由により、このような変換はデータの損失につながる可能性があり、共通のデータベースシステムでは許可されていません。

一般的な使用方法

```
select user_id+age+'12345',
           concat(user_name,user_id,age)
  from user;
```

注：

- サポート対象外の型の暗黙的な変換を行うと、例外が発生することがあります。
- 実行中に変換に失敗すると、例外が発生します。
- **MaxCompute** では、コンテキスト環境に基づいて暗黙の型変換が自動的に実行されます。型が一致しない場合は、**CAST** を使用して明示的な変換を実行することを推奨します。
- 暗黙の型変換ルールは、特定の範囲のスコープに適用できます。 一部のスコープでは、一部のルールのみが有効です。 詳細については、「暗黙的な変換のスコープ」をご参照ください。

- 関係演算子に関する暗黙的な変換

  関係演算子は、等しい (=)、等しくない (<>)、より小さい (<)、以下 (<=)、より大きい (>)、以上 (>=)、**IS NULL**、**IS NOT NULL**、**LIKE**、**RLIKE**、および **IN** です。 **LIKE**、**RLIKE**、および **IN** の暗黙的な変換ルールの特徴ついて個別に説明します。 以下の説明には、これら **3** つの特殊な演算子は含まれていません。

  次の表では、さまざまな型のデータが関係演算に使用されている場合の暗黙の型変換ルールについて説明します。

変換前/変換後	Bigint	Double	String	Datetime	Boolean	Decimal
Bigint	–	**Double**	**Double**	**N**	**N**	**Decimal**
Double	**Double**	–	**Double**	**N**	**N**	**Decimal**
String	**Double**	**Double**	–	**Datetime**	**N**	**Decimal**
Datetime	**N**	**N**	**Datetime**	–	**N**	**N**
Boolean	**N**	**N**	**N**	**N**	–	**N**
Decimal	**Decimal**	**Decimal**	**Decimal**	**N**	**N**	-

注：

- **2** つの型を暗黙的に変換できない場合、例外が発生して関係演算は中断されます。
- 関係演算子に関する詳細は、「関係演算子」をご参照ください。

- 特殊な関係演算子での暗黙的な型変換

  特殊な関係演算子には、**LIKE**、**RLIKE**、および **IN** があります。

  - **LIKE** と **RLIKE** の使い方は次のとおりです。

    ```
    source like pattern;
    source rlike pattern;
    ```

    暗黙的な変換での **LIKE** および **RLIKE** の注意事項を次に示します。

    ■ **LIKE** と **RLIKE** の **source** パラメーターと **pattern** パラメーターは、**STRING** 型である必要があります。

    ■ 他の型を演算で使用することも、暗黙的に **STRING** 型に変換することもできません。

  - **IN** の使い方は次のとおりです。

    ```
    key in (value1, value2, …)
    ```

    IN の暗黙の型変換ルール

    ■ 値列のデータ型は一貫している必要があります。

    ■ キーと値を比較する場合、**BIGINT**、**DOUBLE**、および **STRING** 型の比較では、それらは **DOUBLE** 型に変換されます。 **DATETIME** 型と **STRING** 型を比較する場合は、**DATETIME** 型に変換されます。 他の型間の変換は行えません。

- 算術演算子での暗黙的な変換

  算術演算子には、足し算 (+)、引き算 (-)、掛け算 (*)、割り算 (/)、剰余演算 (%)、単項プラス (+) および単項マイナス (-) があります。 算術演算子の暗黙的な型変換ルールは、次のとおりです。

  - 演算には、**STRING**、**BIGINT**、**DOUBLE**、および **DECIMAL** 型のみ使用できます。
  - **STRING** 型は、演算の前に暗黙的に **DOUBLE** 型に変換されます。
  - **BIGINT** と **DOUBLE** 型が演算に含まれている場合、**BIGINT** 型は暗黙的に **DOUBLE** 型に変換されます。
  - **DATETIME** 型と **BOOLEAN** 型は、算術演算では使用できません。

- 論理演算子での暗黙的な変換

  論理演算子には、**AND**、**OR**、**NOT** があります。 論理演算子の暗黙的な変換ルールは、次のとおりです。

  - 論理演算では、**BOOLEAN** 型のみを使用できます。
  - 他の型を論理演算で使用することはできず、暗黙的に他の型に変換することもできません。

組み込み関数の暗黙的な変換

MaxCompute SQL は、多数のシステム関数を実装しています。 任意の行の **1** つまたは複数の列を計算し、任意の型のデータを出力できます。 組み込み関数の暗黙的な変換ルールは、次のとおりです。

- 関数を呼び出すには、入力パラメーターのデータ型が関数で定義されているデータ型と異なる場合、入力パラメーターのデータ型が関数で定義されたデータ型に変換されます。
- **MaxCompute SQL** の各種組み込み関数のパラメーターには、暗黙的な変換に関するさまざまな要件があります。 詳細については、「組み込み関数」をご参照ください。

CASE WHEN での暗黙的な変換

CASE WHEN の詳細については、「*CASE WHEN* 式」をご参照ください。 **CASE WHEN** の暗黙的な型変換ルールは、次のとおりです。

- 戻り値の型が **BIGINT** および **DOUBLE** の場合、すべて **DOUBLE** 型に変換されます。
- 戻り値の型に **STRING** 型が含まれている場合は、すべて **STRING** 型に変換されます。 変換が失敗した場合 **(BOOLEAN** 型変換など)、エラーが返されます。
- 他の型間の変換は許可されません。

STRING 型と DATETIME 型の間の変換

MaxCompute は、**STRING** 型と **DATETIME** 型間の変換に対応しています。 変換形式は `yyyy-mm-dd hh:mi:ss` です。

単位	文字列 (大文字小文字を区別しない)	有効な値の範囲
年	**yyyy**	**0001～9999**
月	**mm**	**01～12**
日	**dd**	**01～28、29、30、31**
時間	**hh**	**00～23**
分	**mi**	**00～59**
秒	**ss**	**00～59**



注：

- 各単位の値で、上の桁が **0** の場合、省略できません。 たとえば、`2014-1-9 12:12:12` は無効な **DATETIME** 形式であり、**STRING** 型から **DATETIME** 型に変換できません。 `2014-01-09 12:12:12`と記述する必要があります。

- 上記の形式要件を満たす **STRING** 型のみを **DATETIME** 型に変換できます。 たとえば、`cast(“2013-12-31 02:34:34” as datetime)` は、**STRING** 型の `2013-12-31 02:34:34` から **DATETIME** 型に変換されます。 同様に、**DATETIME** 型を **STRING** 型に変換する場合、デフォルトの変換形式は **yyyy-mm-dd hh:mi:ss** です。

たとえば、次の変換では例外が発生します。

```
cast("2013/12/31 02/34/34" as datetime)
cast("20131231023434" as datetime)
cast("2013-12-31 2:34:34" as datetime)
```

dd のしきい値は、**1** か月の日数によって異なります。 値がその月の実際の日数を超えた場合、例外が発生して変換は中断されます。

例：

```
cast("2013-02-29 12:12:12" as datetime) -- エラーが返されます。2013 年 2
 月 29 日は存在しません。
cast("2013-11-31 12:12:12" as datetime) -- エラーが返されます。2013 年 11
 月 31 日は存在しません。
```

MaxCompute には、**DATETIME** 形式を満たさない **STRING** 型を **DATETIME** 型に変換する **TO_DATE** 関数が実装されています。 詳細については、「*TO_DATE*」をご参照ください。

5.6 DDL SQL

5.6.1 テーブル操作

ここでは、クライアントからテーブルを作成、表示、削除する方法を説明します。また、テーブル名やテーブル情報の変更方法についても説明します。

テーブルの作成

SQL 文の形式

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[STORED BY StorageHandler] -- Limited to external tables
[WITH SERDEPROPERTIES (Options)] -- Limited to external tables
[LOCATION OSSLocation];-- Limited to external tables
[LIFECYCLE days]
[As select_statement]
 CREATE TABLE [IF NOT EXISTS] table_name
 LIKE existing_table_name
```

次の点に注意してください。

- テーブルの作成時に **"if not exists"** オプションを指定せず、同じ名前のテーブルが既に存在する場合は、エラーが返されます。 **"if not exists"** オプションを指定すると、同じ名前のテーブルが存在するかどうかにかかわらず、またソーステーブル構造とターゲットテーブル構造が異なっていても、すべての操作は正常に終了します。 既存のテーブルのメタ情報は変わりません。
- テーブル名と列名はどちらも大文字と小文字が区別されず、特殊文字を含めることはできません。 英字で始まる必要があり、**a** から **z**、**A** から **Z**、数字、およびアンダースコア (_) を含めることができます。 名前の長さは **128** バイトを超えることはできません。
- **1** つのテーブルに **1200** 列を定義できます。
- データ型は、**Bigint**、**Double**、**Boolean**、**Datetime**、**Decimal**、**String** を使用できます。**MaxCompute2.0** では、サポート対象のデータ型が増えました。

  > 注：
  > **SQL** 文の実行時に、**Tinyint**、**Smallint**、**Int**、**Float**、**Varchar**、**TIMESTAMP BINARY** などのデータ型が含まれている場合は、`set odps.sql.type.system.odps2=true;` を **SQL** 文の前に追加する必要があります。 **set** 文と **SQL** 文は同時に送信されます。

- パーティションを指定するには、**Partitioned by** を使用します。現在サポート対象のデータ型は、**Tinyint**、**Smallint**、**Int**、**Bigint**、**Varchar**、**String** です。

  パーティション値に **2** バイト文字 (例: 中国語) を使用することはできません。先頭は大文字または小文字で始め、続いて文字または数字を使用します。 名前の長さは **128** バイトを超えることはできません。 スペース、コロン **(:)**、アンダースコア **(_)**、ドル記号 **($)**、シャープ記号 **(#)**、ピリオド **(.)**、感嘆符 **(!)**、**'@'** などの特殊文字を使用できます。 **(\t)**、**(\n)**、**(/)** などの文字は、未定義文字とみなされます。 パーティションテーブルでパーティションフィールドを使用すると、パーティション内のデータを追加、更新、および読み取る際に全テーブルスキャンを実行する必要がなくなり、処理効率が向上します。
- 現在、テーブルには **60,000** のパーティションを作成できます。パーティション階層は、**6** レベルまでにする必要があります。
- コメントの内容は、有効な文字列で、**1024** バイト以内にする必要があります。
- ライフサイクルはテーブルのライフサイクルを示し、単位は '日' です。 **create table like** 文では、ソーステーブルからライフサイクルの属性はコピーされません。
- 外部テーブルの詳細については、「*OSS へのアクセス (Access OSS)*」をご参照ください。

例：

販売記録を保存するためのテーブル **sale_detail** を作成します。 このテーブルでは、**sale_date** と **region** をパーティション列として使用します。 テーブルを作成するための文は、次のとおりです。

```
create table if not exists sale_detail(
(
shop_name string,
customer_id string,
total_price double)
)
partitioned by (sale_date string,region string);
-- パーティションテーブル sale_detail を作成します。
```

`create table...as select ...` 文を使用してテーブルを作成することもできます。 テーブルを作成した後、データは次のように新しいテーブルにコピーされます。

```
create table sale_detail_ctas1 as
select * from sale_detail;
```

テーブル **sale_detail** にデータがある場合、前述の **SQL** 文では、**sale_detail** のすべてのデータがテーブル **sale_detail_ctas1** にコピーされます。

 注：

sale_detail はパーティションテーブルですが、`create table...as select...` 文でテーブルを作成すると、パーティション属性はコピーされません。 ソーステーブルのパーティション列は、オブジェクトテーブルの通常の列になります。 つまり、**sale_detail_ctas1** は **5** 列の非パーティションテーブルです。

`create table ...` 文で、**Select** 句の列値として定数を使用する場合は、次のように列名を指定することを推奨します。

```
create table sale_detail_ctas2 as
        select shop_name,
            customer_id,
            total_price,
            '2013' as sale_date,
            'China' as region
        from sale_detail;
```

列名を指定しない場合、文は次のようになります。

```
create table sale_detail_ctas3 as
        select shop_name,
            customer_id,
            total_price,
            '2013',
            'China'
```

```
        from sale_detail;
```

作成されたテーブル **sale_detail_ctas3** の **4** 番目の列と **5** 番目の列は、`_c3`、`_c4` のような、システムで生成した名前になります。

作成するテーブルをソーステーブルと同じ構造にするには、次のように `create table … like'` 文を使用します。

```
create table sale_detail_like like sale_detail;
```

これで、`sale_detail_like` のテーブル構造は、`sale_detail` のテーブル構造とまったく同じになります。 ライフサイクルを除き、**2** つのテーブルの属性 **(**列名、列コメント、テーブルコメントなど**)** は同じです。 しかし、`sale_detail` のデータをテーブル `sale_detail_like`にコピーすることはできません。

テーブル情報の表示

SQL 文の形式**:**

```
desc <table_name>;
desc extended <table_name>; --View external table information.
```

例：

- 前述のテーブル **sale_detail** の情報を表示するには、次の文を実行します。

  ```
  desc sale_detail;
  ```

  返される情報：

  ```
  odps@ $odps_project>desc sale_detail;
  +------------------------------------------------------------------------
  +
  | Owner: ALIYUN$lili.ll@alibaba-inc.com | Project: $odps_project
                    |
  | TableComment:
     |
  +------------------------------------------------------------------------
  +
  | CreateTime: 2017-06-28 15:05:17
     |
  | LastDDLTime: 2017-06-28 15:05:17
     |
  | LastModifiedTime: 2017-06-28 15:05:17
     |
  +------------------------------------------------------------------------
  +
  | InternalTable: YES | Size: 0
     |
  +------------------------------------------------------------------------
  +
  | Native Columns:
     |
  ```

```
+------------------------------------------------------------------------
+
| Field | Type | Label | Comment
   |
+------------------------------------------------------------------------
+
| shop_name | string | |
   |
| customer_id | string | |
   |
| total_price | double | |
   |
+------------------------------------------------------------------------
+
| Partition Columns:
   |
+------------------------------------------------------------------------
+
| sale_date | string |
   |
| region | string |
   |
+------------------------------------------------------------------------
+
OK
```

- 前述のテーブル **sale_detail_like** の情報を表示するには、次の文を実行します。

```
desc sale_detail_like
```

返される情報：

```
odps@ $odps_project>desc sale_detail_like;
+------------------------------------------------------------------------
+
| Owner: ALIYUN$lili.ll@alibaba-inc.com | Project: $odps_project
                  |
| TableComment:
   |
+------------------------------------------------------------------------
+
| CreateTime: 2017-06-28 15:42:17
   |
| LastDDLTime: 2017-06-28 15:42:17
   |
| LastModifiedTime: 2017-06-28 15:42:17
   |
+------------------------------------------------------------------------
+
| InternalTable: YES | Size: 0
   |
+------------------------------------------------------------------------
+
| Native Columns:
   |
+------------------------------------------------------------------------
+
| Field | Type | Label | Comment
   |
+------------------------------------------------------------------------
+
| shop_name | string | |
```

```
    |
| customer_id | string | |
    |
| total_price | double | |
    |
+------------------------------------------------------------------------
+
| Partition Columns:
    |
+------------------------------------------------------------------------
+
| sale_date | string |
    |
| region | string |
    |
+------------------------------------------------------------------------
+
OK
```

上記の例では、ライフサイクルを除き、**sale_detail_like** と **sale_detail** の属性が同じであることが分かります。 詳細については、「テーブルの照会 *(Describe Table)*」をご参照ください。

sale_detail_ctas1 の情報を確認すると、**sale_date** と **region** は通常の列にすぎず、テーブルのパーティションではないことが分かります。

テーブルの削除

SQL 文の形式**:**

```
DROP TABLE [IF EXISTS] table_name;
```

注：

- **[if exists]** オプションを指定しておらず、テーブルが存在しない場合は、例外が返されます。このオプションを指定すると、テーブルが存在するかどうかにかかわらず、すべての命令文が正常に終了します。
- 外部テーブルが削除されても、**OSS** 内のデータは削除されません。

例：

```
create table sale_detail_drop like sale_detail;
    drop table sale_detail_drop;
    -- テーブルが存在する場合、正常終了します。そうでない場合、例外が返されます。
    drop table if exists sale_detail_drop2;
```

```
--テーブル sale_detail_drop2 の有無にかかわらず、すべてが正常終了します。
```

テーブル名の変更

SQL 文の形式**:**

```
ALTER TABLE table_name RENAME TO new_table_name;
```

 注**:**

- 名前変更操作では、テーブル名だけが更新されます。テーブル内のデータは更新されません。
- **new_table_name** と同じ名前のテーブルが存在する場合、エラーが発生します。
- テーブル **table_name** が存在しない場合は、エラーが発生します。

例：

```
create table sale_detail_rename1 like sale_detail;
alter table sale_detail_rename1 rename to sale_detail_rename2;
```

テーブルコメントの変更

コメント形式**:**

```
ALTER TABLE table_name SET COMMENT 'tbl comment';
```

 注**:**

- テーブル **table_name** は存在していなければなりません。
- コメントの長さは **1024** バイト以内にする必要があります。

例：

```
alter table sale_detail set comment 'new comments for table sale_detai
l';
```

変更したコメントを表示するには、コマンド desc を使用します。 詳細については、「テーブルの照会 *(Describe Table)*」をご参照ください。

テーブルの LastDataModifiedTime の変更

MaxCompute SQL では、touch 操作を使用して、テーブルの **LastDataModifiedTime** を変更できます。 この操作を実行すると、テーブルの **LastDataModifiedTime** は、現在時刻になります。

SQL 文の形式:

```
ALTER TABLE table_name TOUCH;
```

 注:

- テーブル **table_name** が存在しない場合、エラーが返されます。
- この操作でテーブルの **LastDataModifiedTime** の値が変更されると、**MaxCompute** ではテーブルデータの変更を認識し、対応するライフサイクルを計算し直します。

非パーティションテーブルのデータ消去

指定した非パーティションテーブルのデータが消去されます。このコマンドは、パーティションテーブルには使用できません。 パーティションテーブルの場合、パーティション内のデータを消去するには、`ALTER TABLE table_name DROP PARTITION` を使用します。

コマンド形式:

```
TRUNCATE TABLE table_name;
```

5.6.2 テーブルのライフサイクル

MaxCompute には、データのライフサイクル管理機能が搭載されているので、ストレージスペースのリリースを簡単に行うことができます。また、データのリカバリ処理が簡素化されます。

テーブルのライフサイクルの変更

SQL 文の形式

```
ALTER TABLE table_name SET lifecycle days;
```

 注:

- パラメーター **'days'** は、ライフサイクルの完了に必要な時間を示します。 正の整数で、単位は '日' です。
- テーブル **'table_name'** が非パーティションテーブルの場合、 最終更新日から計算して、**N** 日経過してもデータが 変更されない場合は、ユーザーの介入がなくても、テーブルは自動的にリサイクルされます **('dropTable'** 操作と同様)。
- **MaxCompute** では、テーブル内のデータが変更されると、**LastDataModifiedTime** が更新されます。 **LastDataModifiedTime** とライフサイクルの設定に基づいて、このテーブルをリサイクルするかどうかが判断されます。

- テーブル’table_name’がパーティションテーブルの場合、各パーティションの**LastDataModifiedTime**に従って、テーブルをリサイクルするかどうかが判断されます。
- 非パーティションテーブルとは異なり、パーティションテーブルの最後のパーティションがリサイクルされても、テーブルは削除されません。
- ライフサイクルはテーブルに対して設定できますが、パーティションに対しては設定できません。
- ライフサイクルは、テーブル作成時に指定できます。

例:

```
create table test_lifecycle(key string) lifecycle 100;
 -- Create a new table test_lifecycle and the lifecycle is 100 days.
 alter table test_lifecycle set lifecycle 50;
 -- Alter the lifecycle for the table test_lifecycle and set it to be
50 days.
```

テーブルのライフサイクルの無効化

特定のパーティション内のデータは、ライフサイクル機能によるリサイクルを必要としない場合があります。たとえば、月初めのデータや、中国の独身の日期間中のデータが、これに当てはまります。特定のパーティションを使用することで、ライフサイクルを無効にできます。

SQL文の形式

```
ALTER TABLE table_name partition_spec ENABLE|DISABLE LIFECYCLE;
```

例は次のとおりです。

```
ALTER TABLE trans PARTITION(dt='20141111') DISABLE LIFECYCLE;
```

5.6.3 列とパーティションの操作

ここでは、テーブルパーティションの追加、削除、変更コマンドについて説明します。

パーティションの追加

SQL文の形式

```
ALTER TABLE TABLE_NAME ADD [IF NOT EXISTS] PARTITION partition_spec
partition_spec:(partition_col1 = partition_col_value1, partition_col2
 = partiton_col_value2, ...)
```



注:

- パーティション名は、小文字にする必要があります。
- 実行できるのは、’パーティションの作成’だけです。’パーティション列の作成’は実行できません。

- 同じ名前のパーティションが既に存在する場合、**[if not exists]** オプションが指定されていないと、例外が返されます。
- **1** つの **MaxCompute** テーブル内に作成できるパーティションの最大数は、**60,000** です。
- マルチレベルのパーティションテーブルの場合、新しいパーティションを追加するには、すべてのパーティション値を指定する必要があります。

例:

‘sale_detail’ テーブルに新しいパーティションを追加します。

```
alter table sale_detail add if not exists partition (sale_date='201312
', region='hangzhou');
-- 2013 年 12 月の杭州（hangzhou）リージョンにおける売上詳細を保存するパーティ
ションが追加されます。
alter table sale_detail add if not exists partition (sale_date='201312
', region='shanghai');
-- 2013 年 12 月の上海（shanghai）リージョンにおける売上詳細を保存するパーティ
ションが追加されます。
alter table sale_detail add if not exists partition(sale_date='
20111011');
-- パーティション sale_date のみを指定すると、エラーが返されます。
alter table sale_detail add if not exists partition(region='shanghai
');
-- パーティション region のみを指定すると、エラーが返されます。
```

パーティションの削除

パーティションの削除構文の形式は、次のとおりです。

```
ALTER TABLE TABLE_NAME DROP [IF EXISTS] PARTITION partition_spec;
partition_spec:(partition_col1 = partition_col_value1, partition_col2
 = partiton_col_value2, ...)
```

 注:

パーティションが存在しない場合、オプション **[if exists]** が指定されていないと、エラーが返されます。

例:

sale_detail テーブルからパーティションを削除します。

```
alter table sale_detail drop if exists partition(sale_date='201312',
region='hangzhou');
```

```
-- -2013 年 12 月の杭州（Hangzhou）の売上詳細が削除されます。
```

列の追加

SQL 文の形式

```
ALTER TABLE table_name ADD COLUMNS (col_name1 type1, col_name2 type2
...)
```

```
ALTER TABLE table_name ADD COLUMNS (col_name1 type1 comment 'XXX',
col_name2 type2 comment 'XXX');
```

 注：

新規列の順番は指定できません。 デフォルトでは、新規列は最後の列になります。

列名の変更

SQL 文の形式

```
ALTER TABLE table_name CHANGE COLUMN old_col_name RENAME TO new_col_na
me;
```

 注：

・ ’**old_col_name**’ は、既存の列を示します。

・ テーブル内に ’**new_col_name**’ という名前の列は存在できません。

列やパーティションのコメントの変更

列やパーティションのコメントを変更する **SQL** 文は、次のとおりです。

```
ALTER TABLE table_name CHANGE COLUMN col_name COMMENT comment_string;
```

 注：

コメントは、**1024** バイト以内にする必要があります。

列名と列コメントの同時変更

SQL 文の形式

```
ALTER TABLE table_name CHANGE COLUMN old_col_name new_col_name
column_type COMMENT column_comment;
```

 注：

・ ’**old_col_name**’ は既存の列でなければなりません。

・ テーブル内に ’**new_col_name**’ という名前の列は存在できません。

- コメントは、1024 バイト以内にする必要があります。

テーブル/パーティションの LastDataModifiedTime **の変更**

MaxCompute MaxCompute SQL では、’**touch**’ 操作を使用して、パーティションの **LastDataModifiedTime** を変更できます。 この操作を実行すると、パーティションの ’**LastDataModifiedTime**’ は、現在の時刻に変更されます。

SQL 文の形式

```
ALTER TABLE table_name TOUCH PARTITION(partition_col='partition_
col_value', ...)
```

注：

- ’**table_name**’ または ’**partition_col**’ が存在しない場合は、エラーが返されます。
- 指定した **partition_col_value** が存在しない場合は、エラーが返されます。
- この操作でテーブルの ’**LastDataModifiedTime**’ の値を変更すると、**MaxCompute** はテーブルまたはパーティションのデータが変更されたとみなし、ライフサイクルを計算し直します。

パーティション値の変更

MaxCompute SQL では、**'rename'** 操作により、パーティションの値を変更できます。

SQL 文の形式

```
ALTER TABLE table_name PARTITION (partition_col1 = partition_
col_value1, partition_col2 = partiton_col_value2, ...)
RENAME TO PARTITION (partition_col1 = partition_col_newvalue1,
partition_col2 = partiton_col_newvalue2, ...)
```

注：

- パーティションの列名は変更できません。 列の値のみ変更できます。
- マルチレベルのパーティションの値を変更するには、レベルごとにパーティション値を上書きする必要があります。

5.6.4 ビュー操作

ビューの作成

SQL 文の形式

```
CREATE [OR REPLACE] VIEW [IF NOT EXISTS] view_name
    [(col_name [COMMENT col_comment], ...)]
    [COMMENT view_comment]
```

```
    [AS select_statement]
```

 注：

- ビューを作成するには、ビューの参照先テーブルの '読み取り' 権限が必要です。
- ビューには、有効な **'select'** 文を **1** つだけ指定できます。
- ビューから他のビューを参照できますが、ビュー自身を参照することはできません。 循環参照には対応していません。
- `insert into` や `insert overwrite` などを使用してビューを操作し、ビューにデータを書き込むことはできません。
- ビューの作成後、参照先のテーブルが変更された場合 (参照先テーブルが削除されるなど)、アクセスできなくなることがあります。 参照先のテーブルとビューの対応関係を維持する必要があります。
- **[if not exists]** オプションが指定されておらず、ビューが既に存在する場合、`create view` を使用すると異常が発生します。このような状況が発生した場合は、`create or replace view` を使用して、ビューを再作成します。 ビューを再構築しても、権限は変わりません。

例:

```
create view if not exists sale_detail_view
(store_name, customer_id, price, sale_date, region)
comment 'a view for table sale_detail'
as select * from sale_detail;
```

ビューの削除

SQL 文の形式

```
DROP VIEW [IF EXISTS] view_name;
```

 注：

[if exists] オプションが指定されておらず、ビューが存在しない場合、エラーが発生します。

例:

```
DROP VIEW IF EXISTS sale_detail_view;
```

ビュー名の変更

SQL 文の形式

```
ALTER VIEW view_name RENAME TO new_view_name;
```

 注:

同一名のビューが既に存在する場合、エラーが発生します。

例:

```
create view if not exists sale_detail_view
        (store_name, customer_id, price, sale_date, region)
        comment 'a view for table sale_detail'
        as select * from sale_detail;
    alter view sale_detail_view rename to market;
```

5.7 Insert 操作

5.7.1 INSERT OVERWRITE と INSERT INTO

関数の定義:

```
INSERT OVERWRITE|INTO TABLE tablename [PARTITION (partcol1=val1,
partcol2=val2 ...)] [(col1,col2 ...)]
select_statement
FROM from_statement;
```

 注:

- MaxCompute の INSERT 構文は、MySQL や Oracle の INSERT 構文とは異なります。 `insert overwrite`、`insert into` に続けて、TABLE キーワードを指定します。tablename をそのまま使用しません。
- INSERT の書き込み先のテーブルがパーティションテーブルの場合、`[PARTITION ( partcol1=val1, partcol2=val2 …)]` には、関数などの式を指定できません。
- 現時点では、INSERT OVERWRITE で列を挿入することはできません。 代わりに INSERT INTO を使用します。

INSERT OVERWRITE および INSERT INTO の処理結果は、最終テーブルに保存されます。

INSERT INTO と INSERT OVEWRITE の違いは、INSERT INTO は、テーブルやパーティションに追加データが挿入されるのに対し、INSERT OVERWRITE は、テーブルやパーティションから元のデータを削除してからデータが挿入されます。

 注：

同一パーティションに対して、値を指定した **INSERT OVERWRITE** を繰り返し実行すると、**MaxComputer** のパーティションテーブルのパーティションサイズは、異なるデータパーティションサイズになります。 これは、同じテーブルの同じパーティションを選択して、同じテーブルの同じパーティションに **INSERT OVERWRITE** を実行すると、ファイル分割ロジックが変わり、データのサイズが変わるためです。 しかし、データの合計長は、**INSERT OVERWRITE** 前後では一定なので、料金やストレージを心配する必要はありません。

MaxCompute SQL でデータを処理する際に、最もよく使用される文が INSERT OVERWRITE および INSERT INTO です。 処理結果をテーブルに保存することができるため、後続の処理に使用できます。たとえば、**sale_detail** テーブルで、異なるリージョンの売上詳細を処理するには、次の文を実行します。

```
create table sale_detail_insert like sale_detail;
alter table sale_detail_insert add partition(sale_date='2013', region
='china');
insert overwrite table sale_detail_insert partition (sale_date='2013
', region='china')
select shop_name, customer_id, total_price from sale_detail;
```

 注：

元のテーブルと結果テーブルとの対応づけには、**SELECT** 句内の列の順序が適用されます。両テーブルの列名が適用されるわけではありません。 次の **SQL** 文も有効です。

```
insert overwrite table sale_detail_insert partition (sale_date='2013
', region='china')
select customer_id, shop_name, total_price from sale_detail;
-- sale_detail_insert テーブルの作成時、列の順序は次のとおりです。
-- shop_name string、customer_id string、total_price bigint
-- sale_detail のデータを sale_detail_insert に挿入する際、sale_detail の
挿入順序は次のとおりです。
-- customer_id、shop_name、total_price
-- sale_detail.customer_id のデータを sale_detail_insert.shop_name に挿
入します。
-- sale_detail.shop_name のデータを sale_detail_insert.customer_id に挿
入します。
```

パーティションにデータを挿入する場合、**SELECT** リストにパーティション列を指定することはできません。

```
insert overwrite table sale_detail_insert partition (sale_date='2013
', region='china')
```

```
select shop_name, customer_id, total_price, sale_date, region from
sale_detail;
-- エラーが返されます。 項目 sale_date と region はパーティション列です。静的
パーティションの INSERT 文には指定できません。
```

また、パーティションの値には、定数のみを指定します。式は指定できません。 次の文は無効です。

```
insert overwrite table sale_detail_insert partition (sale_date=
datepart('2016-09-18 01:10:00', 'yyyy') , region='china')
select shop_name, customer_id, total_price from sale_detail;
```

5.7.2 MULTI INSERT

MaxCompute SQL では、**1** つの **SQL** 文で複数の結果テーブルやパーティションを挿入できます。

SQL 文の形式

```
FROM from_statement
INSERT OVERWRITE | INTO TABLE tablename1 [PARTITION (partcol1=val1,
partcol2=val2 ...)]
select_statement1 [FROM from_statement]
[INSERT OVERWRITE | INTO TABLE tablename2 [PARTITION (partcol1=val3,
partcol2=val4 ...)]
select_statement2 [FROM from_statement]]
```

注:

- 通常、**1** つの **SQL** 文で、最大 **256** の出力を書き込めます。 出力先が **256** を超える場合は、構文エラーが発生します。
- **MULTI INSERT** 文では、
  - パーティションテーブルの場合、書き込み先パーティションを複数回指定することはできません。
  - 非パーティションテーブルの場合、そのテーブルを複数回指定することはできません。
- パーティションテーブル内の別のパーティションに **INSERT OVERWRITE** 操作および **INSERT INTO** 操作を同時に指定することはできません。指定するとエラーが返されます。

非パーティションテーブルの場合、そのテーブルを複数回指定することはできません。

```
create table sale_detail_multi like sale_detail;
from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010
', region='china' )
select shop_name, customer_id, total_price where .....
insert overwrite table sale_detail_multi partition (sale_date='2011
', region='china' )
select shop_name, customer_id, total_price where .....
-- Return result successfully. Insert the data of sale_detail into
the 2010 sales records and 2011 sales records in China region.
```

```
from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010
', region='china' )
select shop_name, customer_id, total_price
insert overwrite table sale_detail_multi partition (sale_date='2010
', region='china' )
select shop_name, customer_id, total_price;
-- An error is thrown. The same partition appears for multiple times.
from sale_detail
insert overwrite table sale_detail_multi partition (sale_date='2010
', region='china' )
select shop_name, customer_id, total_price
insert into table sale_detail_multi partition (sale_date='2011',
region='china' )
select shop_name, customer_id, total_price;
-- An error is thrown. Different partitions within a partition table
cannot have both an ‘insert overwrite’ operation and an ‘insert into
’ operation.
```

5.7.3 動的パーティション

パーティションテーブルに **INSERT OVERWRITE** 操作を実行するには、文にパーティションの値を指定する必要があります。 または、パーティションテーブルのパーティション列のみを指定し、値は指定しないという柔軟な方法もあります。

概要

Select 句の列を使用して、パーティションの値を指定できます。

SQL 文の形式:

```
insert overwrite table tablename partition (partcol1, partcol2 ...)
select_statement from from_statement;
```

 注:

- **'select_statement'** に続くフィールドにターゲットテーブルの動的パーティション値を指定します。 ターゲットテーブルが **1** レベルの動的パーティションの場合、**select_statment** の最後のフィールドの値が、ターゲットテーブルの動的パーティション値となります。
- 現時点では、分散環境で各プロセスの出力できる動的パーティション数は最大 **512** です。**512** を超えると、エラーが発生します。
- 現時点では、動的パーティション **SQL** で生成できる動的パーティションは、最大 **2,000** です。**2,000** を超えると、エラーが発生します。
- 動的パーティションの値を **NULL** にすることはできません。また、特殊文字、漢字も使用できません。この制約に沿っていない場合は、例外が返されます。 例外:

```
FAILED: ODPS-0123031:Partition exception - invalid dynamic
partition value:
```

```
                province=xxx
```

- ターゲットテーブルにマルチレベルパーティションがある場合、**'INSERT** 文**'** で一部のパーティションを静的パーティションに設定できます。ただし、静的パーティションは上位パーティションである必要があります。

例

動的パーティションについて説明するための簡単な例：

```
create table total_revenues (revenue bigint) partitioned by (region
string);
    insert overwrite table total_revenues partition(region)
        select total_price as revenue, region
            from sale_detail;
```

上記の例のとおり、**SQL** を実行する前にどのパーティションが生成されるか、ユーザーにはわかりません。 **SELECT** 文の実行が完了しないと、**region** 値で作成されたパーティションがどれかユーザーは確認できません。 そのため、動的パーティションといいます。

その他の例**:**

```
create table sale_detail_dypart like sale_detail; --Create target
table.
```

- -- 例 **1:**

```
insert overwrite table sale_detail_dypart partition (sale_date,
region)
select shop_name,customer_id,total_price,sale_date,region from
sale_detail;
    -- 正常終了します。
```

  - **'sale_detail'** テーブルの **sale_date** の値によって、ターゲットテーブルの **sale_date** パーティション値が決まります。また、**region** の値によって、ターゲットテーブルの **region** パーティションの値が決まります。
  - 動的パーティションでは、**select_statement** フィールドとターゲットテーブルの動的パーティションとの対応付けは、フィールドの順序で決まります。 たとえば、**SELECT** 文が次のように記述されているとします。

    ```
    select shop_name,customer_id,total_price,region,sale_date from
              sale_detail;
    ```

    region の値によってターゲットテーブルの **sale_date** パーティション値が決まり、**sale_date** の値によってターゲットテーブルの **region** パーティション値が決まります。

- -- 例 2:

```
insert overwrite table sale_detail_dypart partition (sale_date='2013
', region)
        select shop_name,customer_id,total_price,region from
sale_detail;
    -- 正常に終了します。マルチパーティション。セカンダリパーティションを指定しま
す。
```

- -- 例 3:

```
insert overwrite table sale_detail_dypart partition (sale_date='2013
', region)
        select shop_name,customer_id,total_price from sale_detail;
    -- エラー情報が返されます。 動的パーティションを挿入する場合、動的パーティ
ション列を Select リストに含めなければなりません。
```

- -- 例 4:

```
insert overwrite table sales partition (region='china', sale_date)
select shop_name,customer_id,total_price,sale_date from sale_detail;
    -- エラー情報が返されます。 下位のサブパーティションのみを指定することはでき
ません。上位パーティションを動的に挿入する必要があります。
```

MaxCompute の旧バージョンでは、動的パーティション処理を行う場合に、パーティション列および対応する **SELECT** リストの型が同じでない場合には、エラーが報告されました。**MaxCompute 2.0** では、次のとおり、暗黙的変換が実行されます。

```
create table parttable(a int, b double) partitioned by (p string);
insert into parttable partition(p) select key, value, current_ti
mestmap() from src;
select * from parttable;
```

結果は次のとおりです。

a	b	c
0	NULL	2017-01-23 22:30:47.130406621
0	NULL	2017-01-23 22:30:47.130406621

5.7.4 VALUES

ここでは、`INSERT … VALUES` コマンド操作について説明します。

検証用に、小さなデータテーブルに基本データを用意します。`INSERT … VALUES` 文を使用して、迅速にテストデータを検証用のテーブルに書き込むことができます。

 注:

現時点では、**INSERT OVERWRITE** で列を挿入することはできません。代わりに `INSERT INTO` をご使用ください。

文の形式:

```
INSERT INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)][co1name1,colname2...]
[VALUES (col1_value,col2_value,...),(col1_value,col2_value,...),...]
```

例 1:

```
drop table if exists srcp;
create table if not exists srcp (key string ,value bigint) partitioned
 by (p string);
insert into table srcp partition (p='abc') values ('a',1),('b',2),('c
',3);
```

上記の文が正常に実行されると、パーティション 'abc' は次のようになります。

```
| key | value | p |

| a | 1 | abc |
| b | 2 | abc |
| c | 3 | abc |
```

テーブルに複数の列があり、一部の列にデータを挿入するには、次のように INSERT 関数にリストを使用します。

例 2:

```
drop table if exists srcp;
create table if not exists srcp (key string ,value bigint) partitioned
 by (p string);
insert into table srcp partition (p)(key,p) values ('d','20170101'),('
e','20170101'),('f','20170101');
```

上記の文が正常に実行されると、パーティション '20170101' は、次のようになります。

```
| key | value | p |

| d | NULL | 20170101 |
| e | NULL | 20170101 |
| f | NULL | 20170101 |
```

VALUES で指定されない列は、デフォルトで NULL になります。 INSERT 関数にリストを指定する際、すべての列を VALUES で指定する必要はありません。'INSERT INTO…SELECT…' も同様に使用できます。

実際、VALUES は、INSERT 文でのみ使用するわけではありません。DML 文でも使用できます。

INSERT…VALUES メソッドには制約があり、値を指定する必要があります。 挿入されたデータに対して単純な操作を実行するには、**MaxCompute** のテーブル関数である **VALUES** を使用します。詳細は、例 **3** をご参照ください。

例 **3:**

```
drop table if exists srcp;
create table if not exists srcp (key string ,value bigint) partitioned
 by (p string);
insert into table srcp partition (p) select concat(a,b), length(a)+
length(b),'20170102' from values ('d',4),('e',5),('f',6) t(a,b);
```

VALUES (…), (…) t (a, b) は、テーブル **t** の定義であり、列は **a** および **b**、各列のデータ型は **VALUES** のリストから派生した **(a string, b bigint)** となります。 そうすることにより、物理テーブルを用意することなしに、任意のデータを含む複数行の仮想テーブルに対して、任意の処理を実行できます。

上記の文が正常に実行されると、**'20170102'** のパーティションは、次のようになります。

```
| key | value | p |

| d4 | 2 | 20170102 |
| e5 | 2 | 20170102 |
| f6 | 2 | 20170102 |
```

VALUES TABLE を使用して、次のように **select * from dual** や **union all** の組み合わせを置き換えて、定数を指定することもできます。 例

```
select 1 c from dual
union all
select 2 c from dual;
--等同于
select * from values (1), (2) as t (c);
```

VALUES の特殊な使用方法は次のとおりです。

```
select abs(-1), length('abc'), getdate();
```

上記の文のように、**SELECT** 文の式のリストに、アップストリームテーブルのデータが含まれない場合、**SELECT** は **FROM** なしで実行できます。 実際の処理としては、**1** 行 **0** 列の、データの存在しないダミーテーブルより抽出しています。 これにより、**DUAL** テーブルを作成せずに、**UDF** などの関数を検証することが可能となります。

 注：

- **VALUES** には定数のみを指定でき、**ARRAY** といった複雑なデータ型を含む関数を使用することはできません。 現時点では、**MaxCompute** では、対応する定数を構成できません。 次のような文に書き換えます。

```
insert into table srcp (p ='abc') select 'a',array('1', '2', '3');.
```

この文でも、同様の結果が得られます。

- **VALUES** で **datetime** または **timestamp** を書き込むには、**VALUES** 文に型名を指定します。

```
insert into table srcp (p ='abc') values (datetime'2017-11-11
            00:00:00',timestamp'2017-11-11 00:00:00.123456789');
```

5.8 側面図

側面図は、分割、爆発などの **UDTF** と組み合わせて使用されます。 データの行を複数の行に分割し、分割されたデータをこれに基づいて集計できます。 横方向のビューは、最初に元のテーブルの各行に対して **UDTF** を呼び出します。 **UTDF** は **1** つの行を **1** つ以上の行に分割します。 側面図は結果を組み合わせて、エイリアステーブルをサポートする仮想テーブルを生成します。

シングルラテラルビューステートメント

構文：

```
lateralView：LATERAL VIEW [OUTER] udtf（式）tableAlias AS columnAlias（
 '、' columnAlias）* from句：FROM baseTable（lateralView）*
```

ノート：

- 外側ビュー外側：表関数が行を出力しない場合、対応する入力行は側面ビューの結果に残り、すべての表関数出力リストは **NULL** になります。

例：

2 列のデータを持つ **"pageAds"**というテーブルがあるとします。最初の列は **"pageid** 文字列**"**、**2** 番目の列は **"adid_list"**、**AD ID** のカンマ区切りのコレクションです。

→string pageid	Array<int> adid_list
“front_page”	[1, 2, 3]
“contact_page”	[3, 4, 5]

要件は、すべての **AD ID** が出現した回数を数えることです。 構成の手順は次の通りです。

1. 次のように **AD ID** を分割します。

```
SELECT pageid, adid
 FROM pageAds LATERAL VIEW explode(adid_list) adTable AS adid;
```

実行結果は次のとおりです。

string pageid	int adid
“front_page”	1
“front_page”	2
“front_page”	3
“ contact_page”	3
“ contact_page”	4
“ contact_page”	5

2. 集計の統計

```
SELECT adid, count(1)
    FROM pageAds LATERAL VIEW explode(adid_list) adTable AS adid
GROUP BY adid;
```

結果：

int adid	count(1)
1	1
2	1
3	2
4。	1
50	1

複数の側面ビューステートメント

from ステートメントの後に複数の **Lateral View** ステートメントを続けることができます。後続の **Lateral View** ステートメントは、以前のすべてのテーブルと列を参照できます。

次の表が例として示されています。

Array<int> col1	Array<string> col2
[1, 2]	[“a” , “b” , “c”]
[3, 4]	[“d” , “e” , “f”]

・単一のステートメントを実行します。

```
SELECT myCol1, col2 FROM baseTable
    LATERAL VIEW explode(col1) myTable1 AS myCol1;
```

結果：

int mycol1	Array<string> col2
1	[“a” , “b” , “c”]
2	[“a” , “b” , “c”]
3	[d” , “e” , “f”]
4	[d” , “e” , “f”]

・次のように **Lateral View** ステートメントを追加します。

```
SELECT myCol1, myCol2 FROM baseTable
    LATERAL VIEW explode(col1) myTable1 AS myCol1
    LATERAL VIEW explode(col2) myTable2 AS myCol2;
```

結果は次のとおりです。

int myCol1	string myCol2
1	“a”
1	“b”
1	“c”
2	“あ”
2	“ b”
2	“ c”
3	“d”
3	“e”
3	“f”
4	“d”
4	“e”
4	“f”

5.9 Select 操作

5.9.1 SELECT 構文の説明

ここでは、**Select** 構文の形式を説明します。また、**Select** 構文を使用して、ネスト化されたクエリ、ソート操作、グループ化されたクエリを実行する際の注意事項について説明します。

SELECT 構文の説明

コマンド形式：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[ORDER BY order_condition]
[DISTRIBUTE BY distribute_condition [SORT BY sort_condition] ]
[LIMIT number]
```

注：

- **Select** 文でテーブルからデータを読み込む場合、列名を指定して読み込むか、アスタリスク (*) を使用してすべての列を読み込みます。 基本的な **Select** 文を以下に示します。

  ```
  select * from sale_detail;
  ```

  sale_detail から **shop_name** 列のみを読み込むには、次の文を使用します。

  ```
  select shop_name from sale_detail;
  ```

  フィルタリング条件を指定するには、**where** 句を使用します。 例：

  ```
  select * from sale_detail where shop_name like 'hang%';
  ```

  Select 文を実行すると、結果は最大で **10,000** 行のみ表示されます。 ただし、**Select** 文が句の場合は、すべての結果が上位クエリに返されます。
- パーティションテーブルを選択すると、フルテーブルスキャンは実行できません。

  2018 年 **1** 月 **10** 日 **20:00 (UTC+8)** 以降に作成された新規プロジェクトについては、**SQL** 実行時にパーティションテーブルに対してフルテーブルスキャンを実行できないようにデフォルト設定されています。 不必要な **SQL I/O**、コンピューティングリソース、コストを削減するために、パーティション条件でスキャン対象のパーティションを指定する必要があります。
  注**:** 課金方法が従量課金の場合、データ入力量が課金パラメーターの **1** つになります。

  テーブル定義が `t1(c1,c2) partitioned by(ds)` の場合、新規プロジェクトで次の文を実行しないようにしてください。エラーが発生する可能性があります。

  ```
  Select * from t1 where c1=1;
  ```

```
Select * from t1 where (ds='20180202' or c2=3);
Select * from t1 left outer join t2 on a.id =b.id and a.ds=b.ds and
b.ds='20180101');
--Join 文の実行時、パーティションのクリッピングが where 句に指定されている場
合、パーティションのクリッピングは有効です。 このコマンドを on 句で指定すると、サ
ブテーブルのパーティションがクリッピングされ、メインテーブルでフルテーブルスキャン
が実行されます。
```

パーティションテーブルでフルテーブルスキャンを実行する場合は、パーティションテーブル全体のスキャンを実行する **SQL** 文の前に、`set odps.sql.allow.fullscan=true;` 文を追加します。 **set** 文は、**SQL** 文と一緒に送信する必要があります。 **sales_detail** テーブルがパーティションテーブルとします。 フルテーブルスキャンを実行するには、次の基本的なクエリ文を一緒に送信します。

```
set odps.sql.allow.fullscan=true;
select * from sale_detail;
```

- **table_reference** では、ネストしたサブクエリを使用できます。例：

```
select * from (select region from sale_detail) t where region = '
shanghai';
```

- **where** 句で使用できるフィルタリング条件は、次のとおりです。

フィルタリング条件	説明
>、 <、 =、 >=、 <=、 <>	関係演算子
like、rlike	**like** と **rlike** のソースパラメーターとパターンパラメーターは **String** 型にする必要があります。

フィルタリング条件	説明
in、not in	**in** や **not in** 条件にサブクエリを追加すると、**1** つの列の値しか返されず、戻り値の数は **1000** を超えることはできません。

テーブル全体をスキャンせずに、特定のパーティションをスキャンするには、次のように **Select** 文の **where** 句でパーティションの範囲を指定します。

```
SELECT sale_detail. * FROM sale_detail WHERE sale_detail.sale_date
 >= '2008' AND sale_detail.sale_date <= '2014';
```

MaxCompute SQL の **where** 句では、**between**…**and** 条件でのクエリを使用できません。前述した **SQL** 文は、次のように書き換えることができます。

```
SELECT sale_detail. * FROM sale_detail WHERE sale_detail.sale_date
BETWEEN '2008' AND '2014';
```

- `distinct`**:** 重複した行がある場合、フィールドの前に **Distinct** オプションを指定すると、重複行は取り除かれます。 この場合、**1** つの行のみ返されます。 **All** オプションを使用する、または **Distinct** オプションを使用しない場合、フィールド内のすべての重複値が返されます。

  Distinct オプションを使用すると、**1** 行のレコードのみが返されます。以下に例を示します。

```
select distinct region from sale_detail;
select distinct region, sale_date from sale_detail;
-- 複数の列に対して Distinct オプションを実行します。 Distinct オプションは、
1つの 行ではなく、Select 列のセットに対して有効です。
```

- `group by`**:** グループ化のクエリです。 通常、集約関数で使用されます。 **Select** 文に集約関数が含まれる場合、次のルールに従ってください。

  1. **group by** のキーは、入力テーブルの列名にすることができます。
  2. 入力テーブルの列から構成される式でも構いません。 **Select** 文の出力列の別名をキーにすることはできません。
  3. 優先度は、ルール **1** のほうが **2** よりも高いです。 ルール **1** と **2** が競合する場合、つまり **group by** のキーが、**Select** 文の入力テーブルの列または式、および **Select** 文の出力列の場合、ルール **1** の優先度が高くなります。

  例**:**

```
select region from sale_detail group by region;
-- グループ化列としてそのまま使用される入力テーブル内の列名を指定しており、正常に
実行されます。
select sum(total_price) from sale_detail group by region;
-- region 値でグループ化したテーブルを指定しており、正常に実行されます。各グルー
プの合計売上が返されます。
Select region, sum (total_price) from sale_detail group by region;
-- region 値でグループ化したテーブルを指定しており、正常に実行されます。region
 値（グループ内で一意）と各グループの合計売上が返されます。
```

```
select region as r from sale_detail group by r;
-- Select 列の別名を指定して実行すると、エラーが返されます。
select 2 + total_price as r from sale_detail group by 2 + total_pric
e;
-- 列の完全な式が必要です。
Select region, total_price from sale_detail group by region;
-- エラーが返されます。Select 文で集計関数を使用していない列は、group by に存
在しなければなりません。
select region, total_price from sale_detail group by region,
total_price;
-- 正常に実行されます。
```

Select 操作の前に **group by** があるので、**SQL** 解析時に制限事項が適用されます。 したがって、**group by** 文では、入力テーブルの列や式は、キーとしてのみ使用できます。



注：

詳細は、「集計関数 *(Aggregate Functions)*」をご参照ください。

- **order by:** 特定の列に基づいて、すべてのデータがソートされます。 レコードを降順でソートするには、**DESC** キーワードを使用します。 全体をソートする場合、**order by** を **limit** と一緒に使用する必要があります。 ソートの際に **order by** を使用すると、**NULL** が最小値とみなされます。 この動作は、**MySQL** でも同じですが、**Oracle** では異なります。

  group by とは異なり、**order by** の後ろには **Select** 列の別名を指定する必要があります。 **Select** 操作を任意の列で実行する際に、列の別名が指定されていない場合は、列の別名として列名が使用されます。

  ```
  select * from sale_detail order by region;
  -- エラーが返されます。order by と一緒に limit を使用していません。
  select * from sale_detail order by region limit 100;
  select region as r from sale_detail order by region limit 100;
  -- エラーが返されます。ORDER BY の後ろに列の別名が指定されていません。
  select region as r from sale_detail order by r limit 100;
  ```

  `limit number` の数字は、出力行数を制限するための定数です。 **limit** オプションを指定せずに、**Select** 文の結果を画面に表示すると、最大 **10000** 行が表示されます。 画面表示の上限は、プロジェクトによって異なる場合があります。制限数は、`setproject` コンソールで制御できます。

- **Distribute by:** 特定の列の値に基づき、データに対してハッシュベースのシャーディングを行います。 **Select** の出力列の別名を使用する必要があります。

  ```
  select region from sale_detail distribute by region;
  -- 列名が別名なので、正常に実行されます。
  select region as r from sale_detail distribute by region;
  -- エラーが返されます。DISTRIBUTE BY の後ろに列の別名が指定されていません。
  ```

```
select region as r from sale_detail distribute by r;
```

- **Sort by:** 部分的に並べ替えるには、**distribute by** を最初に追加する必要があります。**sort by** は、**distribute by** の結果を部分的にソートします。 **Select** 文の出力列の別名を使用する必要があります。

```
select region from sale_detail distribute by region sort by region;
select region as r from sale_detail sort by region;
-- エラーが返され、終了します。distribute by が指定されていません。
```

- **order by** と **group by** は、**distribute by** や **sort by** 句と同時に使用できません。 **Select** 出力列の別名を使用する必要があります。

 注:

- **order by**、**sort by**、**distribute by** のキーは、**Select** 文の出力列 (列の別名)でなければなりません。
- **MaxCompute** の **SQL** 解析では、**order by**、**sort by**、**distribute by** 操作は **Select** 操作より後に実行されます。 したがって、**Select** 文の出力列のみをキーとして指定できます。

5.9.2 SELECT 実行順序

SELECT 構文に従って記述された **SELECT** 文の実際の論理的な実行順序は、標準の記述順序とは異なります。

次の例をご参照ください。

```
SELECT  key
        ,MAX(value)
FROM    src t
WHERE   value > 0
GROUP BY key
HAVING  SUM(value) > 100
ORDER BY key
LIMIT   100
;
```

実際の論理的な実行順序は、`FROM->WHERE->GROUP BY->HAVING->SELECT->ORDER BY->LIMIT` です。

- **ORDER BY** で参照されるのは、**SELECT** リストに生成された列のみです。**FROM** ソーステーブル内の列にはアクセスしません。
- **HAVING** 句では、**GROUP BY** キーと集計関数にアクセスします。 **SELECT** 操作の実行時、**GROUP BY** が指定されている場合、**SELECT** でアクセスできるのは、グループキーと集計関数だけです。 **FROM** ソーステーブルの列にはアクセスしません。
- **SELECT** リストに生成された列のみ参照できます。**From** のソーステーブルの列にはアクセスできません。

混乱を避けるため、**MaxCompute** では、クエリ文を実行順に記述できます。 たとえば、前述の文は、次のように記述できます。

```
FROM    src t
WHERE   value > 0
GROUP BY key
HAVING  SUM(value) > 100
SELECT  key
        ,MAX(value)
ORDER BY key
LIMIT   100
;
```

例 **2**：

```
SELECT  shop_name
        ,total_price
        ,region
FROM    sale_detail
WHERE   total_price > 150
DISTRIBUTE BY region
SORT BY region
;
```

実際の論理的な実行順序は、`FROM->WHERE->SELECT->DISTRIBUTE BY->SORT BY`です。

5.9.3 サブクエリ

サブクエリの基本定義

通常の **SELECT** 操作では、複数のテーブルからデータが読み込まれます。例として、**select** `column_1, column_2 … from table_name` などが挙げられます。 しかし、クエリオブジェクトは、次のように別の **SELECT** 操作で記述することもできます。

```
select * from (select shop_name from sale_detail) a;
```



注：

サブクエリでは、別名を指定する必要があります。

FROM 句では、サブクエリをテーブルとして使用して、他のテーブルやサブクエリとの **JOIN** 操作を実行できます。例：

```
create table shop as select * from sale_detail;
select a.shop_name, a.customer_id, a.total_price from
(select * from shop) a join sale_detail on a.shop_name = sale_detail.
shop_name;
```

IN サブクエリ/NOT IN サブクエリ

IN サブクエリは、**LEFT SEMI JOIN** と同様の処理を行います。

例:

```
SELECT * from mytable1 where id in (select id from mytable2);
-- 上の文は下の文と同じです。
SELECT * from mytable1 a LEFT SEMI JOIN mytable2 b on a.id=b.id;
```

現在、MaxCompute では、IN サブクエリと相関条件の両方に対応しています。

例:

```
SELECT * from mytable1 where id in (select id from mytable2 where
value = mytable1.value);
```

サブクエリ内の `where value = mytable1.value` は、相関条件です。 MaxCompute の以前のバージョンでは、サブクエリと外部クエリの両方でソーステーブルを参照する式があると、エラーが返されました。 現在、MaxCompute は、このような式にも対応しています。 実際、このようなフィルタリング条件は、SEMI JOIN の ON 条件の一部です。

NOT IN サブクエリは、LEFT ANTI JOIN と同様の処理を行います。 ただし、大きな違いが 1 つあります。

例:

```
SELECT * from mytable1 where id not in (select id from mytable2);
-- mytable2 で、どの ID も NULL でなければ、上の文は下の文と同じです。
SELECT * from mytable1 a LEFT ANTI JOIN mytable2 b on a.id=b.id;
```

mytable2 に ID が NULL の列がある場合、NOT IN 式は NULL になるため、WHERE 条件は無効となり、データは返されません。 これが LEFT ANTI JOIN と異なる点です。

MaxCompute 1.0 では、 [NOT] IN サブクエリが JOIN 条件として機能しなければ (例：WHERE 文以外)、使用できます。WHERE 文の中であっても、JOIN 条件への変換時にエラーが発生します。 MaxCompute 2.0 では、 引き続きこの機能を使用できます。 ただし、[NOT] IN サブクエリを SEMI JOIN に変換できないので、サブクエリを実行するには、別のジョブを開始する必要があります。 [NOT] IN サブクエリは、相関条件には対応していません。

例:

```
SELECT * from mytable1 where id in (select id from mytable2) OR value
 > 0;
```

WHERE 句に OR が含まれるので、[NOT] IN サブクエリは SEMI JOIN に変換できません。 サブクエリを実行するには、別のジョブを開始する必要があります。

また、パーティションテーブルを特別に処理します。

```
SELECT * from sales_detail where ds in (select dt from sales_date);
```

ds がパーティション列の場合、`select dt from sales_date` は **SEMI JOIN** を変換せずに、サブクエリを実行するためのジョブを開始します。 実行後、結果は **ds** と **1** つずつ比較されます。 **sales_detail** の **ds** 値が、戻された結果の中にない場合、パーティションプルーニングの有効性を保証するために、パーティションは読み取られません。

EXISTS **サブクエリ**/NOT EXISTS **サブクエリ**

EXISTS サブクエリでは、少なくとも **1** つのデータ行がサブクエリ内にあれば、**TRUE** が返されます。**1** 行もない場合には、**FALSE** が返されます。 **NOT EXISTS** サブクエリは、これと全く逆です。

現在、**MaxCompute** では、相関 **WHERE** 条件を含むサブクエリのみ使用できます。 **EXISTS** サブクエリや **NOT EXISTS** サブクエリは、**LEFT SEMI JOIN**、**LEFT ANTI JOIN** に変換されてから実行されます。

例**:**

```
SELECT * from mytable1 where exists (select * from mytable2 where id
 = mytable1.id);
-- 上の文は下の文と同じです。
Select * From mytable1 a left semi join mytable2 B on A. ID = B. ID;
```

また、

```
SELECT * from mytable1 where not exists (select * from mytable2 where
id = mytable1.id);
-- 上の文は下の文と同じです。
SELECT * from mytable1 a LEFT ANTI JOIN mytable2 b on a.id=b.id;
```

5.9.4 UNION ALL/UNION [DISTINCT]

UNION 演算子は、**2** つ以上の **SELECT** 文の結果をマージするために使用します。

構文形式は、次のとおりです。

```
select_statement UNION ALL select_statement;
select_statement UNION [DISTINCT] select_statement;
```

- `UNION ALL`**: SELECT** 操作によって戻される **2** つ以上のデータセットを**1**つに結合します。 結果に重複行が含まれる場合、重複行は除外されず、条件を満たすすべての行が戻されます。
- `UNION [DISTINCT]`**:** この文では、**DISTINCT** を省略できます。 **SELECT** 操作によって戻される **2** つ以上のデータセットが **1** つのデータセットに結合されます。 結果に重複行が含まれる場合、重複行は除外されます。

以下は、**UNION ALL** 操作の例です。

```
Select * From sale_detail where region = 'Hangzhou'
        union all
select * from sale_detail where region = 'shanghai';
```

以下は、**UNION** 操作の例です。

```
SELECT * FROM src1 UNION SELECT * FROM src2;
--The execution effect is equivalent to
SELECT DISTINCT * FROM (SELECT * FROM src1 UNION ALL SELECT * FROM
src2) t;
```

 注:

- **UNION ALL** および **UNION** 操作の対象になる照会列の数と型は、一致していなければなりません。
- 通常、**MaxCompute** で **UNION ALL** や **UNION** 操作を実行できるのは、最大 **256** テーブルまでです。 テーブル数がこの制限を超えると、構文エラーが返されます。

UNION の後に指定する **LIMIT** 句の説明

UNION の後に **CLUSTER BY**、**DISTRIBUTE BY**、**SORT BY**、**ORDER BY**、**LIMIT** 句が続く場合、その句が適用されるのは **UNION** の最後の **SELECT** 文に対してではなく、その前の **UNION** 全体に対してです。現在、**MaxCompute** では、この処理が、`set odps.sql.type.system.odps2=true;` で設定されます。

例

```
set odps.sql.type.system.odps2=true;
SELECT explode(array(3, 1)) AS (a) UNION ALL SELECT explode(array(0, 4
, 2)) AS (a) ORDER BY a LIMIT 3;
```

次の結果が戻されます。

```
| a |

| 0 |
| 1 |
| 2 |
```

5.9.5 JOIN 操作

MaxCompute の **JOIN** 操作では、複数の結合条件を指定できますが、**ON** 条件なしで結合するデカルト積には対応していません。

関数の定義

```
join_table:
```

```
        table_reference join table_factor [join_condition]
        | table_reference {left outer|right outer|full outer|inner}
join table_reference join_condition
    table_reference:
        table_factor
        | join_table
    table_factor:
        tbl_name [alias]
        | table_subquery alias
        | ( table_references )
    join_condition:
        on equality_expression ( and equality_expression )*
```

 注:

equality_expression は等価式です。

LEFT OUTER JOIN: 右テーブル **(sale_detail)** に一致するレコードがない場合でも、 左テーブル **(shop)** からすべてのレコードが返されます。

```
select a.shop_name as ashop, b.shop_name as bshop from shop a
        left outer join sale_detail b on a.shop_name=b.shop_name;
    -- テーブル shop とテーブル sale_detail には、どちらも列 shop_name があ
り、select 句ではエイリアスを使用して区別する必要があります。
```

RIGHT OUTER JOIN: 右結合を示します。 左テーブルに一致するレコードがない場合でも、右テーブルからすべてのレコードが返されます。

例:

```
select a.shop_name as ashop, b.shop_name as bshop from shop a
        right outer join sale_detail b on a.shop_name=b.shop_name;
```

FULL OUTER JOIN: 全結合を示します。 左右両方のテーブルからすべてのレコードが返されます。

例:

```
select a.shop_name as ashop, b.shop_name as bshop from shop a
        full outer join sale_detail b on a.shop_name=b.shop_name;
```

テーブルに一致するレコードが **1** つ以上ある場合、**INNER JOIN** はその行を返します。 キーワード **INNER** は、省略できます。

```
select a.shop_name from shop a inner join sale_detail b on a.shop_name
=b.shop_name;
```

```
select a.shop_name from shop a join sale_detail b on a.shop_name=b.
shop_name;
```

JOIN 条件には、**and** を使用した等価結合条件のみ使用できます。 **MAPJOIN** でのみ、非等価結合条件や、**or** を使用して複数の結合条件を指定することができます。

```
select a.* from shop a full outer join sale_detail b on a.shop_name=b.
shop_name
         full outer join sale_detail c on a.shop_name=c.shop_name;
    -- 複数の結合例に対応
select a.* from shop a join sale_detail b on a.shop_name ! = b.
shop_name;
    -- エラーが返されます。非等価結合条件はサポートされていません。
```

IMPLICIT JOIN: MaxCompute は、次の **JOIN** メソッドに対応しています。

```
SELECT * FROM table1, table2 WHERE table1.id = table2.id;
--以下の実行結果と同じです。
SELECT * FROM table1 JOIN table2 ON table1.id = table2.id;
```

5.9.6 SEMI JOIN

MaxCompute は **SEMI JOIN** をサポートします。 **SEMI JOIN** では、右側のテーブルは結果セットに表示されず、左側のテーブルのデータをフィルタ処理するためにのみ使用されます。 サポートされている構文は次のとおりです。**LEFT SEMI JOIN** と **LEFT ANTI JOIN**。

LEFT SEMI JOIN

JOIN 条件が有効な場合は、左の表のデータが返されます。 つまり、**mytable 1** の行の **ID** が **mytable 2** のすべての **ID** に含まれている場合、この行は結果セットに保存されます。

例：

```
mytable1 aから左の SEMI に参加 mytable2 b a.id = b.id を選択します。
```

mytable 1 の **ID** が **mytable 2** の **ID** に現れると、**mytable 1** のデータのみが返されます。

LEFT ANTI JOIN

JOIN 条件が無効な場合は、左の表のデータが返されます。 つまり、**mytable1**の行の**ID**が **mytable2**のどの**ID**にも現れない場合、この行は結果セットに格納されます。

例：

```
mytable 1 から左の ANTI JOIN mytable 2 を選択します。a.id = b.id;
```

mytable 1 の **ID** が **mytable 2** の **ID** に表示されない場合、**mytable 1** のデータのみが返されます。

5.9.7 MAPJOIN のヒント

MapJoin は、大きなテーブルと **1** つまたは複数の小さなテーブルを結合するのに役立ちます。通常の結合操作よりも高速です。

MapJoin の典型的なシナリオは次のとおりです。データ量が少ない場合、**SQL** は結合操作を実行するプログラムのメモリに、指定されたすべての小さいテーブルをロードして、**JOIN** 実行を高速化します。

 注：

MapJoin を使用するときは、以下にご注意ください。

- 「左外部結合」の左側のテーブルは大きなテーブルである必要があります。
- 右外部結合の右表は大きな表でなければなりません。
- **INNER JOIN** では、左右両方のテーブルを大きなテーブルにすることができます。
- **FULL OUTER JOIN** の場合、**MapJoin** は使用できません。
- **MapJoin** は小さなテーブルをサブクエリとしてサポートします。
- **MapJoin** が使用されていて、小さなテーブルまたはサブクエリを参照する必要がある場合は、エイリアスを参照する必要があります。
- **MapJoin** は、等価でない **JOIN** 条件、または **OR** を使用して接続された複数の条件をサポートしています。
- 現在、**MaxCompute** では、最大 **8** つの小さなテーブルを **MapJoin** で指定できます。 それ以外の場合は、エラーが返されます。
- **MapJoin** が使用される場合、すべての小さなテーブルによって占有される合計メモリは **512 MB** を超えることはできません。 **MaxCompute** は圧縮ストレージを使用するため、小さなテーブルがメモリにロードされるとデータサイズは急激に拡大します。 **512 MB** の制限は、小さなテーブルがメモリにロードされた後のサイズを表します。 **We can config the total memory by** `SET odps.sql.mapjoin.memory.max=2048;` **. The maximum configurable memory is 2048MB.**
- **When JOIN is performed on the multiple tables, the two leftmost tables cannot be tables for MapJoin at the same time.**

例：

```
select /* + mapjoin(a) */
        a.shop_name,
        b.customer_id,
        b.total_price
    from shop a join sale_detail b
```

```
    on a.shop_name = b.shop_name;
```

MaxCompute SQL does not support complex JOIN conditions, such as non-equivalent expressions and the OR logic, in the ON condition of common JOIN operations. However, MapJoin supports such operations.

例：

```
select /*+ mapjoin(a) */
        a.total_price,
        b.total_price
    from shop a join sale_detail b
    on a.total_price < b.total_price or a.total_price + b.total_price
 < 500;
```

5.9.8 HAVING 句

MaxCompute SQL の **Where** キワードは集計関数と一緒に使用できないので、**HAVING** 句を使用します。

関数の定義

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

例

テーブル **Orders** は、**Customer**、**OrderPrice**、**Order_date**、**Order_id** の **4** フィールドから構成されます。 **OrderPrice** が **2,000** 未満の顧客を照会する **SQL** 文は、次のとおりです。

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

5.10 共通テーブル式 (CTE)

SQL 文の信頼性と実行効率を高めるため、**MaxCompute** は、標準 **SQL** の **CTE** に対応しています。

CTE の構文構造は、次のとおりです。

```
WITH
     cte_name AS

        cte_query

    [,cte_name2 AS

     cte_query2
```

```
    ,……]
```

- cte_name **CTE** 名を示します。**WITH** 句内で一意でなくてはいけません。 クエリ内の任意の位置にあっても、**cte_name** の識別子によって **CTE** であることが示されます。
- cte_query は **SELECT** 文です。結果セットは **CTE** を読み込むために使用されます。

例：

```
INSERT OVERWRITE TABLE srcp PARTITION (p='abc')
SELECT * FROM (
    SELECT a.key, b.value
    FROM (
        SELECT * FROM src WHERE key IS NOT NULL ) a
    JOIN (
        SELECT * FROM src2 WHERE value > 0 ) b
    ON a.key = b.key
) c
UNION ALL
SELECT * FROM (
    SELECT a.key, b.value
    FROM (
        SELECT * FROM src WHERE key IS NOT NULL ) a
    LEFT OUTER JOIN (
        SELECT * FROM src3 WHERE value > 0 ) b
    ON a.key = b.key AND b.key IS NOT NULL
)d;
```

最上位層の **UNION** の両側に **JOIN** 句が記述され、**JOIN** の左側のテーブルにも同じクエリが形成されます。 サブクエリを記述する場合は、このコードを繰り返す必要があります。

上記 **SQL** 文は、**CTE** を使用して次のように書き換えることができます。

```
with
  a as (select * from src where key is not null),
  b as (select * from src2 where value>0),
  c as (select * from src3 where value>0),
  d as (select a.key,b.value from a join b on a.key=b.key ),
  e as (select a.key,c.value from a left outer join c on a.key=c.key
and c.key is not null )
insert overwrite table srcp partition (p='abc')
select * from d union all select * from e;
```

書き換えた後、**"a"** に対応するサブクエリは、一度書き換えるだけで、その後再利用することができます。 **CTE** の **WITH** 句によって複数のサブクエリを指定でき、文全体で変数のように繰り返し使用できます。 再利用できるだけでなく、サブクエリは繰り返しネストする必要はありません。

5.11 Explain

MaxCompute SQL の **Explain** 操作では、**DML**文に対応する最終実行計画の構造についての説明が表示されます。 実行計画は、**SQL** セマンティクスを実行する際の最終ステージで使用されるプログラムです。

関数の定義

```
EXPLAIN <DML query>;
```

'explain' を実行すると、次の情報が表示されます。

- 指定した **DML** 文に対応する全タスクの依存構造
- タスク内のすべてのタスク依存構造
- タスク内のすべての演算子依存構造

例**:**

```
EXPLAIN
SELECT abs(a.key), b.value FROM src a JOIN src1 b ON a.value = b.value
;
```

Explain の出力結果には、次の項目が含まれます。

- ジョブ間の依存性**:** `job0 is root job`。このクエリでは **1** つのジョブ **(job0)** のみなので、**1** 行分の情報のみが必要です。
- タスク間の依存性**:**

```
In Job job0:
root Tasks: M1_Stg1, M2_Stg1
```

```
J3_1_2_Stg1 depends on: M1_Stg1, M2_Stg1
```

Job0 は **3** つのタスクから構成されます。**M1_Stg1** と **M2_Stg1** が最初に実行され、次に **J3_1_2_Stg1** が実行されます。

タスクの命名規則は、次のとおりです。

- **MaxCompute** には、**MapTask**、**ReduceTask**、**JoinTask**、**LocalWork** の **4** 種類のタスクがあります。
- タスク名の最初の文字は、タスクの種類を示します。 たとえば、**M2Stg1** は、**MapTask** を示します。
- 最初の文字に続く数字は、タスク **ID** を示します。タスク **ID** は現行のクエリに対応するすべてのタスクの中で一意でなければいけません。
- アンダースコア **(_)** で区切られた数字は、現行タスクとの直接の依存性を示します。 たとえば、**J3_1_2_Stg1** は、現行タスク **(**タスク **ID : 3)** が **2** つのタスク **(**タスク **ID: 1** と **2)** に依存していることを示します。

・ **3** つめの部分は、タスク内の演算子構造です。 演算子の文字列により、タスクの実行セマンティクスを示します。

```
In Task M1_Stg1:
  Data source: yudi_2.src # Data source は、このタスクの入力コンテンツを示
します
  TS: alias: a # TableScanOperator
      RS: order: + # ReduceSinkOperator
          keys:
               a.value
          values:
               a.key
          partitions:
               a.value
In Task J3_1_2_Stg1:
  JOIN: a INNER JOIN b # JoinOperator
      SEL: Abs(UDFToDouble(a._col0)), b._col5 # SelectOperator
          FS: output: None # FileSinkOperator
In Task M2_Stg1:
  Data source: yudi_2.src1
  TS: alias: b
      RS: order: +
          keys:
               b.value
          values:
               b.value
          partitions:
```

```
                b.value
```

- 演算子の説明:

  ■ TableScanOperator: クエリ文中の FROM 文ブロックの論理式を示します。 EXPLAIN の結果には、入力テーブル名 (エイリアス) が表示されます。

  ■ SelectOperator: クエリ文中の SELECT 文ブロックの論理式を示します。 Explain の結果には、次の演算子に渡される列が、カンマ (,) で区切って表示されます。

  ■ 列の参照が渡される場合、`< alias >.< column_name >` と表示されます。

  ■ 式の結果が渡される場合、`func1(arg1_1, arg1_2, func2(arg2_1, arg2_2))` のように、関数として表示されます。

  ■ 定数が渡される場合、その値が直接表示されます。

  ■ FilterOperator: クエリ文中の WHERE 文ブロックの論理式を示します。 SelectOperator の場合と同様に、Explain の結果には、WHERE 条件式が表示されます。

  ■ JoinOperator: クエリ文中の JOIN 文ブロックの論理式を示します。 Explain の結果には、結合されるテーブルと JOIN メソッドの両方が表示されます。

  ■ GroupByOperator: 集計演算の論理式を示します。 クエリ文中に集計関数を使用すると、その構造が表示されます。 Explain の結果には、集計関数の内容が表示されます。

  ■ ReduceSinkOperator: タスク間のデータ分散の論理式を示します。 現在のタスクの結果を別のタスクに渡す場合、現在のタスクの最後に ReduceSinkOperator を使用して、データ分散を実行する必要があります。 出力結果のソート方法、分散キー、ハッシュ値の計算に使用された値と列が、Explain の結果に表示されます。

  ■ FileSinkOperator: 最終データの保存操作を示します。 クエリ文中に Insert 文ブロックがある場合、挿入先のテーブル名が Explain の結果に表示されます。

  ■ LimitOperator: クエリ文中の LIMIT 文ブロックの論理式を示します。 Explain の結果には、制限数が表示されます。

  ■ MapjoinOperator: JoinOperator の場合と同様に、ターゲットテーブルの JOIN 操作を示します。

 注:

非常に複雑なクエリ文の場合、Explain の出力結果が多くなりすぎ、API 制限がトリガーされるため、Explain の結果がすべて表示されません。 このような場合には、クエリを分割して、部分ごとに Explain 操作を実行すると、ジョブ構造を把握することができます。

5.12 組み込み関数

5.12.1 日付関数

ここでは、**MaxCompute SQL**で **Datetime** 型を操作するためのさまざまな関数について説明します。

DATEADD

コマンド形式**:**

```
datetime dateadd(datetime date, bigint delta, string datepart)
```

コマンドの説明**:**

指定した単位「**datepart**」と範囲「**delta**」に従って、日付の値を変更します。

パラメーターの説明：

- `date`**: Datetime** 型で、日付の値です。 入力値が **String** 型の場合、暗黙の変換によって「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- `delta`**: Bigint** 型で、変更対象の日付範囲です。 入力値が「**String**」型または「**Double**」型の場合、暗黙の変換によって「**Bigint**」型に変換されます。 それ以外のデータ型の場合は、例外が発生します。「**delta**」が **0** より大きい場合は「足し算」、そうでない場合は、「引き算」を行います。
- `datepart`**: String** 型の定数です。 このフィールド値は、「**String**」と「**Datetime**」型の変換規則に従い指定します。つまり、「**yyyy**」は年を示し、「**mm**」は月を示します。

  「*String* 型 と *Datetime* 型の変換」をご参照ください。 さらに、拡張日付形式 (年 -「**year**」、月 -「**month**」または 「**mon**」、日 -「**day**」、 時間 -「**hour**」) も使用できます。 定数でない場合、サポート対象外の形式の場合、その他のデータ型の場合は、例外が報告されます。

戻り値**: Datetime** 型。入力値に **NULL** がある場合は、 **NULL** が返されます。



注 **:**

- 指定した単位に応じて「**delta**」が増減し、単位の上位桁が溢れたり、スペースで埋められることがあります 日、月、時、分、秒は、それぞれ **10** 進数、**12** 進数、**24** 進数、**60** 進数、**60** 進数で計算されます。
- 「**delta**」の単位が月の場合の計算ルールは次のとおりです。

  「**delta**」を加えた後に、「**Datetime**」の月部分によって日が溢れない場合は、日付は変わりません。そうでない場合、日の値は結果月の最終日に設定されます。

- 「**datepart**」の値は「**String**」型と「**Datetime**」型の変換規則に従います。つまり、「**yyyy**」は年、「**mm**」は月を示します。 特別な説明がない限り、関連する **Datetime** 型の組み込み関数はすべてこの規則に従います。 また、特別な指示がない限り、すべての **Datetime** 組み込み関数も拡張日付形式（年-「**year**」、月-「**month**」または「**mon**」、日-「**day**」、 時間-「**hour**」）に対応します。

例：

```
 if trans_date = 2005-02-28 00:00:00
dateadd(trans_date, 1, 'dd') = 2005-03-01 00:00:00
-- 1 日追加します。 結果は 2 月の最終日より大きくなります。 実際の値は、翌月の初日です。
dateadd(trans_date, -1, 'dd') = 2005-02-27 00:00:00
-- 1 日引きます。
dateadd(trans_date, 20, 'mm') = 2006-10-28 00:00:00
-- 20 か月追加します。 月が溢れ、年に ‘1’ が追加されます。
If trans_date = 2005-02-28 00:00:00, dateadd(transdate, 1, 'mm') = 2005-03-28 00:00:00
If trans_date = 2005-01-29 00:00:00, dateadd(transdate, 1, 'mm') = 2005-02-28 00:00:00
-- 2005 年 2 月 29 日 この日付が、現在の月の最終日に割り込まれます。
 If trans_date = 2005-03-30 00:00:00, dateadd(transdate, -1, 'mm') = 2005-02-28 00:00:00
```

 注：

"trans_date" の値は、例としてのみ使用します。 この基本的な式は、ファイルの日付と時間を表示するためによく使用されます。

MaxCompute SQLでは、**Datetime** 型は直接の定数表現を持たないので、次の使い方は間違っています。

```
select dateadd(2005-03-30 00:00:00, -1, 'mm') from tbl1;
```

Datetime 型の定数を記述する必要がある場合は、次の方法で記述します。

```
select dateadd(cast("2005-03-30 00:00:00" as datetime), -1, 'mm') from tbl1;
-- String 型の定数は、暗黙の変換によって datatime 型に変換されます。
```

DATEDIFF

コマンド形式**:**

```
bigint datediff(datetime date1, datetime date2, string datepart)
```

コマンドの説明**:**

指定した時間単位「**datepart**」で、**Datetime** 型である **date1** と **date2** の **2** つの差を計算します。

パラメーターの説明：

- datet1、date2: **Datetime** 型、被減数、減数分裂。入力値が「**String**」型の場合、暗黙の変換によって「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- datepart: **String** 型の定数です。拡張日付形式に対応しています。 「**datepart**」が指定した形式になっていない場合、または他のデータ型の場合、例外が報告されます。

戻り値:

Bigint 型が返されます。 入力値に **NULL** がある場合は、**NULL** が返されます。 **date1** が **date2** より小さい場合、戻り値は負になる可能性があります。

 注：
計算過程で「**datepart**」に従って単位の下部分が切り捨てられ、計算結果が返されます。

例：

```
If start = 2005-12-31 23:59:59, end = 2006-01-01 00:00:00:
    datediff(end, start, 'dd') = 1
    datediff(end, start, 'mm') = 1
    datediff(end, start, 'yyyy') = 1
    datediff(end, start, 'hh') = 1
    datediff(end, start, 'mi') = 1
    datediff(end, start, 'ss') = 1
    datediff('2013-05-31 13:00:00', '2013-05-31 12:30:00', 'ss') =
1800
    datediff('2013-05-31 13:00:00', '2013-05-31 12:30:00', 'mi') = 30
If start = 19:33:23. 234, end = 19:33:23. 250 .Dates with milliseconds
 do not belong to the standard datetime style, and cannot be converted
 implicitly directly.Explicit conversion is required here:

datediff(to_date('2018-06-04 19:33:23.250', 'yyyy-MM-dd hh:mi:ss.ff3
'),to_date('2018-06-04 19:33:23.234', 'yyyy-MM-dd hh:mi:ss.ff3') , '
ff3') = 16
```

DATEPART

コマンド形式:

```
bigint datepart(datetime date, string datepart)
```

コマンド形式:

「**date**」の「**datepart**」で指定された時間単位の値を取得します。

パラメーターの説明：

戻り値:

- date: **Datetime** 型。 入力値が「**String**」型の場合、「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- datepart: **String** 型の定数です。 拡張日付形式を使用できます。 「**datepart**」が指定された形式を満たさないか、または他のデータ型である場合、例外が報告されます。

Bigint 型が返されます。 入力値に **NULL** がある場合は、 **NULL** が返されます。

例：

```
datepart('2013-06-08 01:10:00', 'yyyy') = 2013
datepart('2013-06-08 01:10:00', 'mm') = 6
```

DATETRUNC

コマンド形式:

```
datetime datetrunc (datetime date, string datepart)
```

使用法：

指定した時間単位「**datepart**」が切り捨てた後、残りの日数値を返します。

パラメーターの説明：

- date: **Datetime** 型。 入力値が「**String**」型の場合、「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- datepart: **String** 型の定数です。 拡張日付形式を使用できます。「**datepart**」が指定された形式を満たさないか、または他のデータ型である場合、例外が報告されます。

戻り値:

Datetime 型。 入力値に **NULL** がある場合は、 **NULL** が返されます。

例：

```
datetrunc('2011-12-07 16:28:46', 'yyyy') = 2011-01-01 00:00:00
datetrunc('2011-12-07 16:28:46', 'month') = 2011-12-01 00:00:00
datetrunc('2011-12-07 16:28:46', 'DD') = 2011-12-07 00:00:00
```

GETDATE

コマンド形式:

```
datetime getdate()
```

コマンドの説明:

現在のシステム時間を取得します。 **MaxCompute** の標準時として **UTC+8** が使用されます。

戻り値:

Datetime 型。現在の日付と時刻が返されます。

 注:
(分散方式で実行される）**MaxCompute SQL**タスクでは、「**getdate**」は常に固定値を返します。 返される結果は、**MaxCompute** の **SQL** 実行期間内の任意の時刻になり、時間の精度は秒になります。

ISDATE

コマンド形式:

```
boolean isdate(string date, string format)
```

コマンドの説明:

対応する形式の文字列に従って、日付文字列を **Datetime** 型の値に変換できるかどうかを判断します。 変換が成功した場合は **TRUE** を返し、失敗した場合は **FALSE** を返します。

パラメーターの説明：

- `date`: **String** 型の日付値です。 入力値が「**Bigint** 型」、「**Double** 型」または「**Datetime** 型」の場合、「**String**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- `format`: **String** 型の定数です。 日付の拡張形式は使用できません。 冗長な形式の文字列が「**format**」で指定されると、形式の最初の文字列に対応する **Datatime** 値ガ取得されます。 他の文字列は区切り文字として扱われます。 たとえば、**isdate (**「**1234-yyyy**」、「**yyyy-yyyy**」**)** の場合は「**TRUE**」が返されます。

戻り値:

Boolean 型。 **NULL** の入力パラメータ値がある場合は、**NULL** が返されます。

LASTDAY

コマンド形式:

```
datetime lastday(datetime date)
```

コマンド形式:

その月の最後の日を取得します。日までの単位は切り捨てられ、「**hh:mm:ss**」部分は「**00:00:00**」となります。

パラメーターの説明：

`date`: **Datetime** 型。 入力値が「**String**」型の場合、「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。

戻り値:

Datetime 型になります。 入力パラメーター値が **NULL** の場合は、 **NULL** が返されます。

TO_DATE

コマンド形式:

```
datetime to_date(string date, string format)
```

コマンドの説明:

指定された形式に従って、「**date**」の文字列を **Datetime** 型の値に変換します。

パラメーターの説明：

- `date`: **String** 型で、変換対象の日付値です。 入力値が「**Bigint**」型、「**Double**」型、「**Datetime**」型の場合、暗黙の変換によって「**String**」型に変換されます。 それ以外のデータ型または **NULL** の場合は、例外が報告されます。
- `format`: **String** 型の定数で日付形式です。定数でないか、他のデータ型である場合は、例外が発生します。 フィールド **"format"** は拡張形式に対応しておらず、他の文字は分析プロセスで 無効な文字として無視されます。

  パラメーター `format` には少なくとも「**yyyy**」が含まれている必要があります。そうでなければ、例外が報告されます。 冗長な形式の文字列が `format` で指定されると、形式の最初の文字列に対応する **Datatime** 型の値が取得されます。他の文字列は区切り文字として扱われます。 たとえば、`to_date ('1234-2234', 'yyyy-yyyy')` は、「**1234-01-01 00:00:00**」と返されます。

  format 形式：**yyyy** は **4** 桁の年、**mm** は **2** 桁の月、**DD** は **2** 桁の日、**HH** は **24** 時間制、**MI** は **2** 桁の分、**SS** は **2** 桁の秒、**FF3** は **3** 桁の精度のミリ秒です。

戻り値:

Datetime 型で、形式は **yyyy-mm-dd hh: mi: ss** になります。 入力値に **NULL** がある場合は、 **NULL** が返されます。

例：

```
to_date('Alibaba2010-12*03', 'Alibabayyyy-mm*dd') = 2010-12-03 00:00:
00
to_date('20080718', 'yyyymmdd') = 2008-07-18 00:00:00
to_date('200807182030','yyyymmddhhmi')=2008-07-18 20:30:00
to_date('2008718', 'yyyymmdd') = null
```

```
-- 形式が要件を満たしていません。 例外が発生します。
to_date('Alibaba2010-12*3', 'Alibabayyyy-mm*dd') = null
-- 形式が非互換のため、例外が発生します。
to_date('2010-24-01', 'yyyy') = null
-- 形式が非互換のため、例外が発生します。
to_date('20181030 15-13-12.345','yyyymmdd hh-mi-ss.ff3')=2018-10-30 15
:13:12
```

TO_CHAR

コマンド形式:

```
string to_char(datetime date, string format)
```

コマンドの説明:

指定した形式に従って、**Datetime** 型の「日付」を **String** 型に変換します。

パラメーターの説明：

- `date`: **Datetime** 型で、変換対象の日付値です。 入力値が「**String**」型の場合、暗黙の変換によって「**Datetime**」型に変換されます。 それ以外のデータ型の場合は、例外が報告されます。
- `format`: **String** 型の定数です。 定数でない場合、または他のデータ型の場合は、例外が報告されます。 「**format**」の日付形式の部分が対応するデータに置き換えられ、他の文字は直接出力されます。

戻り値:

String 型が返されます。 **NULL** の入力パラメータ値がある場合は、**NULL** が返されます。

例：

```
to_char('2010-12-03 00:00:00', 'Alibabayyyy-mm*dd') = 'Alibaba2010-12*
03'
to_char('2008-07-18 00:00:00', 'yyyymmdd') = '20080718'
to_char('Alibaba2010-12*3', 'Alibabayyyy-mm*dd') -- -- 形式が非互換のた
め、例外が発生します。
to_char('2010-24-01', 'yyyy') -- -- 形式が非互換のため、例外が発生します。
to_char('2008718', 'yyyymmdd') -- -- 形式が非互換のため、例外が発生します。
```

他のデータ型から **String** 型への変換については、「*TO_CHAR*」をご参照ください。

UNIX_TIMESTAMP

コマンド形式:

```
bigint unix_timestamp(datetime date)
```

コマンドの説明:

Datetime 型の日付を **Bigint** 型の **UNIX** 形式の日付に変換します。

パラメーターの説明：

date: **Datetime** 型の日付値です。 入力値が「**String**」型の場合、「**Datetime**」型に変換され、計算で使用されます。 それ以外のデータ型の場合は、例外が報告されます。

戻り値:

Bigint 型で、**UNIX** 形式の日付値が返されます。 「**date**」が **NULL** の場合は **NULL** が返されます。

FROM_UNIXTIME

コマンド形式:

```
datetime from_unixtime(bigint unixtime)
```

コマンドの説明:

UNIXの時間値である「**unixtime**」を**Datetime** 型の日時値に変換します。

パラメーターの説明：

unixtime: **Bigint** 型の秒数です。**UNIX** 形式の日時値です。 入力値が、「**String**」型、「**double**」型の場合は、暗黙的な変換によって「**Bigint**」型に変換されます。

戻り値:

Datetime 型の日付値になります。 「**unixtime**」が **NULL** の場合は、**NULL** が返されます。

例：

```
from_unixtime(123456789) = 1973-11-30 05:33:09
```

WEEKDAY

コマンド形式:

```
bigint weekday(datetime date)
```

コマンドの説明:

日付に対応する現在の週の **n** 日目を返します。

パラメーターの説明：

date: **Datetime** 型。 入力値が「**String**」型の場合は、「**Datetime**」型に変換されてから演算に使用されます。 それ以外のデータ型の場合は、例外が報告されます。

戻り値:

Bigint 型。 入力パラメータ値が **NULL** の場合は **NULL** が返されます。 月曜日は週の最初の日とみなされ、対応する戻り値は **0** です。 他の曜日は **0** から昇順となります。 日曜日の場合、戻り値は **6** です。

WEEKOFYEAR

コマンド形式**:**

```
bigint weekofyear(datetime date)
```

コマンドの説明**:**

日付が含まれる年の **n** 番目の週を返します。 月曜日は週の最初の日とします。

 注 **:**
今週が今年に属しているか翌年に属しているかは、今週の時間の大部分 **(4**日以上**)** がどの年に属しているかによって決まります。

パラメーターの説明：

`date`**: Datetime** 型。 入力値が「**String**」型の場合、「**Datetime**」型に変換されてから演算に使用されます。 それ以外のデータ型の場合は、例外が報告されます。

戻り値**:**

Bigint 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例：

```
select weekofyear(to_date("20141229", "yyyymmdd")) from dual;
結果：
+------------+
| _c0 |
+------------+
| 1 |
+------------+
 -20141229 は 2014 に属しますが、この週の日付の大部分は 2015 です。したがって、戻される結果は 1 で、2015 の第 1 週であることを示します。
 select weekofyear(to_date("20141231", "yyyymmdd")) from dual ;
-- 1 を戻します。
 select weekofyear(to_date("20141229", "yyyymmdd")) from dual ;
```

```
-- 53 を戻します。
```

Maxcomputerte 2.0 に新しく追加された算術関数

MaxCompute 2.0 のアップグレード版では、新規の日付関数がいくつか追加されています。 新規関数を使用して、**Hive** モードと互換性のある新しいデータ型を設計する場合は、新規関数の **SQL** 文の前に次の **2** つの **set** 文を追加する必要があります。

```
set odps.sql.type.system.odps2=true;--Enable the new type.
```

両方を同時に送信するには、次の **SQL** 文を実行します。

```
set odps.sql.type.system.odps2=true;
select year('1970-01-01 12:30:00')=1970 from dual;
```

新規拡張関数の説明は、次のとおりです。

YEAR

コマンド形式**:**

```
INT year(string date)
```

コマンドの説明**:**

日付が属する年を返します。

パラメーターの説明：

`date`**: String** 型の日付値です。 形式には少なくとも「**yyyy-mm-dd**」を含める必要があり、追加の文字列を含めることはできません。 それ以外の場合は、**NULL** が返されます。

戻り値**:**

Int 型 。

例：

```
year('1970-01-01 12:30:00') = 1970
year('1970-01-01') = 1970
year('70-01-01') = 70
year(1970-01-01) = null
year('1970/03/09') = null
```

```
year(null) Returns an exception
```

QUARTER

コマンド形式:

```
INT quarter(datetime/timestamp/string date )
```

 注:

新しいデータ型を使用するには、**QUARTER** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の四半期で、**1**～**4** が返されます。

パラメーターの説明：

`date`: **Datetime** 型、**Timestamp** 型、**String** 型のいずれかの日付値です。 形式には少なくとも「**yyyy-mm-dd**」を含める必要があります。 それ以外の場合は、**NULL** が返されます。

戻り値:

Int 型。入力値が **NULL** の場合は、**NULL** が返されます。

例：

```
quarter('1970-11-12 10:00:00') = 4
quarter('1970-11-12') = 4
```

MONTH

コマンド形式:

```
INT month(string date)
```

 注:

新しいデータ型を使用するには、**MONTH** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付が属する月を返します。

パラメーターの説明：

`date`: **String** 型の日付値です。 それ以外のデータ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例：

```
month('2014-09-01') = 9
month('20140901') = null
```

DAY

コマンド形式:

```
INT day(string date)
```



注：

新しいデータ型を使用するには、**DAY** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の曜日を返します。

パラメーターの説明：

`date`: **String** 型の日付値です。 それ以外の データ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例：

```
day('2014-09-01') = 1
day('20140901') = null
```

DAYOFMONTH

コマンド形式:

```
INT dayofmonth(date)
```



注：

新しいデータ型を使用するには、**DAYOFMONTH** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の日の部分を返します。

たとえば、コマンド `int dayofmonth(2017-10-13)` を実行すると、**13** が返されます。

パラメーターの説明：

`date`: **String** 型の日付値です。 それ以外のデータ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例：

```
dayofmonth('2014-09-01') = 1
dayofmonth('20140901') = null
```

HOUR

コマンド形式:

```
INT hour(string date)
```

 注：

新しいデータ型を使用するには、**HOUR** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の時間を返します。

パラメーターの説明：

`date`: **String** 型の日付値です。 それ以外のデータ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例：

```
hour('2014-09-01 12:00:00')=12
hour('12:00:00')=12
```

```
hour('20140901120000')=null
```

MINUTE

コマンド形式:

```
INT minute(string date)
```

 注:

新しいデータ型を使用するには、**MINUTE** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の分の部分を返します。

パラメーターの説明:

`date`: **String** 型の日付値です。 それ以外のデータ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例:

```
minute('2014-09-01 12:30:00') = 30
minute('12:30:00') = 30
minute('20140901120000') = null
```

SECOND

コマンド形式:

```
INT second(string date)
```

 注:

新しいデータ型を使用するには、**SECOND** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

日付の秒の部分を返します。

パラメーターの説明:

`date`: **String** 型の日付値です。 それ以外のデータ型の値の場合は、例外が返されます。

戻り値:

Int 型。

例：

```
second('2014-09-01 12:30:45') = 45
second('12:30:45') = 45
second('20140901123045') = null
```

CURRENT_TIMESTAMP

コマンド形式:

```
timestamp current_timestamp()
```

 注：

新しいデータ型を使用するには、**CURRENT_TIMESTAMP** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

現在のタイムスタンプを **Timestamp** 型の値として返します。 この値は固定値ではありません。

戻り値:

Timestamp 型。

例：

```
select current_timestamp() from dual;--Returns '2017-08-03 11:50:30.
661'
```

ADD_MONTHS

コマンド形式:

```
string add_months(string startdate, int nummonths)
```

 注：

新しいデータ型を使用するには、**ADD_MONTHS** 関数を指定する **SQL** 文の前に、 `set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

「**startdate**」に「**nummonths**」の値を加えた日付を返します。

パラメーターの説明：

- `startdate`: **String** 型の値です。 形式には少なくとも「**yyyy-mm-dd**」を含める必要があり それ以外の場合は、**NULL** が返されます。
- `num_months`: **Int** 型の値です。

戻り値:

String 型の日付が「**yyyy-mm-dd**」の形式で返されます。

例：

```
Add_months ('2017-02-14', 3) = '2017-05-14'
add_months('17-2-14',3) = '0017-05-14'
add_months('2017-02-14 21:30:00',3) = '2017-05-14'
add_months('20170214',3) = null
```

LAST_DAY

コマンド形式:

```
string last_day(string date)
```

 注：
新しいデータ型を使用するには、**LAST_DAY** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

指定した日付を含む月の最後の日付を返します。

パラメーターの説明：

`date`: **String** 型で、形式は「**yyyy-MM-dd HH:mi:ss**」または「**yyyy-MM-dd**」です。

戻り値:

String 型の日付が「**yyyy-mm-dd**」の形式で返されます。

例：

```
last_day('2017-03-04') = '2017-03-31'
last_day('2017-07-04 11:40:00') = '2017-07-31'
```

```
last_day('20170304') = null
```

NEXT_DAY

コマンド形式:

```
string next_day(string startdate, string week)
```


注：
新しいデータ型を使用するには、**NEXT_DAY** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

startdate パラメーターで指定した 日付以降で、**week** パラメーターで指定した曜日に一致する最初の日付を返します。 つまり、翌週の特定の曜日の日付です。

パラメーターの説明：

- `startdate`: **String** 型で、形式は「**yyyy-MM-dd HH:mi:ss**」または「**yyyy-MM-dd**」です。
- `week`: **String** 型で、曜日の最初の **2** 文字または **3** 文字、または曜日のフルネームです。 たとえば、**Mo**、**TUE**、**FRIDAY** です。

戻り値:

String 型の日付が「**yyyy-mm-dd**」の形式で返されます。

例：

```
next_day('2017-08-01','TU') = '2017-08-08'
next_day('2017-08-01 23:34:00','TU') = '2017-08-08'
Next_day ('20170801 ', 'tu') = NULL
```

MONTHS_BETWEEN

コマンド形式:

```
double months_between(datetime/timestamp/string date1, datetime/
timestamp/string date2)
```


注：
新しいデータ型を使用するには、**MONTHS_BETWEEN** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明:

date1と **date2** の間の月数を返します。

パラメーターの説明：

- `date1`**: Datetime** 型、**Timestamp** 型、**String** 型のいずれかで、形式は「**yyyy-MM-dd HH:mi:ss**」または「**yyyy-MM-dd**」です。
- `date2`**: Datetime** 型、**Timestamp** 型、**String** 型のいずれかで、形式は「**yyyy-MM-dd HH:mi:ss**」または「**yyyy-MM-dd**」です。

戻り値**:**

Double 型。

- **date1** が **date2** より後の場合、戻り値は正の数になります。 **date2** が **date1** より後の場合、戻り値は負の数になります。
- **date1** と **date2** が 、**2** か月の最後の日の場合、戻り値は月数を表す整数です。 それ以外の場合の式は、**(date1 - date2)/31** です。

例：

```
months_between('1997-02-28 10:30:00', '1996-10-30') = 3.9495967741
935485
months_between('1996-10-30','1997-02-28 10:30:00' ) = -3.9495967741
935485
months_between('1996-09-30','1996-12-31') = -3.0
```

5.12.2 算術関数

ここでは、**MaxCompute SQL** で使用できる算術関数コマンドと実行方法について説明します。

ABS

関数の定義**:**

```
Double abs(Double number)
Bigint abs(Bigint number)
Decimal abs(Decimal number)
```

使用法：

絶対値を返します。

パラメーターの説明：

`number`**: Double** 型、**Bigint** 型、**Decimal** 型の任意の数字です。

- 入力値が **Bigint** 型の場合は、**Bigint** 型が返されます。
- 入力値が **Double** 型の場合は、**Double** 型が返されます。
- 入力値が **Decimal** 型の場合は、**Decimal** 型が返されます。

入力値が **String** 型の場合は、暗黙の型変換で **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

返される結果は、入力パラメーター値のデータ型によって異なります。 たとえば、入力パラメーター値が **NULL** の場合は **NULL** が返されます。

 注**:**

Bigint 型の入力値が **Bigint** 型の最大値を超える場合は、**Double** 型が返されます。 この場合、精度が欠けている可能性があります。

例：

```
abs(null) = null
abs(-1) = 1
abs(-1.2) = 1.2
abs("-2") = 2.0
abs(12232083745629837659238745692374 8) = 1.2232083745629837e32
```

以下は、**SQL**で **ABS** 関数を使用する場合の例です。 他の組み込み関数 (ウィンドウ関数と集約関数を除く) の使用方法と似ています。

```
select abs(id) from tbl1;
-- tbl1 の id フィールド絶対値を取り出します。
```

ACOS

関数の定義**:**

```
Double acos(Double number)
Decimal acos(Decimal number)
```

使用法：

数値の逆余弦を計算します。

パラメーターの説明：

number**: Double** 型または **Decima** 型で、 **-1<=number <=1**です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合、エラーが発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。値は **0** から **π** の間です。 数値が **NULL** の場合は、**NULL** が返されます。

例：

```
acos("0.87") = 0.5155940062460905
acos(0) = 1.5707963267948966
```

ASIN

関数の定義:

```
Double asin(Double number)
Decimal asin(Decimal number)
```

使用法：

数値の逆正弦関数を計算します。

パラメーターの説明：

number: **Double** 型または **Decima** 型で、**-1<=number <=1** です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。値は **-π/2** から **π/2** の間です。 数値が **NULL** の場合は、**NULL** が返されます。

例：

```
asin(1) = 1.5707963267948966
asin(-1) = -1.5707963267948966
```

ATAN

関数の定義:

```
Double atan(Double number)
```

使用法：

数値のバックカット関数を計算します。

パラメーターの説明：

number: **Double** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力パラメーター値がそれ以外のデータ型の場合、エラーが発生します。

戻り値:

Double 型が返されます。値は **-π/2** から **π/2** の間です。 数値が **NULL** の場合は、**NULL** が返されます。

例：

```
atan(1) = 0.7853981633974483
atan(-1) = -0.7853981633974483
```

CEIL

関数の定義**:**

```
Bigint ceil(Double value)
Bigint ceil(Decimal value)
```

使用法：

この関数は、引数以上で最小の整数値を返します。

パラメーターの説明：

`value`**: Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合、エラーが発生します。

戻り値**:**

Bigint 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

例：

```
ceil(1.1) = 2
ceil(-1.1) = -1
```

CONV

関数の定義**:**

```
String conv(String input, Bigint from_base, Bigint to_base)
```

使用法：

数値を **16** 進数に変換します。

パラメーターの説明：

- `input`**:** 変換対象の整数で、 **String** 型で表します。 **Bigint** 型および **Double** 型の場合、暗黙の変換で **String** 型に変換されます。

- from_base, to_base: **Decimal** 値です。使用できる値は、**2**、**8**、**10**、**16** です。 **Bigint** 型および **Double** 型の場合、暗黙の変換で **String** 型に変換されます。

戻り値:

String 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。 変換プロセスは、**64** ビット精度で実行されます。 オーバーフローが発生すると、例外が発生します。 入力値が負の値 **(**「-」で始まる**)** の場合、例外が発生します。 入力値が **10** 進数の場合は、**16** 進数変換の前に整数に変換されます。 小数部は除外されます。

例：

```
conv('1100', 2, 10) = '12'
conv('1100', 2, 16) = 'c'
conv('ab', 16, 10) = '171'
conv('ab', 16, 16) = 'ab'
```

COS

関数の定義:

```
Double cos(Double number)
Decimal cos(Decimal number)
```

使用法:

入力はラジアン値です。

パラメーターの説明：

number: **Double** 型または **Decimal** 型です。 入力値が **String** 型の場合は、暗黙の型変換で **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

例：

```
cos(3.1415926/2)=2.6794896585028633e-8
cos(3.1415926)=-0.9999999999999986
```

COSH

関数の定義:

```
Double cosh(Double number)
Decimal cosh(Decimal number)
```

使用法：

双曲線余弦関数です

パラメーターの説明：

number**: Double** 型または **Decimal** 型です。 入力値が **String** 型の場合は、暗黙の型変換で **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

COT

関数の定義**:**

```
Double cot(Double number)
Decimal cot(Decimal number)
```

使用法：

ラジアン値を入力します。

パラメーターの説明：

number**: Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

EXP

関数の定義**:**

```
Double exp(Double number)
Decimal exp(Decimal number)
```

使用法：

指数関数です。

戻り値**:**

数値の絶対値が返されます。

パラメーターの説明：

number**: Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

FLOOR

関数の定義**:**

```
Bigint floor(Double number)
Bigint floor(Decimal number)
```

使用法：

引数以下で最大の整数値が返されます。

パラメーターの説明：

number**: Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

Bigint 型が返されます。入力値が **NULL** の場合は **NULL** が返されます。

例：

```
floor(1.2)=1
floor(1.9)=1
floor(0.1)=0
floor(-1.2)=-2
floor(-0.1)=-1
floor(0.0)=0
Floor (-0.0) = 0
```

LN

関数の定義**:**

```
Double ln(Double number)
Decimal ln(Decimal number)
```

使用法：

数値の自然対数が返されます。

パラメーターの説明：

number: **Double** 型または **Decimal** 型です。

- 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。
- 数値が **NULL** の場合は、**NULL** が返されます。 **number** が負または **0** の場合、例外が発生します。

戻り値:

Double 型または **Decimal** 型が返されます。

LOG

関数の定義:

```
Double log(Double base, Double x)
Decimal log (decimal base, decimal X)
```

使用法：

基数が **base** の **x** の対数を返します。

パラメーターの説明：

- number: **Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。
- x: **Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型の対数値が返されます。

- **base** または **x** が **NULL** の場合、**NULL** が返されます。
- **base** または **x** の一方が負またはゼロの場合、異常の原因となります。
- **base** が **1** の場合も、異常の原因になります。

POW

関数の定義:

```
Double pow(Double x, Double y)
Decimal pow(Decimal x, Decimal y)
```

使用法：

x の **y** 乗、つまり **x ^ y** が返されます。

パラメーターの説明：

- X**: Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。
- Y**: Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。 **X** または **Y** が **NULL** の場合、**NULL** が返されます。

RAND

関数の定義**:**

```
Double rand(Bigint seed)
```

使用法：

乱数 (行ごとに異なる) が返されます。シードを指定すると、生成された乱数列が確定的であることが確認されます。戻り値の範囲は **0** から **1** です。

パラメーターの説明：

seed**: Bigint** 型です。乱数列のシードにより、乱数列の開始値が決定されます。

戻り値**:**

Double 型が返されます。

例：

```
select rand() from dual;
select rand(1) from dual;
```

ROUND

関数の定義**:**

```
Double round(Double number, [Bigint Decimal_places])
Decimal round(Decimal number, [Bigint Decimal_places])
```

使用法：

特定の小数点位置に四捨五入します。

パラメーターの説明：

- number: **Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。
- Decimal_place: **Bigint** 型の定数で、四捨五入する小数点からの位置です。 それ以外のデータ型の場合は、例外が発生します。このパラメータを省略すると、**1** の位に四捨五入されます。 デフォルト値は **0** です。

戻り値:

Double 型または **Decimal** 型が返されます。number または Decimal_places が **NULL** の場合、**NULL** が返されます。

> 注：
>
> Decimal_places は負になることがあります。 負数の場合、小数点から左へ数えられます。 小数部は削除されます。 **decimal_place** が整数部分より長い場合、**0** が返されます。

例：

```
round(125.315) = 125.0
round(125.315, 0) = 125.0
Round (125.315, 1) = 125.3
round(125.315, 2) = 125.32
round(125.315, 3) = 125.315
round(-125.315, 2) = -125.32
round(123.345, -2) = 100.0
round(null) = null
round(123.345, 4) = 123.345
round(123.345, -4) = 0.0
```

SIN

関数の定義:

```
Double sin(Double number)
Decimal sin(Decimal number)
```

使用法：

数値の正弦関数を計算します。入力はラジアン値です。

パラメーターの説明：

number: **Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

SINH

関数の定義:

```
Double sinh(Double number)
Decimal sinh(Decimal number)
```

使用法:

数値の逆正弦関数を計算します。

パラメーターの説明：

number: **Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

SQRT

関数の定義:

```
Double sqrt(Double number)
Decimal sqrt(Decimal number)
```

使用法：

数値の平方根を計算します。

パラメーターの説明：

number: **Double** 型または **Decimal** 型で、ゼロより大きくなければなりません。ゼロ未満の場合、例外が発生します。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

TAN

関数の定義:

```
Double tan(Double number)
Decimal tan(Decimal number)
```

使用法:

数値の正接関数を計算します。入力はラジアン値です。

パラメーターの説明：

number: **Double** 型または**r Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

TANH

関数の定義:

```
Double tanh(Double number)
Decimal tanh(Decimal number)
```

使用法:

数値の双曲線正接関数を計算します。

パラメーターの説明：

number: **Double** 型または **Decimal** 型です。入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。

戻り値:

Double 型または **Decimal** 型が返されます。 数値が **NULL** の場合は、**NULL** が返されます。

TRUNC

関数の定義:

```
Double trunc(Double number[, Bigint Decimal_places])
Decimal trunc(Decimal number[, Bigint Decimal_places])
```

使用法：

この関数は、指定した小数の位まで、入力値を切り捨てる際に使用します。

パラメーターの説明：

- `number`: **Double** 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されます。 入力値がそれ以外のデータ型の場合は、エラーが発生します。
- `Decimal_places`: **Bigint** 型の定数で、数値を切り捨てる小数の位です。 その他のデータ型は **Bigint** 型に変換されます。 このパラメーターを省略すると、小数第 **1** 位が切り捨てられます。

戻り値:

Double 型または **Decimal** 型が返されます。 `number` または `Decimal_places` が **NULL** の場合、 **NULL** が返されます。

 注：

- **Double** 型が返される場合、返される結果の表示は、**trunc(125.815, 1)** のように予想通りにならない可能性があります (この問題はすべてのシステムに存在します)。
- 切り捨てられる部分はゼロが補われます。
- **Decimal_places** は、負にすることもできます。 負の場合は、小数点から左に数えられ、そこから小数点部分までが切り捨てられます。 **Decimal_place** が整数部分より長い場合、**0** が返されます。

例：

```
trunc(125.815) = 125.0
trunc(125.815, 0) =125.0
trunc(125.815, 1) = 125.80000000000001
trunc(125.815, 2) = 125.81
trunc(125.815, 3) = 125.815
trunc(-125.815, 2) = -125.81
trunc(125.815, -1) = 120.0
trunc(125.815, -2) = 100.0
trunc(125.815, -3) = 0.0
trunc(123.345, 4) = 123.345
```

```
trunc(123.345, -4) = 0.0
```

Maxcomputerte2.0 の新規算術関数

MaxCompute 2.0 へのアップグレードに伴い、算術関数がいくつか追加されました。 新規関数で新規データ型を使用する場合は、新規関数の **SQL** 文を使用する前に、次の **set** 文を追加します。

```
set odps.sql.type.system.odps2=true;
```

新しい拡張機能は次のとおりです。

LOG2

関数の定義**:**

```
Double log2(Double number)
Double log2(Decimal number)
```

 注:

新規のデータ型関数を使用するには、**LOG2** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

特定の数の **2** を底とする対数を返します。

パラメーターの説明：

`number`**: Double** 型または **Decimal** 型。

戻り値**:**

Double 型。 入力値がゼロまたは **NULL** の場合、戻り値は **NULL** になります。

以下に例を示します。

```
log2(null)=null
log2(0)=null
log2(8)=3.0
```

LOG10

関数の定義**:**

```
Double log10(Double number)
Double log10(Decimal number)
```

 注:

新規のデータ型関数を使用するには、**LOG10** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

特定の数の **10** を底とする対数を返します。

パラメーターの説明：

`number`**: Double** 型または **Decimal** 型。

戻り値**:**

Double 型が返されます。 入力値がゼロまたは **NULL** の場合、戻り値は **NULL** になります。

以下に例を示します。

```
log10(null)=null
log10(0)=null
log10(8)=0.9030899869919435
```

BIN

関数の定義**:**

```
String bin(Bigint number)
```



注：

新規のデータ型関数を使用するには、**BIN** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

特定の数値のバイナリコード式を返します。

パラメーターの説明：

`number`**: Bigint** 型。

戻り値**:**

String 型。 入力値がゼロの場合、ゼロが返されます。入力値が **NULL** の場合、**NULL** が返されます。

例：

```
bin(0)='0'
bin(null)='null'
```

```
bin(12)='1100'
```

HEX

関数の定義:

```
String hex(Bigint number)
String hex(String number)
String hex (binary number)
```



注:

新規のデータ型関数を使用するには、**HEX** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法:

この関数は、整数または文字を **16** 進形式に変換するために使用します。

パラメーターの説明:

`number`: **number** が **Bigint** 型の場合、数値の **16** 進形式が返されます。 この変数が **String** 型の場合、文字列の **16** 進形式が返されます。

戻り値:

String 型が返されます。 入力値がゼロの場合、ゼロが返されます。入力値が **NULL** の場合、例外が返されます。

例:

```
hex(0)=0
hex('abc')='616263'
hex(17)='11'
hex('17')='3137'
hex(null) results in an exception and returns failed.
```



注:

入力パラメータ値が **Binary** 型の場合、 新しいデータ型を使用するには、`set odps.sql.type.system.odps2=true;`を追加して、**SQL** 文と一緒に送信します。

UNHEX

関数の定義:

```
BINARY unhex(String number)
```



注:

新規のデータ型関数を使用するには、**UNHEX** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

指定した **16** 進数文字列で表される文字列を返します。

パラメーターの説明：

`number`**: 16** 進数の文字列です。

戻り値**:**

Binary 型が返されます。 入力値がゼロの場合、失敗が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

例：

```
Unhex ('616263') = 'abc'
unhex(616263)='abc'
```

RADIANS

関数の定義**:**

```
Double radians(Double number)
```


注：
新規のデータ型関数を使用するには、**RADIANS** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数は、度をラジアンに変換するために使用します。

パラメーターの説明：

`number`**: Double** 型。

戻り値**:**

Double 型が返されます。入力値が **NULL** の場合、**NULL** が返されます。

例：

```
radians(90)=1.5707963267948966
radians(0)=0.0
```

```
radians(null)=null
```

DEGREES

関数の定義:

```
Double degrees(Double number)
Double degrees(Decimal number)
```

 注:

新規のデータ型関数を使用するには、**DEGREES** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法:

この関数はラジアンを度に変換するために使用します。

パラメーターの説明:

`number`: **Double** 型または **Decimal** 型。

戻り値:

Double データ型が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

例:

```
degrees(1.5707963267948966)=90.0
degrees(0)=0.0
Degrees (null) = NULL
```

SIGN

関数の定義:

```
Double sign(Double number)
Double sign(Decimal number)
```

 注:

新規のデータ型関数を使用するには、**SIGN** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法:

入力データの符号を適用します。 **1.0** は正数、**-1.0** は負数を表します。 それ以外の場合は、 **0.0** が返されます。

パラメーターの説明:

number: **Double** 型または **Decimal** 型。

戻り値:

Double データ型が返されます。 入力値が **0** の場合、**0.0** が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

例：

```
sign(-2.5)=-1.0
Sign (2.5) = 1.0
sign(0)=0.0
sign(null)=null
```

E

関数の定義:

```
Double e()
```

 注：
新規のデータ型関数を使用するには、**E** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数は **e** 値を返すために使用します。

戻り値:

Double 型が返されます。

例：

```
e()=2.718281828459045
```

PI

関数の定義:

```
Double pi()
```

 注：
新規のデータ型関数を使用するには、**PI** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数はπ値を返すために使用します。

戻り値:

Double 型が返されます。

例：

```
pi()=3.141592653589793
```

FACTORIAL

関数の定義:

```
Bigint factorial(Int number)
```


注：
新規のデータ型関数を使用するには、**FACTORIAL** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数は、特定の数の階乗を返すために使用します。

パラメーターの説明：

`number`: **Int** 型のデータで、データの範囲は **0**～**20** です。

戻り値:

Bigint 型が返されます。入力値がゼロの場合、**1** が返されます。 入力値が **NULL** または **0**～**20** の範囲外の場合、**NULL** が返されます。

例：

```
factorial(5)=120 --5! = 5*4*3*2*1 = 120
```

CBRT

関数の定義:

```
Double cbrt(Double number)
```


注：
新規のデータ型関数を使用するには、**CBRT** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数は、立方根を返すために使用します。

パラメーターの説明：

number: **Double** 型。

戻り値:

Double データ型が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

例：

```
cbrt(8)=2
cbrt(null)=null
```

SHIFTLEFT

関数の定義:

```
Int shiftleft(Tinyint|Smallint|Int number1, Int number2)
Bigint shiftleft(Bigint number1, Int number2)
```

 注：

新規のデータ型関数を使用するには、**SHIFTLEFT** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

指定した桁数分だけ左にシフトします。

パラメーターの説明：

- number1: **Tinyint** 型、**Smallint** 型、**Int** 型、**Bigint** 型の整数。
- number2: **Int** 型の整数。

戻り値:

Int 型または **Bingint** 型が返されます。

例：

```
shiftleft(1,2)=4  --バイナリ値 1 を左に 2 桁分シフトします (1<<2,0001 を
0100 にシフト)
shiftleft(4,3)=32  --バイナリ値 4 を左に 3 桁分シフトします (4<<3,0100 を 10
,0000 にシフト)
```

SHIFTRIGHT

関数の定義:

```
Int shiftright(Tinyint|Smallint|Int number1, Int number2)
```

```
Bigint shiftright(Bigint number1, Int number2)
```

 注：

新規のデータ型関数を使用するには、**SHIFTRIGHT** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

使用法：

この関数は、指定した桁数だけ右にシフトするために使用します。

パラメーターの説明：

- `number1`: **Tinyint** 型、**Smallint** 型、**Int** 型、**Bigint** 型の整数。
- `number2`: **Int** 型の整数。

戻り値:

Int 型または **Bigint** 型が返されます。

例：

```
shiftright(4,2)=1 -- バイナリ値 4 を右に 2 桁分シフトします (4>>2,0100 を
0001 にシフト)
shiftright(32,3)=4 -- バイナリ値 32 を右に 3 桁分シフトします (32>>3,100000
 を 0100 にシフト)
```

SHIFTRIGHTUNSIGNED

コマンド形式は次のとおりです。

```
Int shiftrightunsigned(Tinyint|Smallint|Int number1, Int number2)
Bigint shiftrightunsigned(Bigint number1, Int number2)
```

 注：

新規のデータ型関数を使用するには、**SHIFTRIGHTUNSIGNED** 関数を指定する **SQL** 文の前に、`set odps.sql.type.system.odps2=true;` を追加します。

コマンドの説明は次のとおりです。

この関数は、指定した桁数だけ、符号を付けずに右にシフトするために使用します。

パラメーターの説明：

- `number1`: **Tinyint** 型、**Smallint** 型、**Int** 型、**Bigint** 型の整数。
- `number2`: **Int** 型の整数。

戻り値:

Int 型または **Bigint** 型が返されます。

例：

```
shiftrightunsigned(8,2)=2 -- 符号なしバイナリ値 8 を右に 2 桁分シフトします（
8>>>2,1000 を 0010 にシフト）
shiftrightunsigned(-14,2)=1073741820  -- 符号なしバイナリ値 -14 を右に
 2 桁分シフトします（-14>>>2, 11111111 11111111 11111111 11110010 を
00111111 11111111 11111111 11111100 にシフト）
```

5.12.3 ウィンドウ関数

MaxCompute SQLでは、ウィンドウ関数を使用して、ワークフローを柔軟に分析および処理できます。 ウィンドウ関数は **'select'** 句でのみ指定できます。 ただし、ウィンドウ 関数では、ネストしたウィンドウ関数と集計関数を使用できません。 また、集計関数と同レベルで一緒に使用することもできません。

現在、**MaxCompute SQL** 文では、**5** 種類のウィンドウ関数を使用できます。

ウィンドウ関数の構文**:**

```
window_func() over (partition by [col1,col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] windowing_clause)
```

- partition by では、ウィンドウ関数の対象となる列を指定します。 列が区切られ、同じ値を持つ行が同じウィンドウ関数で処理されます。 現在、ウィンドウ関数では、最大 **100,000,000** 行のデータを指定できます。 **5,000,000** 行を超えないようにすることを推奨します。超えると、実行時にエラーが発生します。
- order by 句では、ウィンドウ関数でデータを並び替える方法を指定します。
- windowing_clause 部分には、**rows** メソッドを使用して、ウィンドウ関数の対象範囲を指定します。 **2** 通りの方法を以下に示します。
  - **Rows between x preceding|following and y preceding|following** は、ウィンドウ関数の範囲が、**x** 行前または **x** 行後から **y** 行前または **y** 行後までであることを示します。
  - **Rows x preceding|following:** ウィンドウ関数の範囲は、現在の行から **x** 行前または **x** 行後までになります。
  - **x**、**y** は **0** 以上の整数定数で、対応する値の範囲は **0**～**10000** です。 値が **0** の場合、現在の行を示します。 **order by** 句を指定した場合は、**rows** メソッドを使用してウィンドウ関数の範囲を指定します。

 注：

すべてのウィンドウ関数で、**rows** メソッドを使用してウィンドウ関数の対象範囲を指定できるわけではありません。 ウィンドウ関数では、**AVG**、**count**、**Max**、**min**、**StdDev**、**sum** などを使用できます。

COUNT

関数の定義:

```
Bigint count([distinct] expr) over(partition by [col1, col2…]
 [order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

取得した行の総数を計算します。

パラメーターの説明:

- `expr`: 任意のデータ型です。 **NULL** の行はカウントされません。 **distinct** キーワードが指定されている場合は、一意のカウント値が使用されます。
- `partition by [col1, col2…]`: ウィンドウ関数で使用する列を指定します。
- `order by col1[asc|desc], col2[asc|desc]`: **order by** 句を指定していない場合、現在のウィンドウの **expr** のカウント値が返されます。 **order by** 句を指定すると、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在行までの間のカウント値が返されます。

戻り値:

Bigint 型。

 注:
キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

例:

テーブル **test_src** は既に存在し、このテーブルには **Bigint** 型の列 **user_id** があるとします。

```
select user_id,
        count(user_id) over (partition by user_id) as count
    from test_src;

    | user_id | count |

    | 1 | 3 |
    | 1 | 3 |
    | 1 | 3 |
    | 2 | 1 |
    | 3 | 1 |

    -- order by 句が指定されていません。現在のウィンドウのカウント値を戻します。
```

```
    select user_id,
        count(user_id) over (partition by user_id order by user_id) as
 count
    from test_src;

    | user_id | count |

    | 1 | 1 | -- ウィンドウの開始行
    | 1 | 2 | --開始行から現在の行までに 2 レコード存在します。 2 を戻します。
    | 1 | 3 |
    | 2 | 1 |
    | 3 | 1 |

    -- order by 句が指定されていないので、現在のウィンドウの開始行から現在の行ま
での累積カウント値を戻します。
```

AVG

関数の定義:

```
avg([distinct] expr) over(partition by [col1, col2…]
 [order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

平均を計算します。

パラメーターの説明:

- `distinct`: キーワード **distinct** が指定されている場合、一意の値の平均が計算されます。
- `expr`: **Double** 型。
  - 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換によって **Double** 型に変換されてから、演算に使用されます。 別のデータ型の場合は、例外が発生します。
  - 入力値が **NULL** の場合、その行は計算に使用されません。
  - データ型が **Boolean** 型の場合、その行は計算から除外されます。
- `partition by [col1, col2…]`: ウィンドウ関数で使用する列を指定します。
- `order by col1[asc|desc], col2[asc|desc]`: **order by** 句を指定していない場合、現在のウィンドウのすべての値の平均が返されます。 **order by** 句を指定すると、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの間の累加平均が返されます。

戻り値:

Double 型。

 注:

キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

MAX

関数の定義:

```
max([distinct] expr) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

最大値を計算します。

パラメーターの説明:

- expr: **Boolean** 型以外の任意のデータ型です。 入力値が **NULL** の場合、その行は計算に使用されません。 キーワード **distinct** が指定されている場合は、一意の値の最大値が求められます。
- partition by [col1, col2…]: ウィンドウ関数を使用する列を指定します。
- order by col1[asc|desc], col2[asc|desc]: **order by** 句を指定していない場合、現在のウィンドウの最大値が返されます。 **order by** 句を指定すると、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの間の最大値が返されます。

戻り値:

expr のデータ型と同じになります。

 注:
キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

MIN

関数の定義:

```
min([distinct] expr) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

列の最小値を計算します。

パラメーターの説明:

- expr **Boolean** 型 以外の任意のデータ型です。 パラメーター値が **NULL** の場合、この行は計算に使用されません。 キーワード **distinct** が指定されている場合は、一意の最小値を使用することを示します。
- partition by [col1, col2…]: ウィンドウ関数で使用する列を指定します。

- `order by col1[asc|desc], col2[asc|desc]`: **order by** 句を指定していない場合、現在のウィンドウの最小値が返されます。 **order by** 句を指定している場合、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの間の最小値が返されます。

戻り値:

expr と同じデータ型になります。



注:

キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

MEDIAN

関数の定義:

```
Double median(Double number1,number2...) over(partition by [col1, col2
…])
Decimal median(Decimal number1,number2...) over(partition by [col1,
col2…])
```

使用法:

中央値を計算します。

パラメーターの説明:

- `number1,number1…`: **1 to 255 digits of a Double or Decimal type.**
  - 入力値が **String** 型または **Bigint** 型の場合、暗黙の変換で **Double** 型になった後に演算が実行されます。それ以外の型の場合は、例外が発生します。
  - 入力値が **NULL** の場合は、**NULL** が返されます。
  - 入力値が **Double** 型の場合、デフォルトで **Double** 型の配列に変換されます。
- `partition by [col1, col2…]`: ウィンドウ関数で使用する列を指定します。

戻り値:

Double 型。

STDDEV

関数の定義:

```
Double stddev([distinct] expr) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
Decimal stddev([distinct] expr) over(partition by [col1, col2…]
```

```
[order by [col1[asc|desc], col2[asc|desc…]] [windowing_clause])
```

使用法:

母集団の標準偏差を計算します。

パラメーターの説明:

- expr: Double 型です。
  - 入力値が String 型または Bigint 型の場合、Double 型に変換されてから、演算で使用されます。 別のデータ型の場合は、例外が発生します。
  - 入力値が NULL の場合、この行は除外されます。
  - キーワード distinct が指定されている場合、一意の値の母集団標準偏差が計算されます。
- Partition by [col1, col2..]: ウィンドウ関数で使用する列を指定します。
- order by col1[asc|desc], col2[asc|desc]: order by 句を指定していない場合、現在のウィンドウの母集団標準偏差が返されます。 order by 句を指定している場合、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの間の母集団標準偏差が返されます。

戻り値:

入力値が Decimal 型の場合は、Decimal 型が返されます。そうでなければ、Double 型が返されます。

例:

```
select window, seq, stddev_pop('1\01') over (partition by window order
 by seq) from dual;
```



注:

- キーワード distinct が指定されている場合、order by 句は使用できません。
- Stddev_pop stddev 関数の別名です。使用方法は stddev と同じです。

STDDEV_SAMP

関数の定義:

```
Double stddev_samp([distinct] expr) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
Decimal stddev_samp([distinct] expr) over((partition by [col1,col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

標準偏差を計算します。

パラメーターの説明:

- `Expr`: **Double** 型です。
  - 入力値が **String** 型または **Bigint** 型の場合、**Double** 型に変換されてから、演算で使用されます。 別のデータ型の場合は、例外が報告されます。
  - 入力値が **NULL** の場合、この行は除外されます。
  - キーワード **distinct** が指定されている場合、一意の値の標準偏差が計算されます。
- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。
- `Order by col1[asc|desc], col2[asc|desc]`:**order by** 句を指定していない場合、現在のウィンドウのサンプル標準偏差が返されます。 **order by** 句を指定している場合、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの間のサンプル標準偏差が返されます。

戻り値:

入力値が **Decimal** 型の場合は、**Decimal** 型が返されます。そうでなければ、**Double** 型が返されます。

 注:
キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

SUM

関数の定義:

```
sum([distinct] expr) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause])
```

使用法:

要素の合計を計算します。

パラメーターの説明:

- `Expr`: **Double** 型。
  - 入力値が **String** 型または **Bigint** 型の場合、**Double** 型に変換されてから、演算で使用されます。 別のデータ型の場合は、例外が報告されます。
  - 入力値が **NULL** の場合、この行は除外されます。
  - キーワード **distinct** が指定されている場合、一意の値の合計が計算されます。
- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。

- `Order by col1[asc|desc], col2[asc|desc]`:「**order by**」句を指定していない場合、現在のウィンドウの合計が返されます。**order by** 句を指定している場合、返される結果は指定した順序に従って並べ替えられ、現在のウィンドウの開始行から現在の行までの合計が返されます。

戻り値:

- 入力パラメーター値が **Bigint** 型の場合は、**Bigint** 型が返されます。
- 入力パラメーター値が **Decimal** 型の場合は、**Decimal** 型が返されます。
- 入力パラメーター値が **Double** 型または **String** 型の場合は、**Double** 型が返されます。

 注:

キーワード **distinct** が指定されている場合、**order by** 句は使用できません。

DENSE_RANK

関数の定義:

```
Bigint dense_rank() over(partition by [col1, col2…]
order by [col1[asc|desc], col2[asc|desc]…])
```

使用法:

密集ランクを計算します。**col2** と同じ行にあるデータは、同じランクになります。

パラメーターの説明:

- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。
- `order by col1[asc|desc], col2[asc|desc]`: ランクの基準となる値を指定します。

戻り値:

Bigint 型。

例:

テーブル **emp** のデータは、次のとおりです。

```
| empno | ename | job | mgr | hiredate| sal| comm | deptno |
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600,300,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250,500,30
7566,JONES,MANAGER,7839,1981-04-02 00:00:00,2975,,20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250,1400,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450,,10
7788,SCOTT,ANALYST,7566,1987-04-19 00:00:00,3000,,20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500,0,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100,,20
```

```
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300,,10
7948,JACCKA,CLERK,7782,1981-04-12 00:00:00,5000,,10
7956,WELAN,CLERK,7649,1982-07-20 00:00:00,2450,,10
7956,TEBAGE,CLERK,7748,1982-12-30 00:00:00,1300,,10
```

すべての従業員を部門別にグループ化し、各グループを **SAL** に従って降順にソートして、グループ独自のシリアル番号を取得する必要があるとします。

```
SELECT deptno
        , ename
        , sal
        , DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC
) AS nums--Deptno as a window column, and sort in descending order
according to sal.
    FROM emp;
--結果は次のとおりです。

| deptno | ename | sal | nums |

| 10 | JACCKA | 5000.0 | 1 |
| 10 | KING | 5000.0 | 1 |
| 10 | CLARK | 2450.0 | 2 |
| 10 | WELAN | 2450.0 | 2 |
| 10 | TEBAGE | 1300.0 | 3 |
| 10 | MILLER | 1300.0 | 3 |
| 20 | SCOTT | 3000.0 | 1 |
| 20 | FORD | 3000.0 | 1 |
| 20 | JONES | 2975.0 | 2 |
| 20 | ADAMS | 1100.0 | 3 |
| 20 | SMITH | 800.0 | 4 |
| 30 | BLAKE | 2850.0 | 1 |
| 30 | ALLEN | 1600.0 | 2 |
| 30 | TURNER | 1500.0 | 3 |
| 30 | MARTIN | 1250.0 | 4 |
| 30 | WARD | 1250.0 | 4 |
| 30 | JAMES | 950.0 | 5 |
```

RANK

関数の定義:

```
Bigint rank() over(partition by [col1, col2…]
order by [col1[asc|desc], col2[asc|desc]…])
```

使用法:

ランクを計算します。 **col2** と同じ行データのランキングは下がります。

パラメーターの説明:

- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。
- `Order by col1[asc|desc], col2[asc|desc]`: ランクの基準となる値を指定します。

戻り値:

Bigint 型。

例**:**

テーブル **emp** のデータは次のとおりです。

```
| empno | ename | job | mgr | hiredate| sal| comm | deptno |
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600,300,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250,500,30
7566,JONES,MANAGER,7839,1981-04-02 00:00:00,2975,,20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250,1400,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450,,10
7788,SCOTT,ANALYST,7566,1987-04-19 00:00:00,3000,,20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500,0,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100,,20
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300,,10
7948,JACCKA,CLERK,7782,1981-04-12 00:00:00,5000,,10
7956,WELAN,CLERK,7649,1982-07-20 00:00:00,2450,,10
7956,TEBAGE,CLERK,7748,1982-12-30 00:00:00,1300,,10
```

すべての従業員を部門別にグループ化し、各グループを **SAL** に従って降順にソートして、グループ独自のシリアル番号を取得する必要があるとします。

```
SELECT deptno
        , ename
        , sal
        , RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) AS nums
--Deptno as a window column, and sort in descending order according to
 sal.
    FROM emp;
--結果は次のとおりです。

| deptno | ename | sal | nums |

| 10 | JACCKA | 5000.0 | 1 |
| 10 | KING | 5000.0 | 1 |
| 10 | CLARK | 2450.0 | 3 |
| 10 | WELAN | 2450.0 | 3 |
| 10 | TEBAGE | 1300.0 | 5 |
| 10 | MILLER | 1300.0 | 5 |
| 20 | SCOTT | 3000.0 | 1 |
| 20 | FORD | 3000.0 | 1 |
| 20 | JONES | 2975.0 | 3 |
| 20 | ADAMS | 1100.0 | 4 |
| 20 | SMITH | 800.0 | 5 |
| 30 | BLAKE | 2850.0 | 1 |
| 30 | ALLEN | 1600.0 | 2 |
| 30 | TURNER | 1500.0 | 3 |
| 30 | MARTIN | 1250.0 | 4 |
| 30 | WARD | 1250.0 | 4 |
```

```
| 30 | JAMES | 950.0 | 6 |
```

LAG

関数の定義:

```
lag(expr, Bigint offset, default) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]])
```

コマンドの説明:

offset に従って現在行の **n** 行前の値を取得します。 現在の行番号が**rn**の場合は、行番号が **rn - offset** である行の値を取得します。

パラメーターの説明:

- `expr`: 任意のデータ型です。
- `offset`: **Bigint** 型の定数です。 入力値が **String** 型または **Double** 型の場合は、暗黙の変換によって **Bigint** 型に変換されます。 **Offset > 0** 。
- `default`: 指定した オフセットの範囲が制限を超えた場合のデフォルト値を定義します。 定数で、デフォルトは **NULL** です。
- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。
- `order by col1[asc|desc], col2[asc|desc]`: **Specifies the order method for return result.**

戻り値:

expr と同じデータ型が返されます。

LEAD

コマンド形式:

```
lead(expr, Bigint offset, default) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]])
```

コマンドの説明:

offset に従って、現在行の **n** 行後にある値を取得します。 現在の行番号が **rn** の場合、行番号が **rn + offset** である行の値が取得されます。

パラメーターの説明:

- `expr`: 任意のデータ型です。
- `offset`: **Bigint** 型の定数です。 入力値が **String** 型、**Decimal** 型、**Double** 型の場合は、暗黙の変換によって **Bigint** 型に変換されます。 **Offset > 0** です。

- default: 指定した オフセットの範囲が制限を超えた場合のデフォルト値を定義します。 定数です。
- Partition by [col1, col2..]: ウィンドウ関数で使用する列を指定します。
- order by col1[asc|desc], col2[asc|desc]: 戻される結果の並び替え方法を指定します。

戻り値:

expr と同じデータ型が返されます。

例:

```
select c_Double_a,c_String_b,c_int_a,lead(c_int_a,1) over(partition by
 c_Double_a order by c_String_b) from dual;
select c_String_a,c_time_b,c_Double_a,lead(c_Double_a,1) over(
partition by c_String_a order by c_time_b) from dual;
select c_String_in_fact_num,c_String_a,c_int_a,lead(c_int_a) over(
partition by c_String_in_fact_num order by c_String_a) from dual;
```

PERCENT_RANK

コマンド形式:

```
Percent_rank () over (partition by [col1, col2...]
order by [col1[asc|desc], col2[asc|desc]…])
```

コマンドの説明:

データグループ内の特定の行の相対的な順位を計算します。

パラメーターの説明:

- Partition by [col1, col2..]: ウィンドウ関数で使用する列を指定します。
- order by col1[asc|desc], col2[asc|desc]: ランクの基準となる値を指定します。

戻り値:

Double 型が返されます。値の範囲は **[0、1]**です。 相対的な順位の計算式は、`(rank-1)/(number of rows -1)`です。



注:

現在、単一ウィンドウ内の行の制限数は、最大 **10,000,000** です。

ROW_NUMBER

コマンド形式:

```
row_number() over(partition by [col1, col2…]
```

```
order by [col1[asc|desc], col2[asc|desc]…])
```

コマンドの説明:

1 から始まる行番号を計算します。

パラメーターの説明:

- `Partition by [col1, col2..]`: ウィンドウ関数で使用する列を指定します。
- `order by col1[asc|desc], col2[asc|desc]`: 戻される結果の並び替え方法を指定します。

戻り値:

Bigint 型が返されます。

例:

テーブル **emp** のデータは次のとおりです。

```
| empno | ename | job | mgr | hiredate| sal| comm | deptno |
7369,SMITH,CLERK,7902,1980-12-17 00:00:00,800,,20
7499,ALLEN,SALESMAN,7698,1981-02-20 00:00:00,1600,300,30
7521,WARD,SALESMAN,7698,1981-02-22 00:00:00,1250,500,30
7566, Jones, Manager, fig-04-02 00:00:00, 2975, 20
7654,MARTIN,SALESMAN,7698,1981-09-28 00:00:00,1250,1400,30
7698,BLAKE,MANAGER,7839,1981-05-01 00:00:00,2850,,30
7782,CLARK,MANAGER,7839,1981-06-09 00:00:00,2450,,10
7788, Scott, analyst, fig-04-19 00:00:00, 3000, 20
7839,KING,PRESIDENT,,1981-11-17 00:00:00,5000,,10
7844,TURNER,SALESMAN,7698,1981-09-08 00:00:00,1500,0,30
7876,ADAMS,CLERK,7788,1987-05-23 00:00:00,1100,,20
7900,JAMES,CLERK,7698,1981-12-03 00:00:00,950,,30
7902,FORD,ANALYST,7566,1981-12-03 00:00:00,3000,,20
7934,MILLER,CLERK,7782,1982-01-23 00:00:00,1300,,10
7948,JACCKA,CLERK,7782,1981-04-12 00:00:00,5000,,10
7956,WELAN,CLERK,7649,1982-07-20 00:00:00,2450,,10
7956, tebage, clerk, maid-12-30 00:00:00, 1300, 10
```

すべての従業員を部門別にグループ化し、各グループを **SAL** に従って降順にソートして、グループ独自のシリアル番号を取得する必要があるとします。

```
SELECT deptno
        , ename
        , Sal
        , Row_number () over (partition by deptno order by Sal DESC
) as Nums  --Deptno as a window column, and sort in descending order
according to sal.
    FROM emp;
--結果は次のとおりです。

| deptno | ename | sal | nums |

| 10 | JACCKA | 5000.0 | 1 |
| 10 | KING | 5000.0 | 2 |
| 10 | CLARK | 2450.0 | 3 |
```

```
| 10 | WELAN | 2450.0 | 4 |
| 10 | TEBAGE | 1300.0 | 5 |
| 10 | MILLER | 1300.0 | 6 |
| 20 | SCOTT | 3000.0 | 1 |
| 20 | FORD | 3000.0 | 2 |
| 20 | JONES | 2975.0 | 3 |
| 20 | ADAMS | 1100.0 | 4 |
| 20 | SMITH | 800.0 | 5 |
| 30 | BLAKE | 2850.0 | 1 |
| 30 | ALLEN | 1600.0 | 2 |
| 30 | TURNER | 1500.0 | 3 |
| 30 | MARTIN | 1250.0 | 4 |
| 30 | WARD | 1250.0 | 5 |
| 30 | JAMES | 950.0 | 6 |
```

CLUSTER_SAMPLE

コマンド形式**:**

```
boolean cluster_sample([Bigint x, Bigint y])
over(partition by [col1, col2..])
```

コマンドの説明**:**

グループサンプリングに使用します。

パラメーターの説明**:**

- x**: Bigint** 型の定数で、**x>=1** です。 パラメータ **y** を指定した場合、**x** はウィンドウを **x** 個の部分に分割することを示します。 パラメータ **y** を指定しない場合、**x** はウィンドウ内で **x** 行のレコードを選択することを示します **(x** 行がこのウィンドウ内にあれば、**true** が返されます**)**。 **x** が **NULL** の場合、**NULL** が返されます。
- y**: Bigint** 型の定数で、**y>=1**、**y<=x** です。ウィンドウ内の **x** 部分から **y** 部分レコードを選択することを示します **(**つまり、**y** 部分のレコードが存在する場合、戻り値は **true** になります**)**。**y** が **NULL** の場合、**NULL** が返されます。
- [col1、col2]**:** ウィンドウ関数を使用する列を指定します。

戻り値**:**

Boolean 型が返されます。

例**:**

テーブル **test_tbl** に **2** つの列 **key** と **value** がある場合、**key** がグループ化フィールドです。 以下に示すように、**key** に対応する値には **groupa** と **groupb** があり、**value** フィールドはキー値を示します。

```
| key | value |
| groupa | -1.34764165478145 |
```

```
    | groupa | 0.740212609046718 |
    | groupa | 0.167537127858695 |
    | groupa | 0.630314566185241 |
    | GroupA | 0.0112401388646925 |
    | groupa | 0.199165745875297 |
    | groupa | -0.320543343353587 |
    | groupa | -0.273930924365012 |
    | groupa | 0.386177958942063 |
    | groupa | -1.09209976687047 |
    | groupb | -1.10847690938643 |
    | groupb | -0.725703978381499 |
    | groupb | 1.05064697475759 |
    | groupb | 0.135751224393789 |
    | groupb | 2.13313102040396 |
    | groupb | -1.11828960785008 |
    | groupb | -0.849235511508911 |
    | groupb | 1.27913806620453 |
    | groupb | -0.330817716670401 |
    | groupb | -0.300156896191195 |
    | groupb | 2.4704244205196 |
    | groupb | -1.28051882084434 |
```

各グループから **10**% の値を選択するには、以下の **MaxCompute SQL** を推奨します。

```
Select key, Value
    from (
        Select key, value, cluster_sample (10, 1) over (partition by
key) as flag
        from tbl
        ) sub
    where flag = true;

| Key | value |

| groupa | 0.167537127858695 |
| groupb | 0.135751224393789 |
```

NTILE

コマンド形式**:**

```
BIGINT ntile(BIGINT n) over(partition by [col1, col2…]
[order by [col1[asc|desc], col2[asc|desc]…]] [windowing_clause]))
```

コマンドの説明**:**

グループ化されたデータを順番に **N** 個のスライスにカットして、現在のスライス値を返すために使用します。スライスが不均等な場合、最初のスライスの分布はデフォルトで増加します。

パラメーターの説明**:**

N: Bigint データ型。

戻り値**:**

Bigint 型が返されます。

例：

テーブル **EMP** のデータは、以下のとおりです。

```
| Empno | ename | job | Mgr | hiredate | Sal | REM | deptno |
7369, Smith, clerk, maid-12-17 00:00:00, 800, 20
7499, Allen, salesman, maid-02-20 00:00:00, 1600,300, 30
7521, Ward, salesman, maid-02-22 00:00:00, 1250,500, 30
7566, Jones, Manager, fig-04-02 00:00:00, 2975, 20
7654 Martin, salesman, fig-09-28 00:00:00, fig, 30
7698, Blake, Manager, fig-05-01 00:00:00, 2850, 30
7782, Clark, Manager, fig-06-09 00:00:00, 2450, 10
7788, Scott, analyst, fig-04-19 00:00:00, 3000, 20
00:00:00, King, President, 1991-11-17 5000, 7839, 10
7844, Turner, salesman, fig-09-08 00:00:00, 1500,0, 30
7876, Adams, clerk, maid-05-23 00:00:00, 1100, 20
7900 James, clerk, maid-12-03 00:00:00, 950, 30
7902 Ford, analyst, fig-12-03 00:00:00, 3000, 20
7934 Miller, clerk, fig-01-23 00:00:00, 1300, 10
7948, jaccka, clerk, fig-04-12 00:00:00, 5000, 10
7956, welan, clerk, fig-07-20 00:00:00, 2450, 10
7956, tebage, clerk, maid-12-30 00:00:00, 1300, 10
```

全従業員を **Sal** の高低に応じて **3** つのグループに分け、従業員自身のグループのシリアル番号を取得する必要があるとします。

```
Select deptno, ename, Sal, ntile (3) over (partition by depno order by
 Sal DESC) as nt3 from EMP;
-- 実行結果は次のとおりです

| Deptno | ename | Sal | nt3 |

| 10 | jaccka | 5000.0 | 1 |
| 10 | King | 5000.0 | 1 |
| 10 | welan | 2450.0 | 2 |
| 10 | Clark | 2450.0 | 2 |
| 10 | tebage | 1300.0 | 3 |
10 | Miller | 1300.0 | 3 |
| 20 | Scott | 3000.0 | 1 |
| 20 | Ford | 3000.0 | 1 |
| 20 | Jones | 2975.0 | 2 |
| 20 | Adams | 1100.0 | 2 |
| 20 | Smith | 800.0 | 3 |
| 30 | Blake | 2850.0 | 1 |
| 30 | Allen | 1600.0 | 1 |
| 30 | Turner | 1500.0 | 2 |
| 30 | Martin | 1250.0 | 2 |
| 30 | ward | 1250.0 | 3 |
```

```
| 30 | James | 950.0 | 3 |
```

5.12.4 集計関数

集計関数の入力と出力の関係は、多対 **1** です。つまり、複数の入力レコードが **1** つの出力レコードに集約されます。 **SQL** 文では、**group by** 句と一緒に使用します。

COUNT

関数の定義**:**

```
bigint count([distict|all] value)
```

使用法**:**

レコード数を数えます。

パラメーターの説明**:**

- **distinct|all:** カウント時に重複レコードを除外するかどうかを指定します。 デフォルトの **all** では、すべてのレコードが数えられます。 フィールド「**distinct**」を指定すると、一意のカウント値が使用されます。
- **value:** 任意のデータ型。 入力値が **NULL** の場合、対応する行はカウントされません。 **Count (*)** にすると、すべての行がカウントされます。

戻り値**:**

Bigint 型が返されます。

例**:**

```
-- テーブル tbla に列 col1 があり、データ型が Bigint の場合
+------+
| COL1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
select count(*) from tbla;  -- 値は 3.
select count(col1) from tbla;  -- 値は 2
```

集計関数は、**group by** 句と一緒に使用します。 たとえば、テーブル **"test_src"** に **2** つの列があり、**key** が **String** 型、**value** が **Double** 型であるとします。

```
-- test_src pデータは、次のように表示されます。
+-----+-------+
| key | value |
+-----+-------+
| a | 2.0 |
+-----+-------+
```

```
| a | 4.0 |
+-----+-------+
| b | 1.0 |
+-----+-------+
| b | 3.0 |
+-----+-------+
-- 次の文を実行し、結果を取得します。
select key, count(value) as count from test_src group by key;
+-----+-------+
| key | count |
+-----+-------+
| a | 2 |
+-----+-------+
| b | 2 |
+-----+-------+
-- 集計関数は、同じキー値を持つ集計値を計算します。前述のルールは、以降の関数にも適用されます。
```

AVG

関数の定義:

```
double avg(double value)
decimal avg(decimal value)
```

使用法:

平均値を計算します。

パラメーターの説明:

value: Double 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合は、暗黙の変換で **Double** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。 入力値が **NULL** の場合、対応する行は演算で使用されません。 入力値は **Boolean** 型にはできません。

戻り値:

入力値が **Decimal** 型の場合、**Decimal** 型が返されます。 それ以外の有効なデータ型の場合、**Double** 型が返されます。

例:

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
+-------+
| value |
+-------+
| 1 |
| 2 |
| NULL |
+-------+
-- この列の平均: (1+2)/2=1.5
select avg(value) as avg from tbla;
+------+
| avg |
+------+
```

```
| 1.5 |
+------+
```

MAX

関数の定義:

```
max(value)
```

使用法:

最大値を計算します。

パラメーターの説明:

value: 任意のデータ型。 列値が **NULL** の場合、対応する行は演算で使用されません。 **Boolean** 型の値は計算から除外されます。

戻り値:

戻り値と入力値のデータ型は同じです。

例:

```
-- テーブル tbla に列 clo1 があり、データ型が Bigint の場合
+------+
| col1 |
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
Select max (value) from tbla; -- 戻り値は 2
```

MIN

関数の定義:

```
MIN(value)
```

使用法：

列の最小値を計算します。

パラメーターの説明：

任意のデータ型。 列値が **NULL** の場合、対応する行は演算で使用されません。 **Boolean** 型は演算から除外されます。

例：

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
```

```
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+

+------+
Select min (value) from tbla; -- 戻り値は 1
```

MEDIAN

関数の定義:

```
double median(double number)
decimal median(decimal number)
```

使用法：

中央値を計算します。

パラメーターの説明：

number: Double 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合は、**Double** 型に変換されてから演算で使用されます。 それ以外のデータ型の場合は、例外が発生します。

戻り値:

Double 型または **Decimal** 型が返されます。

例：

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| 3 |
+------+
| 4 |
+------+
| 5 |
+------+
select MEDIAN(value) from tbla;  -- 戻り値は 3.0
```

STDDEV

関数の定義:

```
double stddev(double number)
```

```
decimal stddev(decimal number)
```

使用法：

母集団の標準偏差を計算します。

パラメーターの説明：

number: Double 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合は、**Double** 型に変換されてから演算で使用されます。 それ以外のデータ型の場合は、例外が発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。

例：

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| 3 |
+------+
| 4 |
+------+
| 5 |
+------+
select STDDEV(value) from tbla;  -- 戻り値は 1.4142135623730951
```

STDDEV_SAMP

関数の定義**:**

```
double stddev_samp(double number)
decimal stddev_samp(decimal number)
```

使用法：

母集団の標準偏差を計算します。

パラメーターの説明：

number: Double 型または **Decimal** 型です。 入力値が **String** 型または **Bigint** 型の場合は、**Double** 型に変換されてから演算で使用されます。 それ以外のデータ型の場合は、例外が発生します。

戻り値**:**

Double 型または **Decimal** 型が返されます。

例：

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| 3 |
+------+
| 4 |
+------+
| 5 |
+------+
select STDDEV_SAMP(value) from tbla; -- 戻り値は 1.5811388300841898
```

SUM

関数の定義**:**

```
sum(value)
```

使用法：

要素の合計を計算します。

パラメーターの説明：

value: Double 型、**Decimal** 型、**Bigint** 型のいずれかです。 入力値が **String** 型の場合は、**Double** 型に変換されてから演算で使用されます。 列の値が **NULL** の場合、その行はカウントされます。**Boolean** 型のデータは、計算から除外されます。

戻り値**:**

入力パラメータ値が **Bigint** 型の場合は、**Bigint**型が返されます。 入力パラメータ値が **Double** 型または **String** 型の場合は、**Double** 型が返されます。

例：

```
-- テーブル tbla に列値が含まれていて、データ型が Bigint の場合
+------+
| value|
+------+
| 1 |
+------+
| 2 |
+------+
| NULL |
+------+
```

```
select sum(value) from tbla;   -- 戻り値は 3
```

WM_CONCAT

関数の定義:

```
string wm_concat(string separator, string str)
```

使用法：

特定の区切り文字を使用して、**str** の値を連結します。

パラメーターの説明：

- **Separator: String** 型の定数です。 それ以外の型の定数の場合、または定数でない場合は、例外が発生する可能性があります。
- **Str: String** 型。 入力値が **String** 型の場合は、**Double** 型に変換されてから演算で使用されます。 別のデータ型の場合は、例外が発生します。

戻り値:

String 型が返されます。

 注：

`select wm_concat(',', name) from test_src;`の **SQL** 文では、**test_src** が空の場合には **NULL** が返されます。

COLLECT_LIST

関数の定義:

```
ARRAY collect_list(col)
```

使用法：

特定のグループ内で、**col** で指定された式を使用してデータを配列に集約します。

パラメーターの説明：

col: テーブル列は任意のデータ型にすることができます。

戻り値:

ARRAY 型が返されます。

 注：

新規データ型を使用するには、この関数を指定する SQL 文の前に `set odps.sql.type.system.odps2=true;` を追加し、 SQL 文と一緒に送信します。

COLLECT_SET

関数の定義:

```
ARRAY collect_set(col)
```

使用法：

特定のグループ内で、col で指定した式を使用して、データを非繰り返し要素の配列に集約します。

パラメーターの説明：

col: テーブル列は任意のデータ型にすることができます。

戻り値:

Array 型が返されます。

 注：

新規データ型を使用するには、この関数を指定する SQL 文の前に `set odps.sql.type.system.odps2=true;` を追加し、SQL 文と一緒に送信します。

5.12.5 文字列関数

ここでは、MaxCompute で使用できる CHAR_MATCHCOUNT、CHR、CONCAT、GET_JSON_OBJECT、INSTR、IS_ENCODING などの文字列関数について説明します。

CHAR_MATCHCOUNT

コマンド形式:

```
bigint char_matchcount(string str1, string str2)
```

使用法:

str1 内の各文字が str2 内で複製される合計回数を計算します。

パラメーターの説明:

- str1、str2: String 型で、有効な UTF-8 文字列でなければなりません。 照合処理中に無効な文字が検出されると、負の値が返されます。
- 戻り値: Bigint 型。入力値に NULL があると、NULLが返されます。

例:

```
char_matchcount('abd','aabc') = 2
-- str1 内の 2 つの文字列 'a' と 'b' が str2 に出現
```

CHR

コマンド形式:

```
string chr(bigint ascii)
```

使用法:

指定した **ASCII**コード「**ascii**」を文字に変換します。

パラメーターの説明:

- **ascii: Bigint** 型の **ASCII** 値。 入力値が「**String**」型または「**Double**」型の場合、暗黙の変換によって「**Bigint**」型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- 戻り値: **String** 型。 パラメータ値の範囲は、**0**～**255** です。 この範囲を超えると、例外が発生します。 入力値が **NULL** の場合は、**NULL** が返されます。

CONCAT

コマンド形式:

```
string concat(string a, string b...)
```

使用法:

すべての文字列を接続した結果を戻します。

パラメーターの説明:

- **a, b…: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- 戻り値: **String** 型。 パラメータが存在しない場合、または特定のパラメータ値が **NULL** の場合は、**NULL** が返されます。

例:

```
concat('ab','c') = 'abc'
concat() = NULL
```

```
concat('a', null, 'b') = NULL
```

GET_JSON_OBJECT

コマンド形式:

```
STRING GET_JSON_OBJECT(STRING json,STRING path)
```

使用法:

標準の **json** 文字列では、指定された文字列がパスに従って抽出されます。

パラメーターの説明:

- **json: String** 型で、標準の **json** 形式の文字列です。
- **path: String** 型で、ドル記号 **($)** で始まる **json** のパスを表す文字列です。 新規関数の実装については、「*JsonPath*」をご参照ください。
  - **$:** ルートオブジェクト
  - 。**:** 子演算子
  - **[]:** 配列の添字演算子
  - ***: []**のワイルドカード
- **String: String** 型が返されます。

 注 **:**

- **json** が **NULL** または無効な **json** 形式の場合は、**NULL** が返されます。
- **path** が **Null** または無効 **(json** に存在しない**)** の場合、**NULL** が返されます。
- **json** が有効で **path** も存在する場合は、対応する文字列が返されます。

例:

```
+-----+
json
+-----+
{"store":
{"fruit":[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}],
"bicycle":{"price":19.95,"color":"red"}
},
"email":"amy@only_for_json_udf_test.net",
"owner":"amy"
}
```

JSON オブジェクト内の情報を抽出するには、次の クエリ処理を実行します。

```
odps> SELECT get_json_object(src_json.json, '$.owner') FROM src_json;
amy
odps> SELECT get_json_object(src_json.json, '$.store.fruit\[0]') FROM
src_json;
```

```
{"weight":8,"type":"apple"}
odps> SELECT get_json_object(src_json.json, '$.non_exist_key') FROM
src_json;
NULL
```

例:

```
get_json_object('{"array":[["aaaa",1111],["bbbb",2222],["cccc",3333
]]}','$.array[1][1]')= "2222"
get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["
h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[*]') = "["h0","h1","h2"]"
get_json_object('{"aaa":"bbb","ccc":{"ddd":"eee","fff":"ggg","hhh":["
h0","h1","h2"]},"iii":"jjj"}','$.ccc.hhh[1]') = "h1"
```

INSTR

コマンド形式:

```
bigint instr(string str1, string str2[, bigint start_position[, bigint
 nth_appearance]])
```

使用法:

部分文字列 **str2** が **str1** のどこにあるかを計算します。

パラメーターの説明:

- **str1: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- **str2: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- **start_position: Bigint** 型です。他のデータ型の場合は、例外が発生します。 このパラメーターは、**str1** のどの文字から検索を開始する必要があるかを示します。デフォルトの開始位置は、最初の文字位置 **1** です。 **0** 未満の場合は、エラーが発生します。
- **nth_appearance: Bigint**型 (**0**より大きい) で、文字列中の部分文字列の **2** 番目に一致する位置を表します。 チェーンのデータ型が異なる場合、または **0** 以下の場合、例外が発生します。
- 戻り値: **Bigint** 型。

 注:

- **str2** が **str1** で検出されない場合は、**0** が返されます。-
- いずれかの入力パラメータ値が **NULL** の場合、**NULL** が返されます。
- **str2** が **NULL** で常に一致する場合、**instr (‘abc’, ‘’)** では **1** が返されます。

例:

```
instr('Tech on the net', 'e') = 2
instr('Tech on the net', 'e', 1, 1) = 2
instr('Tech on the net', 'e', 1, 2) = 11
instr('Tech on the net', 'e', 1, 3) = 14
```

IS_ENCODING

コマンド形式:

```
boolean is_encoding(string str, string from_encoding, string
to_encoding)
```

使用法:

指定した文字セット「**from_encoding**」から文字セット 「**to_encoding**」に入力文字列「**str**」を変更できるかどうかを判断します。 入力値が文字化けしているかを判断するために使用できます。 一般的な使い方では、「**from_encoding**」を 「**utf-8**」に、「**'to_encoding**」を「**gbk**」に設定します。

パラメーターの説明:

- **str: String** 型。入力値が **NULL** の場合は **NULL** が返されます。 空の文字列は、任意の文字セットに属していると見なすことができます。
- **from_encoding**、**to_encoding: String** 型で転送元、転送先の文字セットです。 入力値が **NULL** の場合は、**NULL** が返されます。
- 戻り値: **Boolean** 型。 「**str**」が正常に変換できる場合は、**true** が返されます。そうでない場合は、**false** が返されます。

例:

```
is_encoding('test', 'utf-8', 'gbk') = true
is_encoding('test', 'utf-8', 'gbk') = true
-- これら 2 つの繁体字は、中国の GBK 語グループです。
is_encoding('test', 'utf-8', 'gb2312') = false
-- ‘GB2312’ の書記素一覧には、これら 2 つの繁体字が含まれていません。
```

KEYVALUE

コマンド形式:

```
KEYVALUE(STRING srcStr,STRING split1,STRING split2, STRING key)
KEYVALUE(STRING srcStr,STRING key) //split1 = ";", split2 = ":"
```

使用法:

split1 で ‘**srcStr**’ を ‘**key-value**’ のペアに分割し、**split2** で ‘**key-value**’ のペアを分割します。 キーに対応する値を取得します。

パラメーターの説明:

- **srcStr:** 分割するソース文字列です。
- **key: n** 番目の文字列を返すように指定します。 ソース文字列が「**split1**」と「**split2**」で分割された後、「**key**」に指定した値に従って、対応する値が返されます。
- **split1**、**split2:**「**srcStr**」を分割するための区切り文字として使用される文字列です。 これら **2** つのパラメータを式で指定しない場合、「**split1**」のデフォルト値は「**;**」、「**split2**」のデフォルト値は「**:**」になります。 **split1** によって分割され、かつ **split2** が複数ある文字列の場合、戻り結果は定義されません。

戻り値:

- **String** 型。
- 「**split1**」または「**split2**」が **NULL** の場合は、**NULL** が返されます。
- 「**scrStr**」と「**key**」が **NULL** の場合、または「**key**」と一致しない場合は **NULL** が返されます。
- 複数の「**Key-Value**」が一致する場合は、最初に一致したキーに対応する値が返されます。

例**1:**

```
keyvalue('0:1\;1:2', 1) = '2'
```

注:

ソース文字列は、「**0:1\;1:2**」です。 **split1** と **split2** が指定されていないため、デフォルトの **split1** は「**;**」で、**split2** は「**:**」です。

split1 の分割後、キーと値のペアは **0:1\,1:2** です。

split2 splitの後は、次のようになります。

```
0 1/
1 2
```

1 に対応するキーの値 **(2)** が返されます。

例 **2:**

```
keyvalue("\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;
shipping:2\;pf:0\;market:shoes\;instPayAmount:0\;","\;",":","tf") = "
21910" value:21910.
```

注:

ソース文字列は次のとおりです。

```
“\;decreaseStore:1\;xcard:1\;isB2C:1\;tf:21910\;cart:1\;shipping:2\;
pf:0\;market:shoes\;instPayAmount:0\;”
```

split1 '\;' に従って分割された後に、分割から派生したキーと値の組み合わせは、以下のとおりです。

```
decreaseStore:1, xcard:1, isB2C:1, tf:21910, cart:1, shipping:2, pf:0,
market:shoes, instPayAmount:0
```

分割後、**split2 ":"** に従うと、結果は次のようになります。

```
decreaseStore 1
 xcard 1
 isB2C 1
 tf 21910
 cart 1
 shipping 2
 pf 0
 market shoes
 instPayAmount 0
```

キーパラメータの値は「**tf**」になります。対応する値パラメータの戻り値は、**21910**です。

LENGTH

コマンド形式**:**

```
bigint length(string str)
```

使用法**:**

文字列の長さを返します。

パラメーターの説明**:**

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- 戻り値**: Bigint** 型。 「**str**」が **NULL** の場合、**NULL** が返されます。 「**str**」が **UTF-8** 以外のコーディング形式の場合、**-1** が返されます。

例:

```
length('hi! China') = 6
```

LENGTHB

コマンド形式:

```
bigint lengthb(string str)
```

使用法:

「**str**」の長さが返されます。単位はバイトです。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- 戻り値: **Bigint** 型。 「**str**」が **NULL** の場合、**NULL** が返されます。

例:

```
lengthb('hi! 中国') = 10
```

MD5

コマンド形式:

```
string md5(string value)
```

使用法:

入力文字列の **MD5** 値を計算します。

パラメーターの説明:

- **value: String**型。 入力値が **Bigint**型、**Double**型、**Decimal**型、**Datetime** 型の場合、計算前に暗黙的に **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。 入力値が **NULL** の場合は、**NULL** が返されます。
- 戻り値: **String** 型。

REGEXP_EXTRACT

コマンド形式:

```
string regexp_extract(string source, string pattern[, bigint
occurrence])
```

使用法:

パターンに従って文字列ソースを分割し (正規表現の規則) 、出現 (n番目) グループの文字を返します。

パラメーターの説明:

- **source: String** 型。検索対象の文字列です。
- **pattern: String** 型の定数です。 **pattern** が **NULL** 文字列の場合は、例外が発生します。「**group**」が **pattern** で指定されていない場合も、例外が発生します。
- **Occurrence: Bigint** 型の定数で、**0** 以上にする必要があります。 それ以外のデータ型、または **0** より小さい場合、例外が発生します。 このパラメータを指定しない場合、デフォルト値は **1** になります。これは、最初のグループを返すことを示します。「**Occurrence**」が **0** の場合、「**pattern**」全体を満たす部分文字列が返されます。
- 戻り値: **String** 型。 入力値に **NULL** がある場合は、**NULL** が返されます。

例:

```
regexp_extract('foothebar', 'foo(. *?)( bar)', 1) = the
regexp_extract('foothebar', 'foo(. *?)( bar)', 2) = bar
regexp_extract('foothebar', 'foo(. *?)( bar)', 0) = foothebar
regexp_extract('8d99d8', '8d(\\d+)d8') = 99
-- 標準 SQL が MaxCompute 上で送信された場合、2 つの "\" をシフト文字として使用
する必要があります。
regexp_extract('foothebar', 'foothebar')
-- 例外が発生します。‘pattern’ に ‘group’ が指定されていません。
```

REGEXP_INSTR

関数の定義

```
bigint regexp_instr(string source, string pattern[,
bigint start_position[, bigint nth_occurrence[, bigint return_option
]]])
```

使用法:

部分文字列の開始位置や終了位置を返します。これは、**start_position** と **nth_occurrence** のソースと **pattern** が一致します。 入力パラメータが **NULL** の場合、**NULL** を返します。

パラメーターの説明:

- **source: String** 型で、検索対象です。
- **pattern: String** 型の定数です。 「**pattern**」が **NULL** の場合、例外が発生します。
- **start_position: Bigint** 型の定数で、検索の開始位置です。 指定しない場合、デフォルト値は **1** です。 **Bigint** 以外のデータ型の場合、または入力値が **0** 以下の場合は、例外が発生します。
- **nth_occurrence: Bigint** 型の定数です。 指定しない場合、デフォルト値は **1** です。 検索すると、最初の位置に表示されます。 入力値が **0** 以下の場合、または **Bigint** 以外のデータ型の場合、例外が発生します。
- **return_option: Bigint** 型の定数です。 値は **0** または **1** です。 それ以外のデータ型、または無効な値の 場合、例外が発生します。 **0** は一致した値の開始位置を返すことを示します。 **1** は一致した値の終了位置を返すことを示します。
- **Return value: Bigint** 型。**return_option** で指定したソース内の一致した部分文字列の開始位置または終了位置です。

例**:**

```
regexp_instr("i love www.taobao.com", "o[[:alpha:]]{1}", 3, 2) = 14
```

REGEXP_REPLACE

コマンド形式**:**

```
string regexp_replace(string source, string pattern, string replace_st
ring[, bigint occurrence])
```

使用法**:**

n 回目の出現で **'pattern'**と一致した **source** の部分文字列を指定された文字列 **'replace_st ring'** に置き換えてから戻ります。

パラメーターの説明**:**

- **source: String** 型。置換対象の文字列です。
- **pattern: String** 型の定数で、 照合するパターンです。 入力値が **NULL** の場合、例外が発生します。
- **replace_string: String** 型。マッチしたパターンを置き換えた後の文字列です。
- **Occurrence: Bigint** 型の定数で、**0** 以上にする必要があります。 **n** 番目のマッチングを **replace_string** に置き換えることを示します。 **0** の場合、一致したすべての部分文字列が置き換えられたことを示します。 それ以外のデータ型、または **0** より小さい場合、例外が発生します。 デフォルトは **0** になります。

- 戻り値: **String** 型。 存在しないグループを参照するときは、文字列を置き換えないでください。 **source**、**pattern**、**occurrence** パラメータに **NULL** が入力されると、**NULL** が返されます。**replace_string** が **NULL** でパターンが一致しない場合、**NULL** が返されます。**replace_string** が **NULL** でパターンが一致する場合は、元の文字列が返されます。

 注 :

参照グループが存在しない場合は、未定義と見なされます。

例:

```
regexp_replace("123.456.7890", "([[:digit:]]{3})\\.([[:digit:]]{3})\\.
([[:digit:]]{4})",
"(\\1)\\2-\\3", 0) = "(123)456-7890"
regexp_replace("abcd", "(.)", "\\1 ", 0) = "a b c d "
regexp_replace("abcd", "(.)", "\\1 ", 1) = "a bcd"
regexp_replace("abcd", "(.)", "\\2", 1) = "abcd"
-- pattern にはグループのみが定義されていて、参照先の 2 番目のグループは存在しま
せん。
-- これは回避してください。 存在しないグループの参照結果は定義されていません。
regexp_replace("abcd", "(. *)(.)$", "\\2", 0) = "d"
regexp_replace("abcd", "a", "\\1", 0) = "bcd"
-- pattern にグループ定義が存在しないので、'\1' は存在しないグループを参照しま
す。
-- これは回避してください。 存在しないグループの参照結果は定義されていません。
```

REGEXP_SUBSTR

コマンド形式:

```
string regexp_substr(string source, string pattern[, bigint start_posi
tion[, bigint nth_occurrence]])
```

使用法:

start_position から検索を開始し、指定されたパターンと **n** 回目に一致する **source** 内の部分文字列を見つけます。

パラメーターの説明:

- **source: String** 型で、検索対象の文字列です。
- **pattern: String** 型の定数で、 照合するパターンです。 入力値が **NULL** の場合、例外が発生します。
- **start_position: Bigint** 型の定数で、**0** より大きくなければなりません。 **Bigint** 以外のデータ型の場合、または **0** 以下の場合、例外が発生します。 指定しない場合、デフォルト値は **1** です。これは、照合が **source** の最初の文字から始まることを示します。 指定しない場合、デフォルト値は **1** です。 これは、照合が **source** の最初の文字から始まることを示します。

- **nth_occurrence: Bigint** 型の定数で、**0** より大きくなければなりません。 指定しない場合、デフォルト値は **1** です。 これは、戻される部分文字列が、最初に一致した値 であることを示します。 指定しない場合、デフォルト値は **1** です。 これは、戻される部分文字列が、最初に一致した値 であることを示します。
- 戻り値**: String** 型。 入力値に **NULL** がある場合は、**NULL** が返されます。 一致するレコードが存在しない場合は、**NULL**が返されます。

例**:**

```
regexp_substr ("I love aliyun very much", "a[[:alpha:]]{5}") = "aliyun
"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum
:]]*', 1, 1) = " have"
regexp_substr('I have 2 apples and 100 bucks!', '[[:blank:]][[:alnum
:]]*', 1, 2) = "2"
```

REGEXP_COUNT

コマンド形式**:**

```
bigint regexp_count(string source, string pattern[, bigint start_posi
tion])
```

使用法**:**

source 内の **start_position** から始まる、部分文字列が指定したパターンと一致する回数をカウントします。

パラメーターの説明**:**

- **source: String** 型で、検索対象の文字列です。 **String** 型以外の場合、例外が発生します。
- **Pattern: String** 型の定数で、照合するパターンです。 入力値が **NULL** の文字列の場合、または **String** 型以外のデータ型の場合は、例外が発生します。
- **start_position: Bigint** 型の定数で、**0** より大きくなければなりません。 **Bigint** 以外のデータ型の場合、または入力値が **0** 以下の場合は、例外が発生します。 指定しない場合、デフォルト値は **1** です。これは、照合が **source** の最初の文字から始まることを示します。
- 戻り値**: Bigint** 型になります。 一致するものがない場合は、**0** が返されます。 いずれかの入力パラメータ値が **NULL** の場合、**NULL** が返されます。

例**:**

```
regexp_count('abababc', 'a.c') = 1
```

```
regexp_count('abcde', '[[:alpha:]]{2}', 3) = 1
```

SPLIT_PART

コマンド形式:

```
string split_part(string str, string separator, bigint start[, bigint
end])
```

使用法:

区切り文字に従って文字列 **str** を分割し、**n** 番目の開始部分から **n** 番目の終了部分までの部分文字列を返します。

パラメーターの説明:

- **str: String** 型で、分割される文字列です。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- **separator: String** 型 の定数で、文字列を分割するために使用する区切り文字です。 文字でも文字列でも構いません。 入力値が他のデータ型の場合は、例外が発生します。
- **start: Bigint** 型の定数で、**0** より大きくなければなりません。 定数でない場合、または **Bigint** 以外のデータ型の場合、例外が発生します。 戻り部分の開始番号 **(1**から開始**)** を示します。 **end** が指定されていない場合は、「**start**」で指定した部分が返されます。
- **end': Bigint** 型の定数で、「**start**」以上にする必要があります。そうでなければ、例外が発生します。 戻される部分の終了番号を示します。 定数でない場合、または **Bigint** 以外のデータ型の場合、例外が発生します。 最後の部分を示しているので、省略できます。

戻り値**: String** 型。 いずれかの入力パラメータ値が **NULL** の場合、**NULL** が返されます。 **separator** が空の文字列の場合、ソース文字列 **str** が返されます。

 注 **:**

- 「**delimiter**」が **str** に存在しない場合は、 「**start**」を**1**として指定し、**str** 全体が返されます。 入力値が空の文字列の場合、出力値は空の文字列になります。
- 開始値が分割後の部分数よりも大きい場合、たとえば、分割は**6** つに分割され、「**start**」の値が **6** より大きい場合、空の文字列が返されます。

例**:**

```
split_part('a,b,c,d', ',', 1) = 'a'
split_part('a,b,c,d', ',', 1, 2) = 'a,b'
```

```
split_part('a,b,c,d', ',', 10) = ''
```

SUBSTR

コマンド形式:

```
string substr(string str, bigint start_position[, bigint length])
```

使用法:

start_positionから「**str**」の部分文字列を指定した長さ返します。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 入力値が他のデータ型の場合は、例外が発生します。
- **start_position: Bigint** 型で、**1** から始まります。 **start_position** が **0** の場合、空の文字列が返されます。**start_position** が負の場合、開始位置は文字列の末尾から逆方向に数えられ、最後の文字は **-1**、その直前の数は **-2**、**-3** などとなります。 **Bigint** 型以外の場合、例外が発生します。
- **length: Bigint** 型で、**0** より大きくなければなりません。 それ以外のデータ型、または **0** より小さい場合、例外が発生します。 このパラメータは、子文字列の長さを示します。
- 戻り値**: String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

 注 :
length を省略すると、最初から最後までの部分文字列が返されます。

例:

```
substr("abc", 2) = "bc"
substr("abc", 2, 1) = "b"
substr("abc",-2,2)="bc"
substr("abc",-3)="abc"
```

SUBSTRING

コマンド形式:

```
string  substring(string|binary str, int start_position[, int length])
```

使用法:

start_positionから「**str**」の指定した長さの部分文字列を返します。

パラメーターの説明:

- **str: String** 型または **Binary** 型です。それ以外のデータ型の場合、**NULL** が返されるか、例外が発生します。
- **start_position: 1** から始まる **Int** 型です。 **start_position** が **0** の場合、空の文字列が返されます。**start_position** が負の場合、開始位置は文字列の末尾から逆方向に数えられ、最後の文字は **-1**、その直前の数は **-2**、**-3** などとなります。 **Int** 型以外の場合、例外が発生します。
- **length: Bigint** 型で、**0** より大きくなければなりません。 それ以外のデータ型、または **0** より小さい場合、例外が発生します。 このパラメータは、子文字列の長さを示します。
- 戻り値**: String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

 注 **:**

length を省略すると、最初から最後までの部分文字列が返されます。

例**:**

```
substring('abc', 2) = 'bc'
substring('abc', 2, 1) ='"b'
substring('abc',-2,2)='bc'
substring('abc',-3,2)='ab'
substring(BIN(2345),2,3)='001'
```

TOLOWER

コマンド形式**:**

```
string tolower(string source)
```

使用法**:**

英語の文字列ソースに対応する小文字の文字列を入力 します。

パラメーターの説明**:**

- **source: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 他のデータ型の場合は、例外が発生します。
- 戻り値**: String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例**:**

```
tolower("aBcd") = "abcd"
```

```
tolower("Haha Cd") = "haha cd"
```

TOUPPER

コマンド形式:

```
string toupper(string source)
```

使用法:

英語の文字列「**source**」に対応する大文字の文字列を出力します。

パラメーターの説明:

- **source: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。
- 戻り値: **String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例:

```
toupper("aBcd") = "ABCD"
toupper("HahaCd") = "HAHACD"
```

TO_CHAR

コマンド形式:

```
string to_char(boolean value)
string to_char(bigint value)
string to_char(double value)
string to_char(decimal value)
```

使用法:

Boolean 型、**Bigint** 型、**Double** 型を対応する**String** 型に変換します。

パラメーターの説明:

- **Value: Boolean** 型、**Bigint** 型、**Double** 型を指定できます。 それ以外のデータ型の場合、例外が発生します。 **Datetime** 型の形式に出力する場合は、同じ名前の別の関数「**TO_CHAR**」をご参照ください。
- 戻り値: **String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例:

```
to_char(123) = '123'
to_char(true) = 'TRUE'
to_char(1.23) = '1.23'
```

```
to_char(null) = NULL
```

TRIM

コマンド形式:

```
string trim(string str)
```

使用法:

入力文字列 **str** の左スペースと右スペースを削除します。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合、例外が発生します。
- 戻り値: **String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

LTRIM

コマンド形式:

```
string ltrim(string str)
```

使用法:

入力文字列 **str** の左側のスペースを削除します。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合、例外が発生します。
- 戻り値: **String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例:

```
select ltrim(' abc ') from dual;
Returns:
+-----+
| _c0 |
+-----+
| abc |
+-----+
```

RTRIM

コマンド形式:

```
string rtrim(string str)
```

使用法:

入力文字列 **str** の右側のスペースを削除します。

パラメーターの説明**:**

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。
- 戻り値**: String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例**:**

```
select rtrim('a abc ') from dual;
Returns:
+------+
| _c0 |
+------+
| a abc |
+------+
```

REVERSE

コマンド形式**:**

```
STRING REVERSE(string str)
```

使用法**:**

逆順の文字列を返します。

パラメーターの説明**:**

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。
- 戻り値**: String** 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例**:**

```
select reverse('abcedfg') from dual;
Returns:
+------+
| _c0 |
+------+
| gfdecba |
```

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

SPACE

コマンド形式:

```
STRING SPACE(bigint n)
```

使用法:

長さ **n** の文字列を返すスペース文字列関数です。

パラメーターの説明:

- **n: Bigint** 型。長さは **2 MB** を超えることはできません。 入力値が **NULL** の場合、例外が発生します。
- 戻り値: **String** 型。

例:

```
select length(space(10)) from dual; ---- 10 を戻します。
select space(400000000000) from dual; ----エラー。長さが 2 MB を超えています。
```

REPEAT

コマンド形式:

```
STRING REPEAT(string str, bigint n)
```

使用法:

n 回繰り返される **str** 文字列を返します。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。
- **n: Bigint** 型。 長さは **2 MB** を超えることはできません。 入力値が **NULL** の場合、例外が発生します。
- 戻り値: **String** 型。

例:

```
select repeat('abc',5) from lxw_dual;
```

```
Returns:abcabcabcabcabc
```

ASCII

コマンド形式:

```
Bigint ASCII(string str)
```

使用法:

str の最初の文字の **ascii** を返します。

パラメーターの説明:

- **str: String** 型。 入力値が **Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型の場合、暗黙の変換によって **String** 型に変換されます。 それ以外のデータ型の場合は、例外が発生します。
- 戻り値: **Bigint** 型。

例:

```
select ascii('abcde') from dual;
戻り値 : 97
```

Maxcomputerte2.0 拡張機能

MaxCompute 2.0 へのアップグレードに伴い、いくつかの算術関数が追加されました。 新規関数で新規データ型を使用する場合は、新規関数の **SQL** 文を指定する前に、次の **set** 文を追加する必要があります。

```
set odps.sql.type.system.odps2=true;
```



注:

新規データ型関数を使用するには、関数を指定する **SQL** 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、**SQL** 文と一緒に実行します。

拡張された関数および拡張された文字列関数は、次のとおりです。

CONCAT_WS

コマンド形式:

```
string concat_ws(string SEP, string a, string b...)
string concat_ws(string SEP, array)
```



注:

新規データ型関数を使用するには、関数を指定する **SQL** 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、**SQL** 文と一緒に実行します。

使用法**:**

パラメータ内のすべての文字列を指定した区切り文字で接続して連結します。

パラメーターの説明**:**

- **SEP: String** 型の区切り文字です。 指定しないと、例外が返されます。

戻り値**:**

String 型。 パラメータが存在しない場合、またはパラメータが **NULL** の場合は、**NULL** が返されます。

例**:**

```
concat_ws(':','name','hanmeimei')='name:hanmeimei'
concat_ws(':','avg',null,'34')=null
```

LPAD

コマンド形式**:**

```
string lpad(string a, int len, string b)
```

 注 **:**
新規データ型関数を使用するには、関数を指定する **SQL** 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、**SQL** 文と一緒に実行します。

使用法**:**

文字列 **b** を使用して、文字列 **a** を **len** で指定した位置の左に挿入します。

パラメーターの説明**:**

- **len: Int** 型の整数です。
- **a/b…: String** 型。

戻り値**:**

String 型。 **len**が **a** の桁数よりも小さい場合、**a** が左から切り捨てられ、**len** で指定された桁数の文字列が取得されます。 **len** が **0** の場合、空が返されます。

例**:**

```
lpad('abcdefgh',10,'12')='12abcdefgh'
lpad('abcdefgh',5,'12')='abcde'
```

```
lpad('abcdefgh',0,'12') Returns a blank result
```

RPAD

コマンド形式:

```
string rpad(string a, int len, string b)
```

 注:

新規データ型関数を使用するには、関数を指定する SQL 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、SQL 文と一緒に実行します。

使用法:

文字列 b を使用して、文字列 a を len で指定した位置の右に挿入します。

 注:

関数 `set odps.sql.type.system.odps2 = true` を使用する SQL 文の前に set odps 文を追加する必要があります。そうしないと、エラーが報告されます。

パラメーターの説明:

- len: Int 型の整数。
- a/b…: String 型。

戻り値:

String 型。 len が a の桁数よりも小さい場合、a が左から切り捨てられ、len で指定された桁数の文字列が取得されます。 len が 0 の場合、空が返されます。

例:

```
rpad('abcdefgh',10,'12')='abcdefgh12'
rpad('abcdefgh',5,'12')='abcde'
rpad('abcdefgh',0,'12') Returns a blank result
```

REPLACE

コマンド形式:

```
string replace(string a, string OLD, string NEW)
```

 注:

新規データ型関数を使用するには、関数を指定する SQL 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、SQL 文と一緒に実行します。

使用法:

文字列 a の string OLD と完全に一致する部分を string NEW に置き換えて、文字列 a を返します。

パラメーターの説明:

パラメータはすべて String 型です。

戻り値:

String 型。 入力値が NULL の場合は、NULL が返されます。

例:

```
replace('ababab','abab','12')='12ab'
replace('ababab','cdf','123')='ababab'
replace('123abab456ab',null,'abab')=null
```

SOUNDEX

コマンド形式:

```
string soundex(string a)
```

 注:

新規データ型関数を使用するには、関数を指定する SQL 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、SQL 文と一緒に実行します。

使用法:

通常の文字列を SOUNDEX 文字列に変換します。

パラメータの説明: a は String 型です。

戻り値: String 型。 入力値が NULL の場合は、NULL が返されます。

例:

```
soundex('hello')='H400'
```

SUBSTRING_INDEX

コマンド形式:

```
string substring_index(string a, string SEP, int count))
```

 注:

新規データ型関数を使用するには、関数を指定する SQL 文の前に `set odps.sql.type.system.odps2 = true;` を追加し、SQL 文と一緒に実行します。

使用法:

count で指定した区切り文字の前部分を残し、文字列 **a** を切り捨てます。 **count** が正の場合は、区切り文字の左側の部分が残されます。 **count** が負の場合は、右側の部分が残されます。

パラメータの説明: **a** と **SEP** 部分は **String** 型で、**count** 部分は **Int** 型です。

戻り値:

String 型。 入力値が **NULL** の場合は、**NULL** が返されます。

例:

```
substring_index('https://www.alibabacloud.com, '.', 2)='https://www.
alibabacloud'
substring_index('https://www.alibabacloud.com', '.', -2)='alibabacloud
.com'
substring_index('https://www.alibabacloud.com', null, 2)=null
```

5.12.6 その他の関数

ここでは、**cast**、**decode**、**least**、**array**、**split**、**map** などの関数の使用方法について説明します。

CAST

関数の定義

```
cast(expr as <type>)
```

式の結果をオブジェクト型に変換します。 たとえば、**cast (‘1’ as bigint)** を実行すると、**String** 型の「**1**」が **Bingint** 型の「**1**」に変換されます。 変換が失敗した場合、またはサポート対象外の場合は、例外が発生します。

 注 :

- **cast (double as bigint)** を実行すると、**Double** 型の値が **Bigint** 型の値に変換されます。
- **cast (string as bigint)** を実行すると、**String** 型の値が **Bigint** 型の値に変換されます。 文字列が整数形式で表現された数字で構成されている場合は、**Bigint** 型の値に直接変換されます。
- 文字列が浮動形式または指数形式で表現された数値で構成される場合、最初に **Double** 型の値に変換され、次に **Bigint** 型の値に変換されます。

- **cast(string as datetime)** または **cast(datetime as string)** では、デフォルト形式 `yyyy-mm-dd hh:mi:ss`が使用されます。

COALESCE

関数の定義

```
coalesce(expr1, expr2, ...)
```

使用法:

リストから **NULL** ではない最初の値を返します。 リスト内のすべての値が **NULL** の場合は、**NULL** が返されます。

パラメーターの説明：

expr: テスト対象の値 これらの値はすべて同じデータ型か、**NULL** でなければなりません。そうでない場合は、例外が発生します。

戻り値:

戻り値のデータ型は、パラメーターのデータ型と同じです。

 注：
少なくとも **1** つのパラメータが必要です。そうでない場合は例外が発生します。

DECODE

関数の定義

```
decode(expression, search, result[, search, result]...[, default])
```

使用法：

選択関数に **if-then-else** 分岐を実装します。

パラメーターの説明：

- **expression:** 比較する式。
- **search: expression** と比較する検索文字列です。
- **result: search** と **expression** の値が一致したときに戻す値です。
- **default:** 任意のパラメーターです。 すべての検索項目が式と一致しない場合は、このデフォルト値が返されます。 指定しない場合は、**NULL** が返されます。

戻り値:

- 一致した検索レコードが返されます。

- 一致するレコードがない場合は、デフォルト値が返されます。
- デフォルト値が指定されていない場合は、 **NULL** が返されます。

> 注：
>
> - 少なくとも **3** つのパラメーターを指定しなければなりません。
> - 結果のデータ型はすべて同じか **NULL** でなければなりません。 データ型が一致しない場合は、例外が発生します。 すべての検索と式のデータ型は同じでなければなりません。そうでなければ例外が報告されます。
> - デコード中のオプション検索で、レコードが繰り返し一致した場合は、最初の値が返されます。

例：

```
Select
decode(customer_id,
1, 'Taobao',
2, 'Alipay',
3, 'Aliyun',
Null, 'N/A',
'Others') as result
from sale_detail;
```

前述した **DECODE** 関数では、次の **if-then-else** 文で関数が実装されます。

```
if customer_id = 1 then
result := 'Taobao';
elsif customer_id = 2 then
result := 'Alipay';
elsif customer_id = 3 then
result := 'Aliyun';
...
else
result := 'Others';
end if;
```

> 注：
>
> - **MaxCompute SQL**で **NULL** = **NULL** が計算され、**NULL** が返されますが、**DECODE** 関数では **NULL** と **NULL** の値は同じです。
> - 上記の例で、 `customer_id` の値が **NULL** の場合、**DECODE** 関数では **N/A** が結果として返されます。

GET_IDCARD_AGE

関数の定義

```
get_idcard_age(idcardno)
```

使用法：

現在の年と、**ID** で識別された生年との差である **ID** 番号に基づいて現在の年齢を返します。

パラメーターの説明：

idcardno: String 型で、**15** 桁または **18** 桁の **ID** 番号です。 計算では、都道府県コードと最後の桁に従って **ID** の有効性がチェックされ、チェックが失敗した場合は **NULL** が返されます。

戻り値**:**

Bigint 型が返されます。 入力値が **NULL** の場合、**NULL** が返されます。 現在の年と誕生年の差が **100** より大きい場合は、**NULL** が返されます。

GET_IDCARD_BIRTHDAY

関数の定義

```
get_idcard_birthday(idcardno)
```

使用法：

ID 番号に基づいて生年月日を返します。

パラメーターの説明：

idcardno: String 型で、**15** 桁または **18** 桁の **ID** 番号です。 計算では、都道府県コードと最後の桁に従って **ID** の有効性がチェックされ、チェックが失敗した場合は **NULL** が返されます。

戻り値**:**

Datetime 型が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

GET_IDCARD_SEX

関数の定義

```
get_idcard_sex(idcardno)
```

使用法：

ID 番号に基づいて性別を返します。値は **M (**男性**)** または **F (**女性**)** です。

パラメーターの説明：

idcardno: String 型で、**15** 桁または **18** 桁の **ID** 番号です。 計算では、都道府県コードと最後の桁に従って **ID** の有効性がチェックされ、チェックが失敗した場合は **NULL** が返されます。

戻り値**:**

String 型が返されます。 入力値が **NULL** の場合、**NULL** が返されます。

GREATEST

関数の定義

```
greatest(var1, var2, ...)
```

使用法：

最大の入力パラメータ値を返します。

パラメーターの説明：

var1/var2: Bigint 型、**Double** 型、**Decimal** 型、**Datetime** 型、**String** 型のいずれかです。 すべての値が **NULL** の場合は、**NULL** が返されます。

戻り値**:**

- 入力パラメータの最大値が返されます。 暗黙の型変換が必要ない場合、戻り値のデータ型は入力パラメータのデータ型と同じになります。
- **NULL** が最小値です。

  入力パラメータのデータ型によって、戻り値は次のようになります。

  - **Double** 型、**Bigint** 型、**Decimal** 型、**String** 型の場合は、**Double** 型に変換されます。
  - **String** 型と **Datetime** 型の場合は、**Datetime** 型に変換されます。
  - それ以外の暗黙の変換はできません。

ORDINAL

関数の定義

```
ordinal(bigint nth, var1, var2, ...)
```

使用法：

入力変数を小さい順から大きい順にソートした後、「**nth**」で指定された位置に値を返します。

パラメーターの説明：

- **nth: Bigint**型で、値を返す位置を指定します。 **NULL** の場合は **NULL** が返されます。

- **var1/var2: Bigint** 型、**Double** 型、**Decimal** 型、**Datetime** 型、**String** 型のいずれかです。

戻り値**:**

- **n** 番目のビット値です。 暗黙の型変換が必要ない場合、戻り値のデータ型は入力パラメータのデータ型と同じになります。
- 暗黙の型変換が必要な入力パラメータ値がある場合、
  - **Double** 型、**Bigint** 型、**String** 型の場合は、**Double** 型に変換されます。
  - **String** 型と **Datetime** 型の場合は、**Datetime** 型に変換されます。
  - それ以外の暗黙の変換はできません。
- **NULL** は最小値です。

例：

```
ordinal(3, 1, 3, 2, 5, 2, 4, 6) = 2
```

LEAST

関数の定義

```
least(var1, var2, ...)
```

使用法：

入力パラメータの最小値を返します。

パラメーターの説明：

var1/var2: Bigint 型、**Double** 型、**Decimal** 型、**Datetime** 型、**String** 型のいずれかです。すべての値が **NULL** の場合は、**NULL** が返されます。

戻り値**:**

- 入力パラメータの最小値が返されます。暗黙の型変換が必要ない場合、戻されるデータ型は入力パラメータのデータ型と同じになります。
- 暗黙の型変換が必要な入力パラメータ値がある場合
  - **Double** 型、**Bigint** 型、**String** 型の場合は、**Double** 型に変換されます。
  - **String** 型と **Datetime** 型の場合は、**Datetime** 型に変換されます。
  - **Decimal** 型が **Double** 型、**Bigint** 型、**String** 型と比較されるときに、**Decimal** 型に変換されます。
  - それ以外の暗黙の変換はできません。
- **NULL** は最小値です。

MAX_PT

関数の定義

```
max_pt(table_full_name)
```

使用法：

パーティションテーブルの場合、レベル 1 にあるパーティションの最大値が返されます。これはアルファベット順にソートされます。また、そのパーティションに対応するデータファイルがあります。

パラメーターの説明：

table_full_name: String 型。テーブル名を指定します。プロジェクト名と同じでなければいけません (例：**prj.src**)。 このテーブルに対する読み取り権限を所有している必要があります。

戻り値**:**

レベル 1 のパーティションの最大値が返されます。

例：

「**tbl**」はパーティションテーブルであり、そのテーブルのすべてのパーティションが次のとおりだとします。また、データファイルがあるとします。

```
pt =‘20120901’
pt =‘20120902’
```

次の文を実行すると、 `max_pt` の戻り値は、「**20120902**」になります。 **MaxCompute SQL** 文で、「**20120902**」パーティション内のデータが読み取られます。

```
select * from tbl where pt=max_pt('myproject.tbl');
```



注**:**

alter table を使用して新しいパーティションを追加したが、このパーティションにデータファイルがない場合、このパーティションは返されません。

UUID

関数の定義

```
string uuid()
```

使用法：

ランダム **ID**を返します。 例**:** `29347a88-1e57-41ae-bb68-a9edbdd94212`

 注：

UUID により、重複の可能性が低いランダムなグローバル **ID** が返されます。

SAMPLE

関数の定義

```
boolean sample(x, y, column_name)
```

使用法：

x と **y** の設定に従って **column_name** のすべての値をサンプリングし、サンプリング条件を満たさない行を除外します。

パラメーターの説明：

- **x**、**y: Bigint** 型。**x** 部分へのハッシュを示し、**y** 番目の部分が読み取られます。 **y** は省略できます。
  - **y** を省略すると、最初の部分が読み取られます。 パラメーター内で **y** を省略すると、**column_name** も同時に無視されます。
  - **x** と **y** は **Bigint** 型の定数で、**0** より大きい値です。 それ以外のデータ型の場合、または **0** 以下の場合は、例外が発生します。 **y&gt, x** の場合も、例外が発生します。 **x** と **y** のいずれかの入力値が **NULL** の場合、**NULL** が返されます。
- **column_name:** サンプリング対象の列です。
  - **column_name** は省略することができます。その場合、**x** と **y** の値に従って、ランダムにサンプルが取得されます。
  - 任意のデータ型にすることができ、列値は **NULL** にすることができます。 暗黙的な型変換は必要ありません。
  - **column_name**が **NULL** の場合、例外が報告されます。

戻り値**:**

Boolean 型。

 注：

NULL 値によるデータの偏りを避けるため、**column_name** に **NULL** 値があると、 **x** 部分で一様ハッシュが実行されます。 **column_name** が追加されていない場合、データサイズが小さいため、出力は必ずしも一様ではありません。 そのため、出力結果を向上させるには、**column_name** を追加することを推奨します。

例：

テーブル **"tbla"** が存在し、列 **"cola"** がこのテーブルにあるとします。

```
select * from tbla where sample (4, 1 , cola) = true;
-- The values are carried out Hash into 4 portions and take the first
portion.
select * from tbla where sample (4, 2) = true;
-- The values do random Hash into 4 portions for each row of data and
take the second portion.
```

CASE WHEN EXPRESSION

MaxCompute では、次に示す **2** 種類の **case when** 構文を使用できます。

```
case
when (_condition1) then result1
when (_condition2) then result2
...
else resultn
end
case
when (_condition1) then result1
when (_condition2) then result2
when (_condition3) then result3
...
else resultn
end
```

case when 式を使用することで、式の値の計算結果に応じて、柔軟に値を戻すことができます。

shop_name に基づいてリージョンを取得するための **SQL** 文は、次のとおりです。

```
select
case
when shop_name is null then 'default_region'
when shop_name like 'hang%' then 'zj_region'
end as region
From sale_detail;
```

 注：

- 結果のデータ型に **Bigint** 型と **Double** 型が含まれる場合は、**Double** 型に変換後、結果が返されます。
- 結果のデータ型に **String** 型が含まれる場合は、**String** 型に変換後、結果が返されます。 変換が失敗した場合は、エラーが報告されます **(Boolean** 型などの場合**)**。
- それ以外のデータ型の変更は実行できません。

If 式

関数の定義

```
if(testCondition, valueTrue, valueFalseOrNull)
```

使用法：

testConditionが真かどうかを判断します。 真の場合は **True** が返され、偽の場合は **valueFalse** または **Null** を返します。

パラメーターの説明：

- **testCondition:** 真偽を判定するための式です。 **Boolean** 型です。
- **valueTrue:** 式 **testCondition** が真の時に返される値です。
- **valueFalseOrNull:** 式 **testCondition** が真ではない場合、および **NULL** の可能性がある場合に返される値です。

戻り値**:**

戻り値のデータ型は、**valueTrue** または**valueFalseOrNu** のデータ**l**型と同じです。

例：

```
select if(1=2,100,200) from dual;
--Returned results:
+ ------------ +
| _c0 |
+------------+
| 200 |
+------------+
```

SPLIT

関数の定義

```
split(str, pat)
```

使用法：**STR** が **Pat** で分割された後、配列が返されます。

パラメーターの説明：

- **str: String** 型。分割する文字列を指定します。
- **pat: String** 型。区切り文字を指定します。正規表現を使用できます。

戻り値**:**

```
array <string>
```

結果は、**pat** で区切られた **str** の要素です。

例：

```
select split("a,b,c",",") from dual;
Results:
+------+
| _c0 |
+------+
| [a, b, c] |
+------+
```



注：

MaxCompute SQL および **MapReduce for MaxCompute 2.0** で利用できる **Set** コマンドは、次のとおりです。

- **SQL** 文の実行時に、**Tinyint** 型、**Smallint** 型、**Int** 型、**Float** 型、**Varchar** 型、**TIMESTAMP BINARY** 型のデータが処理の対象になる場合は、`set odps.sql.type.system.odps2=true;` を **SQL** 文の前に追加する必要があります。 **set** 文と **SQL** 文は同時に送信します。
- プロジェクトレベル: 新規データ型をプロジェクトレベルで使用可能にできます。 プロジェクトオーナーが必要に応じて、次のコマンドを使用して、プロジェクトに設定できます。

```
set odps.sql.type.system.odps2=true;
```

STR_TO_MAP

関数宣言：

```
str_to_map(text [, delimiter1 [, delimiter2]])
```

使用法： **‘delimiter1’** を使って **” text"** をキー値ペアに分割し、次に **‘delimiter2’** を使って各キー値ペアを分割します。

パラメータの説明

- **text: String** 型。分割する文字列を指定します。
- **delimiter1: String** 型の区切り文字です。指定しない場合、デフォルトは「**,**」です。
- **delimiter1: String** 型の区切り文字です。指定しない場合、デフォルトは「**,**」です。

戻り値: **map <string, string >.** 要素は、文字列 **'delimiter1'** と **'delimiter2'** によって **'text'** が分割された後のキー値の結果です。

例：

```
Select fig ('test1 &amp;amp; 1-test2 & 2 ','-','&amp;amp;');
```

以下の結果が返されます。

```
+-------------+
| A |
+-------------+
| {Test1: 1, Test2: 2} |
```

EXPLODE

関数の定義

```
explode(var)
```

使用法：

1 行のデータを複数行の UDTF に変換します。

- var が Array 型の場合、列に格納されている配列が複数行に変換されます。
- var が Map 型の場合、列に格納されているマップの各キー値のペアは、2 つの列から成る行に変換されます。1 列はキー用、もう 1 列は値用です。

パラメーターの説明：

`var: Array<T> 型または Map<K, V> 型です。`

戻り値:

変換後の行が返されます。

 注：

UDTF を使用する場合、以下の制限があります。

- 1 つの select 文では 1 つの UDTF しか指定できず、他の列は指定できません。
- group by、cluster by、distribution by、sort by と一緒には使用できません。

例：

```
explode(array(null, 'a', 'b', 'c')) col
```

MAP

関数の定義

```
MAP map(K key1, V value1, K key2, V value2, ...)
```

使用法：

特定のキーと値のペアを使って、マップを作成します。

パラメータの説明

key/value

- 暗黙的な変換後も含め、すべてのキーのデータ型は一致しており、基本のデータ型でなければなりません。
- 暗黙の型変換後の値も含めて、すべての値のデータ型は一致している必要がありますが、どの型でも構いません。

戻り値**:**

Map 型が返されます。

例：

たとえば、**"t_table"** のフィールド **(c1 Bigint** 型、**c2 String** 型、**c3 String** 型、**c4 Bigint** 型、**c5 Bigint** 型**)** に次のデータが格納されているとします。

```
+------------+----+----+------------+------------+
| c1         | c2 | c3 | c4         | c5         |
+------------+----+----+------------+------------+
| 1000       | k11 | k21 | 86         | 15         |
| 1001       | k12 | k22 | 97         | 2          |
| 1002       | k13 | k23 | 99         | 1          |
+------------+----+----+------------+------------+
```

次の **SQL** 文を実行します。

```
select  map(c2,c4,c3,c5) from t_table;
```

結果は次のとおりです。

```
+ ---- +
| _c0 |
+------+
| {k11:86, k21:15} |
| {k12:97, k22:2} |
| {k13:99, k23:1} |
```

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

MAP_KEYS

関数の定義

```
ARRAY map_keys(map<K, V>)
```

使用法：

map パラメータ内のすべてのキーの配列を返します。

パラメーターの説明：

map：**Map** 型のデータです。

戻り値**:**

Array 型が返されます。入力値が **NULL** の場合、**NULL** が返されます。

例：

たとえば、t_table_map のフィールドは (`c1 bigint,t_map map<string,bigint>`) で、データは次のとおりです。

```
+------------+-------+
| C1 | t_map |
+ ------------ + ------- +
| 1000       | {k11:86, k21:15} |
| 1001       | {k12:97, k22:2} |
| 1002       | {k13:99, k23:1} |
+------------+-------+
```

次の **SQL** 文を実行します。

```
select  c1,map_keys(t_map) from t_table_map;
```

結果は次のとおりです。

```
+------------+------+
| c1         | _c1  |
+------------+------+
| 1000       | [k11, k21] |
| 1001       | [k12, k22] |
| 1002       | [k13, k23] |
+------------+------+
```

MAP_VALUES

関数の定義**:**

```
ARRAY map_values(map<K, V>)
```

使用法：

map パラメータ内のすべての値の配列を返します。

パラメーターの説明：

map：**Map** 型のデータです。

戻り値**:**

Array 型が返されます。入力値が **NULL** の場合、**NULL** が返されます。

例：

```
select map_values(map('a',123,'b',456));
結果:
[123, 456]
```

ARRAY

関数の定義

```
ARRAY array(value1,value2, ...)
```

使用法：

特定の値を使って配列を作成します。

パラメーターの説明：

value: このパラメータは任意のデータ型にできますが、すべての値は同じデータ型にする必要があります。

戻り値**:**

Array 型が返されます。

例：

たとえば、`t_table` フィールド **(c1 Bigint** 型、**c2 String** 型、**c3 String** 型、**c4 Bigint** 型、**c5 Bigint** 型**)** に次のデータが格納されているとします。

```
+------------+----+----+------------+------------+
| c1         | c2 | c3 | c4         | c5         |
+------------+----+----+------------+------------+
| 1000       | k11 | k21 | 86         | 15         |
| 1001       | k12 | k22 | 97         | 2          |
| 1002       | k13 | k23 | 99         | 1          |
```

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

次の **SQL** 文を実行します。

```
select array(c2,c4,c3,c5) from t_table;
```

結果**:**

```
+------+
| _c0 |
+------+
| [k11, 86, k21, 15] |
| [k12, 97, k22, 2] |
| [k13, 99, k23, 1] |
+ ---- +
```

SIZE

関数の定義

```
INT size(map)
INT size(array)
```

使用法：

- `size(map<K, V>)` は、特定のマップのキー値ペア―の数を返します。
- `size(array<T>)` は、特定の配列の要素数を返します。

パラメーターの説明：

- `map<K, V>`**: Map** 型のデータです。
- `array<T>`**: Array** 型のデータです。

戻り値**:**

Int 型が返されます。

例：

```
select size(map('a',123,'b',456)) from dual;--Returns 2
select size(map('a',123,'b',456,'c',789)) from dual;--Returns 3
select size(array('a','b')) from dual;--Returns 2
select size(array(123,456,789)) from dual;--Returns 3
```

ARRAY_CONTAINS

関数の定義**:**

```
boolean array_contains(ARRAY<T> a,value v)
```

使用法：

指定された配列 **a** に **v** が含まれているかどうかを調べます。

パラメーターの説明：

- **a: Array** 型のデータです。
- **v:** 指定された **v** は、配列内のデータと同じデータ型ータ型でなければなりません。

戻り値**:**

Boolean 型が返されます。

例：

t_table_array のフィールドが (c1 bigint, t_array array<string>) で、データが次のとおりだとします。

```
+ ------------ + --------- +
| c1          | t_array |
+-------------+---------+
| 1000        | [k11, 86, k21, 15] |
| 1001        | [k12, 97, k22, 2] |
| 1002        | [k13, 99, k23, 1] |
+-------------+---------+
```

次の **SQL** 文を実行します。

```
select c1, array_contains(t_array,'1') from t_table_array;
```

結果**:**

```
+-------------+------+
| c1          | _c1  |
+-------------+------+
| 1000        | false |
| 1001        | false |
| 1002        | true |
+-------------+------+
```

SORT_ARRAY

関数の定義

```
ARRAY sort_array(ARRAY<T>)
```

使用法：

この関数は、指定された配列をソートするために使用します。

パラメーターの説明：

ARRAY<T>**: Array** 型のデータで、配列内のデータ型は、どれでも構いません。

戻り値**:**

Array 型が返されます。

例：

```
select sort_array(array('a','c','f','b')),sort_array(array(4,5,7,2,5,8
)),sort_array(array('You','Me','He')) from dual;
結果:
[a, b, c, f] [2, 4, 5, 5, 7, 8] [He, You, Me]
```

次の **SQL** 文を実行します。

```
Select sort_array (C1), sort_array (C2), sort_array (C3) from t_array;
```

以下の結果が返されます。

```
[a, b, c, f] [2, 4, 5, 5, 7, 8] [He, You, Me]
```

POSEXPLODE

関数の定義**:**

```
posexplode(ARRAY<T>)
```

使用法：

指定された配列を分けます。 各値が行に割り当てられ、各行は添字 **(0**から始まる**)** と配列要素に対応する**2**つの列から構成されます。

パラメーターの説明：

ARRAY：**Array** 型のデータ。配列内のデータは任意の型にできます。

戻り値**:**

テーブル生成関数が返されます。

例：

```
select posexplode(array('a','c','f','b')) from dual;
結果:
+------------+-----+
| pos | val |
+------------+-----+
| 0 | a |
| 1 | c |
| 2 | f |
| 3 | b |
```

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

STRUCT

関数の定義:

```
STRUCT struct(value1,value2, ...)
```

使用法：

指定された値リストを使って構造体を作成します。

パラメーターの説明：

value: 各値は任意のデータ型にできます。

戻り値:

`STRUCT<col1:T1, col2:T2, ... >` が返されます。フィールド名は、**col1, col2,** … のように順になります。

例：

```
select struct('a',123,'ture',56.90) from dual;
結果:
{col1:a, col2:123, col3:ture, col4:56.9}
```

NAMED_STRUCT

関数の定義:

```
STRUCT named_struct(string name1, T1 value1, string name2, T2 value2
, ...)
```

使用法：

指定された名前と値のリストを使って構造体を作成します。

パラメーターの説明：

- **value:** 各値は任意のデータ型にできます。
- **name: String** 型のフィールド名を指定します。

戻り値:

`STRUCT<name1:T1, name2:T2, ... >` 型が返されます。 生成される構造体のフィールド名は、**name1**、**name2**… のように連続しています。

例：

```
select named_struct('user_id',10001,'user_name','LiLei','married','F
','weight',63.50) from dual;
```

```
結果:
{user_id:10001, user_name:LiLei, married:F, weight:63.5}
```

INLINE

コマンド形式:

```
inline(array<struct<f1:T1, f2:T2, ... >>)
```

下図をご参照ください。

指定された構造体配列を展開します。 各要素には 1 行が与えられ、各構造体要素は各行の 1 列に対応します。

パラメーターの説明：

`STRUCT<f1:T1, f2:T2, ... >` :配列内の値は任意の型にすることができます。

戻り値:

テーブル生成関数が返されます。

例：

Table t_table 内のフィールド (`t_struct struct<user_id:bigint,user_name:string,married:string,weight:double>` <user_id: bigint,="" user_name:="" string,="" married:="" weight:="" double="">) の場合、テーブルデータは </user_id:> のようになります。

```
+-----------+
| T_struct |
+-----------+
{user_id:10001, user_name:LiLei, married:F, weight:63.5}
{user_id:10001, user_name:LiLei, married:F, weight:63.5}
+-----------+
```

次の SQL 文を実行します。

```
select inline(array(t_struct)) from t_table;
```

以下の結果が返されます。

```
+------------+----+----+------------+------------+
| user_id    | user_name | married | weight     |
+------------+----+----+------------+------------+
| 10001      | LiLei     | N       | 63.5       |
| 10002      | HanMeiMei | Y       | 43.5       |
```

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

TRANS_ARRAY

関数の定義:

```
trans_array (num_keys, separator, key1,key2,…,col1, col2,col3) as (
key1,key2,…,col1, col2)
```

使用法：

1 行のデータを複数行に変換し、列内の固定区切り記号形式で区切られた配列を複数行に変換する UDTF です。

パラメーターの説明：

- num_keys: Bigint 型の定数。0 以上でなければなりません。 複数行に変換するときに、キーを転置するための列番号として使用されます。
- Key: 1つの行を複数行に変換するときは、複数行にある列を複製します。
- separator: String 型の定数です。 文字列を複数の要素に分割するために使用される区切り文字です。 NULL の場合は例外が発生します。
- keys: 転置するときのキーの列として、 num_keys で指定します。 num_keys がすべての列をキーとして指定されている場合 (つまり、num_keys がすべての列の数に等しい場合)、1 行のみが返されます。
- cols: 行に変換するための配列です。 キーの後のすべての列は転置される配列と見なされます。 String 型です。 格納されているコンテンツは、「Hangzhou; Beijing; shanghai」といった String 形式の配列になります。配列は、「;」で区切られます。

戻り値:

転置行、新しい列名は as で指定されます。 キーとしての列のデータ型は変更されず、他のすべての列は、String 型になります。 分割される行数は、配列に設定された最大数によって異なります。値のないロケールは NULL で補完されます。

 注 :

UDTF を使用する場合、以下の制限があります。

- キーと見なされるすべての列を前面に配置し、転置する列を背面に配置する必要があります。
- 1 つの selct 文では1つの UDTF しか指定できず、他の列は指定できません。
- 1 つの selct 文では1つの UDTF しか指定できず、他の列は指定できません。

例：

t_table テーブルのデータは次のとおりです。

```
+----------+----------+------------+
| login_id | login_ip | login_time |
+----------+----------+------------+
wangwangA 192.168.0.1,192.168.0.2 20120101010000,20120102010000
| Wangwangb | 192.168.25.10, 192.168.67.22, 192,168.6. 3 | maid,
20120223080000 |
+----------+----------+------------+
```

次の **SQL** 文を実行します。

```
trans_array(1, ",", login_id, login_ip, login_time) as (login_id,
login_ip,login_time)
```

結果**:**

```
+----------+----------+------------+
| Login_id | login_ip | login_time |
+----------+----------+------------+
| wangwangB | 192.168.45.10 | 20120111010000 |
| wangwangB | 192.168.67.22 | 20120112010000 |
| wangwangB | 192.168.6.3 | 20120223080000 |
| wangwangA | 192.168.0.1 | 20120101010000 |
| wangwangA | 192.168.0.2 | 20120102010000 |
+----------+----------+------------+
```

テーブルに次のデータが格納されているとします。

```
Login_id LOGIN_IP LOGIN_TIME
wangwangA 192.168.0.1,192.168.0.2 20120101010000
```

NULLは、配列内で値のないロケールを補完するものです。

```
Login_id Login_ip Login_time
wangwangA 192.168.0.1 20120101010000
wangwangA 192.168.0.2 NULL
```

5.13 UDF

5.13.1 UDF の概要

MaxCompute には、ユーザーのコンピューティングに関する要求を満たす数多くの組み込み関数が実装されています。

ユーザー定義関数 **(UDF)** は、他の組み込み関数と似ています。 **Java** と **MaxCompute** のデータ型の対応関係については、「パラメーターと戻り値のデータ型」をご参照ください。

Maven をご使用の場合は、*Maven* で「**odps-sdk-udf**」を検索すると、**Java SDK** の各種バージョンを入手できます。設定は、以下のとおりです。

```
<dependency>
    <groupId>com.aliyun.odps</groupId>
    <artifactId>odps-sdk-udf</artifactId>
   <version>0.29.10-public</version>
</dependency>
```

MaxCompute では、次の **2** 種類の **UDF** を拡張できます。

UDF クラス	説明
UDF (ユーザー定義スカラー関数)	ユーザー定義スカラー関数。 入力と出力の関係は **1** 対 **1** です。 **1** 行データを読み込み、出力値を **1** つ書き出します。
UDTF (ユーザ定義テーブル値関数)	ユーザ定義テーブル値関数は、**1** つの関数を呼び出すと複数行のデータが出力されるというシナリオで使用します。 **UDF** では戻り値を **1** つしか出力できませんが、**UDTF** は複数フィールドを返すことができる独自のユーザー定義関数です。
UDAF (ユーザ定義集計関数)	ユーザー定義集計関数 (**UDAF**) の入力と出力の関係は、**1** 対多です。 複数の入力レコードが集約されて、出力値が求められます。**Group By** 句と一緒に使用できます。 詳細については、「集計関数集計関数」をご参照ください。

注：

- ユーザー定義スカラー関数、ユーザー定義集計関数、ユーザー定義テーブル値関数などのユーザー定義関数をまとめて **UDF** といいます。より狭い意味では、ユーザー定義スカラー関数を指します。ここでは、この用語を頻繁に使用しており、文脈に応じて意味を判断できます。
- **SQL** 文で指定された **UDF** がメモリ不足であることを告げるプロンプトが表示されることがあります。この問題を解決するには、`set odps.sql.udf.joiner.jvm.memory = xxxx ;` を設定します。 この原因は、データが巨大でデータの偏りも存在するためです。これにより、メモリサイズがタスクを占有し、デフォルトのメモリサイズを超えます。

MaxCompute UDFは、プロジェクト間で共有できます。 **project_b** 内の **UDF** は、**project_a** 内でも使用できます。 詳細については、「セキュリティガイド」ドキュメントの「権限付与」をご参照ください。

UDF の例

クイックスタートの「*UDF の例*」 をご参照ください。

5.13.2 Java UDF

MaxCompute UDFには、**UDF**、**UDAF**、**UDTF** の**3**つのタイプがあります。 ここでは、これら **3** つの関数を **Java** で実装する方法に焦点を当てます。

パラメーターと戻り値のデータ型

MaxCompute SQLで使用できる **UDF**のデータ型には、**Bigint**、**Double**、**Boolean**、**Datetime**、**Decimal**、**String**、**Tinyint**、**Smallint**、**Int**、**Float**、**Varchar**、**Binary**、**Timestamp** などの基本データ型と、 **Array**、 **Map**、**Struct** の複合データ型があります。

- **Java UDF** を介して **Tinyint**、**Smallint**、**Int**、**Float**、**Varchar**、**Binary**、**Timestamp** などの基本データ型を使用する方法は、次のとおりです。
  - **UDTF** では、 **@Resolve** アノテーションによって「署名」を取得します。たとえば、 `@Resolve("smallint->varchar(10)")`と記述します。
  - **UDF** ではリフレクション分析の「**evaluate**」によって「署名」が取得されます。 この場合、**MaxCompute** 組み込み型と**Java** 型は **1** 対 **1** に対応しています。
  - **UDAF** では **@Resolve** アノテーションを使用して署名が取得されます。**Maxcompute2.0** ではアノテーションで新しいデータ型を使用できます。たとえば、`@Resolve("smallint-> varchar (10 )")` と記述します。
- **JAVA UDF** では、「**Array**」、「**Map**」、「**Struct**」の **3** つのデータ型が使用されます。
  - **UDAF** と **UDTF** では、**@Resolve**アノテーションによって署名を指定します。たとえば、 `@Resolve("array<string>,struct<a1:bigint,b1:string>,string->map<string,bigint>,struct<b1:bigint>")` と記述します。
  - **UDF** では、**evaluate** メソッドの署名を介して **UDF** の入力データ型と出力データ型がマッピングさます。このとき、**Maxcompute** のデータ型と **Java** のデータ型の対応付けが参照されます。 この関係では、**Array** 型 は **java.util.List** を、**Map** 型は **java.util.Map** を、**Struct** 型は **com.aliyun.odps.data.Struct** をそれぞれマッピングします。
  - **UDAF** では **@Resolve** アノテーションを使用して署名が取得されます。**Maxcompute2.0** ではアノテーションで新しいでータ型を使用できます。たとえば、`@Resolve("smallint-> varchar (10 )")` と記述します。

    注：

    ■ **com.aliyun.odps.data.Struct** ではリフレクションからフィールド名とフィールドタイプが認識されないので、**@Resolve**アノテーションで補う必要があります。 つまり、**UDF** で **Struct** を使用するには、**@Resolve** アノテーションを **UDF** クラス

に追加します。 このアノテーションは、パラメーターのオーバーロードまたは **com.aliyun.odps.data.Struct** を含む戻り値にのみ影響します。

■ 現在、*UDF* クラスには *1* つの *@Resolve* アノテーションしか指定できません。 したがって、構造体パラメータまたは戻り値を持つ *UDF* では、オーバーロードは *1* つだけ定義できます。

次の表に、**MaxCompute** と**Java** のデータ型の関係を示します。

MaxCompute のデータ型	**Java** のデータ型
Tinyint	**java.lang.Byte**
Smallint	**java.lang.Short**
Int	**java.lang.Integer**
Bigint	**java.lang.Long**
Float	**java.lang.Float**
Double	**java.lang.Double**
Decimal	**java.math.BigDecimal**
Boolean	**java.lang.Boolean**
String	**java.lang.String**
Varchar	**com.aliyun.odps.data.Varchar**
Binary	**com.aliyun.odps.data.Binary**
Datetime	**java.util.Date**
Timestamp	**java.sql.Timestamp**
array	**java.util.List**
Map	**java.util.Map**
Struct	**com.aliyun.odps.data.Struct**

注 :

- **Java** の対応するデータ型と戻り値のデータ型はオブジェクトです。 最初の文字が大文字であることを確認してください。
- **SQL**の **NULL** 値は、**Java** の **NULL** 参照によって表されます。したがって、「**Java** プリミティブ型」は表現できないため使用できません。 **SQL** の **NULL** 値
- 「**Array**」型に対応する **Java** のデータ型は 「**list**」です。

UDF

UDFを実装するには、クラス「**com.aliyun.odps.udf.UDF**」を継承し、「**evaluate**」メソッドを適用する必要があります。「**evaluate**」メソッドは、非静的なパブリックメソッドでなければなりません。 **evaluate** メソッドのパラメーターのデータ型と戻り値のデータ型は、**SQL** では **UDF** の署名と見なされています。 つまり、ユーザーは **UDF** に複数の **evaluate** メソッドを実装できます。 **UDF** を呼び出すには、フレームワークは **UDF** によって呼び出されたパラメーターの種類に応じて、該当する **evaluate** メソッドを結びつける必要があります。

注**:** クラス名は同じで機能ロジックが異なるクラスは、異なる **jar** パッケージ内で使用する必要があります。 たとえば、**UDF (UDAF/UDTF)** で **2** つの **jar** ファイルに **com.aliyun.UserFunction.class** が含まれる場合、 **udf1** と **udf2** は、それぞれリソース **udf1.jar** と **udf2.jar** に対応します。**1** つの **SQL**文で **2** つの **UDF** が使用される場合、システムはクラスの **1** つをランダムにロードします。 これにより、**UDF** の実行動作に不整合が生じたり、コンパイルが失敗したりします。

UDF のサンプルは次のとおりです。

```
package org.alidata.odps.udf.examples;
  import com.aliyun.odps.udf.UDF;

public final class Lower extends UDF {
  Public String evaluate (string s ){
    If (Stream = NULL ){
        return null;
    }
        return s.toLowerCase();
  }
}
```

`void setup(ExecutionContext ctx)` と `void close()`によって、**UDF** は開始および終了します。

UDF の使用方法は、**MaxCompute SQL** の組み込み関数と似ています。 詳細については、「組み込み関数」をご参照ください。

その他の UDF **例**

次のコードでは、**3** つのオーバーロードを持つ **UDF** が定義されます。 **1** 番目、**2** 番目、**3** 番目のオーバーロードではそれぞれ、**Array** 型、**Map** 型、**Struct** 型のデータがパラメーターとして使用されます。 **3** 番目のオーバーロードではパラメーターや戻り値として **Struct** 型が使用されるので、 特定の **Struct** 型を指定するには、 `@Resolve` アノテーションを **UDF** クラス内に配置する必要があります。

```
@Resolve ("struct, string-> string ")
public class UdfArray extends UDF {
  public String evaluate(List vals, Long len) {
```

```
    return vals.get(len.intValue());
  }
  Public String evaluate (MAP map, string key ){
    return map.get(key);
  }
  public String evaluate(Struct struct, String key) {
  return struct.getFieldValue("a") + key;
 }
}
```

ユーザーは複合データ型を直接 **UDF** に渡すことができます。

```
create function my_index as 'UdfArray' using 'myjar.jar';
select id, my_index(array('red', 'yellow', 'green'), colorOrdinal) as
color_name from colors;
```

UDAF

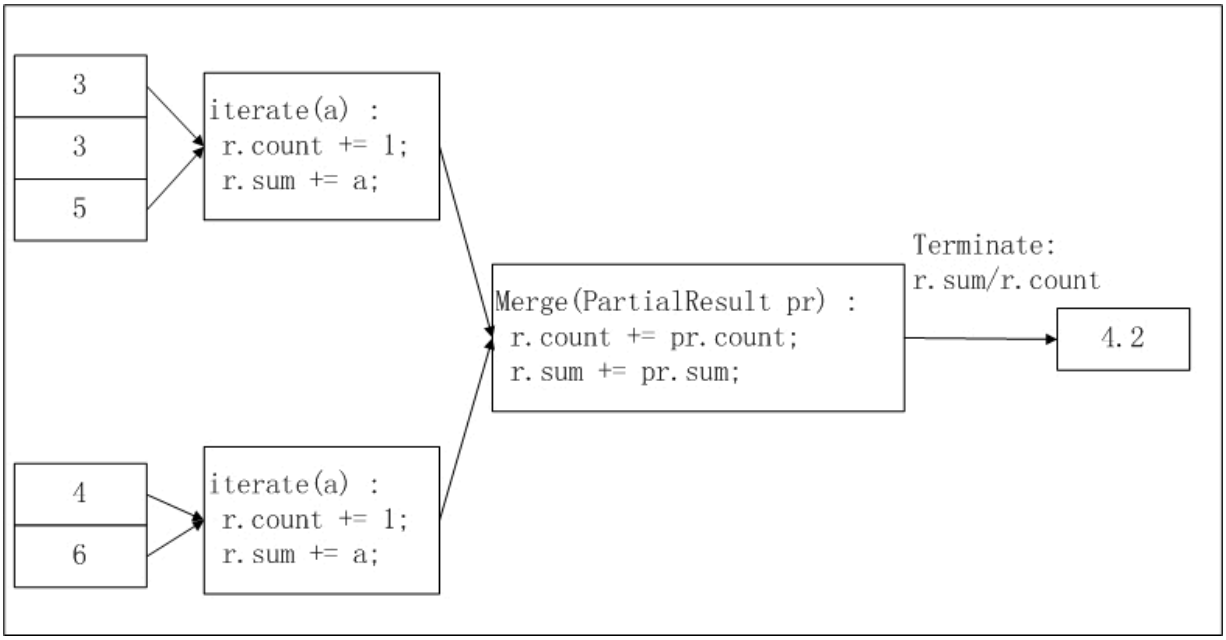
Java UDAF を実装するには、クラス「**com.aliyun.odps.udf.Aggregator**」を継承し、以下のインターフェイスを適用する必要があります。

```
public abstract class Aggregator implements ContextFunction {
  @Override
  public void setup(ExecutionContext ctx) throws UDFException {

  }
  @Override
  public void close() throws UDFException {
  }
  /**
   * Create an aggregate buffer
   * @return Writable - Aggregate buffer
   */
  abstract public Writable newBuffer();
  /**
   * @param buffer: aggregation buffer
   * @param args: specified parameter to call UDAF in SQL
   * @throws UDFException
   */
  abstract public void iterate(Writable buffer, Writable[] args)
throws UDFException;
  /**
   * generate final result
   * @param buffer
   * @return final result of Object UDAF
   * @throws UDFException
   */
  abstract public Writable terminate(Writable buffer) throws
UDFException;
  abstract public void merge(Writable buffer, Writable partial) throws
 UDFException;
}
```

最も重要な **3** つのインターフェイスは、「**iterate**」、「**merge**」、および「**terminate**」です。 **UDAF** の主なロジックは、この **3** つのインターフェイスに依存しています。 さらに、ユーザーは定義済みの書き込み可能バッファを認識する必要があります。

「平均計算」を例に挙げて、下図では **MaxCompute UDAF** でこの関数を実装するための論理手順と計算手順を説明します。



上の図では、入力データは特定のサイズに従ってスライスされています。スライスの詳細については、「*MapReduce*」をご参照ください。 各スライスのサイズは、ワーカーが指定された時間内に処理を完了するのに適しています。 このスライスサイズは、ユーザーが手動で設定する必要があります。

UDAF の計算プロセスは、**2** つのステップに分けられます。

- 第一段階で、各ワーカーはスライス内のデータ量と合計をカウントします。 中間結果として、各スライスのデータ量と合計を考慮することができます。
- 第二段階では、ワーカーが第一段階で生成された各スライスの情報を集めます。 最終出力では、**r.sum / r.count** はすべての入力データの平均になります。

平均を計算するには、次の **UDAF** エンコードの例を使用してください。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.udf.Aggregator;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.annotation.Resolve;
@Resolve("double->double")
public class AggrAvg extends Aggregator {
  private static class AvgBuffer implements Writable {
    private double sum = 0;
    private long count = 0;
    @Override
    public void write(DataOutput out) throws IOException {
      out.writeDouble(sum);
      out.writeLong(count);
```

```
    }
    @Override
    public void readFields(DataInput in) throws IOException {
      sum = in.readDouble();
      count = in.readLong();
    }
  }
  private DoubleWritable ret = new DoubleWritable();
  @Override
  public Writable newBuffer() {
    return new AvgBuffer();
  }
  @Override
  public void iterate(Writable buffer, Writable[] args) throws
UDFException {
    DoubleWritable arg = (DoubleWritable) args[0];
    AvgBuffer buf = (AvgBuffer) buffer;
    if (arg ! = null) {
      buf.count += 1;
      buf.sum += arg.get();
    }
  }
  @Override
  public Writable terminate(Writable buffer) throws UDFException {
    AvgBuffer buf = (AvgBuffer) buffer;
    if (buf.count == 0) {
      ret.set(0);
    } else{
      ret.set(buf.sum / buf.count);
    }
    return ret;
  }
  @Override
  public void merge(Writable buffer, Writable partial) throws
UDFException {
    AvgBuffer buf = (AvgBuffer) buffer;
    AvgBuffer p = (AvgBuffer) partial;
    buf.sum += p.sum;
    buf.count += p.count;
  }
}
```

 注：

- **Writable** の **readFields** 関数の場合、部分的に書き込み可能なオブジェクトを再利用できるため、同じオブジェクトである **readFields** 関数が複数回呼び出されます。 この関数では、呼び出されるたびにオブジェクト全体がリセットされることを想定しています。 オブジェクトにコレクションが含まれている場合は、空にする必要があります。
- **UDAF** の使用方法は、**MaxCompute SQL** の集計関数と同様です。 詳細については、「集計関数」をご参照ください。
- **UDTF** の実行方法は、**UDF** と同様です。 詳細については、「*Java UDF* 関数」をご参照ください。

UDTF

Java UDTF クラスは、クラス「**com.aliyun.odps.udf.UDTF**」を継承する必要があります。 このクラスには **4** つのインターフェイスがあります。

インターフェイスの定義	説明
public void setup(ExecutionContext ctx) throws UDFException	**UDTF** が入力データを処理する前に、ユーザー定義の初期化動作を呼び出すための初期化メソッドです。 「**Setup**」は、最初と各ワーカーごとに **1** 回呼び出されます。
public void process(Object[] args) throws UDFException	このメソッドは、フレームワークで呼び出されます。 **SQL** の各レコードは、それに応じて **1** 回「**process**」を呼び出します。 「**process**」のパラメーターは、**SQL**で指定される **UDTF** 入力パラメーターです。 入力パラメータは **Object []** として渡され、結果は「**forward**」関数を通して出力されます。 出力データを決定するには、ユーザー自身が「**process**」関数で「**forward**」を呼び出す必要があります。
public void close() throws UDFException	**UDTF** の終了メソッドです。 フレームワークでは、このメソッドが一度だけ呼び出されます。つまり、最後のレコードを処理した後です。
public void forward (Object …o) throws UDFException	データを出力するには、ユーザーが「**forward**」メソッドを呼び出します。 各「**forward**」はレコードの出力を表し、**SQL** の **UDTF** の「**as**」句で指定された列に対応します。

UDTF プログラムのサンプルは次のとおりです。

```
package org.alidata.odps.udtf.examples;
import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.UDTFCollector;
import com.aliyun.odps.udf.annotation.Resolve;
import com.aliyun.odps.udf.UDFException;
// TODO define input and output types, e.g., "string,string->string,
bigint".
   @Resolve("string,bigint->string,bigint")
   public class MyUDTF extends UDTF {
     @Override
     public void process(Object[] args) throws UDFException {
       String a = (String) args[0];
       Long b = (Long) args[1];
       for (String t: a.split("\\s+")) {
         forward(t, b);
       }
     }
   }
```

注：

上記の例は参考用です。 **UDTF** の実行方法は、**UDF** を使用した場合と似ています。 詳細については、「*Java UDF 開発*」をご参照ください。

この **UDTF** を **SQL** で使用した例を以下に示します。 **MaxCompute** での登録関数名が「**user_udtf**」であるとします。

```
select user_udtf(col0, col1) as (c0, c1) from my_table;
```

"my_table are" の **col0** と **col1** の値が、以下であるとします。

```
+ ------ + ------ +
| col0 | col1 |
+ ------ + ------ +
| A B | 1 |
| C D | 2 |
+ ------ + ------ +
```

「**SELECT**」を実行すると、 結果は以下のようになります。

```
+----+----+
| c0 | c1 |
+----+----+
| A | 1 |
| B | 1 |
| C | 2 |
| D | 2 |
+----+----+
```

注意事項

UDTF は多くの場合、**SQL**では次のように使用されます。

```
select user_udtf(col0, col1) as (c0, c1) from my_table;
select user_udtf(col0, col1, col2) as (c0, c1) from (select * from
my_table distribute by key sort by key) t;
select reduce_udtf(col0, col1, col2) as (c0, c1) from (select col0,
col1, col2 from (select map_udtf(a0, a1, a2, a3) as (col0, col1, col2
) from my_table) t1 distribute by col0 sort by col0, col1) t2;
```

しかし、**UDTF** を使用する場合、以下の制限があります。

- 同じ **SELECT** 句内では、他の式を使用することはできません。

```
select value, user_udtf(key) as mycol ...
```

- **UDTF** はネストできません。

```
select user_udtf1(user_udtf2(key)) as mycol...
```

- 同じ **SELECT** 句で **group by**、**distribute by**、**sort by** と一緒に使用することはできません。

```
select user_udtf(key) as mycol ... group by mycol
```

その他の UDTF **例**

UDTF では、**MaxCompute** リソースについて、詳しく学習できます。 以下では、**UDTF** を使用して **MaxCompute** リソースを読み取る方法について説明します。

1. **UDTF** プログラムをコンパイルします。 コンパイルが成功したら、**Jar** パッケージ **(udtfexample1.jar)** をエクスポートします。

```
package com.aliyun.odps.examples.udf;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Iterator;
import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.UDFException;
import com.aliyun.odps.udf.UDTF;
import com.aliyun.odps.udf.annotation.Resolve;
/**
 * project: example_project
 * table: wc_in2
 * partitions: p2=1,p1=2
 * columns: colc,colb
 */
@Resolve("string,string->string,bigint,string")
public class UDTFResource extends UDTF {
  ExecutionContext ctx;
  long fileResourceLineCount;
  long tableResource1RecordCount;
  long tableResource2RecordCount;
  @Override
  public void setup(ExecutionContext ctx) throws UDFException {
  this.ctx = ctx;
  try {
   InputStream in = ctx.readResourceFileAsStream("file_resource.txt
");
   BufferedReader br = new BufferedReader(new InputStreamReader(in
));
   String line;
   fileResourceLineCount = 0;
   while ((line = br.readLine()) ! = null) {
     fileResourceLineCount++;
   }
   br.close();
```

```
   Iterator<Object[]> iterator = ctx.readResourceTable("table_reso
urce1").iterator();
   tableResource1RecordCount = 0;
   while (iterator.hasNext()) {
     tableResource1RecordCount++;
     iterator.next();
   }
   iterator = ctx.readResourceTable("table_resource2").iterator();
   tableResource2RecordCount = 0;
   while (iterator.hasNext()) {
     tableResource2RecordCount++;
     iterator.next();
   }
 } catch (IOException e) {
   throw new UDFException(e);
 }
}
   @Override
   public void process(Object[] args) throws UDFException {
     String a = (String) args[0];
     long b = args[1] == null ? 0 : ((String) args[1]).length();
     forward(a, b, "fileResourceLineCount=" + fileResourceLineCount
 + "|tableResource1RecordCount="
     + tableResource1RecordCount + "|tableResource2RecordCount=" +
tableResource2RecordCount);
    }
}
```

2. **MaxCompute** にリソースを追加します。

```
Add file file_resource.txt;
Add jar udtfexample1.jar;
Add table table_resource1 as table_resource1;
Add table table_resource2 as table_resource2;
```

3. **MaxCompute** で **UDTF**（**my_udtf**）を作成します。

```
create function mp_udtf as com.aliyun.odps.examples.udf.UDTFResource
 using
'udtfexample1.jar, file_resource.txt, table_resource1, table_reso
urce2';
```

4. **MaxCompute** で、リソーステーブル **"table_resource1"**、**"table_resource2"**、および物理テーブル **"tmp1"** を作成します。 対応するデータをテーブルに挿入します。

5. この **UDTF** を実行します。

```
select mp_udtf("10","20") as (a, b, fileResourceLineCount) from tmp1
;
以下の結果が返されます。
+-------+------------+-------+
| a | b | fileResourceLineCount |
+-------+------------+-------+
| 10 | 2 | fileResourceLineCount=3|tableResource1RecordCount=0|
tableResource2RecordCount=0 |
| 10 | 2 | fileresourcelinecount = 3 | tableResource1RecordCount = 0
 | tableResource2RecordCount = 0 |
```

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

UDTF の例 - 複合データ型

次の例のコードにより、3 つのオーバーロードを持つ UDF が定義されます。 最初のオーバーロードでは、パラメータとして「Array」型が使用されます。 2 番目はパラメータとして「Map」型が使用されます。そして 3 番目はパラメータとして「Struct」型が使用されます。 3 番目のオーバーロードではパラメータまたは戻り値として「Struct」型が使用されるため、UDF クラスには、特定のタイプの「struct」を指定するための @Resolve アノテーションが必要です。

```
@Resolve("struct<a:bigint>,string->string")
public class UdfArray extends UDF {
  public String evaluate(List<String> vals, Long len) {
    return vals.get(len.intValue());
  }
  public String evaluate(Map<String,String> map, String key) {
    return map.get(key);
  }
  public String evaluate(Struct struct, String key) {
    return struct.getFieldValue("a") + key;
  }
}
```

ユーザーは UDF に複合データ型を渡すことができます。

```
create function my_index as 'UdfArray' using 'myjar.jar';
select id, my_index(array('red', 'yellow', 'green'), colorOrdinal) as
color_name from colors;
```

Hive UDF の互換性の例

MaxCompute 2.0 は、Hive 形式の UDF に対応しています。 Hive UDF と UDTF の中には、MaxCompute で直接使用できるものがあります。



：

現在、互換性のある Hive のバージョンは、2.1.0で、 対応する Hadoop のバージョンは 2.7.2 です。 他の Hive および Hadoop バージョンで開発した UDF は、この Hive/Hadoop バージョンを使用して再コンパイルする必要があります。

例：

```
package com.aliyun.odps.compiler.hive;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInsp
ectorFactory;
import java.util.ArrayList;
import java.util.List;
```

```
import java.util.Objects;
public class Collect extends GenericUDF {
  @Override
  public ObjectInspector initialize(ObjectInspector[] objectInspectors
) throws UDFArgumentException {
    if (objectInspectors.length == 0) {
      throw new UDFArgumentException("Collect: input args should >= 1
");
    }
    for (int i = 1; i < objectInspectors.length; i++) {
      if (objectInspectors[i] ! = objectInspectors[0]) {
        throw new UDFArgumentException("Collect: input oi should be
the same for all args");
      }
    }
    return ObjectInspectorFactory.getStandardListObjectInspector(
objectInspectors[0]);
  }
  @Override
  public Object evaluate(DeferredObject[] deferredObjects) throws
HiveException {
    List<Object> objectList = new ArrayList<>(deferredObjects.length);
    for (DeferredObject deferredObject : deferredObjects) {
      objectList.add(deferredObject.get());
    }
    return objectList;
  }
  @Override
  public String getDisplayString(String[] strings) {
    return "Collect";
  }
}
```

 注：

Hive UDF の使用については、以下をご参照ください。

- *https://cwiki.apache.org/confluence/display/Hive/HivePlugins*
- *https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide+UDTF*
- *https://cwiki.apache.org/confluence/display/Hive/GenericUDAFCaseStudy*

UDF では任意のタイプと数のパラメーターを配列にパッケージ化し、出力できます。出力 **jar** パッケージの名前が **test.jar** だとします。

```
--リソースの追加
Add jar test.jar;
--関数の作成
CREATE FUNCTION hive_collect as 'com.aliyun.odps.compiler.hive.Collect
' using 'test.jar';
--関数の使用
set odps.sql.hive.compatible=true;
select hive_collect(4y,5y,6y) from dual;
+------+
| _c0 |
+------+
| [4, 5, 6] |
```

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



注：

UDF では、 **Array**、**Map**、**Struct** 、その他の複合データ型を含むすべてのデータ型を使用できます。

注：

- **MaxCompute** の **add jar** コマンドにより、プロジェクト内にリソースが恒久的に作成されます。また、**UDF** の作成時に **jar** を指定しますが、すべての **jar** を自動的にクラスパスに追加することはできません。
- 互換性のある **Hive UDF** を使用するには、 `set odps.sql.hive.compatible = true;` を **SQL** 文の前に追加し、**SQL** 文と一緒に送信します。
- 互換性のある **Hive UDF** を使用する場合は、 **MaxCompute** の *JAVA* サンドボックスの制限にご注意ください。

5.14 UDT

5.14.1 UDT

MaxCompute に新世代の **SQL** エンジンを基盤としたユーザー定義型 **(UDT)** が導入されました。 **UDT** を使用すると、**SQL** 文でサードパーティ言語のクラスやオブジェクトを参照してデータを取得したり、メソッドを呼び出すことができます。

シナリオ

UDT は通常、以下のシナリオで適用されます。

- シナリオ **1: MaxCompute** には、他の言語を使用して簡単にタスクを実行するための組み込み関数がありません。 たとえば、組み込み **Java** クラスを **1** 回呼び出すだけで実行できるタスクがあります。 ユーザー定義関数 **(UDF)** を使用して、こうしたタスクを実行する手順は複雑です。
- シナリオ **2: SQL** 文でサードパーティ製ライブラリを呼び出して、対応する関数を実装する必要があります。 関数を **UDF** 内にラップするのではなく、サードパーティのライブラリで提供される関数を直接 **SQL** 文で使用します。
- シナリオ **3: Select Transform** を使用すると、オブジェクトとクラスを **SQL** 文に含めて、**SQL** 文を読みやすく、かつ保守しやすくできます。 **Java** などの一部の言語では、ソースコードはコンパイル後にしか実行できません。 **SQL** 文でこれらの言語のオブジェクトとクラスを参照するとします。

概要

UDT を使用すると、SQL 文でサードパーティ言語のクラスやオブジェクトを参照してデータを取得したり、メソッドを呼び出すことができます。

MaxCompute がサポートする UDT と他の SQL エンジンとの間には大きな違いがあります。他の SQL エンジンでサポートされている UDT は、MaxCompute の構造体複合型に似ています。 Oracle データベースの CREATE TYPE 文など、一部の独自の SQL 言語には、サードパーティのライブラリを呼び出すことができる機能があります。 MaxCompute で使用できる UDT は、CREATE TYPE 文に似ています。 UDT にはフィールドとメソッドの両方が含まれています。 さらに、MaxComputeでは、データ定義言語 (DDL) 文を使用して型マッピングを定義する必要はありません。 代わりに、MaxComputeを使用すると、SQL文で型を直接参照できます。

例：

```
set odps.sql.type.system.odps2=true;
-- MaxCompute でサポートされる新規データ型を有効化します The following
example will use a new type of Integer (int).
SELECT java.lang.Integer.MAX_VALUE;
```

出力は次のとおりです。

```
+------------+
| max_value |
+------------+
| 2147483647 |
+------------+
```

Java と同じ方法で java.lang を削除することで文を短縮できます。

```
set odps.sql.type.system.odps2=true;
SELECT Integer.MAX_VALUE;
```

前述の SELECT 文の式は、Java 式に似ており、Java と同じ方法で実行されます。 式により、MaxCompute で UDT が指定されます。

UDF を使用すると、UDT によって提供されるすべての関数を実装できます。 UDF を使用して同じ関数を実装する場合は、次の手順に従う必要があります。

1. UDF クラスを定義します。

```
package com.aliyun.odps.test;
public class IntegerMaxValue extends com.aliyun.odps.udf.UDF {
public Integer evaluate() {
 return Integer.MAX_VALUE;
}
```

```
}
```

2. **UDF** をコンパイルして、**JAR** パッケージに含めます。 **JAR** パッケージをアップロードして、関数を作成します。

```
add jar odps-test.jar;
「odps-test.jar」を使用して、関数 integer_max_value を「com.aliyun.odps.
test.IntegerMaxValue」として作成します。
```

3. **SQL** 文で関数を呼び出します。

```
select integer_max_value();
```

UDT を使用すると、この手順が簡単になります。 **UDT** を使用すると、他の言語で提供される外部関数をより簡単に **SQL** 文で使用できます。

UDT のしくみ

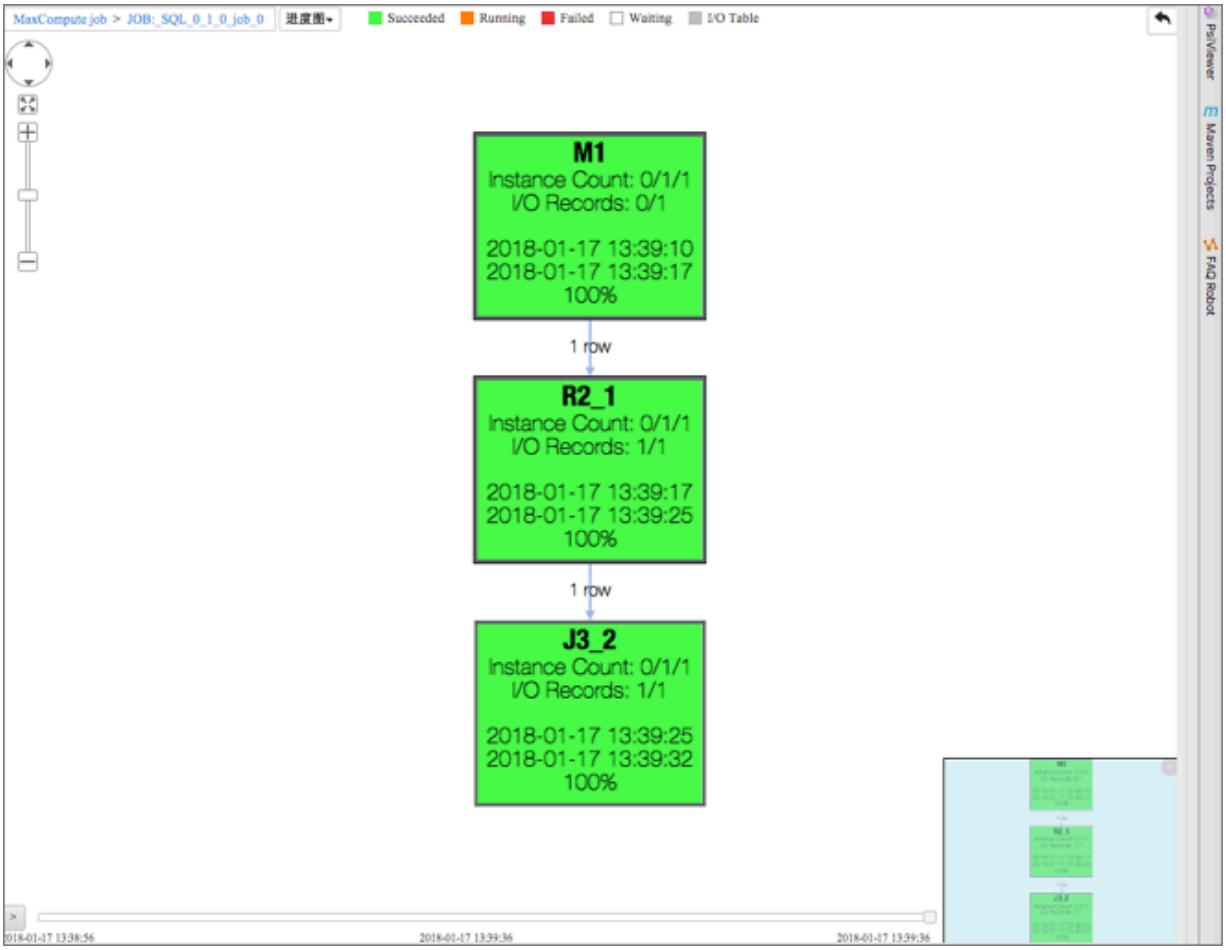
この例は、**UDT** を使用して **Java** の静的フィールドにアクセスする方法を紹介します。 **UDT** を使用して、多くの関数を実装できます。 次の例は、**UDT** の実行手順とその関数を示しています。

```
--  サンプルデータ
@table1 := select * from values ('10000000000000000000') as t(x);
@table2 := select * from values (100L) as t(y);
--  コードのロジック
@a := select new java.math.BigInteger(x) x from @table1;
 -- 新規オブジェクトの作成
@b := select java.math.BigInteger.valueOf(y) y from @table2;
 -- 静的メソッドの呼び出し
select /*+mapjoin(b)*/ x.add(y).toString() from @a a join @b b;
 -- インスタンスメソッドの呼び出し
```

出力は次の通りです。

```
100000000000000000100
```

この例では、サブクエリを使用して **UDT** 列を検索する方法も示しています。これは、**UDF** を使用して実行するのは難しいです。 変数 **a** によって取得される **x** 列は**java.math.BigInteger** 型です。 組み込み型ではありません。 **UDT** データを別の演算子に渡してからそのメソッドを呼び出すことができます。 データシャッフルで **UDT** データを使用することもできます。



この図は、**UDT** に **M1**、**R2**、**J3** の **3** つのステージ、 があることを示しています。 **Join** 句を使用すると、**MapReduce** と同じデータの入れ替えが必要になります。 データは複数のステージで処理されます。 通常、異なるステージでのデータ処理は、異なるプロセスまたは異なる物理マシンで実行されます。

M1 ステージでは、新しい **java.math.BigInteger(x)** メソッドのみが呼び出されます。

java.math.BigInteger.valueOff(y) メソッドと**x.add(y).toString()** メソッドは、**J3** ステージで別々に呼び出されます。 これらのメソッドは、さまざまなプロセスまたは物理マシンのさまざまなステージで呼び出されます。 **UDT** ではこれらの段階がカプセル化されて、**JVM** として機能します。

特徴

- **UDT** は現在 **Java** にのみ対応しています。 他の言語については、今後のバージョンでサポートの対象になる予定です。

- **UDT** では、**JAR** パッケージをアップロードして直接パッケージを参照することもできます。**UDT** でフラグが使用できるようになりました。
  - 参照する必要のあるリソースを **1** つ以上指定するには、`set odps.sql.session.resources` を使用します。このとき、リソースはカンマ **(,)** で区切ります。 例: `set odps.sql.session.resources=foo.sh,bar.txt;`.

     注:

    このフラグは、 **Select Transform** のリソース設定フラグと同じです。 したがって、このフラグは **2** つの関数を制御します。 たとえば、「概要」で説明したように、**UDF JAR** パッケージを参照するために **UDT** を使用できます。

    ```
    set odps.sql.type.system.odps2=true;
    set odps.sql.session.resources=odps-test.jar;
    --JAR パッケージを参照するには、まず対応するプロジェクトにパッケージをアップロードし、それが JAR 型のリソースであることを確認する必要があります。
    select new com.aliyun.odps.test.IntegerMaxValue().evaluate();
    ```

  - **1** つ以上のデフォルト **JAR** パッケージを指定するには、`odps.sql.session.java.imports` を使用します。このとき、コンマ **(,)** で区切って指定します。 これは **Java** の **import** 文に似ています。 `java.math.BigInteger`などのクラスパスを指定できます。あるいは、 `*`を使用することもできます。 現在、`static import` は使用できません。

    たとえば、「概要」で説明したように、**UDF JAR** パッケージを参照するために **UDT** を使用できます。

    ```
    set odps.sql.type.system.odps2=true;
    set odps.sql.session.resources=odps-test.jar;
    set odps.sql.session.java.imports=com.aliyun.odps.test. *;
    -- デフォルトの JAR パッケージを指定します
    select new IntegerMaxValue().evaluate();
    ```

- **UDT** では次の操作を実行できます。
  - **new** 演算子を使用してオブジェクトをインスタンス化します。
  - **new** 演算子を使用して、**ArrayList** の初期化など、配列をインスタンス化します。 例: **new Integer[] { 1, 2, 3 }.**
  - 静的メソッドを含むメソッドを呼び出します。 ファクトリパターンでオブジェクトを作成できます。
  - 静的フィールドを始めとしたフィールドにアクセスします。

     注:

    - パブリックメソッドとパブリックフィールドのみがサポート対象です。

- **UDT** の識別子には、パッケージ名、クラス名、メソッド名、およびフィールド名があります。 すべての識別子は、大文字と小文字が区別されます。
- **UDT** は型変換と **SQL** 構文に対応しています。たとえば、**cast (1 as java.lang.Object)** です。 **UDT**は、**(Object)1** のように、**Java** 構文を使った型変換に対応していません。
- 匿名クラスとラムダ式には対応していません。 今後のバージョンで、サポート対象になる可能性があります。
- 通常、**UDT** は式の中で使用されます。 値を返さない関数は、式の中では呼び出せません。 したがって、現在 **UDT** では、値を返さない呼び出し関数を使用できません。 この問題は、今後のバージョンで解決される予定です。

・ デフォルトでは、**Java SDK** で提供されるすべてのクラスを参照できます。

 注：

ランタイムでは現在 **JDK 1.8** が使用されています。 今後のバージョンではサポート対象外になる可能性があります。

・ すべての演算子では、**MaxCompute SQL**セマンティクスが使用されます。 `String.valueOf(1) + String.valueOf(2)` の結果は **3** です。 **2** つの文字列は暗黙的に **Double** 型の値に変換されて合計されます。 **Java** 文字列連結を使用して文字列をマージすると、結果は **12** になります。

MaxCompute および **Java** の文字列連結メソッドに加えて、等価 (=) 演算子についても混乱が生じる可能性があります。 **SQL** 文の等号 (=) 演算子は、比較演算子として使用されます。 **2** つの **Java** オブジェクトを比較するには、**equals** メソッドを呼び出す必要があります。 **2**つのオブジェクトが等しいかどうかを検証するために、等号 (=) 演算子を使用することはできません。 **UDT** が使用されている場合、あるオブジェクトが別のオブジェクトと等しいことを保証することはできません。 詳細については、以下の説明をご参照ください。

- **Java** データ型は、組み込みデータ型に対応付けられます。 詳細は、「*Java UDF*」にあるデータ型の対応付けの表をご参照ください。 対応表は、**UDT** に適用できます。
  - 組み込み型データでは、その組み込み型と対応付けられている **Java** 型のメソッドを直接呼び出すことができます。 例: `'123'.length() , 1L.hashCode()`.
  - **UDT** は、組み込み関数と **UDF** で使用できます。 たとえば、 `chr(Long.valueOf('100'))`では、`Long.valueOf` によって、`java.lang.Long`型の値が返されます。 組み込み関数 **chr** は、組み込み型の **Bigint** に対応しています。
  - **Java** プリミティブ型データは、自動的にボクシング型データに変換されます。この場合は、前述の **2** つの規則を適用できます。

  > 注:
  > 特定の組み込み型の新規データ型の場合、 `set odps.sql.type.system.odps2=true;` 文を追加して、新規データ型を宣言する必要があります。 そうしないと、エラーが発生します。

- **UDT** は **Java** ジェネリックを完全にサポートしています。 たとえば、パラメーターのデータ型に基づいて、コンパイラは `java.util.Arrays.asList(new java.math.BigInteger('1'))`メソッドによって返される値のデータ型が`java.util.List<java.math.BigInteger>`であることを認識します。

  > 注:
  > **type** パラメーターを構成関数に設定するか、**java.lang.Object** を使用する必要があります。これは **Java** の場合と同じです。 たとえば、 `new java.util.ArrayList(java.util.Arrays.asList('1', '2'))` の結果は、`java.util.ArrayList<Object>`になります。 `new java.util.ArrayList<String>(java.util.Arrays.asList('1', '2'))` の結果は、`java.util.ArrayList<String>`になります。

- **UDT** では、同等のオブジェクトが明確に定義されません。 これはデータの入れ替えが原因です。 結合の例は、オブジェクトが異なるプロセス間または物理マシン間で伝送される可能性があることを示しています。 オブジェクトは送信中に **2** つの異なるオブジェクトとして参照

されることがあります。 たとえば、オブジェクトが 2 台のマシンにシャッフルされてから再シャッフルされることがあります。

したがって、UDT を使用するときは、等号 (=) 演算子を使用するのではなく、equals メソッドを使用して 2 つのオブジェクトを比較する必要があります。

行または列内のオブジェクト間の相関関係は保証されます。 ただし、異なる行または列にあるオブジェクト間の相関は保証されません。

- 現在、join、group by、distribute by、sort by、order by、cluster by の各句で UDT をシャッフルキーとして使用することはできません。

  UDT は式のどの段階でも使用できますが、最終結果として出力することはできません。 たとえば、`group by new java.math.BigInteger('123')`メソッドは呼び出せませんが、その代わりに、 `group by new java.math.BigInteger('123').hashCode()` メソッドを呼び出すことはできます。 これは、hashCode によって返される値がint.class 型であるためです。これは、組み込み型の Iint として使用できます。 これは組み込み型を Java 型マッピング規則に適用します。

- UDT により、データ型の変換規則がより柔軟になりました。

  - UDT オブジェクトは、暗黙的な変換によって基本クラスのオブジェクトに変換されます。
  - UDT オブジェクトは、基本クラスまたはサブクラスのオブジェクトに強制的に変換できます。
  - 継承なしの 2 つのオブジェクトに対するデータ型変換では、ネイティブの変換規則が適用されます。 変換によりデータが変わる可能性があります。 たとえば、java.lang.Long 型のデータをjava.lang.Integer 型のデータに強制的に変換できます。 これは、組み込みの Bigint 型データを Int 型データに変換する処理です。 この変換処理により、データが変更されたり、データの精度が変わることがあります。

- 現在、UDT オブジェクトは保存できません。つまり、UDT オブジェクトをテーブルに追加することはできません。 DDL は UDT に対応していません。 暗黙の型変換を使用してデータを組み込み型に変換しない限り、UDT を含むテーブルを作成することはできません。 さらに、最終出力は UDT にはできません。 すべての Java クラスが toString() メソッドに対応しているため、この問題を解決するには、toString() を呼び出して、UDT データを java.lang.String 型のデータに変換できます。 このメソッドを使用して、デバッグ中に UDT データを確認できます。

  また、`set odps.sql.udt.display.tostring=true;` 文を追加すると、MaxCompute でデバッグを実行するために`java.util.Objects.toString(...)` メソッドを呼び出す

と、すべての **UDT** データを **String** 型に変更できます。 このフラグは通常 **print** 文にしか適用できないため、デバッグに使用されます。 **insert** 文には適用できません。

Binary は組み込み型で、自動シリアル化をサポートしています。 その後、**byte []** 配列を保存できます。 保存した **byte []** 配列は、**Binary** 型に逆シリアル化できます。

いくつかのクラスには、**protobuffer** のような独自のシリアル化メソッドと逆シリアル化メソッドをもっている場合があります。 **UDT** を保存するには、データをバイナリデータに変換するためのシリアル化メソッドと逆シリアル化メソッドを呼び出す必要があります。

- **UDT**を使用して、スカラー関数で提供される機能を実行できます。 組み込み関数*collect list* と *explode* を使用すると、集計関数とテーブル関数で提供される機能を実行できます。

UDT の例

- **Java** 配列の使用例

```
set odps.sql.type.system.odps2=true;
set odps.sql.udt.display.tostring=true;
select
    new Integer[10],    --  10 の要素を含む配列を作成します。
    new Integer[] {c1, c2, c3},  -- ArrayList の初期化により、3 の要素を
含む配列を作成します。
    new Integer[][] { new Integer[] {c1, c2}, new Integer[] {c3, c4
} },  -- マルチディメンション配列を作成します。
    new Integer[] {c1, c2, c3} [2], -- インデックスを使用して配列の要素に
アクセスします。
    java.util.Arrays.asList(c1, c2, c3);    -- 組み込み配列を作成するた
めの別の方法です。 List<Integer> を作成します。array<int> として使用されま
す。
from values (1,2,3,4) as t(c1, c2, c3, c4);
```

- **JSON** の使用例

  UDT のランタイムは、**JSON** で直接使用できる**JSON** 依存関係（バージョン**2.2.4**）を持っています。

```
set odps.sql.type.system.odps2=true;
set odps.sql.session.java.imports=java.util.*,java,com.google.gson.
 *; -- 複数のパッケージをインポートするには、パッケージをカンマ (,) で区切りま
す。
@a := select new Gson() gson;   --  gson オブジェクトを作成します。
select
gson.toJson(new ArrayList<Integer>(Arrays.asList(1, 2, 3))), -- オブ
ジェクトを JSON 文字列に変換します。
cast(gson.fromJson('["a","b","c"]', List.class) as List<String>) --
JSON 文字列を逆シリアル化します。 また、gson は逆シリアル化した結果を List<
Object> 型から List<String> 型へ強制的に変換します。
```

```
from @a;
```

組み込み関数 *get_json_object*と比較すると、このメソッドは簡単です。**JSON** 文字列からコンテンツを抽出してから文字列をサポート対象のデータ型に逆シリアル化されるので、効率が向上します。

JSON の依存関係に加えて、**MaxCompute** ランタイムは **commons-logging (1.1.1)**、**commons-lang (2.5)**、**commons-io (2.4)**、および **protobuf-java (2.4.1)** などの他の依存関係も持っています。

- 複合型の使用例

組み込み型の **Array** と**Map** は、それぞれ **java.util.List** と **java.util.Map** に対応付けられます。 結果**:**

- **java.util.List** や **java.util.Map** インターフェイスを呼び出すクラス内の **Java** オブジェクトは、**MaxCompute SQL** 複合型データ処理で使用できます。
- **MaxCompute** の **Array** 型と **Map** 型データは、**List** または **Map** インターフェイスを直接呼び出すことができます。

```
set odps.sql.type.system.odps2=true;
set odps.sql.session.java.imports=java.util.*;
select
    size(new ArrayList<Integer>()),         -- 組み込み関数 size を呼び出し、ArrayList のサイズを取得します。
    array(1,2,3).size(),                    -- 組み込み型配列の List メソッドを呼び出します。
    sort_array(new ArrayList<Integer>()),   -- ArrayList 内のデータをソートします。
    al[1],                                  -- Java List メソッドは、インデックスをサポートしていません。 ただし、配列型はインデックスをサポートしています。
    Objects.toString(a),          -- このメソッドにより、配列型を文字列型データに変換できるようになります。
    array(1,2,3).subList(1, 2)              -- サブリストを取得します。
from (select new ArrayList<Integer>(array(1,2,3)) as al, array(1,2,3) as a) t;
```

- 集計の例

UDT を使用して集計を行うには、まず組み込み関数 *collect_set* または *collect_list* を使用して、 データを **List** 型に変換してから、データを集計するための **UDT**メソッドを呼び出します。

次の例は、**BigInteger** データから中央値を取得する方法を示しています。 データは **java.math.BigInteger** 型なので、組み込み型の*median* 関数を直接呼び出すことはできません。

```
set odps.sql.session.java.imports=java.math.*;
@test_data := select * from values (1),(2),(3),(5) as t(value);
```

```
@a := select collect_list(new BigInteger(value)) values from @
test_data;  -- データをリストに集計します。
@b := select sort_array(values) as values, values.size() cnt from @a
;  -- 平均を取得するには、まずデータをソートします。
@c := select if(cnt % 2 == 1, new BigDecimal(values[cnt div 2]), new
 BigDecimal(values[cnt div 2 - 1].add(values[cnt div 2])).divide(new
 BigDecimal(2))) med from @b;
-- 最終す出力
select med.toString() from @c;
```

collect_list 関数ではすべてのデータが集計されるため、**collect_list** 関数を使用して部分集計することはできません。 組み込み型の集計関数または **UDAF** オブジェクトを使用する方が効率的です。 可能であれば、組み込み型の集計関数を使用することを推奨します。 すべてのデータを **1** つのグループに集約すると、データスキューのリスクが高まります。

UDAF オブジェクトのロジックで、すべてのデータが集約される場合 **(**組み込み関数 **wm_concat**と同じ**)**、**collect_list** 関数を使用するほうが **UDAF** オブジェクトを使用するよりも効率的です。

・ テーブル値関数の使用例

テーブル値関数を使用すると、複数の行と列を入出力できます。 複数の行と列を入力または出力するには、次の手順に従います。

1. 複数の行または列の入力方法は、集計関数の使用例をご参照ください。
2. 複数行を出力するには、**UDT** を使用して**Collection** 型 **(List** または **Map)** を定義し、次に **explode** 関数を呼び出してコレクションを複数行に分割します。
3. **UDT** には複数のフィールドを含めることができます。 さまざまな取得メソッドを呼び出すことで、フィールドからデータを取得できます。 データは複数行に出力されます。

次の例では、**JSON** 文字列を分割し、分割結果を複数の列に出力する方法を紹介します。

```
@a := select '[{"a":"1","b":"2"},{"a":"1","b":"2"}]' str; -- Sample
data
@b := select new com.google.gson.Gson().fromJson(str, java.util.List
.class) l from @a; -- JSON 文字列を逆シリアル化します。
@c := select cast(e as java.util.Map<Object,Object>) m from @b
lateral view explode(l) t as e;  -- explode 関数を呼び出して、文字列を分
割します。
@d := select m.get('a') as a, m.get('b') as b from @c; -- 複数の列に分
割した結果を出力します。
select a.toString() a, b.toString() b from @d; -- 最終結果 変数 d の列
a と b はオブジェクト型です。
```

機能、パフォーマンス、セキュリティ

UDT には次の機能が備わっています：

・ アクセシビリティが優れているため、 他の関数を定義する必要はありません。

- **JDK** でサポートされているすべての機能を直接使用できます。 これにより、**SQL** の柔軟性が向上します。
- **SQL** 文で他の言語のオブジェクトおよびクラスを直接参照できるので、管理が簡単です。
- 他の言語のライブラリを直接参照できます。 これにより、コードを再利用できます
- オブジェクト指向の概念に基づいた関数を実現できます。

後で **UDT** に加えられる改善点は、次のとおりです。

- **UDT** では、値を返さない関数を使用できます。たとえば、**List** インターフェイスで使用できる **add** メソッドなど、値を返すが、返された値は無視されて、渡されたデータが使用されます。 このメソッドは、渡したリストを返します。
- 匿名クラスとラムダ式には対応していません。
- **UDT** をシャッフルキーとして使用できます。
- **Python** など、サポート対象の言語が今後増える予定です。

UDT の実行手順は **UDF** と似ています。 **UDT** と **UDF** のパフォーマンスはほぼ同じです。 さらに、計算エンジンが大幅に改善されました。 したがって、**UDT** は特定のシナリオでより高いパフォーマンスを発揮します。

- 逆シリアル化は、**1** つのプロセス内のオブジェクトに対してのみ必要です。 逆シリアル化は、オブジェクトがプロセス間で伝送される場合にのみ必要です。 つまり、結合または集計関数の呼び出しなど、データの入れ替えが行われない場合、**UDT** はシリアル化または逆シリアル化のオーバーヘッドを一切発生させません。
- **UDT** のランタイムは **Codegen** に基づいており、 リフレクションに基づいていません。 したがって、パフォーマンスの低下は発生しません。 複数の**UDT**が**1**つの **FunctionCall** にラップされて同時に実行されます。 たとえば、 `values[x].add(values[y]).divide(java.math.BigInteger.valueOf(2))` で複数の **UDT** メソッドが呼び出されると考えるかもしれませんが、実際には **1** つのメソッドしか呼び出されません。 **UDT** は小粒度データ処理に焦点を当てています。 ただし、これにより、複数の関数が呼び出されるインターフェイスに追加のオーバーヘッドが発生することはありません。

UDT は **UDF**と同様に、*Java* サンドボックスモデルによって制限されています。 制限された操作を実行するには、サンドボックスの分離を有効にするか、サンドボックスホワイトリストへの登録を申請する必要があります。

5.15 UDJ

MaxCompute 2.0 コンピューティングエンジンを基盤とした新しいインターフェイスとなるユーザー定義結合 **(UDJ)** が、**MaxCompute** のユーザー定義関数 **(UDF)** のフレームワークに導

入されました。 このインターフェイスを使用すると、ユーザー定義メソッドに基づいて、複数のテーブルをより柔軟に処理できます。 また、**MapReduce** を使用することで、基盤となる分散システムでの実行操作も簡単になります。 これは、**NewSQL** をベースにした **MaxCompute** のビッグデータ処理における 大きな改善点です。

概要

MaxComputeでは、**INNER JOIN**、**RIGHT JOIN**、**OUTER JOIN**、**LEFT JOIN**、**FULL JOIN**、**SEMIJOIN**、**ANTISEMIJOIN**などの複数のネイティブ *Join* メソッドを使用できます。 ほとんどの場合、これらのネイティブな **join** メソッドを利用できます。 ただし、これらのメソッドでは、複数テーブルを処理できません。

ほとんどの場合、**UDF** を使用してコードフレームワークを構築できますが、 現在の **UDF**、**UDTF**、**UDAF** フレームワークは、**1** つのテーブルの処理にのみ対応しています。 複数のテーブルに対してユーザー定義操作を実行するには、ネイティブの **join** メソッド、**UDF**、**UDTF**、および複雑な **SQL** 文を使用する必要があります。 場合によっては、複数のテーブルを処理するときに、必要なコンピューティングタスクを完了するために、**SQL** の代わりにカスタム **MapReduce** フレームワークを使用する必要があります。

どのような場合でも、これらの操作には技術的専門知識が必要であり、以下の問題を引き起こす可能性があります。

- コンピューティングプラットフォームでは、**SQL** 文で複数の **join** メソッドを呼び出すと、「ブラックボックス」が発生する可能性があります。
- **MapReduce** を使用すると、コードの最適な実行も不可能になります。 **MapReduce** コードの大部分は **Java** で書かれています。 **MapReduce** コードの実行は、最適化されたネイティブランタイムで **LLVM** コードジェネレータによって生成された **MaxCompute** コードを実行するよりも非効率的です。

例

次の例では、**MaxCompute** で **UDJ** を使用する方法について説明します。 この例では、支払いテーブルと **"user_client_log"** テーブルを使用しています。

- 支払いテーブル **(user_id string,time datetime,pay_info string)** には、ユーザーの支払い情報が格納されます。 各支払いレコードには、ユーザー**ID**、支払い時間、および支払いの詳細が含まれます。
- **user_client_log (user_id string,time datetime,content string)** テーブルには、ユーザーのクライアントログレコードが格納されます。 各レコードには、ユーザー**ID**、操作時間、および操作データが含まれています。

前提条件: **"user_client_log"** テーブル内の各レコードを対象に、操作時間に最も近い時間を持つ支払いレコードを見つけて、両方のレコードの内容を結合して出力します。

サンプルデータは次のとおりです。

payment

user_id	time	pay_info
2656199	2018-02-13 22:30:00	gZhvdySOQb
8881237	2018-02-13 08:30:00	pYvotuLDIT
8881237	2018-02-13 10:32:00	KBuMzRpsko

user_client_log

user_id	time	content
8881237	2018-02-13 00:30:00	click MpkvilgWSmhUuPn
8881237	2018-02-13 06:14:00	click OkTYNUHMqZzlDyL
8881237	2018-02-13 10:30:00	click OkTYNUHMqZzlDyL

以下は、**"user_client_log"** テーブルのレコードです。

user_id	time	content
8881237	2018-02-13 00:30:00	click MpkvilgWSmhUuPn

次の支払いレコードには、前のクライアントログレコードの操作時間に最も近い時間データが含まれます。

user_id	time	pay_info
8881237	2018-02-13 08:30:00	pYvotuLDIT

これら **2** つのレコードは、次のようにマージされます。

8881237	2018-02-13 00:30:00	click MpkvilgWSmhUuPn, pay pYvotuLDIT

2 つのテーブルのマージ結果は次のとおりです。

user_id	time	content
8881237	2018-02-13 00:30:00	click MpkvilgWSmhUuPn, pay pYvotuLDIT

user_id	time	content
8881237	2018-02-13 06:14:00	click OkTYNUHMqZzlDyL, pay pYvotuLDIT
8881237	2018-02-13 10:30:00	click OkTYNUHMqZzlDyL, pay KBuMzRpsko

ネイティブ結合操作の呼び出し

標準の **Join** メソッドを使用する場合は、共通の **user_id** フィールドに基づいてこれら **2** つのテーブルを結合する必要があります。 各クライアントログレコードで、操作時間に最も近い支払いレコードを見つけなければなりません。 **SQL** 文は次のように記述できます。

```
SELECT
  p.user_id,
  p.time,
  merge(p.pay_info, u.content)
FROM
  payment p RIGHT OUTER JOIN user_client_log u
ON p.user_id = u.user_id and abs(p.time - u.time) = min(abs(p.time - u
.time))
```

テーブル内の **2** つの行を結合するときは、同じ **user_id** の下で、**p.time** と **u.time** の最小差を計算する必要があります。 ただし、結合条件の中で集計関数を呼び出すことはできません。 したがって、このタスクは標準の **join** メソッドを呼び出しても実行できません。

分散システムでは、**join** メソッドは、これらのテーブルによって共有されているフィールドに基づいて、**2** つ以上のテーブルから取得された行をマージします。 標準 **SQL**で **join** メソッドを使用する場合、マージされたデータを処理するための選択肢はいくつかあります。 したがって、**UDJ** などの汎用インターフェイスは、カスタマイズされた方法でマージされたデータを処理し、結果を出力するために必要です。 このインターフェイスはプラグインとして統合することができます。

Java を使用した UDJ コードの記述

以下では、**UDJ** を使用してネイティブの **join** メソッドを呼び出しても実装できない機能を実行する方法を説明します。

前提条件

UDJ は新機能なので、新しい **SDK** が必要です。

```
<dependency>
  <groupId>com.aliyun.odps</groupId>
  <artifactId>odps-sdk-udf</artifactId>
  <version>0.29.10-public</version>
  <scope>provided</scope>
```

```
</dependency>
```

SDK には新しい抽象クラス **UDJ** が実装されています。 この **UDJ** を拡張することで、すべての **UDJ** 機能を実装できます。

```
package com.aliyun.odps.udf.example.udj;
import com.aliyun.odps.Column;
import com.aliyun.odps.OdpsType;
import com.aliyun.odps.Yieldable;
import com.aliyun.odps.data.ArrayRecord;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.udf.DataAttributes;
import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.UDJ;
import com.aliyun.odps.udf.annotation.Resolve;
Import java. util. arraylist;
import java.util.Iterator;
/** For each record of the right table, find the nearest record of the
 left table and
 * merge two records.
 */
@Resolve("->string,bigint,string")
public class PayUserLogMergeJoin extends UDJ {
  private Record outputRecord;
  /** Will be called prior to the data processing phase. User could
implement
   * use this method to do initialization work.
   */
  @Override
  public void setup(ExecutionContext executionContext, DataAttributes
dataAttributes) {
    //
    outputRecord = new ArrayRecord(new Column[]{
      new Column("user_id", OdpsType.STRING),
      new Column("time", OdpsType.BIGINT),
      new Column("content", OdpsType.STRING)
    });
  }
  /** Override this method to implement join logic.
   * @param key Current join key
   * @param left Group of records of the left table corresponding to
the current key
   * @param right Group of records of the right table corresponding to
 the current key
   * @param output Used to output the result of UDJ
   */
  @Override
  public void join(Record key, Iterator<Record> left, Iterator<Record
> right, Yieldable<Record> output) {
    outputRecord.setString(0, key.getString(0));
    if (! right.hasNext()) {
      // Empty the right group and do nothing.
      return;
    } else if (! left.hasNext()) {
      // Empty left group. Output all records of the right group
without merge.
      while (iter.hasNext()) {
        Record logRecord = right.next();
        outputRecord.setBigint(1, logRecord.getDatetime(0).getTime());
        outputRecord.setString(2, logRecord.getString(1));
        output.yield(outputRecord);
      }
```

```
      return;
    }
    ArrayList<Record> pays = new ArrayList<>();
    // The left group of records will be iterated from the start to
the end
    // for each record of the right group, but the iterator cannot be
reset.
    // So we save every record of the left to an ArrayList.
    left.forEachRemaining(pay -> pays.add(pay.clone()));
    while (right.hasNext()) {
      Record log = right.next();
      long logTime = log.getDatetime(0).getTime();
      long minDelta = Long.MAX_VALUE;
      Record nearestPay = null;
      // Iterate through all records of the left, and find the pay
record that has
      // the minimal difference in terms of time.
      for (Record pay: pays) {
        long delta = Math.abs(logTime - pay.getDatetime(0).getTime());
        if (delta < minDelta) {
          minDelta = delta;
          nearestPay = pay;
        }
      }
      // Merge the log record with the nearest pay record and output
the result.
      outputRecord.setBigint(1, log.getDatetime(0).getTime());
      outputRecord.setString(2, mergeLog(nearestPay.getString(1), log.
getString(1)));
      output.yield(outputRecord);
    }
  }
  String mergeLog(String payInfo, String logContent) {
    return logContent + ", pay " + payInfo;
  }
  @Override
  public void close() {
  }
}
```

 注:

この例では、エントリ内の **NULL** 値は処理されません。 データ処理手順を単純化して見やすくするために、この例ではテーブルに **NUL L**値が含まれていないと想定しています。

この **UDJ** の **join** メソッドを呼び出すたびに、**2** つのテーブルの同じキーが一致するレコードが返されます。 したがって、**UDJ** は支払いテーブル内のすべてのレコードを検索して、**user_client_log** テーブルの各レコードに最も近い時間でレコードを見つけます。

ユーザーがいくつかの支払い記録しか持っていないと仮定します。 この場合、支払いテーブルのデータを **RAM** にロードできます。 通常、**RAM** には、一日に生成されたユーザの支払いデータを格納するのに十分なスペースがあります。 この仮定が無効だとどうなるでしょうか。 どうすればこの問題を解決できるでしょうか。 この問題については、「**SORT BY**句の使用」で説明します。

MaxCompute で UDJを作成

UDJ コードを **Java** で作成したら、そのコードをプラグインとして **MaxCompute SQL** にアップロードします。 まず **MaxCompute** にコードを登録する必要があります。 コードが **JAR** パッケージ *odps-udj-example.jar*に含まれているとします。

次のように、**Add JAR** コマンドを使用して、**JAR** パッケージを **MaxCompute** にアップロードします。

```
add jar odps-udj-example.jar;
```

CREATE FUNCTION 文を使用して、対応する **Java** クラスを参照し、**odps-udj-example.jar JAR** パッケージを使用して、**UDJ** 関数 **pay_user_log_merge_join** を作成します。

`com.aliyun.odps.udf.example.udj.PayUserLogMergeJoin:`

```
create function pay_user_log_merge_join
  as 'com.aliyun.odps.udf.example.udj.PayUserLogMergeJoin'
  using 'odps-udj-example.jar';
```

MaxCompute SQL で UDJ を使用

データベースに **UDJ** を登録した後、この関数は **MaxCompute SQL** で使用できます。

1. サンプルソーステーブルを作成します。

```
create table payment (user_id string,time datetime,pay_info string);
create table user_client_log(user_id string,time datetime,content
string);
```

2. サンプルデータを作成します。

```
支払いテーブルにデータを作成します。
INSERT OVERWRITE TABLE payment VALUES
('1335656', datetime '2018-02-13 19:54:00', 'PEqMSHyktn'),
('2656199', datetime '2018-02-13 12:21:00', 'pYvotuLDIT'),
('2656199', datetime '2018-02-13 20:50:00', 'PEqMSHyktn'),
('2656199', datetime '2018-02-13 22:30:00', 'gZhvdySOQb'),
('8881237', datetime '2018-02-13 08:30:00', 'pYvotuLDIT'),
('8881237', datetime '2018-02-13 10:32:00', 'KBuMzRpsko'),
('9890100', datetime '2018-02-13 16:01:00', 'gZhvdySOQb'),
('9890100', datetime '2018-02-13 16:26:00', 'MxONdLckwa')
;
--Create data in the user_client_log table
INSERT OVERWRITE TABLE user_client_log VALUES
('1000235', datetime '2018-02-13 00:25:36', 'click FNOXAibRjkIaQPB
'),
('1000235', datetime '2018-02-13 22:30:00', 'click GczrYaxvkiPultZ
'),
('1335656', datetime '2018-02-13 18:30:00', 'click MxONdLckpAFUHRS
'),
('1335656', datetime '2018-02-13 19:54:00', 'click mKRPGOciFDyzTgM
'),
('2656199', datetime '2018-02-13 08:30:00', 'click CZwafHsbJOPNitL
'),
```

```
('2656199', datetime '2018-02-13 09:14:00', 'click nYHJqIpjevkKToy
'),
('2656199', datetime '2018-02-13 21:05:00', 'click gbAfPCwrGXvEjpI
'),
('2656199', datetime '2018-02-13 21:08:00', 'click dhpZyWMuGjBOTJP
'),
('2656199', datetime '2018-02-13 22:29:00', 'click bAsxnUdDhvfqaBr
'),
('2656199', datetime '2018-02-13 22:30:00', 'click XIhZdLaOocQRmrY
'),
('4356142', datetime '2018-02-13 18:30:00', 'click DYqShmGbIoWKier
'),
('4356142', datetime '2018-02-13 19:54:00', 'click DYqShmGbIoWKier
'),
('8881237', datetime '2018-02-13 00:30:00', 'click MpkvilgWSmhUuPn
'),
('8881237', datetime '2018-02-13 06:14:00', 'click OkTYNUHMqZzlDyL
'),
('8881237', datetime '2018-02-13 10:30:00', 'click OkTYNUHMqZzlDyL
'),
('9890100', datetime '2018-02-13 16:01:00', 'click vOTQfBFjcgXisYU
'),
('9890100', datetime '2018-02-13 16:20:00', 'click WxaLgOCcVEvhiFJ')
;
```

3. **MaxCompute SQL** で、作成した **UDJ** 関数を使用します。

```
SELECT r.user_id, from_unixtime(time/1000) as time, content FROM (
SELECT user_id, time as time, pay_info FROM payment
) p JOIN (
SELECT user_id, time as time, content FROM user_client_log
) u
ON p.user_id = u.user_id
USING pay_user_log_merge_join(p.time, p.pay_info, u.time, u.content)
r
AS (user_id, time, content)
;
```

UDJ の構文は、標準の結合構文と似ています。 唯一の違いは、**USING** 句が **UDJ** に追加されていることです。

- **SQL** の **UDJ** 関数の名前は、**pay_user_log_merge_join** です。
- **(p.time、p.pay_info、u.time、u.content)** は、これら **2** つのテーブルで使用される列です。
- **r** は、**UDJ** 関数によって返される結果の別名です。 この別名は。他の **SQL** 文で参照できます。
- **(user_id、time、content)** は、**UDJ** 関数によって返される列です。

この **SQL** 文を実行すると、結果は次のようになります。

```
+---------+------------+---------+
| user_id | time       | content |
+---------+------------+---------+
| 1000235 | 2018-02-13 00:25:36 | click FNOXAibRjkIaQPB |
| 1000235 | 2018-02-13 22:30:00 | click GczrYaxvkiPultZ |
```

```
| 1335656 | 2018-02-13 18:30:00 | click MxONdLckpAFUHRS, pay
PEqMSHyktn |
| 1335656 | 2018-02-13 19:54:00 | click mKRPGOciFDyzTgM, pay
PEqMSHyktn |
| 2656199 | 2018-02-13 08:30:00 | click CZwafHsbJOPNitL, pay
pYvotuLDIT |
| 2656199 | 2018-02-13 09:14:00 | click nYHJqIpjevkKToy, pay
pYvotuLDIT |
| 2656199 | 2018-02-13 21:05:00 | click gbAfPCwrGXvEjpI, pay
PEqMSHyktn |
| 2656199 | 2018-02-13 21:08:00 | click dhpZyWMuGjBOTJP, pay
PEqMSHyktn |
| 2656199 | 2018-02-13 22:29:00 | click bAsxnUdDhvfqaBr, pay
gZhvdySOQb |
| 2656199 | 2018-02-13 22:30:00 | click XIhZdLaOocQRmrY, pay
gZhvdySOQb |
| 4356142 | 2018-02-13 18:30:00 | click DYqShmGbIoWKier |
| 4356142 | 2018-02-13 19:54:00 | click DYqShmGbIoWKier |
| 8881237 | 2018-02-13 00:30:00 | click MpkvilgWSmhUuPn, pay
pYvotuLDIT |
| 8881237 | 2018-02-13 06:14:00 | click OkTYNUHMqZzlDyL, pay
pYvotuLDIT |
| 8881237 | 2018-02-13 10:30:00 | click OkTYNUHMqZzlDyL, pay
KBuMzRpsko |
| 9890100 | 2018-02-13 16:01:00 | click vOTQfBFjcgXisYU, pay
gZhvdySOQb |
| 9890100 | 2018-02-13 16:20:00 | click WxaLgOCcVEvhiFJ, pay
MxONdLckwa |
+---------+------------+---------+
```

前のコードに示したように、ネイティブな結合メソッドを呼び出しても実行できなかったタスクは、**UDJ** を使用して完了しました。

事前ソート

一致する支払い記録を見つけるために、支払いテーブル内のすべてのレコードを検索するためにイテレータが使用されます。 このタスクを実行するには、同じ **user_id** を持つすべての支払いレコードを **ArrayList** にロードする必要があります。 この方法は、支払いレコード数が少ない場合に適用できます。 **RAM** サイズの制限により、大量の支払いレコードが生成された場合は、データをロードするための別の方法を見つける必要があります。 ここでは、**SORT BY** 句を使用してこの問題に対処する方法について説明します。

支払いデータのサイズが大きすぎて **RAM** に保存できない場合、テーブル内のすべてのデータがすでに時間順にソートされていると、この問題の対処は簡単です。 その後、これら **2** つのリストの最初の要素を比較するだけで済みます。

Java で記述した **UDJ**

```
@Override
public void join(Record key, Iterator<Record> left, Iterator<Record>
right, Yieldable<Record> output) {
  outputRecord.setString(0, key.getString(0));
  if (! right.hasNext()) {
    return;
  } else if (! left.hasNext()) {
```

```
    while (right.hasNext()) {
      Record logRecord = right.next();
      outputRecord.setBigint(1, logRecord.getDatetime(0).getTime());
      outputRecord.setString(2, logRecord.getString(1));
      output.yield(outputRecord);
    }
    return;
  }
  long prevDelta = Long.MAX_VALUE;
  Record logRecord = right.next();
  Record payRecord = left.next();
  Record lastPayRecord = payRecord.clone();
  while (true) {
    long delta = logRecord.getDatetime(0).getTime() - payRecord.
getDatetime(0).getTime();
    if (left.hasNext() && delta > 0) {
      // The delta of time between two records is decreasing, we can
still
      // explore the left group to try to gain a smaller delta.
      lastPayRecord = payRecord.clone();
      prevDelta = delta;
      payRecord = left.next();
    } else {
      // Hit to the point of minimal delta. Check with the last pay
record,
      // output the merge result and prepare to process the next
record of
      // right group.
      Record nearestPay = Math.abs(delta) < prevDelta ? payRecord :
lastPayRecord;
      outputRecord.setBigint(1, logRecord.getDatetime(0).getTime());
      String mergedString = mergeLog(nearestPay.getString(1),
logRecord.getString(1));
      outputRecord.setString(2, mergedString);
      output.yield(outputRecord);
      if (right.hasNext()) {
        logRecord = right.next();
        prevDelta = Math.abs(
          logRecord.getDatetime(0).getTime() - lastPayRecord.
getDatetime(0).getTime()
        );
      } else {
        break;
      }
    }
  }
}
```

ネイティブな **SQL** 言語では、この例を少し変更して **UDJ** 句の末尾に **SORT BY** 句を追加してから、両方のテーブルのデータを時間順に並べ替えるだけで済みます。 注記**: UDJ** コードを変更したら、対応する **JAR** パッケージを更新する必要があります。

```
SELECT r.user_id, from_unixtime(time/1000) as time, content FROM (
  SELECT user_id, time as time, pay_info FROM payment
) p JOIN (
  SELECT user_id, time as time, content FROM user_client_log
) u
ON p.user_id = u.user_id
USING pay_user_log_merge_join(p.time, p.pay_info, u.time, u.content)
r
AS (user_id, time, content)
```

```
SORT BY p.time, u.time
;
```

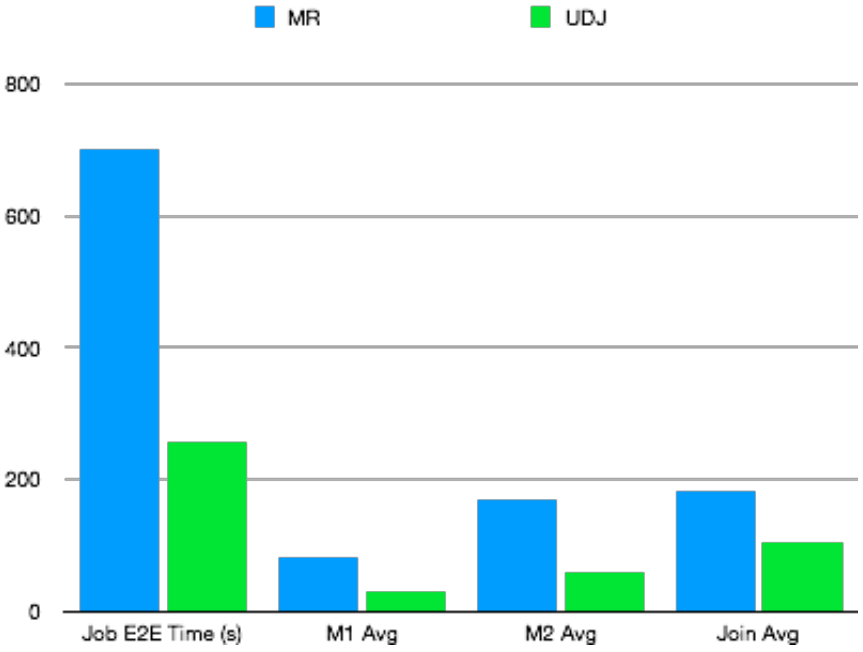
実行結果は、コードが変更される前の結果と同じです。

このメソッドでは、**SORT BY** 句を使用してデータを事前ソートします。 同じ結果を得るには、最大 **3** つのレコードのみをキャッシュする必要があります。

UDJ のパフォーマンス

UDJ がないと、分散システムで複雑なクロステーブルコンピューティングタスクを処理するために **MapReduce** を使用する必要があります。 適用可能なシナリオには、広告や情報検索などの複雑なビジネスシナリオが含まれます。

次の例では、オンライン **MapReduce** ジョブを使用して **UDJ** のパフォーマンスをテストします。 この **MapReduce** ジョブは、複雑なアルゴリズムを使用して **2** つのテーブルを結合します。 この例では、**UDJ**を使用して **MapReduce** ジョブの **SQL** 文を書き換えて、実行結果が正しいかどうかを確認しています。 同じプログラミングの並行処理の下で、パフォーマンスの比較は次のとおりです。



図に示すように、**UDJ** を使用すると、複数のテーブルを処理する複雑なロジックを記述しやすくなり、照会のパフォーマンスが大幅に向上します。 コードは **UD J**関数の内部でのみ実行され、この例の **Map** 段階のロジックなど、コードのロジック全体は高性能 な **MaxCompute** ネイティブランタイムによって実行されます。 **UDJ** により、**MaxCompute** ランタイムエンジンとインタフェース間のデータ交換が最適化されます。 **UDJ** の結合ロジックは、**MapReduce** の **Reduce** ステージよりも効率的です。

5.16 補足

5.16.1 エスケープ文字

MaxCompute SQL では、文字列定数を一重引用符 (') または二重引用符 (“) で囲んで表現することができます。 一重引用符で囲まれた文字列に二重引用符を含めることも、二重引用符で囲まれた文字列に一重引用符を含めることもできます。 それ以外の場合は、エスケープ文字を使用して表現する必要があります。

以下の表現を使用できます。

```
"I'm a happy manong."
'I\'m a happy manong.'
```

MaxCompute SQL では、'\' はエスケープ文字の一種で、文字列内で特殊文字を表現したり、それに続く文字そのものを表現するために使用します。 文字列定数を読み取る際、'\' の後ろに有効な **8** 桁の **16** 進数 が **3** つ続き、 対応する範囲が **001** から **177** の場合、**ASCII** 値に従って対応する文字に変換されます。

次の表は、特殊なエスケープ文字の一覧です。

エスケープ文字	文字の意味
\b	バックスペース
\t	タブ
\n	改行
\r	キャリッジリターン
\’	一重引用符
\”	二重引用符
\ \	バックスラッシュ
\;	セミコロン
\Z	コントロール **Z**

エスケープ文字	文字の意味
\0 または \00	終端文字

```
select length('a\tb') from dual;
```

結果は 3 です。 これは、文字列内に 3 文字あることを示します。 '\t' は 1 文字と見なされます。 その他の文字は、文字そのものとして表現されます。

```
select 'a\ab',length('a\ab') from dual;
```

結果は、'aab'、3 です、'\a' は、通常の 'a’ と表現されます。

5.16.2 LIKE の使用方法

LIKE を使った照合では、'%' は任意の複数文字と一致することを示します。 '_' は、任意の 1 文字と一致することを示します。 '%' または '_' 自体と照合するには、これらの文字をエスケープする必要があります。 '\\%' は '%' と一致し、'_' は '_' と一致します。

```
    'abcd' like 'ab%' -- true
    'abcd' like 'ab_' -- false
    'ab_cde' like 'ab\\_c%'; -- true
```

 :

MaxCompute SQL は UTF-8 文字セットにのみ対応しています。 データが別の形式でエンコードされている場合は、計算結果が正しくない可能性があります。

5.16.3 正規表現

MaxCompute SQL の正規表現では、文字で照合する PCRE 標準が採用されます。 サポート対象のメタ文字は、次のとおりです。

メタ文字	説明
^	行の先頭 (TOL)
$	行末
.	任意の文字
*	0 回または複数回の一致
+	1 回または複数回の一致
?	0 回または 1 回の一致

メタ文字	説明
?	量指定子が一致します。 この文字が他の制約 (*、+、**?**、**{n}**、**{n, {n , m}}** に従っている場合、 非貪欲モードでの照合になります。 非貪欲モードでは、一致する文字列が可能な限り少なくなります。デフォルトの貪欲モードでは、可能な限り多くの文字列が一致するようになります。
A | B	**A** または **B**
(abc)*	**'abc'** と **0** 回または複数回の一致
{n} or {m, n}	複数回の一致
[ab]	角括弧内の任意の文字と一致。 この例では、**a** または **b** と一致します。
[a-d]	**a**、**b**、**c**、**d** の中の任意の文字と一致します。
[^ab]	^ は、**'**以外**'** を示し、**a** 、**b** 以外の任意の文字と一致します。
[::]	次の表の **POSIX** 文字グループをご参照ください。
****	エスケープ文字
\n	**N** は **1** から **9** までの数字で、後方参照されています。
\d	数字
\ D	非数字

POSIX 文字グループは、次のとおりです。

POSIX 文字グループ	説明	範囲
[[:alnum:]]	文字と数字	**[a-zA-Z0-9-]***
[[:alpha:]]	文字	**[a-zA-Z]**
[[:ascii:]]	**ASCII** 文字	**[\x00-\x7F]**
[[:blank:]]	スペース文字とタブ	**[\t]**
[[:cntrl:]]	制御文字	**[\x00-\x1F\x7F]**
[[:digit:]]	数字	**[0-9]**
[[:graph:]]	空白文字以外の文字	**[\x21-\x7E]**
[[:lower:]]	小文字	**[a-z]**
[[:print:]]	**[:graph:]** と空白文字	**[\x20-\x7E]**
[[:punct:]]	句読点	**[][!”[][!”#$%&’()*+,./:;<=>?@\^_`{|}~-]**
[[:space:]]	空白文字	**[\t\r\n\v\f]**

POSIX 文字グループ	説明	範囲
[[:upper:]]	大文字	[A-Z]
[[:xdigit:]]	16 進文字	[A-Fa-f0-9]

システムではエスケープ文字としてバックスラッシュが使用されるので、正規表現パターンに現れるすべての "\" は 2 つのエスケープを実行します。 たとえば、正規表現で文字列 "a + b" と照合する必要があるとします。 "+" は正規表現では特殊文字なので、エスケープして表す必要があります。 システムはエスケープのレイヤーを説明する必要があるため、通常のエンジンでの表現は "a\+b" です。この文字列と一致する表現は "a\\+b" です。

次のようなテーブル test_dual があるとします。

```
select 'a+b' rlike 'a\\\+b' from test_dual;
| _c1 |
| true |
```

"\" は通常のエンジンでは特殊文字なので、極端な場合、文字 "\" と一致させるには、"\" で表す必要があり、システムでは再度エスケープを実行するので、" \\" と記述されます。

```
select 'a\\b', 'a\\b' rlike 'a\\\\b' from test_dual;
| _c0 | _c1 |
| a\b | true |
```

注:
MaxCompute SQL で `a\\b` と記述すると、出力結果は`a\b`になります。

TAB が文字列内に存在する場合、システムでこの 2 つの文字 `\t`が読み取られると、1 文字として保存されます。 したがって、正規表現では一般文字になります。

```
select 'a\tb', 'a\tb' rlike 'a\tb' from test_dual;| _c0 | _c1 |
| a b | true |
```

5.16.4 予約語とキーワード

ここでは、MaxCompute SQL のすべての予約語を示します。

:

- テーブル名、列名、パーティション名には予約語を使用できません。使用するとエラーが発生します。

・ 予約語は、大文字と小文字が区別されません。

```
    % & && ( ) * +
                - . / ; < <= <>
        ADD AFTER ALL
    ALTER ANALYZE AND ARCHIVE ARRAY AS ASC
    BEFORE BETWEEN BIGINT BINARY BLOB BOOLEAN BOTH DECIMAL
    BUCKET BUCKETS BY CASCADE CASE CAST CFILE
    CHANGE CLUSTER CLUSTERED CLUSTERSTATUS COLLECTION COLUMN COLUMNS
    COMMENT COMPUTE CONCATENATE CONTINUE CREATE CROSS CURRENT
    CURSOR DATA DATABASE DATABASES DATE DATETIME DBPROPERTIES
    DEFERRED DELETE DELIMITED DESC DESCRIBE DIRECTORY DISABLE
    DISTINCT DISTRIBUTE DOUBLE DROP ELSE ENABLE END
    ESCAPED EXCLUSIVE EXISTS EXPLAIN EXPORT EXTENDED EXTERNAL
    FALSE FETCH FIELDS FILEFORMAT FIRST FLOAT FOLLOWING
    FORMAT FORMATTED FROM FULL FUNCTION FUNCTIONS GRANT
    GROUP HAVING HOLD_DDLTIME IDXPROPERTIES IF IMPORT IN
    INDEX INDEXES INPATH INPUTDRIVER INPUTFORMAT INSERT INT
    INTERSECT INTO IS ITEMS JOIN KEYS LATERAL
    LEFT LIFECYCLE LIKE LIMIT LINES LOAD LOCAL
    LOCATION LOCK LOCKS LONG MAP MAPJOIN MATERIALIZED
    MINUS MSCK NOT NO_DROP NULL OF OFFLINE
    ON OPTION OR ORDER OUT OUTER OUTPUTDRIVER
    OUTPUTFORMAT OVER OVERWRITE PARTITION PARTITIONED PARTITIONP
ROPERTIES PARTITIONS
    PERCENT PLUS PRECEDING PRESERVE PROCEDURE PURGE RANGE
    RCFILE READ READONLY READS REBUILD RECORDREADER RECORDWRITER
    REDUCE REGEXP RENAME REPAIR REPLACE RESTRICT REVOKE
    RIGHT RLIKE ROW ROWS SCHEMA SCHEMAS SELECT
    SEMI SEQUENCEFILE SERDE SERDEPROPERTIES SET SHARED SHOW
    SHOW_DATABASE SMALLINT SORT SORTED SSL STATISTICS STORED
    STREAMTABLE STRING STRUCT TABLE TABLES TABLESAMPLE TBLPROPERTIES
    TEMPORARY TERMINATED TEXTFILE THEN TIMESTAMP TINYINT TO
    TOUCH TRANSFORM TRIGGER TRUE UNARCHIVE UNBOUNDED UNDO
    UNION UNIONTYPE UNIQUEJOIN UNLOCK UNSIGNED UPDATE USE
    USING UTC UTC_TMESTAMP VIEW WHEN WHERE WHILE DIV
```

5.16.5 Hive データ型の対応表

MaxCompute と **Hive** データ型の対応表は、以下のとおりです。

Hive データ型	MaxCompute データ型
BOOLEAN	**Boolean**
TINYINT	**Tinyint**
SMALLINT	**Smallint**
INT	**Int**
BIGINT	**Bigint**
FLOAT	**Float**
DOUBLE	**Double**
Decimal	**Decimal**
String	**String**

Hive データ型	MaxCompute データ型
Varchar	Varchar
Char	String
BINARY	Binary
Timestamp	Timestamp
Date	Datetime
ARRAY	Array
`Map <key, value>`	MAP
STRUCT	Struct
Union	この機能はサポート対象外です。

6 MapReduce

6.1 概要

6.1.1 MapReduce

ここでは、**MaxCompute** で使用できる **MapReduce** プログラミングインターフェイスとその制限について説明します。

MaxCompute が提供する **MapReduce** プログラミングインターフェイスには、次の **3** つのバージョンがあります。

- **MaxCompute MapReduce: MaxCompute** のネイティブインターフェイスで、他のインターフェイスよりも高速です。 ファイルシステムを公開せずにプログラムを開発するのに便利です。
- *MR2* **(**拡張 **MapReduce):** より複雑なジョブスケジューリングロジックに対応した **MaxCompute** の拡張版インターフェイスです。 **MapReduce** は、**MaxCompute** ネイティブインターフェイスと同じ方法で実装します。
- *Hadoop* 互換バージョン **:** *Hadoop MapReduce* との高い互換性を備えています が、**MaxCompute** ネイティブインターフェイスや **MR2** との互換性はありません。

上記 **3** つのバージョンは、 基本概念、ジョブ送信、入出力、リソースの点で基本的に同じですが、唯一の違いは **Java SDK** です。 ここでは、**MapReduce** の原理を説明します。 **MapReduce** の詳細については、*Hadoop MapReduce* をご参照ください。

 注 **:**
現時点で、**MapReduce** を使って外部テーブルのデータを読み書きすることはできません。

シナリオ

当初、**MapReduce** は分散データ処理モデルとして **Google** によって提案されました。現在は複数のビジネスシナリオで広く採用されています。 以下はその例です。

- 検索：**Web** のクロール、フリップインデックス、**PageRank**。
- **Web** アクセスログ分析
  - 個人向けの情報を提供するために、**Web** アクセスや買い物行動の特性の分析とマイニングを行います。
  - ユーザーのアクセス行動の分析

- テキストの統計と分析
  - **Mo Yan** の小説のワードカウントと **TFIDF** 分析
  - 学術論文および特許文書の参照分析と統計
  - ウィキペディアのデータ分析など
- 大規模データマイニング**:** 非構造化データ、時空間データ、画像データマイニング
- 機械学習**:** 教師あり学習、教師なし学習、決定木や **SVM** などの分類アルゴリズム
- 自然言語処理**:**
  - ビッグデータに基づくトレーニングと予測
  - 現在の単語のマトリックス、頻繁なアイテムセットのデータマイニング、繰り返しのドキュメント検出などを構築するためのコーパスに基づきます。
- 広告の推奨事項**:** ユーザークリック **(CTR)** と購買行動 **(CVR)** の予測

データ処理

MapReduce のデータ処理は、**Map** と **Reduce** の **2** 段階に分けられます。 最初に **Map** を実行してから **Reduce** を実行する必要があります。 **Map** と **Reduce** の処理ロジックはユーザーが定義しますが、**MapReduce** フレームワークのプロトコルに準拠する必要があります。 プロセスは次のとおりです。

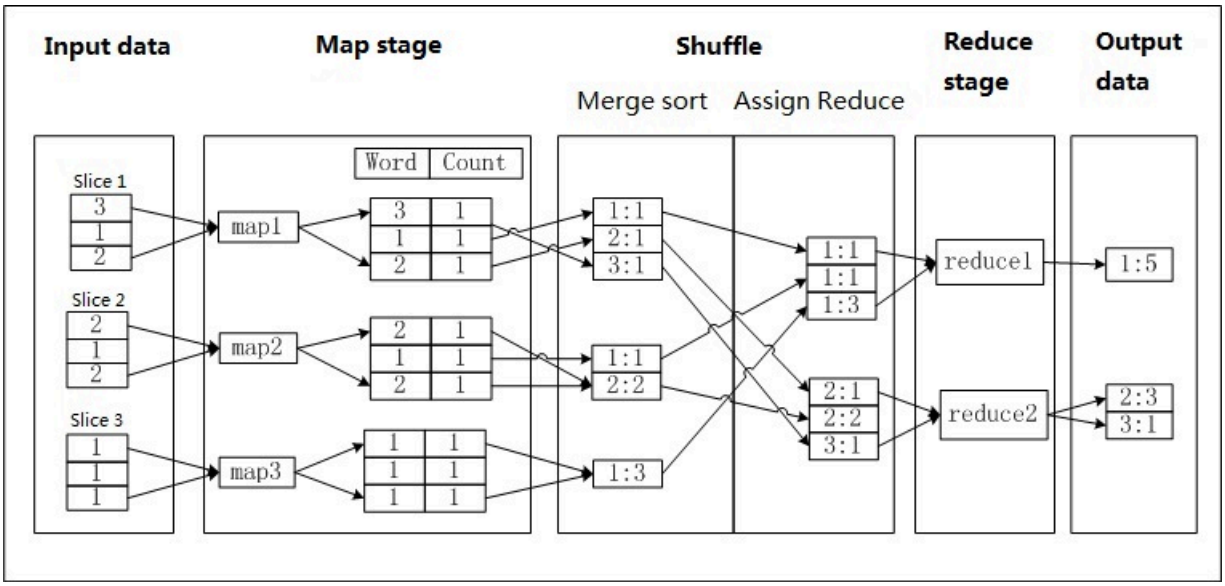
1. **Map** を実行する前に、入力データをスライスする必要があります。つまり、入力データを同じサイズのブロックに分割します。 各ブロックは単一の **Map Worker** の入力データとして処理されるため、複数の **Map Worker** が同時に処理を実行できます。
2. スライスが分割された後、複数の **Map Worker** が同時に処理を行えます。 各 **Map Worker** は、データの読み取り後、計算を実行し、結果を **Reduce** に出力します。 **Map Worker** がデータを出力するには、出力レコードごとにキーを指定する必要があります。 このキー値によって、どの **Reduce Worker** にデータが送信されたのかが分かります。 キー値と **Reduce Worker** の関係は、多対 **1** です。 同じキーを持つデータは同じ **Reduce Worker** に送信されるため、**1** つの **Reduce Worker** が複数のキー値データを受け取ることがあります。
3. **Reduce** 段階の前に、**MapReduce** フレームワークではキー値に従ってデータがソートされ、同じキー値を持つデータがグループ化されます。 **Combiner** を指定した場合、フレームワークでは **Combiner** が呼び出されて、同じキーデータが集約されます。 ユーザーは **Combiner** のロジックを定義する必要があります。 従来の **MapReduce** フレームワークとは異なり、**Combiner** と **MaxCompute** の **Reduce** の入力パラメーターと出力パラメーターは、一致している必要があります。 この処理は一般的にシャッフルと呼ばれます。
4. **Reduce** 段階では、同じキーを持つデータが同じ **Reduce Worker** によってシャッフルされます。 **Reduce Worker** は、複数の **Map Worker** からデータを受け取ります。 各 **Reduce**

Worker は、同じキーを持つ複数のレコードに対して **Reduce** 操作を実行します。 その後、**Reduce** 処理によって、これらの複数レコードから値が求められます。

> 注：
> **MapReduce** フレームワークの簡単な説明は、前述のプロセスで述べました。 詳細は、関連ドキュメントをご参照ください。

次の例では、**WordCount** を使用して **MaxCompute MapReduce** の処理段階について説明します。

'a.txt' という名前のテキストがあり、各行が数字で示されています。それぞれの数字の出現回数を数える必要があるとします。 テキスト内の数字は **'Word'** と呼ばれ、数字の出現回数は **'Count'** と呼ばれます。 **MaxCompute MapReduce** によるこの機能の実行順序を下図に示します。



実行順序：

1. まず、テキストがスライスされ、データがスライスごとに単一の **Map Worker** に入力されます。
2. **Map Worker** により入力が処理されます。 **Map Worker** で数値が取得されると、**Count** が **1** に設定されます。 次に、<**Word,Count**> キューが出力されます。 出力データのキーとして **'Word'** が取得されます。
3. **Shuffle** 段階の初期操作では、**Map Worker** の出力がキー値 (**Word**の値) に従ってソートされます。 ソート後に結合操作が実行され、同じキー値の **Count**（**Word** 値）が累積され、新しい <**Word,Count**> キューが構成されます。 このプロセスはマージソートと呼ばれます。
4. **Shuffle** の後半操作では、データが **Reduce** に送信されます。 データを受信した後、**Reduce Worker** はキー値に基づいてデータを再度ソートします。

5. データの処理中、各 **Reduce Worker** によって **Combiner** と同じロジックで同じキー値 (**Word** 値) を持つ **Count** が累積され、出力値が取得されます。
6. 結果が出力されます。



注 :

MaxCompute 内のデータはテーブルに格納されるため、**MaxCompute MapReduce** の入力と出力はテーブルでなければなりません。 出力をユーザー定義することはできません。また、対応するファイルシステムインターフェイスはありません。

6.1.2 拡張 MapReduce

従来の **MapRudece** と異なり、**MaxCompute** に搭載されている 拡張 **MapReduce** モデルは、基盤となるスケジューリングモデルと **I/O** モデルが改良されているため、実行時の冗長な **I/O** 操作を回避できます。

従来の **MapReduce** モデルでは、**MapReduce** 操作の後に、分散ファイルシステム (**HDFS** や **MaxCompute** テーブルなど) に毎回データをロードする必要があります。しかし、一般的な **MapReduce** アプリケーションは、複数の **MapReduce** ジョブから構成されるため、ジョブごとにデータへの書き込みが必要です。 以下の **Map** タスクは、後続の **Shuffle** 操作の準備段階として、データを読み込むためだけに使用されるタスクの例ですが、結果として、冗長な **I/O** 操作が発生します。

MaxCompute の計算スケジュールロジックは、さらに複雑なプログラミングの枠組みに対応しています。 前述のシナリオでは、次の **Reduce** 操作は **Reduce** 操作後に実行可能で、**Map** 操作の挿入は必要ありません。 このようにして、**MaxCompute** は拡張 **MapReduce** モデルを提供します。つまり、**Map>Reduce> Reduce** など、多数の **Reduce** 操作が **Map** 操作の後に続くことができます。

Hadoop Chain Mapper/Reducer は、アナログシリアル **Map** や **Reduce** 操作にも対応していますが、拡張 **MaxCompute (MR2)** モデルと大きな違いがあります。

Hadoop Chain Mapper/Reducer は、従来の **MapReduce** モデルを基盤とし、元の **Mapper** や **Reducer** 後にいくつかの **Mapper** 操作を加えただけです (**Reducer** 操作は追加できません) 。 拡張 **MapReduce** の利点は、以前使用した **Mapper** のビジネスロジックを再利用できる点です。また、**1** つの **Map** ステージや **Reduce** ステージを複数の **Mapper** ステージに分割できます。基盤となるスケジューリングおよび **I/O** モデルは本質的には変更されていません。

MaxCompute と比べると、**MR2** は基本的に **Map/Reduce** 関数が記述通りに実行されます。 主な違いはパフォーマンスにあります。詳細については、「拡張 *MapReduce* 例」をご参照ください。

6.1.3 オープンソースの MapReduce

ここでは、オープンソースの **MapReduce** アプリケーションの背景と **HadoopMR** プラグインの基本的な使い方を説明します。

MaxCompute は、ネイティブの **MapReduce** プログラミングモデルとインターフェイスのセットを提供しています。 これらのインターフェイスの入出力は、**MaxCompute** テーブルであり、データはレコード形式で処理されるように編成されます。

ただし、**MaxCompute API** は **Hadoop** フレームワーク用の **API** とは大きく異なります。 以前は、**Hadoop MapReduce** ジョブを **MaxCompute** に移行するには、まず、**MapReduce** コードを書き換え、**MaxCompute API** を使用してコードをコンパイルおよびデバッグし、最終コードを **JAR** パッケージに圧縮し、最後にそのパッケージを **MaxCompute** プラットフォームにアップロードする必要がありました。 このプロセスは面倒であり、開発とテストに多くの時間を費やす必要があります。 元の **Hadoop MapReduce** コードを部分的に変更する必要がない場合は、**MaxCompute** コンソールで実行するのが最善の方法です。

現在、**MaxCompute** プラットフォームは、**Hadoop MapReduce** コードを **MaxCompute MapReduce** 仕様に適合させることを可能にするプラグインを提供します。 **MaxCompute** は、**Hadoop MapReduce** ジョブのバイナリレベルの互換性に関して、ある程度の柔軟性を備えています。 つまり、コードを変更せずに、元の **Hadoop MapReduce Jar** パッケージを **MaxCompute** 上で直接実行するように設定できます。使い始めるには、開発プラグインをダウンロードしてください。 このプラグインは現在テスト段階にあるため、カスタムコンパレータやキー入力には対応していません。

次の例では、**WordCount** プログラムを使用したプラグインの基本的な使い方を説明します。

HadoopMR プラグインのダウンロード

ここをクリックして、 **hadoop2openmr-1.0.jar** という名前のプラグインをダウンロードします。

> 注：
> この **Jar** パッケージには、**Hadoop 2.7.2** との依存関係が定義されています。 バージョンの競合を避けるために、ジョブの **Jar** パッケージに **Hadoop** の依存関係を定義しないでください。

Jar パッケージの準備

wordcount_test.jar という名前の **WordCount JAR** パッケージをコンパイルしてエクスポートします。 **WordCount** プログラムのソースコードは次のとおりです。

```
package com.aliyun.odps.mapred.example.hadoop;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
```

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import java.io.IOException;
import java.util.StringTokenizer;
public class WordCount {
    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString
());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
    public static class IntSumReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,
            Context context
        ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static void main (final String [] args) throws IOException{
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

テストデータの準備

1. 入力テーブルと出力テーブルを作成します。

```
create table if not exists wc_in(line string);
```

```
create table if not exists wc_out(key string, cnt bigint);
```

2. **Tunnel** コマンドを実行して、入力テーブルにデータをインポートします。

   インポートする **data.txt** ファイル内のデータは、次のとおりです。

   ```
   hello maxcompute
   hello mapreduce
   ```

   MaxCompute コンソールから `Tunnel コマンド` を使用して **data.txt** から **wc_in** にデータをインポートします。

   ```
   tunnel upload data.txt wc_in;
   ```

テーブルと HDFS **ファイルパスの間のマッピングの設定**

設定ファイルは、**wordcount-table-res.conf** です。

```
{
  "file:/foo": {
    "resolver": {
      "resolver": "com.aliyun.odps.mapred.hadoop2openmr.resolver.
TextFileResolver",
      "properties": {
          "text.resolver.columns.combine.enable": "true",
          "text.resolver.seperator": "\t"
      }
    },
    "tableInfos": [
      {
        "tblName": "wc_in",
        "partSpec": {},
        "label": "__default__"
      }
    ],
    "matchMode": "exact"
  },
  "file:/bar": {
    "resolver": {
      "resolver": "com.aliyun.odps.mapred.hadoop2openmr.resolver.
BinaryFileResolver",
      "properties": {
          "binary.resolver.input.key.class" : "org.apache.hadoop.io.
Text",
          "binary.resolver.input.value.class" : "org.apache.hadoop.io.
LongWritable"
      }
    },
    "tableInfos": [
      {
        "tblName": "wc_out",
        "partSpec": {},
        "label": "__default__"
      }
    ],
    "matchMode": "fuzzy"
  }
```

```
}
```

パラメーター

HDFS ファイルと **MaxCompute** テーブルの間のマッピング関係を記述する **JSON** ファイルで設定を行います。 通常、入力と出力の両方を設定する必要があります。 **1** つの **HDFS** パスは、**1** つのリゾルバ、テーブル情報、マッチモードに対応します。

- **resolver:** ファイルデータの処理メソッドを指定します。 現在、以下の **2** つのリゾルバから選択できます。**com.aliyun.odps.mapred.hadoop2openmr.resolver.TextFileResolver** および **com.aliyun.odps.mapred.hadoop2openmr.resolver.BinaryFileResolver** リゾルバ名の他に、リゾルバのデータ解析に関する次のプロパティを設定します。
  - **TextFileResolver:** データがプレーンテキストの場合、入力または出力をプレーンテキストとして設定します。 入力リゾルバを設定する際、 **text.resolver.columns.combine.enable** や **text.resolver.seperator** などのプロパティを設定します。 **text.resolver.columns.combine.enable** を **true** に設定すると、入力テーブルのすべての列は **text.resolver.seperator** で指定した区切り文字に基づいて、単一の文字列に結合されます。 それ以外の場合は、入力テーブルの最初の **2** 列がキーと値として使用されます。
  - **BinaryFileResolver:** バイナリデータを **MaxCompute** でサポートされている型 **(Bigint**、**Boolean**、**Double**など) に変換します。 出力リゾルバを設定する際、プロパティ **binary.resolver.input.key.class** と **binary.resolver.input.value.class** で、中間結果のキーと値のデータ型をそれぞれ定義します。
- **tableInfos: HDFS** に対応する **MaxCompute** テーブルを指定します。 現在、**tblName** パラメーター (テーブル名) だけが設定可能です。 **partSpec** および **label** パラメーターは、この例のパラメーターに設定されている値と同じにする必要があります。
- **matchMode:** パスのマッチングモードを指定します。 完全一致モードは完全一致を示し、ファジーモードはファジーマッチを示します。 **HDFS** 入力パスと一致させるには、ファジーモードで正規表現を使用してください。

ジョブの送信

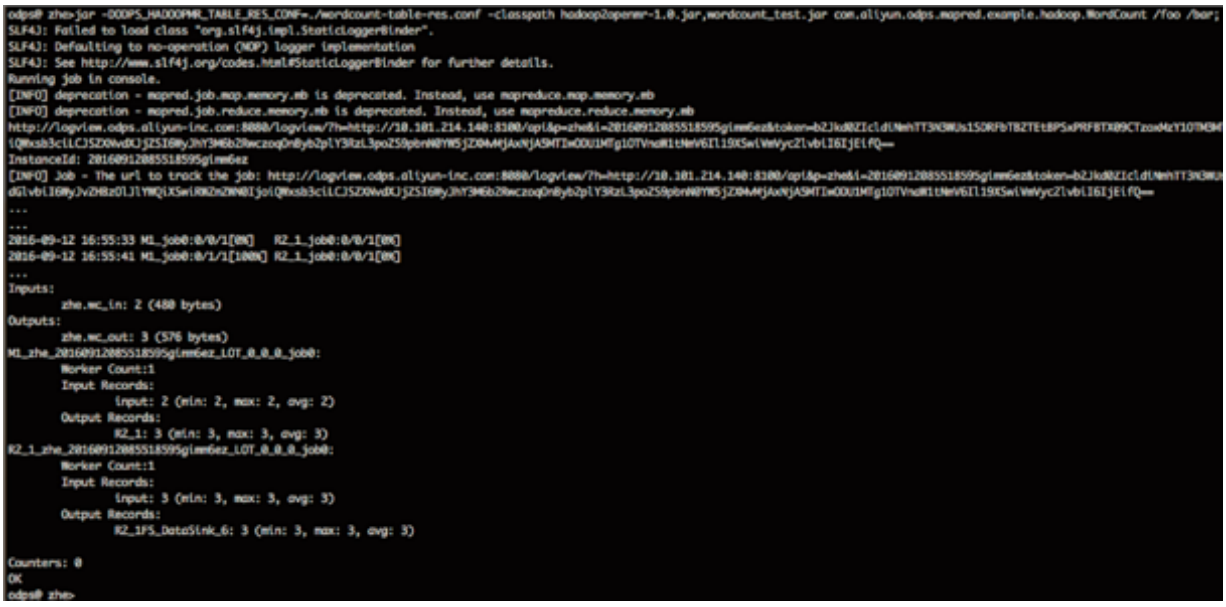
ジョブを送信するには、**MaxCompute** コマンドラインツール **odpscmd** を使用します。 **MaxCompute** コマンドラインツールのインストールと設定については、「コンソール」をご参照ください。 **odpscmd** で、次のコマンドを実行します。

```
jar -DODPS_HADOOPMR_TABLE_RES_CONF=./wordcount-table-res.conf -
classpath hadoop2openmr-1.0.jar,wordcount_test.jar com.aliyun.odps.
mapred.example.hadoop.WordCount /foo/bar;
```

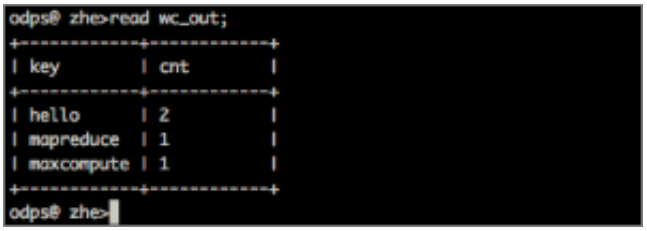
 注:

- **wordcount-table-res.conf** は、**/foo/bar** が設定されたマップです。
- **wordcount_test.jar** は、**Hadoop MapReduce** の **Jar** パッケージです。
- **com.aliyun.odps.mapred.example.hadoop.WordCount** は、実行するジョブのクラス名です。
- **/foo/bar** は **HDFS** 上のパスを指し、**JSON** 構成ファイル内の **wc_in**と**wc_out** にマッピングされます。
- マッピング関係が設定されたら、**DataX** または **DataWorks** のデータ統合機能を使用して、**MR** 計算用に **Hadoop HDFS** 入力ファイルを **wc_in** に手動でインポートし、結果の **wc_out** を **HDFS** 出力ディレクトリ **(/bar)** に手動でエクスポートします。
- 上記の出力で、**hadoop2openmr-1.0.jar**、**wordcount_test.jar**、および **wordcount-table-res.conf** が、**odpscmd** の現在のディレクトリに格納されているとします。 エラーが発生した場合は、設定や **-classpath** を指定するときに、必要な変更を加えてください。

実行プロセスは次のとおりです。



ジョブを実行した後、結果テーブル **wc_out** を調べて、ジョブが完了したかどうかを確認します。



6.2 MR の制限事項

サービスが中断されることのないよう、制限事項を考慮する必要があります。ここでは **MaxCompute MR** の制限事項について概要を説明します。

MaxCompute MapReduce の制限事項は以下のとおりです。

制限事項	値	タイプ	設定項目	デフォルト値	設定可能かどうか	説明
インスタンスのメモリ占有量	**[256MB,12GB]**	メモリ制限	**odps.stage.mapper(reducer).mem** と **odps.stage.mapper(reducer).jvm.mem**	**2048 MB + 1024 MB**	可能	メモリは、フレームワークメモリ (デフォルトでは **2,048 MB**) と **JVM (Java** 仮想マシン) のヒープメモリ (デフォルトでは **1,024 MB**) を含む、単一の **map** インスタンスまたは **reduce** インスタンスによって占有されます。
リソース数	**256**	数制限	**N/A**	なし	不可	**1** つのジョブによって参照されるリソースの数は **256** を超えません。 テーブルとアーカイブは **1** 単位とみなされます。
入出力数	**1024** と **256**	数制限	**N/A**	なし	いいえ	**1** つのジョブの入力数は **1024** を超えません。 (テーブルのパーティションは **1** つの入力と見なされます。 入力テーブルの数は **64** を超えません)。 **1** つのジョブの出力数は **256** を超えません。
カウンター数	**64**	数制限	**N/A**	なし	不可	**1** つのジョブのカスタムカウンターの数は **64** を超えません。 カウンターのグループ名とカウンター名には # を含むことはできません。 カウンターのグループ名とカウンター名の全体の文字数は **100** 文字以内である必要があります。

制限事項	値	タイプ	設定項目	デフォルト値	設定可能かどうか	説明
map インスタンス	**[1, 100000]**	数制限	**odps.stage.mapper.num**	なし	可能	**1** つのジョブの **map** インスタンス数は、分割サイズに基づいたフレームワークによって計算されます。 入力テーブルがない場合は、値を直接 **odps.stage.mapper.num** に設定します。 最終的な範囲は **1** から **100,000** です。
reduce インスタンス	**[0, 2000]**	数制限	**odps.stage.reducer.num**	なし	可能	**1** つのジョブの **reduce** インスタンスの数は、デフォルトでは **map** インスタンスの **1/4** です。 **reduce** インスタンス数を、**0** から **2,000** の範囲で設定します。**reduce** によって処理されるデータ量は、**map** によって処理されるデータ量の数倍になる可能性があります。 この場合は、**reduce** フェーズ は遅くなり、最大で**2000**のインスタンスを開始します。
リトライ回数	**3**	数制限	**N/A**	なし	不可	単一の **map** インスタンスまたは **reduce** インスタンスに許可されているリトライ回数は、最大で **3** 回です。 リトライを許可しない一部の例外は、タスク実行が失敗する原因となる可能性があります。

制限事項	値	タイプ	設定項目	デフォルト値	設定可能かどうか	説明
ローカルデバッグモード	**100**	数制限	**N/A**	なし	不可	ローカルデバッグモードでは、**map** インスタンスの数はデフォルトで **2** で、**100** を超えません。**redece** インスタンスの数はデフォルトで **1** で、**100**を超えません。 **1** つの入力のダウンロードレコード数はデフォルトで **1** で、**100**を超えません。
リソースを繰り返し読み込む回数	**64**	数制限	**N/A**	なし	不可	**map** インスタンスまたは **reduce** インスタンスが **1** つのリソースを繰り返し読み込む回数は、**64** を超えません 。
リソースサイズ	**2 GB**	サイズ制限	**N/A**	なし	不可	ジョブが参照するリソースの合計サイズは **2 GB** を超えません。
分割サイズ	**[1,)**	サイズ制限	**odps.stage.mapper.split.size**	**256 MB**	可能	フレームワークは、設定された分割サイズに基づいて **map** を分割します。分割後に **map** の数が決まります。
文字列の列のコンテンツ長	**8 MB**	サイズ制限	**N/A**	なし	不可	**MaxCompute** テーブルの文字列カラムのコンテンツは **8MB** を超えません。
Worker 実行タイムアウト期間	［**1, 3600**］	時間制限	**odps.function.timeout**	**600**	可能	**map** または **reduce worker** が データを読み書きしない場合、または **context.progress()** を使用してハートビートデータを能動的に送信しない場合の、**worker** のタイムアウト期間。 デフォルト値は **600** 秒です。

制限事項	値	タイプ	設定項目	デフォルト値	設定可能かどうか	説明
MR によって参照されるテーブルで、サポートされるフィールドの型	**Bigint** 型、**Double** 型、**String** 型、**Datetime** 型、**Boolean** 型	データ型制限	**N/A**	なし	不可	**MR** タスクがテーブルを参照するとき、テーブルに異なる型のフィールドが含まれているとエラーが発生します。
OSS からのデータ読み込み		機能制限	**N/A**	なし	不可	サポートされていません。
MaxCompute 2.0 の新しい型		機能制限	**N/A**	なし	不可	サポートされていません。

6.3 機能紹介

6.3.1 基本概念

Map/Reduce

Map と **Reduce** は、対応する **Map/Reduce** メソッド、**setup** メソッド、および **cleanup** メソッドをサポートします。 **setup** メソッドは **Map/Reduce** メソッドの前に呼び出され、各 **worker** はそれを一度だけ呼び出します。

cleanup メソッドは **map/reduce** メソッドの後に呼び出され、各 **worker** はそれを一度だけ呼び出します。

詳細な例については、 「プログラム例」をご参照ください。

ソート/グループ

出力キーレコードの一部の列はソート列と見なされますが、ユーザー定義のコンパレーターはサポートされていません。ソート列から複数の列をグループ列として選択できますが、ユーザー定義のグループコンパレーターはサポートされていません。ソート列はデータのソートに使用され、グループ列はセカンダリソートに使用されます。

詳細は「*SecondarySort の例*」をご参照ください。

パーティション

パーティション列とカスタマイズされたパーティショナーの設定をサポートします。 パーティション列は、カスタマイズされたパーティショナーよりも優先順位が高くなります。

ハッシュ論理に従って、パーティショナーは **Map** 端末上の出力データをさまざまな **Reduce Worker** に分配します。

Combiner

シャッフルステージで隣接レコードを結合します。 さまざまな業務ロジックに従って、**Combiner** を使用するかどうかを選択します。

Combiner は **MapReduce** コンピューティングフレームワークを最適化するのに役立ち、**Combiner** のロジックは一般に **Reduce** と似ています。**Map** がデータを出力した後、フレームワークは **Map** 端末上で同じキー値を持つデータに対してローカル結合操作を実行します。

詳細は「*WordCount のコード例*」をご参照ください。

6.3.2 コマンド

MaxCompute コンソールには、**MapReduce** ジョブを実行するための **JAR** コマンドがあります。 詳細な構文は次のとおりです。

```
使用方法:
jar [<GENERIC_OPTIONS>] <MAIN_CLASS> [ARGS];
        -conf <configuration_file> アプリケーション設定ファイルを指定します
        -resources <resource_name_list> mapper や reducer で使用される
ファイル\テーブルリソース。カンマで区切ります。
        -classpath <local_file_list> mainClass の実行に使用される
classpath
        -D <name>=<value> プロパティ値のペア。mainClass の実行に使用されます
        -l ローカルモードでジョブを実行します
例:
    jar -conf /home/admin/myconf -resources a.txt,example.jar -
classpath ../lib/example.jar:./other_lib.jar -Djava.library.path=./
native -Xmx512M mycompany.WordCount -m 10 -r 10 in out;
```

`<GENERIC_OPTIONS>` には、次のパラメーターが含まれます (オプションパラメーター)。

- **-conf <configuration file>: JobConf** 設定ファイルを指定します。
- **-resources <resource_name_list>: MapReduce** 実行時に使用されるリソース文を示します。 通常、**Map/Reduce** 関数が含まれるリソース名は、**'resource_name_list'** で指定する必要があります。



注：

ユーザーが **Map/Reduce** 関数で他の **MaxCompute** リソースを読み取った場合は、それらのリソース名も **'resource_name_list'** に追加する必要があります。

複数のリソースはカンマ **(,)** で区切ります。 スパンプロジェクトリソースを使用する必要がある場合は、接頭辞 `PROJECT/resources/`、たとえば `-resources otherproject/resources/resfile` を追加します。

Map/Reduce 関数でリソースを読み取る方法については、「リソースの使用例」をご参照ください。

- **-classpath <local_file_list>: ‘main’** クラスのローカル **JAR** パッケージを指定するために使用されるクラスパス (相対パスと絶対パスを含みます)。

  パッケージ名は、システムのデフォルトのファイル区切り文字を使用して区切られます。 通常、区切り文字は **Windows** システムではセミコロン **(;)**、**Linux** システムではカンマ **(,)** です。

  
  注：
  ほとんどの場合、*WordCount* コード例など、**main** クラスと **Map/Reduce** 関数をパッケージに記述します。 つまり、プログラム例の実行期間中、**mapreduce-examples.jar** が **'-resources'** パラメーターと **'-classpath'** パラメーターに表示されますが、**'-resources'** は **Map/Reduce** 関数を参照し、分散環境で動作します。また **'-classpath'** は **'main'** クラスを参照し、ローカルで動作します。 **JAR** パッケージの指定されたパスもローカルパスです。
- **-D <prop_name>=<prop_value>:** ローカルモードの **<mainClass>** の複数の **Java** プロパティを定義します。
- **-l: MapReduce** ジョブをローカルモードで実行します。主にプログラムのデバッグに使用されます。

オプション **'-conf'** で設定ファイル **'JobConf'** を指定します。 このファイルは **SDK** の **JobConf** 設定を変更します。

設定ファイル **‘JobConf’** の例は次のとおりです。

```
<configuration>
       <property>
          <name>import.filename</name>
          <value>resource.txt</value>
       </property>
    </configuration>
```

上記の例では、変数 **‘import.filename’** が定義され、値は **‘resource.txt’** です。

ユーザーは **MapReduce** プログラムの **JobConf** インターフェイスを介してこの変数値を取得します。 または、**SDK** の **JobConf** インターフェイスを介して値を取得することも可能です。 詳細な例については、 「リソースの使用例」をご参照ください。

例:

```
add jar data\mapreduce-examples.jar;
    jar -resources mapreduce-examples.jar -classpath mapreduce-
examples.jar
        org.alidata.odps.mr.examples.WordCount wc_in wc_out;
    add file data\src.txt;
    add jar data\mapreduce-examples.jar;
    jar -resources src.txt,mapreduce-examples.jar -classpath data\
mapreduce-examples.jar
        org.alidata.odps.mr.examples.WordCount wc_in wc_out;
    add file data\a.txt;
    add table wc_in as test_table;
    add jar data\work.jar;
    jar -conf odps-mapred.xml -resources a.txt,test_table,work.jar
        -classpath data\work.jar:otherlib.jar
        -D import.filename=resource.txt org.alidata.odps.mr.examples.
WordCount args;
```

6.3.3 入出力

- 組み込みデータ型には、**BIGINT**、**DOUBLE**、**STRING**、**DATETIME**、および **BOOLEAN** があります。 ユーザー定義型 **(UDF)** はサポートしていません。
- 複数テーブル入力が可能で、入力テーブルのスキーマは異なっていてもかまいません。 **Map** 関数では、現在のレコードに対応するテーブル情報を取得します。
- 入力は **null** 値でもかまいません。 入力としてのビューはサポートされていません。
- **Reduce** は複数の出力を受け入れ、異なるテーブルまたは同じテーブル内の異なるパーティションにデータを出力します。 異なる出力のスキーマは異なっていてもかまいません。 異なる出力はラベルによって区別されますが、デフォルトの出力にはラベルは必要ありません。 出力は空にできません。

入出力例の詳細は、「プログラム例」をご参照ください。

6.3.4 リソース

Map/Reduce セクションで **MaxCompute** リソースの詳細を確認できます。 **Map/Reduce** の **worker** はリソースをメモリにロードして、後で使用するためにコードを適用できます。

詳細は「リソースの使用例」をご参照ください。

6.3.5 ローカル実行

基本ステージの概要

ローカル実行の前提条件: **jar** コマンドの `-local` パラメーターを設定することで、ローカルで **MapReduce** 実行プロセスをシミュレートして、ローカルデバッグを開始することができます。

ローカルオペレーション時: クライアントは、必要な入力テーブルのメタ情報、リソース、出力テーブルのメタ情報を **MaxCompute** からダウンロードし、**warehouse** という名前のローカルディレクトリに保存します。

プログラムの実行後: 計算結果が **warehouse** 内のファイルに出力されます。入力テーブルと参照先リソースがローカル **warehouse** ディレクトリにダウンロードされている場合、**warehouse** ディレクトリ内のデータとファイルは次回の実行時に直接参照されるので、ダウンロードプロセスを繰り返す必要はありません。

ローカル実行環境と実行分散環境の違い

ローカルオペレーションコースでは、複数の **Map** および **Reduce worker** はまだデータ処理を開始していません。 しかし、これらの **worker** は同時に実行するのではなく、順次実行します。

シミュレーションプロセスと実際の分散オペレーションの相違点は次のとおりです。

- 入力テーブルの行数に制限があります。 現在、最大 **100** 行のデータがダウンロード可能です。
- リソース使用量: 分散環境では、**MaxCompute** は参照されるリソースのサイズを制限しています。 詳細は「アプリケーション制限」をご参照ください。 なお、ローカル実行環境では、リソースサイズに制限はありません。
- セキュリティ制限: 分散環境で動作する **MaxCompute**、**MapReduce**、および **UDF** プログラムは、*Java* サンドボックスによって制限されます。ローカルオペレーションでは、この制限は適用されません。

例:

ローカルオペレーションの例を次に示します。

```
    odps:my_project> jar -l com.aliyun.odps.mapred.example.WordCount
wc_in wc_out
    Summary:
    counters: 10
        map-reduce framework
                combine_input_groups=2
                combine_output_records=2
                map_input_bytes=4
                map_input_records=1
                map_output_records=2
                map_output_[wc_out]_bytes=0
                map_output_[wc_out]_records=0
```

```
                reduce_input_groups=2
                reduce_output_[wc_out]_bytes=8
                reduce_output_[wc_out]_records=2
    OK
```

WordCount の例の詳細については、「*WordCount* のコード例」をご参照ください。

初めてローカルデバッグコマンドを実行した場合、コマンドが正常に実行されると、**warehouse** という名前のパスが現在のパスに表示されます。 **warehouse** のディレクトリ構造は以下のとおりです。

```
<warehouse>
   |____my_project(project directory)
          |____ <__tables__>
          | |__wc_in(table directory)
          | | |____ data(file)
          |       |      |
          | | |____ <__schema__> (file)
          | |__wc_out(table data directory)
          | |____ data(file)
          |              |
          | |____ <__schema__> (file)
          |
          |____ <__resources__>
                  |
                  |___table_resource_name (table resource)
                  | |____<__ref__>
                  |
                  |___ file_resource_name (file resource)
```

- **myproject** と同じ階層のディレクトリはプロジェクトを示します。**wcin** と **wc_out** はテーブルを示します。 **JAR** コマンドでユーザーが読み込んだテーブルファイルは、このディレクトリにダウンロードされます。
- <**__schema__**> の内容は、テーブルのメタ情報を示します。 形式は次のように定義されています。

  ```
  project=local_project_name
  table=local_table_name
  columns=col1_name:col1_type,col2_name:col2_type
  partitions=p1:STRING,p2:BIGINT
  ```

  列と列タイプはコロン (**:**) で区切り、列同士はカンマ (**,**) で区切ります。 <**__schema__**> ファイルに応じて、プロジェクト名とテーブル名を宣言し (`project_name.table_name` など)、カンマ (**,**) で区切ってから列定義を宣言する必要があります (`project_name.table_name,col1_name:col1_type,col2_name:col2_type,……`)。

- ファイル **data** はテーブルデータを示します。列数および対応するデータは、"*_schema_*" の定義に従う必要があります。 また、列の追加や欠落は許可されていません。

  wc_in の "*_schema_*" の内容は次のとおりです。

  ```
  my_project.wc_in,key:STRING,value:STRING
  ```

  'data' の内容は次のとおりです。

  ```
  0,2
  ```

  クライアントは **MaxCompute** からテーブルのメタ情報とデータの一部をダウンロードして、前述の **2** つのファイルに保存します。 この例をもう一度実行すると、ディレクトリ **'wc_in'** 内のデータが直接使用され、再度ダウンロードされることはありません。

   注：

  MaxCompute からデータをダウンロードする機能は、**MapReduce** ローカルオペレーションモードでのみサポートされています。*Eclipse* 開発プラグインでローカルデバッグを実行した場合は、**MaxCompute** のデータをローカルにダウンロードすることはできません。

  wc_out の **'_schema_'** の内容は次のとおりです。

  ```
  my_project.wc_out,key:STRING,cnt:BIGINT
  ```

  'data' の内容は次のとおりです。

  ```
  0,1
  2,1
  ```

  クライアントは **MaxCompute** から **wc_out** のメタ情報をダウンロードし、ファイル "*_schema_*" に保存します。 ファイル **'data'** は、ローカルオペレーション後に生成された結果データファイルです。

   注：

  - ユーザーは "*_schema_*" ファイルと **'data'** を編集して、これら **2** つのファイルを対応するテーブルディレクトリに置くこともできます。
  - ローカルで実行中、クライアントはテーブルディレクトリが既に存在することに気付くので、**MaxCompute** からこのテーブル情報をダウンロードしません。ローカルのテーブルディレクトリは、**MaxCompute** に存在しないテーブルであっても構いません。

6.4 プログラム例

6.4.1 WordCount サンプル

前提条件

1. テストプログラムの Jar パッケージを準備します。 パッケージの名前を **mapreduce-examples.jar**、ローカルストレージパスを *data\resources* とします。

   ・ テーブルを作成します。

   ```
   create table wc_in (key string, value string);
   create table wc_out(key string, cnt bigint);
   ```

   ・ リソースを追加します。

   ```
   add jar data\resources\mapreduce-examples.jar -f;
   ```

2. **WordCount** をテストするためのテーブルとリソースを準備します。
3. **tunnel** を実行してデータをインポートします。

   ```
   tunnel upload data wc_in;
   ```

   テーブル **wc_in** にインポートされたデータファイルの内容は次のとおりです。

   ```
   hello,odps
   ```

手順

WordCount を **odpscmd** で実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.WordCount wc_in wc_out
```

予想される出力

出力テーブル **wc_out** の内容は次のとおりです。

```
+------------+------------+
| key | cnt |
+------------+------------+
| hello | 1 |
| odps | 1 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    Import java. util. iterator;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
```

```
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    public class WordCount {
      public static class TokenizerMapper extends MapperBase {
        private Record word;
        private Record one;
        @Override
        public void setup(TaskContext context) throws IOException{
          word = context.createMapOutputKeyRecord();
          one = context.createMapOutputValueRecord();
          one.set(new Object[] { 1L });
          System.out.println("TaskID:" + context.getTaskID().toString
());
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            throws IOException {
          for (int i = 0; i < record.getColumnCount(); i++) {
            word.set(new Object[] { record.get(i).toString() });
            context.write(word, one);
          }
        }
      }
      /**
       * 出力を合計することで map 出力を結合する combiner クラス
       **/
      public static class SumCombiner extends ReducerBase {
        private Record count;
        @Override
        public void setup(TaskContext context) throws IOException{
          count = context.createMapOutputValueRecord();
        }
        // Assemblyer は reducer と同じインターフェイスを実装します。mapper
 上でローカルに実行される reduce の mapper の出力を直ちに減らすことができます。
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            throws IOException {
          long c = 0;
          while(values.hasNext()) {
            Record val = values.next();
            c += (Long) val.get(0);
          }
          count.set(0, c);
          context.write(key, count);
        }
      }
      /**
       * 入力値の合計を出力する reducer クラス
       **/
      public static class SumReducer extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
```

```
          Long Count = 0;
          while(values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
          }
          result.set(0, key.get(0));
          result.set(1, count);
          context.write(result);
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 2) {
           System.err.println("Usage: WordCount <in_table> <out_table
>");
          System.exit(2);
        }
        JobConf job = new JobConf();
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(SumCombiner.class);
        job.setReducerClass(SumReducer.class);
// mapper の中間結果のキーと値を設定するスキーマ。mapper の中間出力もレコード形式
です。
        job.setMapOutputKeySchema(SchemaUtils.fromString("word:string
"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("count:
bigint"));
        // 入出力テーブル情報の設定
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        Jobclient. runjob (job );
      }
    }
```

6.4.2 MapOnly サンプル

MapOnly ジョブの場合、**Map** は <キー、値> ペアを **MaxCompute** のテーブルに直接送信します。 必要なのは、出力テーブルの指定だけです。 ただし、**Map** によって出力される キーおよび値のメタデータの指定はスキップできます。

前提条件

1. テストプログラムの **JAR** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"** 、ローカルストレージパスを "*data\resources*" とします。
2. **MapOnly** をテストするためのテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table wc_in (key string, value string);
     ```

```
create table wc_out(key string, cnt bigint);
```

・ リソースを追加します。

```
add jar data\resources\mapreduce-examples.jar -f;
```

3. **tunnel** コマンドを使用してデータをインポートします。

```
tunnel upload data wc_in;
```

データファイルの内容は“**mr_src**”テーブルにインポートされます。

```
hello,odps
hello,odps
```

手順

odpscmd で **MapOnly** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.MapOnly wc_in wc_out map
```

予想される出力

出力テーブル **wc_out** の内容は次のとおりです。

```
+------------+------------+
| key | cnt |
+------------+------------+
| hello | 1 |
| hello | 1 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.mapred.JobClient;
    Import com. aliyun. ODPS. mapred. mapperbase;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.data.TableInfo;
    public class MapOnly {
      public static class MapperClass extends MapperBase {
        @Override
        public void setup(TaskContext context) throws IOException{
          boolean is = context.getJobConf().getBoolean("option.mapper.
setup", false);
          // Main 関数は、jobconf で option.mapper.setup を true に設定
し、次のロジックを実行します。
          if (is) {
            Record result = context.createOutputRecord();
            result.set(0, "setup");
            result.set(1, 1L);
```

```
            context.write(result);
          }
        }
        @Override
        public void map(long key, Record record, TaskContext context)
throws IOException {
          boolean is = context.getJobConf().getBoolean("option.mapper.
map", false);
          // Main 関数は、jobconf で option.mapper.map を true に設定し、
次のロジックを実行します。
          if (is) {
            Record result = context.createOutputRecord();
            result.set(0, record.get(0));
            result.set(1, 1L);
            context.write(result);
          }
        }
        @Override
        public void cleanup(TaskContext context) throws IOException {
          boolean is = context.getJobConf().getBoolean("option.mapper.
cleanup", false);
          // Main 関数は、jobconf で option.mapper.cleanup を true に設定
し、次のロジックを実行します。
          if (is) {
            Record result = context.createOutputRecord();
            result.set(0, "cleanup");
            result.set(1, 1L);
            context.write(result);
          }
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 2 && args.length ! = 3) {
          System.err.println("Usage: OnlyMapper <in_table> <out_table>
 [setup|map|cleanup]");
          System.exit(2);
        }
        JobConf job = new JobConf();
        job.setMapperClass(MapperClass.class);
        // maponly ジョブの場合、reducer の数は明示的に 0 を設定する必要があり
ます。
        job.setNumReduceTasks(0);
        // Input Output のテーブル情報の設定
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        if (args.length == 3) {
          String options = new String(args[2]);
        // Jobconf はカスタムキーと値を設定でき、getJobConf は getJobConf の
コンテキストを介して  mapper の関連設定を取得できます。
          if (options.contains("setup")) {
            job.setBoolean("option.mapper.setup", true);
          }
          if (options.contains("map")) {
            job.setBoolean("option.mapper.map", true);
          }
          if (options.contains("cleanup")) {
            job.setBoolean("option.mapper.cleanup", true);
          }
        }
        Jobclient. runjob (job );
      }
```

```
    }
```

6.4.3 複数の入出力

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"** 、ローカルストレージパスを "*data\resources*" とします。
2. 複数の入出力をテストするためのテーブルとリソースを準備します。

   ・ テーブルを作成します。

   ```
   create table wc_in1(key string, value string);
   create table wc_in2(key string, value string);
   create table mr_multiinout_out1 (key string, cnt bigint);
   create table mr_multiinout_out2 (key string, cnt bigint)
   partitioned by (a string, b string);
   alter table mr_multiinout_out2 add partition (a='1', b='1');
   alter table mr_multiinout_out2 add partition (a='2', b='2');
   ```

   ・ リソースを追加します。

   ```
   add jar data\resources\mapreduce-examples.jar -f;
   ```

3. **tunnel** を実行してデータをインポートします。

   ```
   tunnel upload data1 wc_in1;
   tunnel upload data2 wc_in2;
   ```

   "wc_in1" テーブルにインポートされたデータは次のとおりです。

   ```
   hello,odps
   ```

   "wc_in2" テーブルにインポートされたデータは次のとおりです。

   ```
   hello,world
   ```

手順

odpscmd で **MultipleInOut** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.MultipleInOut wc_in1,wc_in2
mr_multiinout_out1,mr_multiinout_out2|a=1/b=1|out1,mr_multiinout_out2|
a=2/b=2|out2;
```

予想される出力

‘mr_multiinout_out1’ の内容は次のとおりです。

```
+------------+------------+
| key | cnt |
+------------+------------+
```

```
| default | 1 |
+------------+------------+
```

‘mr_multiinout_out2’ の内容は次のとおりです。

```
+--------+------------+---+---+
| key | cnt | a | b |
+--------+------------+---+---+
| odps | 1 | 1 | 1 |
| world | 1 | 1 | 1 |
| out1 | 1 | 1 | 1 |
| hello | 2 | 2 | 2 |
| out2 | 1 | 2 | 2 |
+--------+------------+---+---+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    Import java.util.iterator;
    import java.util.LinkedHashMap;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * 複数の入出力例
     **/
    public class MultipleInOut {
      public static class TokenizerMapper extends MapperBase {
        Record word;
        Record one;
        @Override
        public void setup(TaskContext context) throws IOException{
          word = context.createMapOutputKeyRecord();
          one = context.createMapOutputValueRecord();
          one.set(new Object[] { 1L });
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws ioexception {
          for (int i = 0; i < record.getColumnCount(); i++) {
            word.set(new Object[] { record.get(i).toString() });
            context.write(word, one);
          }
        }
      }
      public static class SumReducer extends ReducerBase {
        private Record result;
        private Record result1;
        private Record result2;
        @Override
        public void setup(TaskContext context) throws IOException{
          // 異なる出力の場合、異なるレコードを作成する必要があり、ラベルで区別し
ます
```

```
          result = context.createOutputRecord();
          result1 = context.createOutputRecord("out1");
          result2 = context.createOutputRecord("out2");
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          Long Count = 0;
          while(values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
          }
          long mod = count % 3;
          if (mod == 0) {
            result.set(0, key.get(0));
            result.set(1, count);
            // ラベルが指定されていません。 デフォルトの出力が採用されます。
            context.write(result);
          } else if (mod == 1) {
            result1.set(0, key.get(0));
            result1.set(1, count);
            context.write(result1, "out1");
          } else {
            result2.set(0, key.get(0));
            result2.set(1, count);
            context.write(result2, "out2");
          }
        }
        @Override
        public void cleanup(TaskContext context) throws IOException {
          Record result = context.createOutputRecord();
          result.set(0, "default");
          result.set(1, 1L);
          context.write(result);
          Record result1 = context.createOutputRecord("out1");
          result1.set(0, "out1");
          result1.set(1, 1L);
          context.write(result1, "out1");
          Record result2 = context.createOutputRecord("out2");
          result2.set(0, "out2");
          result2.set(1, 1L);
          context.write(result2, "out2");
        }
      }
      // パーティション文字列 ("ds = 1/pt = 2" など) を map 形式に変換
      public static LinkedHashMap<String, String> convertPartSpecToMap
(
          String partSpec) {
        LinkedHashMap<String, String> map = new LinkedHashMap<String,
String>();
        if (partSpec ! = null && ! partSpec.trim().isEmpty()) {
          String[] parts = partSpec.split("/");
          for (String part : parts) {
            String[] ss = part.split("=");
            if (ss.length ! = 2) {
              throw new RuntimeException("ODPS-0730001: error part
spec format: "
                  + partSpec);
            }
            map.put(ss[0], ss[1]);
          }
        }
        return map;
```

```
      }
      public static void main(String[] args) throws Exception {
        String[] inputs = null;
        String[] outputs = null;
        if (args.length == 2) {
          inputs = args[0].split(",");
          outputs = args[1].split(",");
        } else {
          System.err.println("MultipleInOut in... out...") ;
          System.exit(1);
        }
        JobConf job = new JobConf();
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeySchema(SchemaUtils.fromString("word:string
"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("count:
bigint"));
        // ユーザー入力テーブルの文字列を解析します
        for (String in : inputs) {
          String[] ss = in.split("\\|");
          if (ss.length == 1) {
            InputUtils.addTable(TableInfo.builder().tableName(ss[0]).
build(), job);
          } else if (ss.length == 2) {
            LinkedHashMap<String, String> map = convertPartSpecToMap(
ss[1]);
            InputUtils.addTable(TableInfo.builder().tableName(ss[0]).
partSpec(map).build(), job);
          } else {
            System.err.println("Style of input: " + in + " is not
right");
            System.exit(1);
          }
        }
        // ユーザー出力テーブルの文字列を解析します
        for (String out : outputs) {
          String[] ss = out.split("\\|");
          if (ss.length == 1) {
            OutputUtils.addTable(TableInfo.builder().tableName(ss[0]).
build(), job);
          } else if (ss.length == 2) {
            LinkedHashMap<String, String> map = convertPartSpecToMap(
ss[1]);
            OutputUtils.addTable(TableInfo.builder().tableName(ss[0]).
partSpec(map).build(), job);
          } else if (ss.length == 3) {
            if (ss[1].isEmpty()) {
              LinkedHashMap<String, String> map = convertPartSpecToMap
(ss[2]);
              OutputUtils.addTable(TableInfo.builder().tableName(ss[0
]).partSpec(map).build(), job);
            } else {
              LinkedHashMap<String, String> map = convertPartSpecToMap
(ss[1]);
              OutputUtils.addTable(TableInfo.builder().tableName(ss[0
]).partSpec(map)
                  .label(ss[2]).build(), job);
            }
          } else {
            System.err.println("Style of output: " + out + " is not
right");
            System.exit(1);
          }
```

```
        }
        Jobclient. runjob (job );
      }
    }
```

6.4.4 マルチタスクサンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"** 、ローカルストレージパスを "*data\resources*" とします。
2. **MultiJobs** をテストするためのテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table mr_empty (key string, value string);
     create table mr_multijobs_out (value bigint);
     ```

   - リソースを追加します。

     ```
     add table mr_multijobs_out as multijobs_res_table -f;
     Add jar data \ resources \ mapreduce-examples.jar-f;
     ```

手順

odpscmd で **MultiJobs** を実行します。

```
jar -resources mapreduce-examples.jar,multijobs_res_table -classpath
data\resources\mapreduce-examples.jar
 com.aliyun.odps.mapred.open.example.MultiJobs mr_multijobs_out;
```

予想される出力

出力テーブル **'mr_multijobs_out'** は次のとおりです。

```
+------------+
| value |
+------------+
| 0 |
+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    import java.util.Iterator;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.RunningJob;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
```

```
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * MultiJobs
     *
     * 複数のジョブを実行
     *
     **/
    public class MultiJobs {
      public static class InitMapper extends MapperBase {
        @Override
        public void setup(TaskContext context) throws IOException{
          Record record = context.createOutputRecord();
          long v = context.getJobConf().getLong("multijobs.value", 2);
          record.set(0, v);
          context.write(record);
        }
      }
      public static class DecreaseMapper extends MapperBase {
        @Override
        public void cleanup(TaskContext context) throws IOException {
          // main 関数で定義された可変値を JobConf から取得
          long expect = context.getJobConf().getLong("multijobs.expect
.value", -1);
          long v = -1;
          int count = 0;
          // リソーステーブルのデータ（直前のジョブの出力テーブル）の読み取り
          Iterator<Record> iter = context.readResourceTable("
multijobs_res_table");
          while (iter.hasNext()) {
            Record r = iter.next();
            v = (Long) r.get(0);
            if (expect ! = v) {
              throw new IOException("expect: " + expect + ", but: " +
v);
            }
            count++;
          }
          if (count ! = 1) {
            throw new IOException("res_table should have 1 record, but
: " + count);
          }
          Record record = context.createOutputRecord();
          v--;
          record.set(0, v);
          context.write(record);
          // カウンターの設定。ジョブが正常終了した後、main 関数で取得できます。
          context.getCounter("multijobs", "value").setValue(v);
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 1) {
          System.err.println("Usage: TestMultiJobs <table>");
          System.exit(1);
        }
        String tbl = args[0];
        long iterCount = 2;
        System.err.println("Start to run init job.") ;
        JobConf initJob = new JobConf();
        initJob.setLong("multijobs.value", iterCount);
        initJob.setMapperClass(InitMapper.class);
        InputUtils.addTable(TableInfo.builder().tableName("mr_empty").
build(), initJob);
        OutputUtils.addTable(TableInfo.builder().tableName(tbl).build
(), initJob);
```

```
        initJob.setMapOutputKeySchema(SchemaUtils.fromString("key:
string"));
        initJob.setMapOutputValueSchema(SchemaUtils.fromString("value:
string"));
        // Maponly ジョブは、reducer 数を明示的に 0 に設定する必要があります
        initJob.setNumReduceTasks(0);
        JobClient.runJob(initJob);
        while (true) {
          System.err.println("Start to run iter job, count: " +
iterCount);
          JobConf decJob = new JobConf();
          decJob.setLong("multijobs.expect.value", iterCount);
          decJob.setMapperClass(DecreaseMapper.class);
          InputUtils.addTable(TableInfo.builder().tableName("mr_empty
").build(), decJob);
          OutputUtils.addTable(TableInfo.builder().tableName(tbl).
build(), decJob);
          // Maponly job needs to explicitly set reducer number to 0
          decJob.setNumReduceTasks(0);
          RunningJob rJob = JobClient.runJob(decJob);
          iterCount--;
          // Exit the loop if the number of iterations has been
reached
          if (rJob.getCounters().findCounter("multijobs", "value").
getValue() == 0) {
            break;
          }
        }
        if (iterCount ! = 0) {
          throw new IOException("Job failed.") ;
        }
      }
    }
```

6.4.5 セカンダリソートサンプル

前提条件

1. テストプログラムの **JAR** パッケージを準備します。パッケージの名前を **" mapreduce-examples.jar "** と仮定します。ローカルストレージパスは *data\resources* です。
2. **SecondarySort** オペレーションをテストするためのテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table ss_in(key bigint, value bigint);
     ```

```
create table ss_out(key bigint, value bigint)
```

・ リソースを追加します。

```
add jar data\resources\mapreduce-examples.jar -f;
```

3. **tunnel** コマンドでデータをインポートします。

```
tunnel upload data ss_in;
```

テーブル **" ss_in "** にインポートされたデータファイルの内容は 次のとおりです。

```
1,2
2,1
1,1
2,2
```

手順

odpscmd で **SecondarySort** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.SecondarySort ss_in ss_out;
```

予想される出力

出力テーブル “ **ss_out** ” の内容は次のとおりです。

```
| key | value |

| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    import java.util.Iterator;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.data.TableInfo;


     * これは、ODPS Map/Reduce アプリケーションの例です。 1 レコードに 2 つの整
数を含む必要がある
     * 入力テーブルを読み取ります。 出力は、1 番目と 2 番目の数字でソートされ、
     * 1 番目の数字でグループ化されます。
```

```
    public class SecondarySort {

       * 各行から 2 つの整数を読み取り、((left,
       * right), right) としてキーと値のペアを生成します。

      public static class MapClass extends MapperBase {
        private Record key;
        private Record value;
        @Override
        public void setup(TaskContext context) throws IOException {
          key = context.createMapOutputKeyRecord();
          value = context.createMapOutputValueRecord();

        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            throws IOException {
          long left = 0;
          long right = 0;
          if (record.getColumnCount() > 0) {
            left = (Long) record.get(0);
            if (record.getColumnCount() > 1) {
              right = (Long) record.get(1);

            key.set(new Object[] { (Long) left, (Long) right });
            value.set(new Object[] { (Long) right });
            context.write(key, value);



       * 入力値の合計を出力する reducer クラス

      public static class ReduceClass extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException {
          result = context.createOutputRecord();

        @Override
        public void reduce(Record key, Iterator<Record> values,
TaskContext context)
            throws IOException {
          result.set(0, key.get(0));
          while (values.hasNext()) {
            Record value = values.next();
            result.set(1, value.get(0));
            context.write(result);



      public static void main(String[] args) throws Exception {
        if (args.length ! = 2) {
        System.err.println("Usage: secondarysrot <in> <out>");
          System.exit(2);

        JobConf job = new JobConf();
        job.setMapperClass(MapClass.class);
        job.setReducerClass(ReduceClass.class);
        // 複数の列をキーに設定
        // ペアの 1 番目と 2 番目の部分を比較
        job.setOutputKeySortColumns(new String[] { "i1", "i2" });
```

```
        // ペアの 1 番目の部分に基づくパーティション
        job.setPartitionColumns(new String[] { "i1" });
        // ペアの 1 番目の部分に基づくグルーピングコンパレーター
        job.setOutputGroupingColumns(new String[] { "i1" });
        // map 出力は LongPair, Long
        job.setMapOutputKeySchema(SchemaUtils.fromString("i1:bigint,i2
:bigint"));
        Job. Fig (schemeiutils. fromstring ("i2x: bigint "));
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        JobClient.runJob(job);
        System.exit(0);
```

6.4.6 リソースサンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を " **mapreduce-examples.jar** "、ローカルストレージパスを " *data\resources* " とします。
2. テストテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table mr_upload_src(key bigint, value string);
     ```

   - リソースを追加します。

     ```
     add jar data\resources\mapreduce-examples.jar -f;
     add file data\resources\import.txt -f;
     ```

   - **import.txt** の内容:

     ```
     1000,odps
     ```

手順

odpscmd で **Upload** を実行します。

```
jar -resources mapreduce-examples.jar,import.txt -classpath data\
resources\mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.Upload import.txt mr_upload_src;
```

予想される出力

出力テーブル “ **mr_upload_src** ” の内容は次のとおりです。

```
+------------+------------+
| key | value |
+------------+------------+
| 1000 | odps |
```

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

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.BufferedInputStream;
    import java.io.FileNotFoundException;
    import java.io.IOException;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * Upload
     *
     * テキストファイルのデータをテーブルにインポート
     *
     **/
    public class Upload {
      public static class UploadMapper extends MapperBase {
        @Override
        public void setup(TaskContext context) throws IOException{
          Record record = context.createOutputRecord();
          StringBuilder importdata = new StringBuilder();
          BufferedInputStream bufferedInput = null;
          try {
            byte[] buffer = new byte[1024];
            int bytesRead = 0;
            String filename = context.getJobConf().get("import.
filename");
            bufferedInput = context.readResourceFileAsStream(filename
);
            while ((bytesRead = bufferedInput.read(buffer)) ! = -1) {
              String chunk = new String(buffer, 0, bytesRead);
              importdata.append(chunk);
            }
            String lines[] = importdata.toString().split("\n");
            for (int i = 0; i < lines.length; i++) {
              String[] ss = lines[i].split(",");
              record.set(0, Long.parseLong(ss[0].trim()));
              record.set(1, ss[1].trim());
              context.write(record);
            }
          } catch (FileNotFoundException ex) {
            throw new IOException(ex);
          } catch (IOException ex) {
            throw new IOException(ex);
          } finally {
          }
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws ioexception {
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 2) {
```

```
            System.err.println("Usage: Upload <import_txt> <out_table
>");
            System.exit(2);
        }
        JobConf job = new JobConf();
        job.setMapperClass(UploadMapper.class);
        // リソース名の設定。map の jobconf から取得可能。
        job.set("import.filename", args[0]);
        // Maponly ジョブは、明示的に reducer 数を 0 に設定する必要があります
        job.setNumReduceTasks(0);
        job.setMapOutputKeySchema(SchemaUtils.fromString("key:bigint
"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("value:
string"));
        InputUtils.addTable(TableInfo.builder().tableName("mr_empty").
build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        Jobclient. runjob (job );
      }
    }
```

次の方法で **JobConf** を設定できます。

- **SDK** の **JobConf** インターフェイスを使用します。 この方法は前述の例で使用されています。また、これは最も推奨される方法であり、優先順位が最も高くなります。
- **jar** コマンドラインで、新しい **JobConf** ファイルをパラメーター -conf で指定します。 この方法は、優先順位が最も低くなります。

6.4.7 Counter サンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"** 、ローカルストレージパスを "*data\resources*" とします。
2. **UserDefinedCounters** テストテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table wc_in (key string, value string);
     ```

```
create table wc_out(key string, cnt bigint);
```

・ リソースを追加します。

```
add jar data\resources\mapreduce-examples.jar -f;
```

3. **tunnel** コマンドを使用してデータをインポートします。

```
tunnel upload data wc_in;
```

テーブル **"wc_in"** の **"wc_in"** にインポートされたデータは次のとおりです。

```
hello,odps
```

手順

odpscmd で **UserDefinedCounters** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.UserDefinedCounters wc_in wc_out
```

予想される出力

Counters の出力は次のとおりです。

```
Counters: 3
com.aliyun.odps.mapred.open.example.UserDefinedCounters$MyCounter
MAP_TASKS=1
REDUCE_TASKS=1
TOTAL_TASKS=2
```

出力テーブル“**wc_out**”の内容は次のとおりです。

```
+------------+------------+
| key | cnt |
+------------+------------+
| hello | 1 |
| odps | 1 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    import java.util.Iterator;
    import com.aliyun.odps.counter.Counter;
    import com.aliyun.odps.counter.Counters;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.mapred.RunningJob;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
```

```
    import com.aliyun.odps.data.TableInfo;
    /**
     *
     * ユーザー定義 Counters
     *
     **/
    public class UserDefinedCounters {
      enum MyCounter {
        TOTAL_TASKS, MAP_TASKS, REDUCE_TASKS
      }
      public static class TokenizerMapper extends MapperBase {
        private Record word;
        private Record one;
        @Override
        public void setup(TaskContext context) throws IOException{
          super.setup(context);
          Counter map_tasks = context.getCounter(MyCounter.MAP_TASKS);
          Counter total_tasks = context.getCounter(MyCounter.
TOTAL_TASKS);
          map_tasks.increment(1);
          total_tasks.increment(1);
          word = context.createMapOutputKeyRecord();
          one = context.createMapOutputValueRecord();
          one.set(new Object[] { 1L });
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws ioexception {
          for (int i = 0; i < record.getColumnCount(); i++) {
            word.set(new Object[] { record.get(i).toString() });
            context.write(word, one);
          }
        }
      }
      public static class SumReducer extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
          Counter reduce_tasks = context.getCounter(MyCounter.
REDUCE_TASKS);
          Counter maid = context. getcounter (mycounter );
          reduce_tasks.increment(1);
          total_tasks.increment(1);
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          Long Count = 0;
          while(values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
          }
          result.set(0, key.get(0));
          result.set(1, count);
          context.write(result);
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 2) {
          System.err
```

```
                .println("Usage: TestUserDefinedCounters <in_table> <
out_table>");
          System.exit(2);
        }
        JobConf job = new JobConf();
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeySchema(SchemaUtils.fromString("word:string
"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("count:
bigint"));
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        RunningJob rJob = JobClient.runJob(job);
        // ジョブが正常終了した後、ジョブ内のカスタムカウンターの値を取得できます
        Counters counters = rJob.getCounters();
        long m = counters.findCounter(MyCounter.MAP_TASKS).getValue();
        long r = counters.findCounter(MyCounter.REDUCE_TASKS).getValue
();
        long total = counters.findCounter(MyCounter.TOTAL_TASKS).
getValue();
        System.exit(0);
      }
    }
```

6.4.8 Grep サンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"** 、ローカルストレージパスを "*data\resources*" とします。
2. **Grep** をテストするためのテーブルとリソースを準備します。

   ・ テーブルを作成します。

   ```
   create table mr_src(key string, value string);
   create table mr_grep_tmp (key string, cnt bigint);
   create table mr_grep_out (key bigint, value string);
   ```

   ・ リソースを追加します。

   ```
   add jar data\resources\mapreduce-examples.jar -f;
   ```

3. **tunnel** コマンドを使用してデータをインポートします。

   ```
   tunnel upload data mr_src;
   ```

   テーブル“**mr_src**”にインポートされたデータファイルの内容

   ```
    hello,odps
   ```

```
hello,world
```

手順

odpscmd で **Grep** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.Grep mr_src mr_grep_tmp mr_grep_ou
t hello;
```

予想される出力

出力テーブル“**mr_grep_out**”の内容は次のとおりです。

```
+------------+------------+
| key | value |
+------------+------------+
| 2 | hello |
+------------+------------+
```

サンプルコード

```
package com.aliyun.odps.mapred.open.example;
import java.io.IOException;
import java.util.Iterator;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.mapred.JobClient;
import com.aliyun.odps.mapred.Mapper;
import com.aliyun.odps.mapred.MapperBase;
import com.aliyun.odps.mapred.ReducerBase;
import com.aliyun.odps.mapred.RunningJob;
import com.aliyun.odps.mapred.TaskContext;
import com.aliyun.odps.mapred.conf.JobConf;
import com.aliyun.odps.mapred.utils.InputUtils;
import com.aliyun.odps.mapred.utils.OutputUtils;
import com.aliyun.odps.mapred.utils.SchemaUtils;
/**
 *
 * Extracts matching regexs from input files and counts them.
 *
 **/
public class Grep {
  /**
   * RegexMapper
   **/
  public class RegexMapper extends MapperBase {
    private Pattern pattern;
    private int group;
    private Record word;
    private Record one;
    @Override
    public void setup(TaskContext context) throws IOException{
      JobConf job = (JobConf) context.getJobConf();
      pattern = Pattern.compile(job.get("mapred.mapper.regex"));
      group = job.getInt("mapred.mapper.regex.group", 0);
      word = context.createMapOutputKeyRecord();
```

```
          one = context.createMapOutputValueRecord();
          one.set(new Object[] { 1L });
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context) throws IOException {
          for (int i = 0; i < record.getColumnCount(); ++i) {
            String text = record.get(i).toString();
            Matcher = pattern. matcher (text );
            while (matcher.find()) {
              word.set(new Object[] { matcher.group(group) });
              context.write(word, one);
            }
          }
        }
      }
      /**
       * LongSumReducer
       **/
      public class LongSumReducer extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        @Override
        public void reduce(Record key, Iterator<Record> values,
TaskContext context) throws IOException {
          Long Count = 0;
          while(values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
          }
          result.set(0, key.get(0));-
          result.set(1, count);
          context.write(result);
        }
      }
      /**
       * A {@link Mapper} that swaps keys and values.
       **/
      public class InverseMapper extends MapperBase {
        private Record word;
        private Record count;
        @Override
        public void setup(TaskContext context) throws IOException{
          word = context.createMapOutputValueRecord();
          count = context.createMapOutputKeyRecord();
        }
        /**
         * The inverse function. Input keys and values are swapped.
         **/
        @Override
        public void map(long recordNum, Record record, TaskContext
context) throws IOException {
          word.set(new Object[] { record.get(0).toString() });
          count.set(new Object[] { (Long) record.get(1) });
          context.write(count, word);
        }
      }
      /**
       * IdentityReducer
       **/
      public class IdentityReducer extends ReducerBase {
```

```
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        /** Writes all keys and values directly to output. **/
        @Override
        public void reduce(Record key, Iterator<Record> values,
TaskContext context) throws IOException {
          result.set(0, key.get(0));
          while(values.hasNext()) {
            Record val = values.next();
            result.set(1, val.get(0));
            context.write(result);
          }
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length < 4) {
          System.err.println("Grep <inDir> <tmpDir> <outDir> <regex>
 [<group>]");
          System.exit(2);
        }
        JobConf grepJob = new JobConf();
        grepJob.setMapperClass(RegexMapper.class);
        grepJob.setReducerClass(LongSumReducer.class);
        grepJob.setMapOutputKeySchema(SchemaUtils.fromString("word:
string"));
        grepJob.setMapOutputValueSchema(SchemaUtils.fromString("count:
bigint"));
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), grepJob);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), grepJob);
        // grepjob の grep の正規表現の設定
        grepJob.set("mapred.mapper.regex", args[3]);
        if (args.length == 5) {
          grepJob.set("mapred.mapper.regex.group", args[4]);
        }
        @SuppressWarnings("unused")
        RunningJob rjGrep = JobClient.runJob(grepJob);
        // sortjob への入力としての Grepjob 出力
        JobConf sortJob = new JobConf();
        sortJob.setMapperClass(InverseMapper.class);
        sortJob.setReducerClass(IdentityReducer.class);
        sortJob.setMapOutputKeySchema(SchemaUtils.fromString("count:
bigint"));
        sortJob.setMapOutputValueSchema(SchemaUtils.fromString("word:
string"));-
        InputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), sortJob);
        OutputUtils.addTable(TableInfo.builder().tableName(args[2]).
build(), sortJob);
        sortJob.setNumReduceTasks(1); // 単一のファイルの書き込み
        sortJob.setOutputKeySortColumns(new String[] { "count" });
        @SuppressWarnings("unused")
        RunningJob rjSort = JobClient.runJob(sortJob);
      }
```

```
    }
```

6.4.9 Join サンプル

MaxCompute MapReduce フレームワークは、それ自体では結合ロジックをサポートしません。 したがって、自身の **map/reduce** 関数のデータの結合サンプルを適用する必要があり、これには追加の作業が必要になります。

たとえば、**2** つのテーブル **(Key bigint, value string)** と **(key bigint, value string)** を結合するために、出力テーブルが **chain bigint (value1 string, value2 string)** であるとします。ここで、**value1** と **value2** はスキャナの値です。

前提条件

1. 名前を **"maid”** 、ローカルストレージパスを **" data \ resources "** として、テストプログラム用の **jar** パッケージを準備します。
2. **Join** をテストするためのテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table mr_Join_src1(key bigint, value string);
     create table mr_Join_src2(key bigint, value string);
     create table mr_Join_out(key bigint, value1 string,value2 string);
     ```

   - リソースを追加します。

     ```
     add jar data\resources\mapreduce-examples.jar -f;
     ```

3. **tunnel** を実行してデータをインポートします。

   ```
   tunnel upload data1 mr_Join_src1;
   tunnel upload data2 mr_Join_src2;
   ```

   次のように **maid** データの内容をインポートします。

   ```
    1, hello
    2, ODPS
   ```

   次のように **maid** データの内容をインポートします。

   ```
   1, ODPS
   3,hello
   4, ODPS
   ```

手順

次のように **odpscmd** で **Join** します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
```

```
com.aliyun.odps.mapred.open.example.Join mr_Join_src1 mr_Join_src2
mr_Join_out;
```

予想される出力

ジョブが正常に完了すると、次のようにテーブル **"maid"** の内容が出力されます。

```
+------------+------------+------------+
| key | value1 | value2 |
+------------+------------+------------+
| 1 | hello | odps |
+------------+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    Import java. util. arraylist;
    import java.util.Iterator;
    import java.util.List;
    import org.apache.commons.logging.Log;
    import org.apache.commons.logging.LogFactory;
    Import com. aliyun. ODPS. Data. record;-
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * Join, mr_Join_src1/mr_Join_src2(key bigint, value string),
mr_Join_out(key
     * bigint, value1 string, value2 string)
     *
     */
    public class Join {
      public static final Log LOG = LogFactory.getLog(Join.class);
      public static class JoinMapper extends MapperBase {
        private Record mapkey;
        private Record mapvalue;
        private long tag;
        @Override
        public void setup(TaskContext context) throws IOException{
          mapkey = context.createMapOutputKeyRecord();
          mapvalue = context.createMapOutputValueRecord();
          tag = context.getInputTableInfo().getLabel().equals("left
") ? 0: 1;
        }
        @Override
        public void map(long key,Record record, TaskContext context)
            Throws ioexception {
          mapkey.set(0,record.get(0));
          mapkey.set(1,tag);
          for (int i = 1; i< record.getColumnCount();i++) {
            mapvalue.set(i -1, record.get(i));
          }
          context.write(mapkey,mapvalue);
        }
      }
```

```
      public static class JoinReducer extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        //Reduce 関数の全入力レコードは、同じキーになります
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          long k = key.getBigint(0);
          List<Object[]> leftValues = new ArrayList<Object[]>();
          // key + tag の組み合わせです。左テーブルのレコードデータが reduce
 関数の入力レコードの前になるように設定されているからです。
          while(values.hasNext()) {
            Record value = values.next();
            long tag = (Long)key.get(1);
            //左テーブルのデータは、最初にメモリにキャッシュされます
            if (tag == 0) {
              leftValues.add(value.toArray().clone());
            }else {
              //右テーブルと隣接するデータは、結合による出力となり、左テーブルの
すべてのデータを含みます。左テーブルのデータはすべてメモリにあります。
//この実装は、相対的にパフォーマンスが低く、関数を示しただけのものです。実用として推
奨しません。
              for (Object[] leftValue :leftValues) {
                int index = 0;
                result.set(index++,k);
                for (int i = 0;i<leftValue.length;i++) {
                  result.set(index++,leftValue[i]);
                }
                for (int i = 0;i< value.getColumnCount();i++) {
                  result.set(index++,value.get(i));
                }
                context.write(result);
              }
            }
          }
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length ! = 3) {
          System.err.println("Usage: Join <input table1> <input table2
> <out>");
          System.exit(2);
        }
        JobConf job = new JobConf();
        job.setMapperClass(JoinMapper.class);
        job.setReducerClass(JoinReducer.class);
        job.setMapOutputKeySchema(SchemaUtils.fromString("key:bigint,
tag:bigint"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("value:
string"));
        job.setPartitionColumns(new String[]{"key"});
        job.setOutputKeySortColumns(new String[]{"key", "tag"});
        job.setOutputGroupingColumns(new String[]{"key"});
        job.setNumReduceTasks(1);
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
label("left").build(), job);
        InputUtils.addTable(TableInfo.builder().tableName(args[1]).
label("right").build(), job);
        OutputUtils.addTable(TableInfo.builder().tableName(args[2]).
build(), job);
```

```
        Jobclient. runjob (job );
      }
    }
```

6.4.10 Sleep サンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を " **mapreduce-examples.jar** "、ローカルストレージパスを " *data\resources* " とします。
2. **SleepJob** 操作をテストするためのリソースを準備します。

```
Add jar data \ resources \ mapreduce-examples.jar-f;
```

手順

以下のように **odpscmd** で **Sleep** を実行します。

```
  jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
  com.aliyun.odps.mapred.open.example.Sleep 10;
  jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
  com.aliyun.odps.mapred.open.example.Sleep 100;
```

予想される出力

ジョブが正常に実行されます。 さまざまな **sleep** 時間の実行時間を比較して効果を判断します。

サンプルコード

```
package com.aliyun.odps.mapred.open.example;
import java.io.IOException;
import com.aliyun.odps.mapred.JobClient;
Import com.aliyun.odps.mapred.mapperbase;
import com.aliyun.odps.mapred.conf.JobConf;
public class Sleep {
  private static final String SLEEP_SECS = "sleep.secs";
  public static class MapperClass extends MapperBase {
    // データが入力されていないので、map 関数は実行されず、関連するロジックは
setup に書き込まれるだけです。
    @Override
    public void setup(TaskContext context) throws IOException {
      try {
        // jobconf に設定されたスリープの秒数を取得
        Thread.sleep(context.getJobConf().getInt(SLEEP_SECS, 1) * 1000
);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
  public static void main(String[] args) throws Exception {
    if (args.length ! = 1) {
      System.err.println("Usage: Sleep <sleep_secs>");
      System.exit(-1);
    }
```

```
    JobConf job = new JobConf();
    job.setMapperClass(MapperClass.class);
    // このインスタンスは、maponly のため、reductor 数に 0 を設定する必要があり
ます。
    job.setNumReduceTasks(0);
    // 入力テーブルが存在しないため、mapper 数を明示的に指定する必要があります。
    job.setNumMapTasks(1);
    job.set(SLEEP_SECS, args[0]);
    JobClient.runJob(job);
  }
}
```

6.4.11 Unique サンプル

前提条件

1. テストプログラムの JAR パッケージを準備します。 パッケージの名前を "mapreduce-examples.jar" 、ローカルストレージパスを "*data\resources*" とします。
2. Unique 操作をテストするためのテーブルとリソースを準備します。

   ・ テーブルを作成します。

   ```
   create table ss_in(key bigint, value bigint);
   create table ss_out(key bigint, value bigint);
   ```

   ・ リソースを追加します。

   ```
   add jar data\resources\mapreduce-examples.jar -f;
   ```

3. tunnel コマンドを使用してデータをインポートします。

   ```
   tunnel upload data ss_in;
   ```

   データファイルの内容がテーブル " ss_in " にインポートされます。

   ```
   1,1
   1,1
   2,2
   2,2
   ```

手順

odpscmd で Unique を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.Unique ss_in ss_out key;
```

予想される出力

出力テーブル "ss_out" の内容は次のとおりです。

```
+------------+------------+
| key | value |
+------------+------------+
```

```
| 1 | 1 |
| 2 | 2 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    Import java. util. iterator;
    Import com. aliyun. ODPS. Data. record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * Unique で重複ワードを削除
     *
     **/
    public class Unique {
      public static class OutputSchemaMapper extends MapperBase {
        private Record key;
        private Record value;
        @Override
        public void setup(TaskContext context) throws IOException{
          key = context.createMapOutputKeyRecord();
          value = context.createMapOutputValueRecord();
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws ioexception {
          long left = 0;
          long right = 0;
          if (record.getColumnCount() > 0) {
            left = (Long) record.get(0);
            if (record.getColumnCount() > 1) {
              right = (Long) record.get(1);
            }
            key.set(new Object[] { (Long) left, (Long) right });
            value.set(new Object[] { (Long) left, (Long) right });
            context.write(key, value);
          }
        }
      }
      public static class OutputSchemaReducer extends ReducerBase {
        private Record result = null;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          result.set(0, key.get(0));
          while(values.hasNext()) {
            Record value = values.next();
            result.set(1, value.get(1));
```

```
          }
          context.write(result);
        }
      }
      public static void main(String[] args) throws Exception {
        if (args.length > 3 || args.length < 2) {
          System.err.println("Usage: unique <in> <out> [key|value|all
]");
          System.exit(2);
        }
        String ops = "all";
        if (args.length == 3) {
          Ops = ARGs [2];
        }
        // Reduce 入力グルーピングは、スキャナーの設定に基づき決定されます。この
パラメーターが設定されていない場合、
        /Default は mapoutputkeyschema です
        // キー Unique
        if (ops.equals("key")) {
          JobConf job = new JobConf();
          job.setMapperClass(OutputSchemaMapper.class);
          job.setReducerClass(OutputSchemaReducer.class);
          job.setMapOutputKeySchema(SchemaUtils.fromString("key:bigint
,value:bigint"));
          job.setMapOutputValueSchema(SchemaUtils.fromString("key:
bigint,value:bigint"));
          job.setPartitionColumns(new String[] { "key" });
          job.setOutputKeySortColumns(new String[] { "key", "value
" });
          job.setOutputGroupingColumns(new String[] { "key" });
          job.set("tablename2", args[1]);
          job.setNumReduceTasks(1);
          job.setInt("table.counter", 0);
          InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
          OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
          Jobclient. runjob (job );
        }
        // キーと値 Unique
        if (ops.equals("all")) {
          JobConf job = new JobConf();
          job.setMapperClass(OutputSchemaMapper.class);
          job.setReducerClass(OutputSchemaReducer.class);
          job.setMapOutputKeySchema(SchemaUtils.fromString("key:bigint
,value:bigint"));
          job.setMapOutputValueSchema(SchemaUtils.fromString("key:
bigint,value:bigint"));
          job.setPartitionColumns(new String[] { "key" });
          job.setOutputKeySortColumns(new String[] { "key", "value
" });
          job.setOutputGroupingColumns(new String[] { "key", "value
" });
          Job. Set ("tablename2", argS [1]);
          job.setNumReduceTasks(1);
          job.setInt("table.counter", 0);
          InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
          OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
          Jobclient. runjob (job );
        }
        // 値 Unique
        if (ops.equals("value")) {
```

```
          JobConf job = new JobConf();
          job.setMapperClass(OutputSchemaMapper.class);
          job.setReducerClass(OutputSchemaReducer.class);
          job.setMapOutputKeySchema(SchemaUtils.fromString("key:bigint
,value:bigint"));
          job.setMapOutputValueSchema(SchemaUtils.fromString("key:
bigint,value:bigint"));
          job.setPartitionColumns(new String[] { "value" });
          job.setOutputKeySortColumns(new String[] { "value" });
          job.setOutputGroupingColumns(new String[] { "value" });
          job.set("tablename2", args[1]);-
          job.setNumReduceTasks(1);
          job.setInt("table.counter", 0);
          InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), job);
          OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
          Jobclient. runjob (job );
        }
      }
    }
```

6.4.12 ソートサンプル

前提条件

1. テストプログラムの **Jar** パッケージを準備します。 パッケージの名前を **"mapreduce-examples.jar"**、ローカルストレージパスを "*data\resources*" とします。
2. **SORT** をテストするためのテーブルとリソースを準備します。
   - テーブルを作成します。

     ```
     create table ss_in(key bigint, value bigint);
     create table ss_out(key bigint, value bigint);
     ```

   - リソースを追加します。

     ```
     add jar data\resources\mapreduce-examples.jar -f;
     ```

3. **tunnel** コマンドを使用してデータをインポートします。

   ```
   tunnel upload data ss_in;
   ```

   テーブル **"ss_in"** のデータファイルの内容は次のとおりです。

   ```
   2,1
   1,1
   3,1
   ```

手順

次のように、**odpscmd** で **Sort** を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
```

```
com.aliyun.odps.mapred.open.example.Sort ss_in ss_out;
```

予想される出力

出力テーブル **"ss_out"** の内容は以下のとおりです。

```
+------------+------------+
| key | value |
+------------+------------+
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    import java.util.Date;
    import com.aliyun.odps.data.Record;
    import com.aliyun.odps.data.TableInfo;
    import com.aliyun.odps.mapred.JobClient;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.TaskContext;
    import com.aliyun.odps.mapred.conf.JobConf;
    import com.aliyun.odps.mapred.example.lib.IdentityReducer;
    import com.aliyun.odps.mapred.utils.InputUtils;
    import com.aliyun.odps.mapred.utils.OutputUtils;
    import com.aliyun.odps.mapred.utils.SchemaUtils;
    /**
     * This is the trivial map/reduce program that does absolutely
nothing other
     * than use the framework to fragment and sort the input values.
     *
     **/
    public class Sort {
      static int printUsage() {
        System.out.println("sort <input> <output>");
        return -1;
      }
      /**
       * Implements the identity function, mapping record's first two
columns to
       * outputs.
       **/
      public static class IdentityMapper extends MapperBase {
        private Record key;
        private Record value;
        @Override
        public void setup(TaskContext context) throws IOException{
          key = context.createMapOutputKeyRecord();
          value = context.createMapOutputValueRecord();
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws IOException {
          Key.set (new object [] {(long) record.get (0 )});
          value.set(new Object[] { (Long) record.get(1) });
          context.write(key, value);
        }
```

```
      }
      /**
       * The main driver for sort program. Invoke this method to
submit the
       * map/reduce job.
       *
       * @throws IOException
       * When there is communication problems with the job tracker.
       **/
      public static void main(String[] args) throws Exception {
        JobConf jobConf = new JobConf();
        jobConf.setMapperClass(IdentityMapper.class);
        jobConf.setReducerClass(IdentityReducer.class);
        // グローバルオーダーの場合、reducers の数は 1 に設定されています。すべ
てのデータは reducer 上で連結されます。
        // 少量のデータの場合にのみ使用可能です。terasort など、他の方法で考える
必要があります。
        jobConf.setNumReduceTasks(1);
        Jobconf.setmapoutputkeyschema schemautils schemeiutils.
fromstring ("key: bigint "));
        jobConf.setMapOutputValueSchema(SchemaUtils.fromString("value:
bigint"));
        InputUtils.addTable(TableInfo.builder().tableName(args[0]).
build(), jobConf);
        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), jobConf);
        Date starttime = new date ();
        System.out.println("Job started: " + startTime);
        JobClient.runJob(jobConf);
        Date end_time = new Date();
        System.out.println("Job ended: " + end_time);
        System.out.println("The job took "
            + (end_time.getTime() - startTime.getTime()) / 1000 + "
seconds.") ;
      }
    }
```

6.4.13 パーティションサンプル

次の例では、パーティションを入力と出力として使用します。

例 **1:**

```
     public static void main(String[] args) throws Exception {
     JobConf job = new JobConf();

     LinkedHashMap<String, String> input = new LinkedHashMap<String,
String>();
     input.put("pt", "123456");
     InputUtils.addTable(TableInfo.builder().tableName("input_table").
partSpec(input).build(), job);
     LinkedHashMap<String, String> output = new LinkedHashMap<String,
String>();
     output.put("ds", "654321");
     Outpututils. addtable (tableinfo. builder (). tablename ("
output_table "). partspec (output ). build (), job );
     JobClient.runJob(job);
```

例 2:

```
    package com.aliyun.odps.mapred.open.example;

      public static void main(String[] args) throws Exception {
        if (args.length ! = 2) {
          System.err.println("Usage: WordCount <in_table> <out_table
>");
          System.exit(2);

        JobConf job = new JobConf();
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(SumCombiner.class);
        job.setReducerClass(SumReducer.class);
        job.setMapOutputKeySchema(SchemaUtils.fromString("word:string
"));
        job.setMapOutputValueSchema(SchemaUtils.fromString("count:
bigint"));
        Account account = new AliyunAccount("my_access_id", "
my_access_key");
        Odps odps = new Odps(account);
        odps.setEndpoint("odps_endpoint_url");
        odps.setDefaultProject("my_project");
        Table table = odps.tables().get(tblname);
        TableInfoBuilder builder = TableInfo.builder().tableName(
tblname);
        for (Partition p : table.getPartitions()) {
          if (applicable(p)) {
            LinkedHashMap<String, String> partSpec = new LinkedHashMap
<String, String>();
            for (String key : p.getPartitionSpec().keys()) {
              partSpec.put(key, p.getPartitionSpec().get(key));

            InputUtils.addTable(builder.partSpec(partSpec).build(),
conf);


        OutputUtils.addTable(TableInfo.builder().tableName(args[1]).
build(), job);
        Jobclient. runjob (job );
```

注:

- 上記の例では、**MaxCompute SDK** と **MapReduce SDK** を組み合わせて **MapReduce** タスクを実現しています。
- コードはコンパイルできません。**main** 関数の例として示しています。
- **Applicable** 関数は、パーティションを **MapReduce** ジョブの入力として使用できるかどうかを判定するユーザーロジックです。

6.4.14 Pipeline サンプル

前提条件

1. テストプログラムの Jar パッケージを準備します。 パッケージの名前を "mapreduce-examples.jar" 、ローカルストレージパスを "*data\resources*" とします。
2. WordCountPipeline をテストするためのテーブルとリソースを準備します。

   ・ テーブルを作成します。

   ```
   create table wc_in (key string, value string);
   create table wc_out(key string, cnt bigint);
   ```

   ・ リソースを追加します。

   ```
   add jar data\resources\mapreduce-examples.jar -f;
   ```

3. tunnel コマンドを使用してデータをインポートします。

   ```
   tunnel upload data wc_in;
   ```

   テーブル "wc_in" にインポートされたデータは次のとおりです。

   ```
   hello,odps
   ```

手順

odpscmd で WordCountPipeline を実行します。

```
jar -resources mapreduce-examples.jar -classpath data\resources\
mapreduce-examples.jar
com.aliyun.odps.mapred.open.example.WordCountPipeline wc_in wc_out;
```

予想される出力

出力テーブル "wc_out" の内容は次のとおりです。

```
+------------+------------+
| key | cnt |
+------------+------------+
| hello | 1 |
| odps | 1 |
+------------+------------+
```

サンプルコード

```
    package com.aliyun.odps.mapred.open.example;
    import java.io.IOException;
    Import java. util. iterator;
    import com.aliyun.odps.Column;
    import com.aliyun.odps.OdpsException;
    import com.aliyun.odps.OdpsType;
    Import com. aliyun. ODPS. Data. record;
    import com.aliyun.odps.data.TableInfo;
```

```
    import com.aliyun.odps.mapred.Job;
    import com.aliyun.odps.mapred.MapperBase;
    import com.aliyun.odps.mapred.ReducerBase;
    import com.aliyun.odps.pipeline.Pipeline;
    public class WordCountPipelineTest {
      public static class TokenizerMapper extends MapperBase {
        Record word;
        Record one;
        @Override
        public void setup(TaskContext context) throws IOException{
          word = context.createMapOutputKeyRecord();
          one = context.createMapOutputValueRecord();
          one.setBigint(0, 1L);
        }
        @Override
        public void map(long recordNum, Record record, TaskContext
context)
            Throws ioexception {
          for (int i = 0; i < record.getColumnCount(); i++) {
            String[] words = record.get(i).toString().split("\\s+");
            for (String w : words) {
              word.setString(0, w);
              context.write(word, one);
            }
          }
        }
      }
      public static class SumReducer extends ReducerBase {
        private Record value;
        @Override
        public void setup(TaskContext context) throws IOException{
          value = context.createOutputValueRecord();
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          Long Count = 0;
          while(values.hasNext()) {
            Record val = values.next();
            count += (Long) val.get(0);
          }
          value.set(0, count);
          context.write(key, value);
        }
      }
      public static class IdentityReducer extends ReducerBase {
        private Record result;
        @Override
        public void setup(TaskContext context) throws IOException{
          result = context.createOutputRecord();
        }
        @Override
        public void reduce(Record key,Iterator<Record>values,
TaskContext context)
            Throws ioexception {
          while (values.hasNext()) {
            result.set(0, key.get(0));
            result.set(1, values.next().get(0));
            context.write(result);
          }
        }
      }
      public static void main(String[] args) throws OdpsException {
```

```
        if (args.length ! = 2) {
          System.err.println("Usage: WordCountPipeline <in_table> <
out_table>");
          System.exit(2);
        }
        Job job = new Job();
        /***
         * パイプラインの作成プロセスで、mapper の OutputKeySortColumns,
PartitionColumns, OutputGroupingColumns を指定していない場合、
         * この 3 つのデフォルト設定として OutputKey をデフォルトとします。
         ***/
        Pipeline pipeline = Pipeline.builder()
            . Addmapper (maid. Class)
            .setOutputKeySchema(
                    new Column[] { new Column("word", OdpsType.STRING
) })
            .setOutputValueSchema(
                    new Column[] { new Column("count", OdpsType.BIGINT
) })
            .setOutputKeySortColumns(new String[] { "word" })
            .setPartitionColumns(new String[] { "word" })
            .setOutputGroupingColumns(new String[] { "word" })
            .addReducer(SumReducer.class)
            .setOutputKeySchema(
                    new Column[] { new Column("word", OdpsType.STRING
) })
            .setOutputValueSchema(
                    new Column[] { new Column("count", OdpsType.BIGINT
)})
            .addReducer(IdentityReducer.class).createPipeline();
        // assemblyer を設定する必要がない場合、パイプラインを jobconf と
jobconf に設定します
        job.setPipeline(pipeline);
        // Input Output のテーブル情報の設定
        job.addInput(TableInfo.builder().tableName(args[0]).build());
        job.addOutput(TableInfo.builder().tableName(args[1]).build());
        // ジョブを送信し、終了するまで待機
        job.submit();
        job.waitForCompletion();
        System.exit(job.isSuccessful() == true ? 0 : 1);
      }
    }
```

6.5 Java SDK

6.5.1 Java SDK

ここでは、一般的な **MapReduce** インターフェイスを説明します。

Maven を使用している場合は、必要な **Java SDK** (さまざまなバージョンで利用できます) を入手するために *Maven* ライブラリから **"odps-sdk-mapred"** を検索できます。 設定は次のとおりです。

```
<dependency>
    <groupId>com.aliyun.odps</groupId>
    <artifactId>odps-sdk-mapred</artifactId>
    <version>0.20.7-public</version>
```

```
</dependency>
```

インターフェイス	説明
MapperBase	このクラスから継承するには、ユーザー定義の **Map** 関数が必要です。 入力テーブルのレコードオブジェクトを処理し、キー値を処理してオブジェクトにして、値を **Reduce** ステージへ出力します。または結果レコードを **Reduce** ステージを通さずに結果テーブルへ出力します。 **Reduce** ステージを通さず、直接計算結果を出力するジョブを **Map-Only** ジョブと呼びます。
ReducerBase	カスタマイズされた **Reduce** 関数はこのクラスを継承する必要があります。 キーと関連がある値のセットは減少します。
TaskContext	**MapperBase** と **ReducerBase** の複数のメンバ関数の入力パラメータの **1** つです。 タスクについてのコンテキスト情報が含まれます。
JobClient	ジョブを送信し、管理するために使用します。 送信モードはブロッキング (同期) モード、またはノンブロッキング (非同期) モードを含みます。
RunningJob	ジョブ実行中のオブジェクトを表示し、ジョブ実行プロセス中に **MapReduce** ジョブインスタンスをトレースするために利用されます。
JobConf	**MapReduce** タスクの設定を説明します。 **JobConf** オブジェクトは 通常メインプログラム (**main** 関数) で定義され、ジョブは **JobClient** によって **MaxCompute** へ送信されます。

MapperBase

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
void cleanup(TaskContext context)	**Map** メソッドは **map** ステージが終了した後に呼び出されます。
void map(long key, Record record, TaskContext context)	**Map** メソッドは入力テーブルのレコードを処理します。
void setup(TaskContext context)	**Map** メソッドは **map** ステージが開始される前に呼び出されます。

ReducerBase

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
void cleanup(TaskContext context)	**Reduce** メソッドは **reduce** ステージが終了した後に呼び出されます。

インターフェイス	説明
void reduce(Record key, Iterator< Record > values, TaskContext context)	**Reduce** メソッドは入力テーブルのレコードを処理します。
void setup(TaskContext context)	**Reduce** メソッドは **reduce**ステージが開始される前に呼び出されます。

TaskContext

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
TableInfo[] getOutputTableInfo()	出力テーブルの情報を取得します。
Record createOutputRecord()	デフォルトの出力テーブルのレコードオブジェクトを作成します。
Record createOutputRecord(String label)	指定されたラベルを持つ出力テーブルのレコードオブジェクトを作成します。
Record createMapOutputKeyRecord()	**Map** によって出力されたレコードオブジェクトのキーを作成します。
Record createMapOutputValueRecord()	**Map** によって出力されたレコードオブジェクトの値を作成します。
void write(Record record)	レコードをデフォルト出力に書き込みます。**Reduce** クライアントによって出力データの書き込みに使用されます。**Reduce** クライアント上で複数回呼び出すことができます。
void write(Record record, String label)	レコードを指定されたラベルの出力に書き込みます。**Reduce** クライアントによって出力データの書き込みに使用されます。**Reduce** クライアント上で複数回呼び出すことができます。
void write(Record key, Record value)	**Map** は中間結果のレコードを書き込みます。**Map** 関数の中で呼び出すことができ、 **Map** クライアント上で複数回呼び出すことができます。
BufferedInputStream readResour ceFileAsStream(String resourceName)	ファイルタイプリソースを読み取ります。
Iterator<Record > readResourceTable(String resourceName)	テーブルタイプリソースを読み取ります。
Counter getCounter(Enum<? > > name)	指定された名前の **Counter** オブジェクトを取得します。

インターフェイス	説明
Counter getCounter(String group, String name)	指定された名前とグループ名の **Counter** オブジェクトを取得します。
void progress()	ハートビートの情報を **MapReduce** フレームワークへ通知します。 ユーザーのメソッドの処理に長時間かかる場合や、プロセス内で呼び出されるフレームワークがない場合に、タスクのタイムアウトを防ぐためにこのメソッドを呼び出します。 フレームワークのタイムアウトはデフォルトで **600** 秒に設定されています。

:

- **MaxCompute TaskContext** インターフェイスは **progress** 関数を提供しますが、この関数は **Worker** が長時間実行されたときに終了されるのを防ぎます。フレームワークはこの **Worker** をタイムアウトした **Worker** とみなします。このインターフェイスはハートビート情報をフレームワークへ送信するのと似ていますが、**Worker** の進捗は通知しません。
- **MaxCompute MapReduce Worker** のデフォルトのタイムアウトスケジュールは **10** 分間です (システムデフォルト、ユーザーは制御できません)。 スケジュールが **10** 分を超え、 **Worker** がハートビート情報をフレームワークへ送信できない場合 (**progress** インターフェイスを呼び出さない)、フレームワークはこの **Worker** を中断しなければならず、**MapReduce** タスクは失敗して終了します。**worker** がフレームワークによって終了されることを防ぐために、**Mapper/Reducer** 関数内で定期的に **progress** インターフェイスを呼び出すことを推奨します。

JobConf

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
void setResources(String resourceNames)	このジョブで使用されているリソースを宣言します。 **Mapper/Reducer** の処理実行中は、宣言されたリソースのみが **TaskContext** オブジェクトによって読み取られます。
void setMapOutputKeySchema(Column[] schema)	**Mapper** から **Reducer** へ出力されたキー属性を設定します。
void setMapOutputValueSchema(Column[] schema)	**Mapper** から **Reducer** へ出力された属性値を設定します。

インターフェイス	説明
void setOutputKeySortColumns(String[] cols)	**Mapper** から **Reducer** へ出力された、ソートするキー列を設定します。
void setOutputGroupingColumns(String[] cols)	グループ化されたキー列を設定します。
void setMapperClass(Class<? extends Mapper > theClass)	ジョブの **Mapper** 関数を設定します。
void setPartitionColumns(String[] cols)	ジョブで指定されたパーティション列を設定します。デフォルトは **Mapper** によって出力されたキーのすべての列です。
void setReducerClass(Class<? extends Reducer theClass)	ジョブの **Reducer** を設定します。
void setCombinerClass(Class<? extends Reducer theClass)	**Map** クライアント上で実行される、ジョブのコンバイナを設定します。 この関数は単一の **Map** による同一のローカルキーバリューに対する **Reduce** 操作と似ています。
void setSplitSize(long size)	入力スライスのサイズを設定します。 単位: **MB** デフォルト値は **640** です。
void setNumReduceTasks(int n)	**Reduce** タスクの数を設定します。 デフォルトは **Mapper** タスクの **1/4** です。
void setMemoryForMapTask(int mem)	**Mapper** タスク内の単一 **Worker** のメモリサイズを設定します。 単位: **MB** デフォルト値は **2048** です。
void setMemoryForReduceTask(int mem)	**Reduce** タスクの単一 **Worker** のメモリサイズを設定します。 単位: **MB** デフォルト値は **2048** です。

注:

- 通常、**KeySortColumns** と **PartitionColumns** はキーに含まれますが、**GroupingColumns** は **KeySortColumns** に含まれます。
- **Map** 側では、マッパーの出力レコードは **PartitionColumns** を使用して計算されたハッシュ値に従って **reducer** へ配信され、**KeySortColumns** でソートされます。
- **Reduce** 側では、 **KeySortColumns** によってソートされた後、入力レコードは **reduce** 関数の入力グループとして順次グループ化されます。 つまり、同じ **GroupingColumns** 値のレコードは同じ入力グループとして扱われます。

JobClient

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
static RunningJob runJob(JobConf job)	同期 (ブロッキング) モードで **MapReduce** ジョブを送信した直後に返ります。
static RunningJob submitJob(JobConf job)	非同期 (ノンブロッキング) モードで **MapReduce** ジョブを送信した直後に返ります。

RunningJob

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
String getInstanceID()	実行ログとジョブ管理を確認するためのインスタンス **ID** を取得します。
boolean isComplete()	ジョブが完了したかどうかを確認します。
boolean isSuccessful()	ジョブインスタンスが成功したかどうかを確認します。
void waitForCompletion()	ジョブインスタンスが完了するまで待機します。 通常は非同期モードで送信されたジョブに対して使用されます。
JobStatus getJobStatus()	ジョブインスタンスのステータスを確認します。
void killJob()	ジョブを終了します。
Counters getCounters()	**Counter** の情報を取得します。

InputUtils

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
static void addTable(TableInfo table, JobConf conf)	タスク入力にテーブルを追加します。 複数回呼び出すことができます。 新しく追加されたテーブルは、追加モードで入力キューに追加されます。
static void setTables(TableInfo [] tables, JobConf conf)	タスク入力にテーブルを追加します。

OutputUtils

主な関数インターフェイスは次のとおりです。

インターフェイス	説明
static void addTable(TableInfo table, JobConf conf)	タスク出力にテーブルを追加します。 複数回呼び出すことができます。 同様に、新しく追加されたテーブルを、追加モードで出力キューに追加します。
static void setTables(TableInfo [] tables, JobConf conf)	タスク出力に複数のテーブルを追加します。

パイプライン

パイプラインは *MR2* のサブジェクトです。**Pipeline.builder** によって構築できます。 パイプラインは次のとおりです。

```
    public Builder addMapper(Class<? extends Mapper> mapper)
    public Builder addMapper(Class<? extends Mapper> mapper,
           column [] keyschema, column [] valueschema, string []
sortcols,
           SortOrder [] order, string [] partcols,
            Class<? extends Partitioner> theClass, String[] groupCols)
    public Builder addReducer(Class<? extends Reducer> reducer)
    public Builder addReducer(Class<? extends Reducer> reducer,
           column [] keyschema, column [] valueschema, string []
sortcols,
           SortOrder [] order, string [] partcols,
            Class<? extends Partitioner> theClass, String[] groupCols)
    public setoutputkeyschema builder (Column [] keyschema)
    public setoutputvalueschema builder (Column [] valueschema)
    public setoutputkeysortcolumns builder (String [] sortcols)
    public setoutputkeysortorder builder (Sortorder [] order)
    public setpartitioncolumns builder (String [] partcols)
    public Builder setPartitionerClass(Class<? extends Partitioner>
theClass)
    void setOutputGroupingColumns(String[] cols)
```

例:

```
    job job = new job ();
    pipeline pipeline = pipeline. builder ()
     . addmapper (Tokenizermapper. class)
     . setoutputkeyschema (
         new column [] {new column ("word", OdpsType. string)})
     . setoutputvalueschema (
         new column [] {new column ("count", OdpsType. bigint)})
     . addreducer (Sumreducer. class)
     . setoutputkeyschema (
         new column [] {new column ("count", OdpsType. bigint)})
     . setoutputvalueschema (
         new column [] {new column ("word", OdpsType. string),
         new column ("count", OdpsType. bigint)})
     . addreducer (Identityreducer. class). createPipeline ();
    job. setpipeline (pipeline);
```

```
    job. addinput (...)
    job. addoutput (...)
    job. submit ();
```

前述の例で示されるように、ユーザーは **main** クラスで **Map** を構築し、 続けて **2** つの **Reduce** の **MapReduce** タスクを取得できます。**MapReduce** の基本的な機能に慣れている場合は、機能が似ているので、**MR2** も使用できます。

 注：

- 具体的には、ユーザーが **JobConf** によって **MapReduce** タスクの設定を完了することを推奨します。
- **JobConf** は **Map** を設定した後にのみ、単一の **Reduce** の **MapReduce** タスクを取得することができます。

データ型

MapReduce でサポートされているデータ型に含まれるのは、**BIGINT**、**STRING**、**DOUBLE**、**BOOLEAN**、**DATETIME** です。 **MaxCompute** データ型と **Java** データ型の間の **MaxCompute** は次のとおりです。

MaxCompute SQL データ型	Java データ型
Bigint	Long
String	String
Double	Double
Boolean	Boolean
Datetime	Date
Decimal	BigDecimal

6.5.2 SDK の互換バージョンの概要

以下の表に、**Hadoop MapReduce** と互換性がある **MapReduce** の **MaxCompute** 互換バージョンのリストを示します。

タイプ	インターフェイス	互換性の有無
Mapper	void map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)	あり

タイプ	インターフェイス	互換性の有無
Mapper	**void run(org.apache.hadoop.mapreduce.Mapper.Context context)**	あり
Mapper	**void setup(org.apache.hadoop.mapreduce.Mapper.Context context)**	あり
Reducer	**Void cleanup (Org. Apache. hadoop. mapreduce. reducer. Context Context)**	あり
Reducer	**void reduce(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Reducer.Context context)**	あり
Reducer	**void run(org.apache.hadoop.mapreduce.Reducer.Context context)**	あり
Reducer	**void setup(org.apache.hadoop.mapreduce.Reducer.Context context)**	あり
Partitioner	**int getPartition(KEY key, VALUE value, int numPartitions)**	あり
Mapcontext (inheritance)	**InputSplit getInputSplit()**	なし。例外をスローします。
ReduceContext	**nextKey()**	あり
ReduceContext	**getValues()**	あり
TaskInputOutputContext	**getCurrentKey()**	あり
TaskInputOutputContext	**getCurrentValue()**	あり
TaskInputOutputContext	**getOutputCommitter()**	なし。例外をスローします。
TaskInputOutputContext	**nextKeyValue()**	あり
TaskInputOutputContext	**write(KEYOUT key, VALUEOUT value)**	あり
TaskAttemptContext	**getCounter(Enum < > counterName)**	あり
TaskAttemptContext	**getCounter(String groupName, String counterName)**	あり
TaskAttemptContext	**setStatus(String msg)**	空実装
TaskAttemptContext	**getStatus()**	空実装

タイプ	インターフェイス	互換性の有無
TaskAttemptContext	**getTaskAttemptID()**	なし。例外をスローします。
TaskAttemptContext	**getProgress()**	なし。例外をスローします。
TaskAttemptContext	**progress()**	あり
Job	**addArchiveToClassPath(Path archive)**	なし
Job	**addCacheArchive(URI uri)**	なし
Job	**addCacheFile(URI uri)**	なし
Job	**addFileToClassPath(Path file)**	なし
Job	**cleanupProgress()**	なし
Job	**createSymlink()**	なし。例外をスローします。
Job	**failTask(TaskAttemptID taskId)**	なし
Job	**getCompletionPollInterval(Configuration conf)**	空実装
Job	**getCounters()**	あり
Job	**getFinishTime()**	あり
Job	**getHistoryUrl()**	あり
Job	**getInstance()**	あり
Job	**getInstance(Cluster ignored)**	あり
Job	**getInstance(Cluster ignored, Configuration conf)**	あり
Job	**getInstance(Configuration conf)**	あり
Job	**getInstance(Configuration conf, String jobName)**	空実装
Job	**getInstance(JobStatus status, Configuration conf)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
Job	**getJobFile()**	なし。例外をスローします。
Job	**getJobName()**	空実装
Job	**getJobState()**	なし。例外をスローします。
Job	**getPriority()**	なし。例外をスローします。
Job	**getProgressPollInterval(Configuration conf)**	空実装
Job	**getReservationId()**	なし。例外をスローします。
Job	**getSchedulingInfo()**	なし。例外をスローします。
Job	**getStartTime()**	あり
Job	**getStatus()**	なし。例外をスローします。
Job	**getTaskCompletionEvents(int startFrom)**	なし。例外をスローします。
Job	**getTaskCompletionEvents(int startFrom, int numEvents)**	なし。例外をスローします。
Job	**getTaskDiagnostics(TaskAttemptID taskid)**	なし。例外をスローします。
Job	**getTaskOutputFilter(Configuration conf)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
Job	**getTaskReports(TaskType type)**	なし。例外をスローします。
Job	**getTrackingURL()**	あり
Job	**isComplete()**	あり
Job	**isRetired()**	なし。例外をスローします。
Job	**isSuccessful()**	あり
Job	**isUber()**	空実装
Job	**killJob()**	あり
Job	**killTask(TaskAttemptID taskId)**	なし
Job	**mapProgress()**	あり
Job	**monitorAndPrintJob()**	あり
Job	**reduceProgress()**	あり
Job	**setCacheArchives(URI[] archives)**	なし。例外をスローします。
Job	**setCacheFiles(URI[] files)**	なし。例外をスローします。
Job	**setCancelDelegationTokenUponJobCompletion(boolean value)**	なし。例外をスローします。
Job	**setCombinerClass(Class<? extends Reducer> cls)**	あり
Job	**setCombinerKeyGroupingComparatorClass(Class<? extends RawComparator> cls)**	あり
Job	**setGroupingComparatorClass(Class<? extends RawComparator> cls)**	あり
Job	**setInputFormatClass(Class<? extends InputFormat> cls)**	空実装
Job	**setJar(String jar)**	あり

タイプ	インターフェイス	互換性の有無
Job	**setJarByClass(Class<? > cls)**	あり
Job	**setJobName(String name)**	空実装
Job	**setJobSetupCleanupNeeded(boolean needed)**	空実装
Job	**setMapOutputKeyClass(Class<? > theClass)**	あり
Job	**setMapOutputValueClass(Class<? > theClass)**	あり
Job	**setMapperClass(Class<? extends Mapper> cls)**	あり
Job	**setMapSpeculativeExecution(boolean speculativeExecution)**	空実装
Job	**setMaxMapAttempts(int n)**	空実装
Job	**setMaxReduceAttempts(int n)**	空実装
Job	**setNumReduceTasks(int tasks)**	あり
Job	**setOutputFormatClass(Class<? extends OutputFormat> cls)**	なし。例外をスローします。
Job	**setOutputKeyClass(Class<? > theClass)**	あり
Job	**setOutputValueClass(Class<? > theClass)**	あり
Job	**setPartitionerClass(Class<? extends Partitioner> cls)**	あり
Job	**setPriority(JobPriority priority)**	なし。例外をスローします。
Job	**setProfileEnabled(boolean newValue)**	空実装
Job	**setProfileParams(String value)**	空実装
Job	**setProfileTaskRange(boolean isMap, String newValue)**	空実装
Job	**setReducerClass(Class<? extends Reducer> cls)**	あり

タイプ	インターフェイス	互換性の有無
Job	**setReduceSpeculativeExecution(boolean speculativeExecution)**	空実装
Job	**setReservationId(ReservationId reservationId)**	なし。例外をスローします。
Job	**setSortComparatorClass(Class<? extends RawComparator> cls)**	なし。例外をスローします。
Job	**setSpeculativeExecution(boolean speculativeExecution)**	あり
Job	**setTaskOutputFilter(Configuration conf, org.apache.hadoop.mapreduce.Job.TaskStatusFilter newValue)**	なし。例外をスローします。
Job	**setupProgress()**	なし。例外をスローします。
Job	**setUser(String user)**	空実装
Job	**setWorkingDirectory(Path dir)**	空実装
Job	**submit()**	あり
Job	**toString()**	なし。例外をスローします。
Job	**waitForCompletion(boolean verbose)**	あり
Task Execution & Environment	**mapreduce.map.java.opts**	空実装
Task Execution & Environment	**mapreduce.reduce.java.opts**	空実装
Task Execution & Environment	**mapreduce.map.memory.mb**	空実装
Task Execution & Environment	**mapreduce.reduce.memory.mb**	空実装
Task Execution & Environment	**mapreduce.task.io.sort.mb**	空実装
Task Execution & Environment	**mapreduce.map.sort.spill.percent**	空実装
Task Execution & Environment	**mapreduce.task.io.soft.factor**	空実装
Task Execution & Environment	**mapreduce.reduce.merge.inmem.thresholds**	空実装

タイプ	インターフェイス	互換性の有無
Task Execution & Environment	**mapreduce.reduce.shuffle.merge.percent**	空実装
Task Execution & Environment	**mapreduce.reduce.shuffle.input.buffer.percent**	空実装
Task Execution & Environment	**mapreduce.reduce.input.buffer.percent**	空実装
Task Execution & Environment	**mapreduce.job.id**	空実装
Task Execution & Environment	**mapreduce.job.jar**	空実装
Task Execution & Environment	**mapreduce.job.local.dir**	空実装
Task Execution & Environment	**mapreduce.task.id**	空実装
Task Execution & Environment	**mapreduce.task.attempt.id**	空実装
Task Execution & Environment	**mapreduce.task.is.map**	空実装
Task Execution & Environment	**mapreduce.task.partition**	空実装
Task Execution & Environment	**mapreduce.map.input.file**	空実装
Task Execution & Environment	**mapreduce.map.input.start**	空実装
Task Execution & Environment	**mapreduce.map.input.length**	空実装
Task Execution & Environment	**mapreduce.task.output.dir**	空実装
JobClient	**cancelDelegationToken(Token < DelegationTokenIdentifier> token)**	なし。例外をスローします。
JobClient	**close()**	空実装
JobClient	**displayTasks(JobID jobId, String type , String state)**	なし。例外をスローします。
JobClient	**getAllJobs()**	なし。例外をスローします。
JobClient	**getCleanupTaskReports(JobID jobId)**	なし。例外をスローします。
JobClient	**getClusterStatus()**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
JobClient	**getClusterStatus(boolean detailed)**	なし。例外をスローします。
JobClient	**getDefaultMaps()**	なし。例外をスローします。
JobClient	**getDefaultReduces()**	なし。例外をスローします。
JobClient	**getDelegationToken(Text renewer)**	なし。例外をスローします。
JobClient	**getFs()**	なし。例外をスローします。
JobClient	**getJob(JobID jobid)**	なし。例外をスローします。
JobClient	**getJob(String jobid)**	なし。例外をスローします。
JobClient	**getJobsFromQueue(String queueName)**	なし。例外をスローします。
JobClient	**getMapTaskReports(JobID jobId)**	なし。例外をスローします。
JobClient	**getMapTaskReports(String jobId)**	なし。例外をスローします。
JobClient	**getQueueAclsForCurrentUser()**	なし。例外をスローします。
JobClient	**getQueueInfo(String queueName)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
JobClient	**getQueues()**	なし。例外をスローします。
JobClient	**getReduceTaskReports(JobID jobId)**	なし。例外をスローします。
JobClient	**getReduceTaskReports(String jobId)**	なし。例外をスローします。
JobClient	**getSetupTaskReports(JobID jobId)**	なし。例外をスローします。
JobClient	**getStagingAreaDir()**	なし。例外をスローします。
JobClient	**getSystemDir()**	なし。例外をスローします。
JobClient	**getTaskOutputFilter()**	なし。例外をスローします。
JobClient	**getTaskOutputFilter(JobConf job)**	なし。例外をスローします。
JobClient	**init(JobConf conf)**	なし。例外をスローします。
JobClient	**isJobDirValid(Path jobDirPath, FileSystem fs)**	なし。例外をスローします。
JobClient	**jobsToComplete()**	なし。例外をスローします。
JobClient	**monitorAndPrintJob(JobConf conf, RunningJob job)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
JobClient	**renewDelegationToken(Token< DelegationTokenIdentifier> token)**	なし。例外をスローします。
JobClient	**run(String[] argv)**	なし。例外をスローします。
JobClient	**runJob(JobConf job)**	あり
JobClient	**setTaskOutputFilter(JobClient. TaskStatusFilter newValue)**	なし。例外をスローします。
JobClient	**setTaskOutputFilter(JobConf job, JobClient.TaskStatusFilter newValue)**	なし。例外をスローします。
JobClient	**submitJob(JobConf job)**	あり
JobClient	**submitJob(String jobFile)**	なし。例外をスローします。
JobConf	**deleteLocalFiles()**	なし。例外をスローします。
Jobconf	**deleteLocalFiles(String subdir)**	なし。例外をスローします。
Jobconf	**normalizeMemoryConfigValue(long val)**	空実装
Jobconf	**setCombinerClass(Class<? extends Reducer> theClass)**	あり
Jobconf	**setCompressMapOutput(boolean compress)**	空実装
Jobconf	**setInputFormat(Class<? extends InputFormat> theClass)**	なし。例外をスローします。
JobConf	**setJar(String jar)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
JobConf	**setJarByClass(Class cls)**	なし。例外をスローします。
JobConf	**setJobEndNotificationURI(String uri)**	なし。例外をスローします。
JobConf	**setJobName(String name)**	空実装
JobConf	**setJobPriority(JobPriority prio)**	なし。例外をスローします。
JobConf	**setKeepFailedTaskFiles(boolean keep)**	なし。例外をスローします。
JobConf	**setKeepTaskFilesPattern(String pattern)**	なし。例外をスローします。
JobConf	**setKeyFieldComparatorOptions(String keySpec)**	なし。例外をスローします。
JobConf	**setKeyFieldPartitionerOptions(String keySpec)**	なし。例外をスローします。
JobConf	**setMapDebugScript(String mDbgScript)**	空実装
JobConf	**setMapOutputCompressorClass(Class<? extends CompressionCodec> codecClass)**	空実装
JobConf	**setMapOutputKeyClass(Class<? > theClass)**	あり
JobConf	**setMapOutputValueClass(Class<? > theClass)**	あり
JobConf	**setMapperClass(Class<? extends Mapper> theClass)**	あり
JobConf	**setMapRunnerClass(Class<? extends MapRunnable> theClass)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
JobConf	**setMapSpeculativeExecution(boolean speculativeExecution)**	空実装
JobConf	**setMaxMapAttempts(int n)**	空実装
JobConf	**setMaxMapTaskFailuresPercent(int percent)**	空実装
JobConf	**setMaxPhysicalMemoryForTask(long mem)**	空実装
JobConf	**setMaxReduceAttempts(int n)**	空実装
JobConf	**setMaxReduceTaskFailuresPercent(int percent)**	空実装
JobConf	**setMaxTaskFailuresPerTracker(int noFailures)**	空実装
JobConf	**setMaxVirtualMemoryForTask(long vmem)**	空実装
JobConf	**setMemoryForMapTask(long mem)**	あり
JobConf	**setMemoryForReduceTask(long mem)**	あり
JobConf	**setNumMapTasks(int n)**	あり
JobConf	**setNumReduceTasks(int n)**	あり
JobConf	**setNumTasksToExecutePerJvm(int numTasks)**	空実装
JobConf	**setOutputCommitter(Class<? extends OutputCommitter> theClass)**	なし。例外をスローします。
JobConf	**setOutputFormat(Class<? extends OutputFormat> theClass)**	空実装
JobConf	**setOutputKeyClass(Class<? > theClass)**	あり
JobConf	**setOutputKeyComparatorClass(Class<? extends RawComparator> theClass)**	なし。例外をスローします。
JobConf	**setOutputValueClass(Class<? > theClass)**	あり

タイプ	インターフェイス	互換性の有無
JobConf	**setOutputValueGroupingComparator(Class<? extends RawComparator> theClass)**	なし。例外をスローします。
JobConf	**setPartitionerClass(Class<? extends Partitioner> theClass)**	あり
JobConf	**setProfileEnabled(boolean newValue)**	空実装
JobConf	**setProfileParams(String value)**	空実装
JobConf	**setProfileTaskRange(boolean isMap, String newValue)**	空実装
JobConf	**setQueueName(String queueName)**	なし。例外をスローします。
JobConf	**setReduceDebugScript(String rDbgScript)**	空実装
JobConf	**setReducerClass(Class<? extends Reducer> theClass)**	あり
JobConf	**setReduceSpeculativeExecution(boolean speculativeExecution)**	空実装
JobConf	**setSessionId(String sessionId)**	空実装
JobConf	**setSpeculativeExecution(boolean speculativeExecution)**	なし。例外をスローします。
JobConf	**setUseNewMapper(boolean flag)**	あり
JobConf	**setUseNewReducer(boolean flag)**	あり
JobConf	**setUser(String user)**	空実装
JobConf	**setWorkingDirectory(Path dir)**	空実装
FileInputFormat	関連なし	なし。例外をスローします。
TextInputFormat	関連なし	あり
InputSplit	**mapred.min.split.size.**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
FileSplit	**map.input.file**	なし。例外をスローします。
RecordWriter	関連なし	なし。例外をスローします。
RecordReader	関連なし	なし。例外をスローします。
OutputFormat	関連なし	なし。例外をスローします。
OutputCommitter	**abortJob(JobContext jobContext, int status)**	なし。例外をスローします。
OutputCommitter	**abortJob(JobContext context, JobStatus.State runState)**	なし。例外をスローします。
OutputCommitter	**abortTask(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**abortTask(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**cleanupJob(JobContext jobContext)**	なし。例外をスローします。
OutputCommitter	**cleanupJob(JobContext context)**	なし。例外をスローします。
OutputCommitter	**commitJob(JobContext jobContext)**	なし。例外をスローします。
OutputCommitter	**commitJob(JobContext context)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
OutputCommitter	**commitTask(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**needsTaskCommit(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**needsTaskCommit(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**setupJob(JobContext jobContext)**	なし。例外をスローします。
OutputCommitter	**setupJob(JobContext jobContext)**	なし。例外をスローします。
OutputCommitter	**setupTask(TaskAttemptContext taskContext)**	なし。例外をスローします。
OutputCommitter	**setupTask(TaskAttemptContext taskContext)**	なし。例外をスローします。
Counter	**getDisplayName()**	あり
Counter	**getName()**	あり
Counter	**getValue()**	あり
Counter	**increment(long incr)**	あり
Counter	**setValue(long value)**	あり
Counter	**setDisplayName(String displayName)**	あり
DistributedCache	**CACHE_ARCHIVES**	なし。例外をスローします。
DistributedCache	**CACHE_ARCHIVES_SIZES**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
DistributedCache	**CACHE_ARCHIVES_TIMESTAMPS**	なし。例外をスローします。
Distributed cache	**CACHE_FILES**	なし。例外をスローします。
DistributedCache	**CACHE_FILES_SIZES**	なし。例外をスローします。
DistributedCache	**CACHE_FILES_TIMESTAMPS**	なし。例外をスローします。
DistributedCache	**CACHE_LOCALARCHIVES**	なし。例外をスローします。
DistributedCache	**CACHE_LOCALFILES**	なし。例外をスローします。
DistributedCache	**CACHE_SYMLINK**	なし。例外をスローします。
DistributedCache	**addArchiveToClassPath(Path archive, Configuration conf)**	なし。例外をスローします。
DistributedCache	**addArchiveToClassPath(Path archive, Configuration conf, FileSystem fs)**	なし。例外をスローします。
DistributedCache	**addCacheArchive(URI uri, Configuration conf)**	なし。例外をスローします。
DistributedCache	**addCacheFile(URI uri, Configuration conf)**	なし。例外をスローします。
DistributedCache	**addFileToClassPath(Path file, Configuration conf)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
DistributedCache	**addFileToClassPath(Path file, Configuration conf, FileSystem fs)**	なし。例外をスローします。
DistributedCache	**addLocalArchives(Configuration conf, String str)**	なし。例外をスローします。
DistributedCache	**addLocalFiles(Configuration conf, String str)**	なし。例外をスローします。
DistributedCache	**checkURIs(URI[] uriFiles, URI[] uriArchives)**	なし。例外をスローします。
DistributedCache	**createAllSymlink(Configuration conf, File jobCacheDir, File workDir)**	なし。例外をスローします。
DistributedCache	**createSymlink(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getArchiveClassPaths(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getArchiveTimestamps(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getCacheArchives(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getCacheFiles(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getFileClassPaths(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getFileStatus(Configuration conf, URI cache)**	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
DistributedCache	**getFileTimestamps(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getLocalCacheArchives(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getLocalCacheFiles(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getSymlink(Configuration conf)**	なし。例外をスローします。
DistributedCache	**getTimestamp(Configuration conf, URI cache)**	なし。例外をスローします。
DistributedCache	**setArchiveTimestamps(Configuration conf, String timestamps)**	なし。例外をスローします。
DistributedCache	**setCacheArchives(URI[] archives, Configuration conf)**	なし。例外をスローします。
DistributedCache	**setCacheFiles(URI[] files, Configuration conf)**	なし。例外をスローします。
DistributedCache	**setFileTimestamps(Configuration conf, String timestamps)**	なし。例外をスローします。
DistributedCache	**setLocalArchives(Configuration conf, String str)**	なし。例外をスローします。
DistributedCache	**setLocalFiles(Configuration conf, String str)**	なし。例外をスローします。
IsolationRunner	関連なし	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
Profiling	関連なし	空実装
Debugging	関連なし	空実装
Data Compression	関連なし	あり
Skipping Bad Records	関連なし	なし。例外をスローします。
Job Authorization	**mapred.acls.enabled**	なし。例外をスローします。
Job Authorization	**mapreduce.job.acl-view-job**	なし。例外をスローします。
Job Authorization	**mapreduce.job.acl-modify-job**	なし。例外をスローします。
Job Authorization	**mapreduce.cluster.administrators**	なし。例外をスローします。
Job Authorization	**mapred.queue.queue-name.acl-administer-jobs**	なし。例外をスローします。
MultipleInputs	関連なし	なし。例外をスローします。
Multi{anchor:_GoBack}pleOutputs	関連なし	あり
org.apache.hadoop.mapreduce.lib.db	関連なし	なし。例外をスローします。
org.apache.hadoop.mapreduce.security	関連なし	なし。例外をスローします。
org.apache.hadoop.mapreduce.lib.jobcontrol	関連なし	なし。例外をスローします。

タイプ	インターフェイス	互換性の有無
org.apache.hadoop.mapreduce.lib.chain	関連なし	なし。例外をスローします。
org.apache.hadoop.mapreduce.lib.db	関連なし	なし。例外をスローします。

7 Java サンドボックス

分散環境での実行中に、**MaxCompute**、**MapReduce**、**UDF** は **Java** サンドボックスによって制限を受けます。 **MR Main** など、**MapReduce** ジョブのメインプログラムは制限を受けません。 具体的な制限は以下のとおりです。

- ローカルファイルへの直接アクセスは許可されていません。 **MaxCompute MapReduce/Graph** によって提供されたインターフェイスのみを使用して、ファイルにアクセスします。
  - ファイル、**Jar** パッケージ、リソーステーブルを含む、リソースオプションで指定されたリソースを読み込みます。
  - **System.out** と**System.err** を通じて、ログ情報を出力します。 **MaxCompute** コンソールで **Log** コマンドを実行し、ログ情報を表示します。
- 分散ファイルシステムへの直接アクセスは許可されていません。 **MaxCompute MapReduce/Graph** を使用して、テーブルレコードにのみアクセスします。
- **JNI** による呼び出しは許可されていません。
- **Java** スレッドの作成は許可されていません。 **Linux** コマンドを実行するためのサブプロセスの開始は許可されていません。
- ローカル **IP** アドレスの取得などの、ネットワークアクセスは許可されていません。
- **Java** リフレクションは制限されています。**suppressAccessChecks** 権限は拒否されます。**private** 属性を取得するため、または **private** メソッドを呼び出すために、**private** 属性または **private** メソッドをアクセス可能に設定することはできません。

特にユーザーコードについては、これらの手順に従うと、**access denied** がスローされます。

- **java.io.File**

```
public boolean delete()
public void deleteOnExit()
public boolean exists()
public boolean canRead()
public boolean isFile()
public boolean isDirectory()
public boolean isHidden()
public long lastModified()
public long length()
public String[] list()
public String[] list(FilenameFilter filter)
public File[] listFiles()
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
public boolean canWrite()
public boolean createNewFile()
public static File createTempFile(String prefix, String suffix)
public static File createTempFile(String prefix, String suffix,File
directory)
```

```
public boolean mkdir()
public boolean mkdirs()
public boolean renameTo(File dest)
public boolean setLastModified(long time)
public boolean setReadOnly()
```

- **java.io.RandomAccessFile**

```
RandomAccessFile(String name, String mode)
RandomAccessFile(File file, String mode)
```

- **java.io.FileInputStream**

```
FileInputStream(FileDescriptor fdObj)
FileInputStream(String name)
FileInputStream(File file)
```

- **java.io.FileOutputStream**

```
FileOutputStream(FileDescriptor fdObj)
FileOutputStream(File file)
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
```

- **java.lang.Class**

```
public ProtectionDomain getProtectionDomain()
```

- **java.lang.ClassLoader**

```
ClassLoader()
ClassLoader(ClassLoader parent)
```

- **java.lang.Runtime**

```
public Process exec(String command)
public Process exec(String command, String envp[])
public Process exec(String cmdarray[])
public Process exec(String cmdarray[], String envp[])
public void exit(int status)
public static void runFinalizersOnExit(boolean value)
public void addShutdownHook(Thread hook)
public boolean removeShutdownHook(Thread hook)
public void load(String lib)
public void loadLibrary(String lib)
```

- **java.lang.System**

```
public static void exit(int status)
public static void runFinalizersOnExit(boolean value)
public static void load(String filename)
public static void loadLibrary( String libname)
public static Properties getProperties()
public static void setProperties(Properties props)
public static String getProperty(String key) //Only some keys are
allowed for file access.
public static String getProperty(String key, String def) // Only
some keys are allowed for file access.
public static String setProperty(String key, String value)
public static void setIn(InputStream in)
```

```
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
public static synchronized void setSecurityManager(SecurityManager s
)
```

System.getProperty で使用できるキーの一覧

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.specification.vendor
java.specification.name
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
file.encoding
user.timezone
```

- **java.lang.Thread**

```
Thread()
Thread(Runnable target)
Thread(String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, ...)
public final void checkAccess()
public void interrupt()
public final void suspend()
public final void resume()
public final void setPriority (int newPriority)
public final void setName(String name)
public final void setDaemon(boolean on)
public final void stop()
public final synchronized void stop(Throwable obj)
public static int enumerate(Thread tarray[])
public void setContextClassLoader(ClassLoader cl)
```

- **java.lang.ThreadGroup**

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)
public final void checkAccess()
public int enumerate(Thread list[])
public int enumerate(Thread list[], boolean recurse)
public int enumerate(ThreadGroup list[])
public int enumerate(ThreadGroup list[], boolean recurse)
public final ThreadGroup getParent()
public final void setDaemon(boolean daemon)
public final void setMaxPriority(int pri)
public final void suspend()
public final void resume()
```

```
public final void destroy()
public final void interrupt()
public final void stop()
```

- **java.lang.reflect.AccessibleObject**

```
public static void setAccessible(...)
public void setAccessible(...)
```

- **java.net.InetAddress**

```
public String getHostName()
public static InetAddress[] getAllByName(String host)
public static InetAddress getLocalHost()
```

- **java.net.DatagramSocket**

```
public InetAddress getLocalAddress()
```

- **java.net.Socket**

```
Socket(...)
```

- **java.net.ServerSocket**

```
ServerSocket(...)
public Socket accept()
protected final void implAccept(Socket s)
public static synchronized void setSocketFactory(...)
public static synchronized void setSocketImplFactory(...)
```

- **java.net.DatagramSocket**

```
DatagramSocket(...)
public synchronized void receive(DatagramPacket p)
```

- **java.net.MulticastSocket**

```
MulticastSocket(...)
```

- **java.net.URL**

```
URL(...)
public static synchronized void setURLStreamHandlerFactory(...)
java.net.URLConnection
public static synchronized void setContentHandlerFactory(...)
public static void setFileNameMap(FileNameMap map)
```

- **java.net.HttpURLConnection**

```
public static void setFollowRedirects(boolean set)
java.net.URLClassLoader
URLClassLoader(...)
```

- **java.security.AccessControlContext**

```
public AccessControlContext(AccessControlContext acc, DomainCombiner
 combiner)
```

```
public DomainCombiner getDomainCombiner()
```

8 外部テーブル

8.1 外部テーブルの概要

MaxCompute は、**Alibaba Cloud** ビッグデータプラットフォームのコアコンピューティングコンポーネントです。 **MaxCompute** は、強力な計算能力を持ち、大量のノード上で並列計算タスクをスケジュールすることができます。 また、分散コンピューティングのフェイルオーバー、再試行、およびその他の機能に対する実証済みの処理管理メカニズムも提供します。

分散データ処理の入力として、**MaxCompute SQL** はエクサバイトのオフラインデータの迅速な処理と保存を強力にサポートします。 ビッグデータビジネスの継続的な拡大に伴い、データの新しい使用例が数多く出現しており、**MaxCompute** のコンピューティングフレームワークも進化しています。 強力な計算能力へのアクセスは、特殊な形式の内部データではなく、外部データソースにも徐々に広がっています。

現段階で、**MaxCompute SQL** は内部の **MaxCompute** テーブルに **cfile** 形式で構造化データを保存します。 **MaxCompute** テーブル以外のユーザーデータ (テキスト、各種の非構造化データを含む) の場合、最初にさまざまなツールを使用してデータを **MaxCompute** テーブルにインポートしてから計算する必要があります。 データインポートのプロセスには大きな制限があります。 たとえば、**MaxCompute** で **OSS** データを処理するには、**2** つの一般的な方法を使用できます。

- **OSS SDK** または他のツールを使用して **OSS** からデータをダウンロードするために、**MaxCompute** トンネルを介してデータをテーブルにインポートします。
- **UDF** を作成し、**UDF** 内で直接 **OSS SDK** を呼び出して **OSS** データにアクセスします。

しかし、上記 **2** つの方法には短所があります。

- **MaxCompute** システムの外部にデータを中継する必要があります。 **OSS** のデータ量が大きすぎると、並行処理を使用してプロセスの高速化を検討する必要があり、**MaxCompute** の大規模コンピューティング機能を最大限に活用することはできません。
- **2** 番目のタイプは通常、**UDF** ネットワークアクセスに適用する必要があります。開発者がジョブの同時実行数とデータの分割方法を制御することも問題です。

ここでは、外部テーブルの機能について説明します。外部テーブルは、既存の **MaxCompute** テーブル以外のデータを処理できるように設計されています。 このフレームワークでは、**MaxCompute** で外部テーブルを作成するために単純な **DDL** 文を使用します。 その後、**MaxCompute** テーブルを外部データソースと関連付けて、さまざまなデータアクセスと出力機

能を提供できます。 外部テーブルを作成した後、(ほとんどの場合) **MaxCompute** テーブルと同じように使用して、**MaxCompute SQL** の強力な計算機能を最大限に活用できます。



注:

外部テーブル機能を使用すると、外部テーブルのデータはコピーされずに **MaxCompute** に保存されます。

ここでは、さまざまなデータを **2** 次元にカバーします。

さまざまなデータ記憶媒体**:** プラグインフレームワークは、**OSS**、**OTS** などの多種多様なデータ記憶媒体に接続するために使用することができます。

多様なデータ形式**: MaxCompute** テーブルは構造化データですが、外部テーブルは構造化データに限定されません。

- 画像、オーディオ、ビデオファイル、生バインドなど、構造化データはありません。
- **CSV**、**TSV** などの半構造化データは、特定のスキーマテキストファイルを意味します。 **orc/parquet** ファイルや **hbase/OTS** データなどの **cfile** 以外のファイルの構造化データです。

非構造化データの処理についての洞察を得られるよう、いくつかの例を挙げます。

- **OSS** および **OTS** の非構造化データにアクセスするには、「*OSS* 非構造化データへのアクセス」と「*OTS* 非構造化データへのアクセス」をご参照ください。
- 外部テーブルは **OSS** アカウントにアクセスします。**MaxCompute** が **OSS** にアクセスする権限を **RAM** でカスタマイズする方法については、「*OSS STS* モードでの権限付与」をご参照ください。
- **MaxCompute** の非構造化フレームワークは挿入を介して **OSS** に直接 **MaxCompute** の出力をサポートします。詳細については「*OSS* へ出力される非構造化データ」をご参照ください。
- ミドルウェアデータベースのデータを操作するには、「*OSS* 用オープンソース形式データの処理」を参照し、さまざまなオープンソース形式のデータを処理します。

8.2 OSS STS モードの許可

ここでは、**MaxCompute** が **OSS** にアクセスするためのアクセス権限を **RAM** でカスタマイズする方法を説明します。

外部テーブルの作成時、保存先アクセス **OSS** アカウントは、プレーンテキストの `AccessKeyId` と `AccessKeySecret` を入力できますが、アカウント漏洩のリスクがあります。 シナリオによっては、このリスクは許容できないため、**MaxCompute** は **OSS** にアクセスするためのより安全な方法を提供します。

MaxCompute は、**Alibaba Cloud** のアクセス制御サービス **(RAM)** とトークンサービス **(STS)** を組み合わせて、アカウントのセキュリティ問題に対処しています。 権限を与えるには **2** つの方法があります。

- **MaxCompute** と **OSS** の所有者が同じアカウントの場合、ワンクリック認証操作を **RAM** コンソールで直接実行できます。
- カスタム認証。

1. 最初に、**RAM** で **OSS** へのアクセスを **MaxCompute** に許可する必要があります。 `AliyunODPSDefaultRole` や `AliyunODPSRoleForOtherUser` などのロール名を作成し、ポリシーの内容を設定します。

```
-- MaxCompute と OSS の所有者が同じアカウントの場合

"Statement ":[

"Action": "STS: apererole ",
"Effect": "allow ",
"Principal ":{
"Service ":[
"Maid"

}


"Version": "1"

-- MaxCompute と OSS の所有者が同じアカウントでない場合

"Statement ":[

"Action": "STS: apererole ",
"Effect": "allow ",
"Principal ":{
"Service ":[
"MaxCompute's owner cloud account page"




"Version": "1"
```

2. **OSS** * へのアクセスに必要な権限をロールに付与します。 以下のように設定します。

```
Version: "1 ",
"Statement ":[

"Action ":[
 "Oss: listbuckets ",
 "Oss: GetObject ",
 "Oss: maid ",
 "Oss: putobject ",
 "Oss: deleteobject ",
 "Oss: maid ",
 "Oss: listparts"
```

```
"Resource ":"*",
"Effect": "allow"


-- 他の権限をカスタマイズ可能
```

3. その後、権限ボックスがロールに付与されます。



注：

承認が完了したらロールの詳細を表示してロールの **RAN** 情報を取得します。 **OSS** 外部テーブルを作成するときには、この **RAN** 情報を指定する必要があります。

8.3 OSS 非構造化データへのアクセス

ここでは、**MaxCompute** 上で **OSS** データに簡単にアクセスする方法を説明します。

STS モードでの権限付与

MaxCompute が **OSS** に直接アクセスできるよう、事前に **MaxCompute** アカウントに **OSS** データ権限を付与します。 次の **2** つの方法で権限を付与できます。

- *MaxCompute* と *OSS* の所有者が同じアカウントの場合、**Alibaba Cloud** アカウントに直接ログインし、ここをクリックして権限付与を完了します。
- カスタム権限付与

1. まず、*RAM* で、**OSS** へのアクセス権限を **MaxCompute** に付与する必要があります。 *RAM* コンソールにログイン (**Maxcompute** と **OSS** が同一でなく、**OSS** アカウントログインによって権限付与されている場合) し、コンソールのロール管理でロール (`AliyunODPSDefaultRole` や `AliyunODPSRoleForOtherUser` など) を作成します。
2. 次のようにロールのポリシー内容を変更します。

```
--When the MaxCompute and OSS owner are the same account:
{
"Statement": [
{
 "Action": "sts:AssumeRole",
 "Effect": "Allow",
 "Principal": {
   "Service": [
     "odps.aliyuncs.com"
   ]
 }
}
],
"Version": "1"
}
--When the MaxCompute and OSS owner are not the same account:
{
"Statement": [
```

```
{
 "Action": "sts:AssumeRole",
 "Effect": "Allow",
 "Principal": {
   "Service ":[
     "MaxCompute's Owner account: id@odps.aliyuncs.com"
   ]
 }
}
],
"Version": "1"
}
```

3. 以下のように、**OSS** へのアクセスに必要な権限 `AliyunODPSRolePolicy` をロールに付与します。

```
{
"Version": "1",
"Statement": [
{
 "Action": [
   "oss:ListBuckets",
   "oss:GetObject",
   "oss:ListObjects",
   "oss:PutObject",
   "oss:DeleteObject",
   "Oss: maid ",
   "oss:ListParts"
 ],
 "Resource": "*",
 "Effect": "Allow"
}
]
}
--You can customize other permissions.
```

4. 権限 **AliyunODPSRolePolicy** をこのロールに付与します。

組み込みエクストラクターを用いた OSS データの読み取り

外部データソースにアクセスするときは、カスタムエクストラクターを使用する必要があります。 **MaxCompute** の組み込みエクストラクターを使用して、*OSS* に保存されている従来形式のデータを読み取ることもできます。 外部テーブルを作成し、このテーブルをクエリ操作のソーステーブルとして使用します。

この例では、*OSS* に **CSV** データファイルを保存しているとします。 エンドポイントは `oss-cn-shanghai-internal.aliyuncs.com`、バケットは `oss-odps-test`、データファイルの保存先は */demo/vehicle.csv*です。

外部テーブルの作成

次の文を使用して外部テーブルを作成します。

```
CREATE EXTERNAL TABLE IF NOT EXISTS ambulance_data_csv_external
(
```

```
vehicleId int,
recordId int,
patientId int,
calls int,
locationLatitute double,
locationLongtitue double,
recordTime string,
direction string
)
STORED BY 'com.aliyun.odps.CsvStorageHandler' -- (1)
WITH SERDEPROPERTIES (
 'odps.properties.rolearn'='acs:ram::xxxxx:role/aliyunodpsdefaultrole'
) -- (2)
LOCATION 'oss://oss-cn-shanghai-internal.aliyuncs.com/oss-odps-test/
Demo/'; -- (3)(4)
```

上記の文の説明：

- `com.aliyun.odps.CsvStorageHandler` は、**CSV** 形式のファイルを処理するための組み込み `StorageHandler` です。 **CSV** ファイルの読み取りと書き込みの方法を定義します。 この名前を指定します。 関連ロジックはシステムによって実装されます。
- `odps.properties.rolearn` の情報は、**RAM** の `AliyunODPSDefaultRole` の `Arn` 情報から取得しています。 **RAM** コンソールのロールの詳細から取得できます。
- **LOCATION** には **OSS** ディレクトリを指定する必要があります。 デフォルトでは、このディレクトリ内のすべてのファイルが読み取られます。
  - **OSS** データフローの料金が発生しないよう、イントラネットのドメイン名を使用することを推奨します。
  - **OSS** データを保存するリージョンは、**MaxCompute** を有効化しているリージョンと同じにすることを推奨します。 **MaxCompute** は一部のリージョンにのみデプロイできるため、リージョン間のデータ接続は保証されません。
  - **OSS** 接続フォーマットは `oss://oss-cn-shanghai-internal.aliyuncs.com/bucketname/directoryname/` です。 ディレクトリの後にファイル名を追加する必要はありません。 以下に一般的なエラーを示します。

    ```
    http://oss-odps-test.oss-cn-shanghai-internal.aliyuncs.com/Demo/
     -- HTTP connection is not supported.
    https://oss-odps-test.oss-cn-shanghai-internal.aliyuncs.com/Demo/
     -- HTTPS connection is not supported.
    oss://oss-odps-test.oss-cn-shanghai-internal.aliyuncs.com/Demo
     -- The connection address is incorrect.
    oss://oss://oss-cn-shanghai-internal.aliyuncs.com/oss-odps-test/
    Demo/vehicle.csv  -- You do not need to specify the file name.
    ```

- **MaxCompute** システムでは、関連付けられた **OSS** ディレクトリのみを外部テーブルに記録します。 このテーブルを **DROP** (削除) しても、対応する `LOCATION` データは削除されません。

作成した外部テーブルの構造を表示する場合、次の文を実行します。

```
desc extended <table_name>
```

返された情報の **Extended Info** には、**StorageHandler** や **Location** などの外部テーブル情報が含まれています。

外部テーブルを用いたテーブルデータへのアクセス

外部テーブルを作成後、そのテーブルを通常のテーブルとして使用できます。 `/demo/vehicle.csv` のデータを以下とします。

```
1,1,51,1,46.81006,-92.08174,9/14/2014 0:00,S
1,2,13,1,46.81006,-92.08174,9/14/2014 0:00,NE
1,3,48,1,46.81006,-92.08174,9/14/2014 0:00,NE
1,4,30,1,46.81006,-92.08174,9/14/2014 0:00,W
1,5,47,1,46.81006,-92.08174,9/14/2014 0:00,S
1,6,9,1,46.81006,-92.08174,9/14/2014 0:00,S
1,7,53,1,46.81006,-92.08174,9/14/2014 0:00,N
1,8,63,1,46.81006,-92.08174,9/14/2014 0:00,SW
1,9,4,1,46.81006,-92.08174,9/14/2014 0:00,NE
1,10,31,1,46.81006,-92.08174,9/14/2014 0:00,N
```

次の **SQL** 文を実行します。

```
select recordId, patientId, direction from ambulance_data_csv_external
 where patientId > 25;
```



注：

現在、外部テーブルは **MaxCompute SQL** を介してのみ操作が可能です。 **MaxCompute MapReduce** は外部テーブルの操作ができません。

この文は、ジョブを送信し、組み込み **CSV** エクストラクターをスケジュールして **OSS** からデータを読み取り、処理します。 結果は次のとおりです。

```
+------------+------------+-----------+
| recordId   | patientId  | direction |
+------------+------------+-----------+
| 1 | 51 | S |
| 3          | 48         | NE        |
| 4          | 30         | W         |
| 5          | 47         | S         |
| 7          | 53         | N         |
| 8          | 63         | SW        |
| 10         | 31         | N         |
+------------+------------+-----------+
```

カスタムエクストラクターを用いた OSS データの読み取り

OSS データが複雑な形式で、かつ組み込みエクストラクターが要件を満たさない場合、カスタムエクストラクターを使用して **OSS** ファイルからデータを読み取る必要があります。

たとえば、**CSV** 形式ではない **TXT** データファイルがあり、`|` がレコード間の列区切り文字として使用されているとします。 たとえば、`/demo/SampleData/CustomTxt/AmbulanceData/vehicle.csv` のデータは、以下のようになります。

```
1|1|51|1|46.81006|-92.08174|9/14/2014 0:00|S
1|2|13|1|46.81006|-92.08174|9/14/2014 0:00|NE
1|3|48|1|46.81006|-92.08174|9/14/2014 0:00|NE
1|4|30|1|46.81006|-92.08174|9/14/2014 0:00|W
1|5|47|1|46.81006|-92.08174|9/14/2014 0:00|S
1|6|9|1|46.81006|-92.08174|9/14/2014 0:00|S
1|7|53|1|46.81006|-92.08174|9/14/2014 0:00|N
1|8|63|1|46.81006|-92.08174|9/14/2014 0:00|SW
1|9|4|1|46.81006|-92.08174|9/14/2014 0:00|NE
1|10|31|1|46.81006|-92.08174|9/14/2014 0:00|N
```

・ エクストラクターの定義

区切り文字をパラメータとして使用し、共通のエクストラクターを作成します。 これにより、すべてのテキストファイルを同じ形式で処理できます。 以下をご参照ください。

```
/**
 * Text extractor that extract schematized records from formatted
plain-text(csv, tsv etc.)
 **/
public class TextExtractor extends Extractor {
  private InputStreamSet inputs;
  private String columnDelimiter;
  private DataAttributes attributes;
  private BufferedReader currentReader;
  private boolean firstRead = true;
  public TextExtractor() {
    // default to ",", this can be overwritten if a specific
delimiter is provided (via DataAttributes)
    this.columnDelimiter = ",";
  }
  // no particular usage for execution context in this example
  @Override
  public void setup(ExecutionContext ctx, InputStreamSet inputs,
DataAttributes attributes) {
    this.inputs = inputs; // inputs is an InputStreamSet, each call
to next() returns an InputStream. This InputStream can read all the
content in an OSS file.
    this.attributes = attributes;
    // check if "delimiter" attribute is supplied via SQL query
    String columnDelimiter = this.attributes.getValueByKey("
delimiter"); //The delimiter parameter is supplied by a DDL
statement.
    if ( columnDelimiter ! = NULL)
    {
      this.columnDelimiter = columnDelimiter;
    }
    // note: more properties can be inited from attributes if needed
  }
  @Override
  public Record extract() throws IOException {//extractor() calls
return one record, corresponding to one record in an external table.
    String line = readNextLine();
    if (line == null) {
```

```
      return null;  // A return value of NULL indicates that this
table has no readable records.
    }
    return textLineToRecord(line);  // textLineToRecord splits a row
 of data into multiple columns according to the delimiter.
  }
  @Override
  public void close(){
    // no-op
  }
}
```

textLineToRecord 分割データの完全な実装は、こちらをご参照ください。

- **StorageHandler** の定義

  StorageHandler は、カスタム外部テーブルロジックの集中ポータルとして機能します。

```
package com.aliyun.odps.udf.example.text;
public class TextStorageHandler extends OdpsStorageHandler {
  @Override
  public Class<? extends Extractor> getExtractorClass() {
    return TextExtractor.class;
  }
  @Override
  public Class<? extends Outputer>getOutputerClass() {
    return TextOutputer.class;
  }
}
```

- コンパイルとパッケージ

  カスタムコードをパッケージにコンパイルして **MaxCompute** にアップロードします。

```
add jar odps-udf-example.jar;
```

- 外部テーブルの作成

  組み込みエクストラクターの使用方法と同様に、まず外部テーブルを作成する必要があります。 異なる点は、外部テーブルのアクセスデータを指定する際、カスタム **StorageHandler** を使用する必要があるということです。

  次の文を使用して外部テーブルを作成します。

```
CREATE EXTERNAL TABLE IF NOT EXISTS ambulance_data_txt_external
(
vehicleId int,
recordId int,
patientId int,
calls int,
locationLatitute double,
locationLongtitue double,
recordTime string,
direction string
)
STORED BY 'com.aliyun.odps.udf.example.text.TextStorageHandler'   --
STORED BY specifies the custom StorageHandler class name.
  with SERDEPROPERTIES (
```

```
'delimiter'='\\|', -- SERDEPROPERITES can specify parameters, these
 parameters are passed through the DataAttributes to the Extractor
code.
'odps.properties.rolearn'='acs:ram::xxxxxxxxxxxxx:role/aliyunodps
defaultrole'
)
LOCATION 'oss://oss-cn-shanghai-internal.aliyuncs.com/oss-odps-test/
Demo/SampleData/CustomTxt/AmbulanceData/'
USING 'odps-udf-example.jar'; --You must also specify the jar
package containing the class definition.
```

・ 外部テーブルの照会

次の **SQL** 文を実行します。

```
select recordId, patientId, direction from ambulance_data_txt_e
xternal where patientId > 25;
```

カスタムエクストラクターを用いた非構造化データの読み取り

ここまでで、組み込みエクストラクターやカスタムエクストラクターを使用して、**OSS** に保存されている **CSV** などのテキストデータを簡単に処理できるようになりました。 次に、オーディオデータ **(WAV** 形式のファイル**)** を例として、カスタムエクストラクターを使用して **OSS** のテキスト以外のファイルにアクセスし、処理する方法を説明します。

ここでは、最後の **SQL** 文以降、**OSS** に保存されているオーディオファイルの処理ポータルとして **MaxCompute SQL** を使用する方法を説明します。

次のように外部テーブル **SQL** を作成します**:**

```
CREATE EXTERNAL TABLE IF NOT EXISTS speech_sentence_snr_external
(
sentence_snr double,
id string
)
STORED BY 'com.aliyun.odps.udf.example.speech.SpeechStorageHandler'
WITH SERDEPROPERTIES (
    'mlfFileName'='sm_random_5_utterance.text.label' ,
    'speechSampleRateInKHz' = '16'
)
LOCATION 'oss://oss-cn-shanghai-internal.aliyuncs.com/oss-odps-test/
dev/SpeechSentenceTest/'
USING 'odps-udf-example.jar,sm_random_5_utterance.text.label';
```

上記の例のように、外部テーブルを作成する必要があります。 次に、このテーブルのスキーマを使用して、オーディオファイルから抽出する情報を定義します。

・ オーディオファイルの文の **SNR (**信号対雑音比**) : sentence_snr**

・ オーディオファイルの名前**: ID**

外部テーブルの作成後、標準の **SELECT** 文を使用してクエリを実行します。 この操作はエクストラクターを起動して計算を実行します。 **OSS** データを読み取り、処理するときは、テキストファイルの単純な逆シリアル化に加えて、カスタムエクストラクターを使用してより複雑な

データ処理と抽出ロジックを実行できます。 この例では、 `com.aliyun.odps.udf.example.speech.SpeechStorageHandler` にカプセル化されたカスタムエクストラクターを使用し、オーディオファイル内の有効なステートメントの平均 **SNR** を計算し、**SQL** 操作の構造化データを抽出します **(WHERE sentence_snr > 10)**。 操作が完了すると、**10** 以上の **SNR** を持つオーディオファイルと、対応する **SNR** 値が返されます。

複数の **WAV** 形式のファイルが **OSS** アドレス `oss://oss-cn-hangzhou-zmf.aliyuncs.com/oss-odps-test/dev/SpeechSentenceTest/` に保存されています。 **MaxCompute** フレームワークは、このアドレスにあるすべてのファイルを読み取り、必要に応じてファイルレベルのシャーディングを実行します。 ファイルは複数の処理用計算ノードに自動的に割り当てられます。 各計算ノード上で、エクストラクターは、**InputStreamSet** によってノードに割り当てられたファイルセットを処理します。 特別な処理ロジックは、シングルホストプログラムと類似しています。 アルゴリズムは、クラスに従った単一のホストメソッドを使って実装されます。

`SpeechSentenceSnrExtractor` の処理ロジックの詳細は次のとおりです。

まず、 `setup` インターフェイスのパラメーターを読み取り、初期化を実行して、オーディオ処理モデル (リソース導入を使用) をインポートします。

```
public SpeechSentenceSnrExtractor(){
    this.utteranceLabels = new HashMap<String, UtteranceLabel>();
  }
  @Override
  public void setup(ExecutionContext ctx, InputStreamSet inputs,
DataAttributes attributes){
    this.inputs = inputs;
    This. Attributes = attributes;
    this.mlfFileName = this.attributes.getValueByKey(MLF_FILE_A
TTRIBUTE_KEY);
    String sampleRateInKHzStr = this.attributes.getValueByKey(
SPEECH_SAMPLE_RATE_KEY);
    this.sampleRateInKHz = Double.parseDouble(sampleRateInKHzStr);
    try {
      // read the speech model file from resource and load the model
into memory
      BufferedInputStream inputStream = ctx.readResourceFileAsStream(
mlfFileName);
      loadMlfLabelsFromResource(inputStream);
      inputStream.close();
    } catch (IOException e) {
      throw new RuntimeException("reading model from mlf failed with
exception " + e.getMessage());
    }
  }
```

extract() インターフェイスは、オーディオファイルの読み取りと処理ロジックを実装し、オーディオモデルに基づいてデータの **SNR** (信号対雑音比) を計算し、**[snr, id]** の形式で結果を **Record** に書き込みます。

前の例では、実装プロセスが簡略化されており、関連するオーディオ処理アルゴリズムのロジックは含まれていません。 オープンソースコミュニティで **MaxCompute SDK** が提供しているサンプルコードをご参照ください。

```
@Override
  public Record extract() throws IOException {
    SourceInputStream inputStream = inputs.next();
    if (inputStream == null){
      return null;
    }
    // process one wav file to extract one output record [snr, id]
    String fileName = inputStream.getFileName();
    fileName = fileName.substring(fileName.lastIndexOf('/') + 1);
    logger.info("Processing wav file " + fileName);
    String id = fileName.substring(0, fileName.lastIndexOf('.'));
    // read speech file into memory buffer
    long fileSize = inputStream.getFileSize();
    byte[] buffer = new byte[(int)fileSize];
    int readSize = inputStream.readToEnd(buffer);
    inputStream.close();
    // compute the avg sentence snr
    double snr = computeSnr(id, buffer, readSize);
    // construct output record [snr, id]
    Column[] outputColumns = this.attributes.getRecordColumns();
    ArrayRecord record = new ArrayRecord(outputColumns);
    record.setDouble(0, snr);
    record.setString(1, id);
    return record;
  }
  private void loadMlfLabelsFromResource(BufferedInputStream
fileInputStream)
          throws IOException {
    //  skipped here
  }
  // compute the snr of the speech sentence, assuming the input buffer
 contains the entire content of a wav file
  private double computeSnr(String id, byte[] buffer, int validBuffe
rLen){
    // computing the snr value for the wav file (supplied as byte
buffer array), skipped here
  }
```

クエリの実行

```
select sentence_snr, id
    from speech_sentence_snr_external
where sentence_snr > 10.0;
```

結果

```
-----------------------------------------------------------------
| sentence_snr |                     id                        |
-----------------------------------------------------------------
|   34.4703    |          J310209090013_H02_K03_042            |
-----------------------------------------------------------------
|   31.3905    | tsh148_seg_2_3013_3_6_48_80bd359827e24dd7_0 |
-----------------------------------------------------------------
|   35.4774    | tsh148_seg_3013_1_31_11_9d7c87aef9f3e559_0  |
-----------------------------------------------------------------
```

```
|   16.0462    | tsh148_seg_3013_2_29_49_f4cb0990a6b4060c_0  |
-----------------------------------------------------------------
|   14.5568    |   tsh_148_3013_5_13_47_3d5008d792408f81_0   |
-----------------------------------------------------------------
```

カスタムエクストラクターを使用することで、**OSS** に保存されている複数のオーディオデータファイルを **SQL** 文に基づく分散方式で処理できます。 同じ方法で、**MaxCompute** の大規模な計算能力を使用して、画像、ビデオなどのさまざまな種類の非構造化データを簡単に処理することもできます。

パーティションデータ

前のセクションでは、外部テーブルの関連データは **LOCATION** に指定された **OSS** ディレクトリを介して実装されていました。 ただし、処理中、**MaxCompute** はディレクトリ下のすべてのデータ (サブディレクトリ内のすべてのファイルを含む) を読み取ります。 時間の経過とともにデータがディレクトリに蓄積され、データ量が大きくなりすぎた場合、ディレクトリ全体をスキャンすると、不要な **IO** とデータ処理時間が余分に発生する可能性があります。 通常、この問題には **2** つの解決策があります。

- アクセスデータ量の削減**:** データストレージアドレスを計画し、複数の **EXTERNAL TABLE** の使用を検討してデータのさまざまな部分を記述し、各 **EXTERNAL TABLE LOCATION** がデータのサブセットをポイントするようにします。
- パーティションデータ**: EXTERNAL TABLE** は内部テーブルと同じで、パーティションテーブルの機能をサポートしているため、パーティション機能に基づいてデータの体系化を管理できます。

このセクションでは主に **EXTERNAL TABLE** のパーティション機能について説明します。

- *OSS* におけるパーティションデータの標準編成方法とパスの形式

  内部テーブルとは異なり、**MaxCompute** には外部メモリ **(OSS** など) に保存されているデータを管理する権限がありません。 そのため、システムでパーティションテーブル機能を使用する必要がある場合、**OSS** 上のデータファイルのストレージパスは特定の形式に準拠する必要があります。 形式は次のとおりです。

  ```
  partitionKey1=value1\partitionKey2=value2\...
  ```

  関連するサンプルは以下のとおりです。

  毎日の **LOG** ファイルを **OSS** に保存し、**MaxCompute** で処理するときに日付の粒度に基づいてデータの一部にアクセスするとします。 これらの **LOG** ファイルが **CSV** ファイルである

とすると (複雑でカスタマイズされた形式の使用方法は同じ)、次のパーティション化外部テーブルを使用してデータを定義できます。

```
CREATE EXTERNAL TABLE log_table_external (
    click STRING,
    ip STRING,
    url STRING,
  )
  PARTITIONED BY (
    year STRING,
    month STRING,
    day STRING
  )
  STORED BY 'com.aliyun.odps.CsvStorageHandler'
  WITH SERDEPROPERTIES (
 'odps.properties.rolearn'='acs:ram::xxxxx:role/aliyunodpsdefaultrol
e'
)
  LOCATION 'oss://oss-cn-hangzhou-zmf.aliyuncs.com/oss-odps-test/
log_data/';
```

上記のテーブルの文のように、前の例との違いは、外部テーブルを定義する際に、その外部テーブルが **PARTITIONED BY** 構文を介してパーティションテーブルとして指定されていることです。例は**3** 層のパーティションテーブルで、パーティションのキーは年、月、日です。

このようなパーティションを効果的に機能させるには、**OSS** にデータを保存する際に前述のパス形式に従う必要があります。 以下は、有効なパス保存レイアウトの例です。

```
osscmd ls oss://oss-odps-test/log_data/
2017-01-14 08:03:35 128MB Standard oss://oss-odps-test/log_data/year
=2016/month=06/day=01/logfile
2017-01-14 08:04:12 127MB Standard oss://oss-odps-test/log_data/year
=2016/month=06/day=01/logfile. 1
2017-01-14 08:05:02 118MB Standard oss://oss-odps-test/log_data/year
=2016/month=06/day=02/logfile
2017-01-14 08:06:45 123MB Standard oss://oss-odps-test/log_data/year
=2016/month=07/day=10/logfile
2017-01-14 08:07:11 115MB Standard oss://oss-odps-test/log_data/year
=2016/month=08/day=08/logfile
...
```



注：

オフラインデータを準備した場合、つまり **osscmd** などの **OSS** ツールを使用してオフラインデータを **OSS** ストレージサービスにアップロードした場合、データパス形式を定義します。

`ALTER TABLE ADD PARTITIONDDL` 文を使用して、パーティション情報を **MaxCompute** へ追加できます。

対応する **DDL** 文の例は次のとおりです。

```
ALTER TABLE log_table_external ADD PARTITION (year = '2016', month =
 '06', day = '01')
ALTER TABLE log_table_external ADD PARTITION (year = '2016', month =
 '06', day = '02')
ALTER TABLE log_table_external ADD PARTITION (year = '2016', month =
 '07', day = '10')
ALTER TABLE log_table_external ADD PARTITION (year = '2016', month =
 '08', day = '08')
...
```



注：

これらの操作は、標準の **MaxCompute** の内部テーブル操作と同じです。パーティションの詳細については、「*#unique_59*」をご参照ください。 データの準備が整い、パーティション情報がシステムにインポート済みの場合、**OSS** 上の外部テーブルデータのパーティショニングは、**SQL** 文を使用して実行できます。

2016 年 **6** 月 **1** 日の **LOG** 内の **IP** 数を分析する場合、次のコマンドを使用します。

```
SELECT count(distinct(ip)) FROM log_table_external WHERE year = '
2016' AND month = '06' AND day = '01';
```

この時、**log_table_external** では、外部テーブルに対応するディレクトリは、**log_data/** 全体ではなく、`log_data/year=2016/month=06/day=01` サブディレクトリ **(logfile** と**logfile 1)** 下のファイルにのみアクセスします。不要な **I/O** 操作が大量に発生しないようにします。

同様に、**2016** 年後半のデータのみを分析する場合は、次のコマンドを使用します。

```
SELECT count(distinct(ip)) FROM log_table_external
WHERE year = '2016' AND month > '06';
```

この場合、**OSS** に保存されている **LOG** の後半部分にのみにアクセスします。

- *OSS* 上のカスタマイズされたパスのパーティションデータ

  OSS に履歴を保存しているが、`partitionKey1=value1\partitionKey2=value2\...` のパス形式で保存されていない場合、**MaxCompute** のパーティションモードを使用してア

クセスできます。 MaxCompute はカスタマイズされたパスを介してパーティションをインポートする方法も提供しています。

パーティションの値のみがデータパスにあり、パーティションのキー情報は存在しないとします。 以下は、データパスストレージレイアウトの例です。

```
osscmd ls oss://oss-odps-test/log_data_customized/
2017-01-14 08:03:35 128MB Standard oss://oss-odps-test/log_data_c
ustomized/2016/06/01/logfile
2017-01-14 08:04:12 127MB Standard oss://oss-odps-test/log_data_c
ustomized/2016/06/01/logfile. 1
2017-01-14 08:05:02 118MB Standard oss://oss-odps-test/log_data_c
ustomized/2016/06/02/logfile
2017-01-14 08:06:45 123MB Standard oss://oss-odps-test/log_data_c
ustomized/2016/07/10/logfile
2017-01-14 08:07:11 115MB Standard oss://oss-odps-test/log_data_c
ustomized/2016/08/08/logfile
...
```

外部テーブルビルダー DDL では、前の例を参照して、句にパーティションキーを指定できます。

異なるサブディレクトリを異なるパーティションにバインドするには、次のカスタマイズされたパーティションパスのようなコマンドを使用します。

```
ALTER TABLE log_table_external ADD PARTITION (year = '2016', month = '
06', day = '01')
LOCATION 'oss://oss-cn-hangzhou-zmf.aliyuncs.com/oss-odps-test/
log_data_customized/2016/06/01/';
```

PARTITION ADD に LOCATION 情報を追加し、 パーティションデータパスをカスタマイズします。 データが `partitionKey1=value1\partitionKey2=value2\...` の推奨形式で保存されていなくても、サブディレクトリのパーティションデータにアクセスできます。

8.4 OSS のオープンソース形式データの処理

ここでは、MaxCompute の非構造化フレームワークを介して、OSS に保存されている一般的なオープンソースデータ形式 (ORC、PARQUET、SEQUENCEFILE、RCFILE、AVRO、TEXTFILE) を処理する方法について説明します。

「*OSS* 非構造化データへのアクセス」では、MaxCompute 上で OSS に保存されているテキスト、オーディオ、画像、その他の形式のデータにアクセスする方法を説明しています。 非構造的フレームワークは、オープンソースコミュニティの実装を直接呼び出して、オープンソースデータ形式を解析し、MaxCompute システムとシームレスにやりとりします。

 注：

OSS のオープンソース形式データを処理する前に、*STS* モードで *OSS* に権限を付与する必要があります。

外部テーブルの作成

MaxCompute 非構造化データフレームワークは、外部テーブルを介してさまざまなデータに関連付けられています。**OSS** に関連付けられたオープンソース形式データの外部テーブルの構文を以下に示します。

```
DROP TABLE [IF EXISTS] <external_table>;
CREATE EXTERNAL TABLE [IF NOT EXISTS] <external_table>
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
[ROW FORMAT SERDE '<serde class>'
  [With serdeproperties ('ODPS. properties. rolearn '=' $ {roleran
 }'[, 'name2 '= 'value2',...]
]
STORED AS <file format>
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```



注：

構文形式は **hive** の構文と非常に似ていますが、以下の問題に注意する必要があります。

- この文法形式の **STORED AS** キーワードは、通常の非構造化外部テーブルに使用される **STORED BY** キーではなく、オープンソース互換データを読み取るという点が独特です。

  STORED AS の後には *ORC/PARQUET/RCFILE/SEQUENCEFILE/TEXTFILE* などのファイル形式名が続きます。
- 外部テーブルの列スキーマは、**OSS** に保存されているデータのスキーマと一致する必要があります。
- `ROW FORMAT SERDE` オプションは必須ではなく、いくつかの特別な形式でのみ使用可能です。たとえば、**textfile** を使用する必要があります。
- `WITH SERDEPROPERTIES` を使用して、*STS* モードで **OSS** 権限を関連付ける場合、このパラメーターには `odps.properties.rolearn` 属性を指定する必要があります。この属性の値は、特に **RAM** 内で使用される **Role Arn** 情報です。

  STS モードを使用しない場合、**location** に直接平文で `AccessKeyId` と `AccessKeySecret` を渡すために、このプロパティを指定する必要はありません。

- OSS と明確な AK を関連付ける場合は、次のように記述します。

```
LOCATION 'oss://${accessKeyId}:${accessKeySecret}@${endpoint}/${
bucket}/${userPath}/'
```

- OSS 外部テーブルへのアクセスは、現在、外部ネットワークエンドポイントではサポートされていません。
- 現在、STORE AS の単一ファイルサイズは 3 G を超えることはできません。ファイルが大きすぎる場合は分割することを推奨します。

OSS に関連付けられた PARQUET データの例

複数の parquet ファイルが OSS パスに保存され、各ファイルは parquet 形式、スキーマは 16 列 (bigint 4 列の、double 4 列、string 8 列)、テーブル作成 Div 文のデータが以下であるとします。

```
CREATE EXTERNAL TABLE tpch_lineitem_parquet
(
  Rochelle orderkey bigint,
  l_partkey bigint,
  l_suppkey bigint,
  Rochelle linenumber bigint,
  l_quantity double,
  l_extendedprice double,
  l_discount double,
  l_tax double,
  l_returnflag string,
  l_linestatus string,
  l_shipdate string,
  l_commitdate string,
  l_receiptdate string,
  l_shipinstruct string,
  l_shipmode string,
  _Comment string
)
STORED AS PARQUET
LOCATION 'oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.
aliyuncs.com/bucket/parquet_data/';
```

デフォルトの PARQUET データは圧縮されていません。MaxCompute で PARQUET データを圧縮する必要がある場合は、`set odps.sql.hive.compatible=true;` を設定する必要があります。 サポートされている圧縮タイプは、SNAPPY と GZIP です。

OSS に関連付けられた Text データ

データが OSS に TEXTFILE ファイル として保存され (各行は JSON 形式)、OSS 内の複数のディレクトリによって構成されている場合は、MaxCompute パーティションテーブルとデータアソシエーションを使用できます。 パーティションテーブルの場合の DDL 文の例を以下に示します。

```
CREATE EXTERNAL TABLE tpch_lineitem_textfile
(
```

```
  l_orderkey bigint,
  l_partkey bigint,
  l_suppkey bigint,
  l_linenumber bigint,
  l_quantity double,
  l_extendedprice double,
  l_discount double,
  l_tax double,
  l_returnflag string,
  Maid string,
  l_shipdate string,
  Rochelle Commission string,
  l_receiptdate string,
  l_shipinstruct string,
  l_shipmode string,
  l_comment string
)
PARTITIONED BY (ds string)
ROW FORMAT serde 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE
Location 'oss: // $ {accesskeyid}: $ {accesskeysecret} @ fig /';
```

OSS テーブルディレクトリ下のサブディレクトリがパーティション名として構成されている場合、例は次のようになります。

```
oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.aliyuncs.
com/bucket/text_data/ds=20170102/'
oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.aliyuncs.
com/bucket/text_data/ds=20170103/'
...
```

次の **pant** 文を使用してパーティションを追加できます。

```
ALTER TABLE tpch_lineitem_textfile ADD PARTITION(ds="20170102");
ALTER TABLE tpch_lineitem_textfile ADD PARTITION(ds="20170103");
```

OSS パーティションディレクトリがこのように構成されていない場合、またはテーブルディレクトリにまったく含まれていない場合、例は次のようになります。

```
oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.aliyuncs.
com/bucket/text_data_20170102/;
oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.aliyuncs.
com/bucket/text_data_20170103/;
...
```

この場合は、次の **pant** 文を使用して **ADD PARTITION** を実行できます。

```
ALTER TABLE tpch_lineitem_textfile ADD PARTITION(ds="20170102")
LOCATION 'oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.
aliyuncs.com/bucket/text_data_20170102/';
ALTER TABLE tpch_lineitem_textfile ADD PARTITION(ds="20170103")
LOCATION 'oss://${accessKeyId}:${accessKeySecret}@oss-cn-hangzhou-zmf.
aliyuncs.com/bucket/text_data_20170103/';
```

```
...
```

テキストデータは、**serdeproperties (key:** デフォルト**)** をサポートしています。

```
Fields terminator : '\001'
Escape delimitor : '\\'
Collection items terminator : '\002'
Map keys terminator : '\003'
Lines terminate: '\ N'
Null defination : '\\N'
```

OSS に関連付けられた CSV データ

Tasmania 文の形式は次のとおりです。

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2. OpenCSVSerde'
  WITH SERDEPROPERTIES
    ('Separates atorchare' =, ', 'pigeon techar' = '"', 'escarechar '=
 '\\')
STORED AS TEXTFILE
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

上の文からわかるように、**CSV** データの **pant** 文は、**serdeproperties** をサポートします **(key:** デフォルト**)**

```
separatorChar : ','
quoteChar : '"'
Escarechar :'\'
```



注 :

Hive OpenCSVSerde は文字列型のみをサポートします。

OpenCSVSerde は現在 **Builtin Serde** に属していません。 **DML** 文を実行するときは、**odps.sql.hive.compatible** = **true** を設定する必要があります。

OSS に関連付けられた JSON データ

Tasmania ステートメントの形式は次のとおりです。**SERDEPROPERTIES** がサポートされています。

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE
```

```
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

OSS に関連付けられた ORC データ

Tasmania 文の形式は次のとおりです。

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
STORED AS ORC
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

OSS に関連付けられた AVRO データ

DDL 文の形式は次のとおりです。

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
STORED AS AVRO
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

OSS に関連付けられた SEQUENCEFILE データ

```
CREATE EXTERNAL TABLE [IF NOT EXISTS]
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
STORED AS SEQUENCEFILE
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

OSS のオープンソース形式データの読み取りと処理

他のドキュメントで作成した **2** つの外部表現を比較し、ファイルタイプが異なる場合、**STORED AS** 後の形式名を変更します。 以下の例では、上記 **PARQUET** データに対応する外部テーブル **(tpch_lineitem_parquet)** の処理のみを中心に説明します。 異なるファイルタイプを処理する場合は、**DDL** の作成時に外部テーブルを作成する限り、**parquet/ORC/TEXTFILE/RCFILE/TEXTFILE** を指定するだけです。データを処理する文は同じです。

- *OSS* から直接オープンソースデータを読み取り、処理する

  関連付けるデータテーブルを作成した後は、次に示すように、通常の **MaxCompute** テーブルと同じ操作を直接実行できます。

  ```
  SELECT l_returnflag, l_linestatus,
  SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
  AVG(l_quantity) AS avg_qty,
  COUNT(*) AS count_order
  FROM tpch_lineitem_parquet
  WHERE l_shipdate <= '1998-09-02'
  ```

```
Group by l_returnflag, l_linestatus;
```

MaxCompute 内部計算エンジンが **OSS** から直接 **PARQUET** データを読み取り、処理する点を除き、外部テーブル **tpch_lineitem_parquet** は共通の内部テーブルとして使用されます。

`ROW FORMAT` + `STORED AS` が使用されているため、他のドキュメントで作成した関連テキストファイルの外部パーティションテーブルを使用する場合は、手動でフラグを設定する必要があります (デフォルトでは、**STORED AS** のみを使用し、**odps.sql.hive.compatible** は **FALSE** です)。その後、再度読み取ります。そうしなければ、エラーになります。

```
SELECT * FROM tpch_lineitem_textfile LIMIT 1;
Failed: Maid: User Defined Function exception-traceback:
com.aliyun.odps.udf.UDFException: java.lang.ClassNotFoundException:
com.aliyun.odps.hive.wrapper.HiveStorageHandlerWrapper
--You need to manually set up hive compatible flag.
set odps.sql.hive.compatible=true;
Select * from Maid limit 1;
+------------+------------+------------+--------------+------------
+-----------------+------------+------------+--------------
+--------------+------------+--------------+---------------
+----------------+------------+-----------+
| l_orderkey | l_partkey  | l_suppkey  | l_linenumber | l_quantity
 | l_extendedprice | l_discount | l_tax      | l_returnflag
 | l_linestatus | l_shipdate | l_commitdate | l_receiptdate |
l_shipinstruct | l_shipmode | l_comment |
+------------+------------+------------+--------------+------------
+-----------------+------------+------------+--------------
+--------------+------------+--------------+---------------
+----------------+------------+-----------+
| 5640000001 | 174458698  | 9458733    | 1            | 14.0
  | 23071.58       | 0.08       | 0.06       | N            | O
         | 1998-01-26 | 1997-11-16   | 1998-02-18    | TAKE BACK
RETURN | SHIP       | cuses nag silently. quick |
+------------+------------+------------+--------------+------------
+-----------------+------------+------------+--------------
+--------------+------------+--------------+---------------
+----------------+------------+-----------+
```

 注：

外部テーブルを直接使用し、データを読み取るたびに外部 **OSS** を含む **I/O** 操作が必要になります。また、**MaxCompute** システム自体は内部ストレージに対して多くの高パフォーマンス最適化を使用しないため、パフォーマンスが低下します。 したがって、データを繰り返し計算する必要があり、計算効率の影響を受けるシナリオの場合は、まずデータを **MaxCompute** にインポートしてから計算することを推奨します。

これらの複雑なデータ型は、**SQL** に関係しています (**CREATE**、**SELECT**、**INSERT** など)。 文 `set odps. sql. type. system. odps2 = true;` は **SQL** 文の前に追加し、

実行時に **SET** 文と **SQL** 文を一緒に送信し、実行します。 詳しくは、「データ型」をご参照ください。

- *OSS* から *MaxCompute* へオープンソースデータをインポートし、計算する

  まず、外部テーブルスキーマと同じ内部テーブル **tpch_lineitem_internal** を作成し、オープンソースデータを **OSS** から **MaxCompute** の内部テーブルにインポートして、**MaxCompute** の内部データ保存形式で格納します。

```
CREATE TABLE tpch_lineitem_internal LIKE tpch_lineitem_parquet;
INSERT OVERWRITE TABLE tpch_lineitem_internal;
SELECT * FROM tpch_lineitem_parquet;
```

  次に、同じ操作を直接内部テーブルに対して実行します。

```
SELECT l_returnflag, l_linestatus,
SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
AVG(l_quantity) AS avg_qty,
COUNT(*) AS count_order
FROM tpch_lineitem_internal
WHERE l_shipdate <= '1998-09-02'
GROUP BY l_returnflag, l_linestatus;
```

  このように、データを **MaxCompute** システムにインポートして保存することで、同じデータの計算処理がより効率的になります。

8.5 OSS にエクスポートされた非構造化データ

MaxCompute の非構造化フレームワークは、**Insert** 操作により **MaxCompute** データを直接 **OSS** に出力することができます。また、**MaxCompute** は **OSS** をデータ出力用の外部テーブルに関連付けることもできます。

「*OSS* 非構造化データへのアクセス」では、 **OSS** に保存されている外部テーブルの非構造化データの関連付けを介して **MaxCompute** にアクセスし、処理する方法を説明しています。

OSS への出力データは、通常 **2** つのケースがあります。

- **MaxCompute** 内部テーブルは、**OSS** に関連付けられた外部テーブルに出力されます。
- **MaxCompute** が外部テーブルを処理した後、結果は **OSS** に関連付けられた外部テーブルに直接出力されます。

OSS データへのアクセスと同様、**MaxCompute** は組み込み **storagehandler** とカスタム **storagehandler** を介した出力をサポートします。

組み込み StorageHandler を介した OSS への出力

MaxCompute の組み込み **StorageHandler** を使用することは、 規定の形式でデータを **OSS** に出力し、保存するのに非常に便利です。 必要な作業は、組み込み **StorageHandler** を示す外部テーブルを作成するだけです。これはこのテーブルに関連付けることができ、関連ロジックはシステムによって実装されます。

現在、**MaxCompute** は **2** つの組み込み **StorageHandler** をサポートしています。

- **com.aliyun.odps.CsvStorageHandler : CSV** 形式データの読み取りと書き込みの方法を定義します。データ形式の規則**:** 列の区切り文字はカンマ `,`、改行は `\n` です。
- **com.aliyun.odps.TsvStorageHandler : CSV** 形式データの読み取りと書き込みの方法を定義します。データ形式の規則**:** `\t`は列の区切り文字、改行は `\n` です。
- 外部テーブルの作成

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] <external_table>
(<column schemas>)
[PARTITIONED BY (partition column schemas)]
 STORED BY '<StorageHandler>'
[WITH SERDEPROPERTIES ( 'odps.properties.rolearn'='${roleran}') ]
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/';
```

  - **STORED By: OSS** にエクスポートする必要があるデータファイルが **TSV** ファイルの場合は、組み込み `com.aliyun.odps.TsvStorageHandler`。**OSS** にエクスポートする必要があるデータファイルが **CSV** ファイルの場合は、組み込み `com.aliyun.odps.CsvStorageHandler`。
  - **WITH SERDEPROPERTIES: OSS** 権限を **"STS** モードの権限付与**"** の **"**カスタム権限付与**"** に関連付けるときは、このパラメータに **odps.properties.rolearn** 属性を指定する必要があります。この値は、**RAM** で特に使用されるカスタムロールの情報です。

    
    注 **:**
    STS モードの権限付与については、「*OSS* の非構造化データへのアクセス」をご参照ください。
  - **Location: OSS** ストレージに対応するファイルへのパスを指定します。**odps.properties.rolearn** 属性が *`WITH SERDEPROPERTIES`* に設定されていない場合、平文の **AK** が権限付与に使用され、*`LOCATION`* は以下のようになります。

    ```
    LOCATION
    ```

```
    'oss://${accessKeyId}:${accessKeySecret}@${endpoint}/${bucket
}/${userPath}/'
```

- データは外部テーブルの **INSERT** 操作によって **OSS** に出力されます。



注：

insert-to-OSS の単一ファイルサイズは **5 g** を超えることはできません。

外部テーブルを介して **OSS** ストレージパスに関連付けられている場合は、外部テーブルに対して標準の **SQL INSERT OVERWRITE/INSERT INTO** 操作を実行して、両方のデータを **OSS** に出力できます。

```
INSERT OVERWRITE|INTO TABLE <external_tablename> [PARTITION (
partcol1=val1, partcol2=val2 ...)]
select_statement
FROM <from_tablename>
[WHERE where_condition];
```

- *from_tablename*: 内部テーブルにすることも、外部テーブル (関連付けられた **OSS** または **OTS** の外部テーブルを含む) にすることもできます。
- **INSERT** には、**OSS** に保存されている外部テーブルに従って **"StorageHandler"** の形式 (**TSV** または **CSV**) を指定します。

INSERT 操作が正常終了すると、**OSS** 上の対応する **LOCATION** によって一連のファイルが生成されることがわかります。

例: 外部テーブルに対応する場所は *oss://oss-cn-hangzhou-zmf.aliyuncs.com/oss-odps-test/tsv_output_folder/* で、**OSS** の対応するパスに一連のファイルが生成されているのがわかります。

```
osscmd ls oss://oss-odps-test/tsv_output_folder/
2017-01-14 06:48:27 39.00B Standard oss://oss-odps-test/tsv_output
_folder/.odps/.meta
2017-01-14 06:48:12 4.80MB Standard oss://oss-odps-test/tsv_output
_folder/.odps/20170113224724561g9m6csz7/M1_0_0-0.tsv
2017-01-14 06:48:05 4.78MB Standard oss://oss-odps-test/tsv_output
_folder/.odps/20170113224724561g9m6csz7/M1_1_0-0.tsv
2017-01-14 06:47:48 4.79MB Standard oss://oss-odps-test/tsv_output
_folder/.odps/20170113224724561g9m6csz7/M1_2_0-0.tsv
...
```

LOCATION で指定された **OSS** バケット **oss-odps-test** の下のフォルダー **tsv_output_folder** には、いくつかの **.tsv** ファイルと **.meta** ファイルを含む **.odps** フォル

ダーが含まれています。 同様のファイル構造は、**MaxCompute** から **OSS** への出力に固有のものです。

- **MaxCompute** を使用して外部テーブルに対して**INSERT INTO/OVERWRITE** を実行し、**OSS** アドレスに書き込むと、指定した **LOCATION** の **.odps** フォルダにすべてのデータが書き込まれます。
- **.odps** フォルダ内の **.meta** ファイルは、**MaxCompute** が現在のフォルダに有効なデータを記録するために書き込む、追加のマクロデータファイルです。 通常、**INSERT** 操作が成功した場合、現在のフォルダ内のすべてのデータは有効です。 マクロデータは、ジョブが失敗したときにのみ解析する必要があります。 操作が途中で失敗したり、強制終了された場合は、単純に **INSERT OVERWRITE** 文を再実行します。
- パーティションテーブルの場合、**fig** フォルダ下の **INSERT** 文で指定されたパーティション値に基づいて対応するパーティションサブディレクトリーが生成され、その中のパーティションサブディレクトリーは **".odps"** フォルダになります。 例: `test/tsv_output_folder/first-level partition name = partition value/.odps/20170113224724561g9m6csz7/M1_2_0-0.tsv`

MaxCompute によって組み込まれた **TSV/CSV storagehandler** 処理の場合、生成されるファイルの数は対応する **SQL** に対応しています。

`INSER OVERWITE ... SELECT ... FROM ... ;` 操作でソースデータテーブル **(from_tablename)** に **1000** 個のマッパーが割り当てられ、**1000** 個の **TSV/CSV** ファイルが生成されます。

カスタム storagehandler を介した OSS への出力

MaxCompute 非構造化フレームワークは、組み込み **StorageHandler** を使用して、**OSS** に **TSV/CSV** 共通テキスト形式を出力するだけでなく、カスタムデータ形式ファイルの外部出力をサポートする汎用 **SDK** を提供します。

組み込み **StorageHandler** と同様に、外部テーブルを作成し、 外部テーブルの **INSERT** 操作によってデータを **OSS** に出力する必要があります。 違いは、外部テーブルの作成時、カスタム **StorageHandler** を指定するために **STORED BY** が必要なことです。

 注:

MaxCompute 非構造化フレームワークは、**StorageHandler** と呼ばれるインターフェイスを介して、さまざまなデータストレージ形式の処理を記述します。 具体的には、**StorageHandler** は **Wrapper** クラスとして機能し、カスタムエクストラクターを指定することができます (データの読み取り、解析、処理などのため)。 **Outputer** (データ処理と

出力などのため) も同様です。 カスタム **StorageHandler** は **Odps StorageHandler** を継承し、**getExtractorClass** と **getOutputerClass** インターフェイスを実装する必要があります。

次に、*OSS データへのアクセス*例を使用します。 **OSS** 非構造化データの「カスタムエクストラクターで **OSS** にアクセス」では、**StorageHandler** をカスタマイズすることで、**MaxCompute** がどのように **OSS** テキストファイルにデータを出力するのかを説明し、列の区切り文字として **'|'** を、改行コードとして **'\ n'** を使用します。



注：

MaxCompute Studio を *MaxCompute Java module* に設定した後、例の中のサンプルコードを参照することができます。 または、完全なコードを参照するには、こちらをクリックします。

- *Outputer* の定義

  どちらの出力ロジックも、**outputer** のインターフェイスを実装する必要があります。

```
package com.aliyun.odps.examples.unstructured.text;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.io.OutputStreamSet;
import com.aliyun.odps.io.SinkOutputStream;
import com.aliyun.odps.udf.DataAttributes;
import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.Outputer;
import java.io.IOException;
public class TextOutputer extends Outputer {
    private SinkOutputStream outputStream;
    private DataAttributes attributes;
    private String delimiter;
    public TextOutputer (){
        // default delimiter, this can be overwritten if a delimiter
 is provided through the attributes.
        this.delimiter = "|";
    }
    @Override
    public void output(Record record) throws IOException {
        this.outputStream.write(recordToString(record).getBytes());
    }
    // no particular usage of execution context in this example
    @Override
    public void setup(ExecutionContext ctx, OutputStreamSet
outputStreamSet, DataAttributes attributes) throws IOException {
        this.outputStream = outputStreamSet.next();
        this.attributes = attributes;
        this.delimiter = this.attributes.getValueByKey("delimiter");
        if ( this.delimiter == null)
        {
            this.delimiter=",";
        }
        System.out.println("Extractor using delimiter [" + this.
delimiter + "].");
    }
    @Override
    public void close() {
        // no-op
    }
```

```
    private String recordToString(Record record){
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i &lt; record.getColumnCount(); i++)
        {
            if (null == record.get(i)){
                sb.append("NULL");
            }
            else{
                sb.append(record.get(i).toString());
            }
            if (i ! = record.getColumnCount() - 1){
                sb.append(this.delimiter);
            }
        }
        sb.append("\n");
        return sb.toString();
    }
}
```

すべての出力ロジックは **Outputer API** を呼び出す必要があります。 **3** つの **outputer API (setup**、**output**、**close)** があり、これらはすべて **3** つの **extractor API (setup**、**extract**、**close)** に対応しています。 **setup()** と**close()** は、**outputer** 内で一度だけ呼び出されます。 **setup** で初期化準備を行うことができます。 さらに、**setup()** によって返された **3** つのパラメーターは、**output()** または **close() API** で使用される **outputer** クラス変数として保存する必要があります。 インターフェイス **close ()** は、コードの終わりに使用されます。

通常、データ処理の大部分は **output (Record)** インターフェイスで行われます。 **MaxCompute** システムは、現在の **outputer** の割り当てによって処理された各入力レコードに基づいて、**output (Record)** を **1** 回呼び出します。 **output (Record)** の呼び出しが返された際、**Code** はすでに **Record** を消費しているとします。したがって、現在の **output (Record)** が戻された後、システムはそのレコードで使用されているメモリを使用します。複数の **output()** 関数が呼び出されると、現在のプロセスのレコードを呼び出す必要があります。現在のレコードを保存するには、**invoked.clone()** メソッドを呼び出します。

- エクストラクターの定義

エクストラクターはデータの読み取り、解析、処理などに使用されます。最終的に出力テーブルを **MaxCompute** などで読取る必要がなくなった場合、エクストラクターを定義する必要はありません。

```
package com.aliyun.odps.examples.unstructured.text;
import com.aliyun.odps.Column;
import com.aliyun.odps.data.ArrayRecord;
import com.aliyun.odps.data.Record;
import com.aliyun.odps.io.InputStreamSet;
import com.aliyun.odps.udf.DataAttributes;
import com.aliyun.odps.udf.ExecutionContext;
import com.aliyun.odps.udf.Extractor;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
```

```
/**
 * Text extractor that extract schematized records from formatted
plain-text(csv, tsv etc.)
 **/
public class TextExtractor extends Extractor {
    private InputStreamSet inputs;
    private String columnDelimiter;
    private DataAttributes attributes;
    private BufferedReader currentReader;
    private boolean firstRead = true;
    public TextExtractor() {
        // default to ",", this can be overwritten if a specific
delimiter is provided (via DataAttributes)
        this.columnDelimiter = ",";
    }
    // no particular usage for execution context in this example
    @Override
    public void setup(ExecutionContext ctx, InputStreamSet inputs,
DataAttributes attributes) {
        this.inputs = inputs;
        this.attributes = attributes;
        // check if "delimiter" attribute is supplied via SQL query
        String columnDelimiter = this.attributes.getValueByKey("
delimiter");
        if ( columnDelimiter ! = null)
        {
            this.columnDelimiter = columnDelimiter;
        }
        System.out.println("TextExtractor using delimiter [" + this.
columnDelimiter + "].");
        // note: more properties can be inited from attributes if
needed
    }
    @Override
    public Record extract() throws IOException {
        String line = readNextLine();
        if (line == null) {
            return null;
        }
        return textLineToRecord(line);
    }
    @Override
    public void close(){
        // no-op
    }
    private Record textLineToRecord(String line) throws IllegalArg
umentException
    {
        Column[] outputColumns = this.attributes.getRecordColumns();
        ArrayRecord record = new ArrayRecord(outputColumns);
        if (this.attributes.getRecordColumns().length ! = 0 ){
            // string copies are needed, not the most efficient one
, but suffice as an example here
            String[] parts = line.split(columnDelimiter);
            int[] outputIndexes = this.attributes.getNeededIndexes
();
            if (outputIndexes == null){
                throw new IllegalArgumentException("No outputIndexes
 supplied.");
            }
            if (outputIndexes.length ! = outputColumns.length){
                throw new IllegalArgumentException("Mismatched
output schema: Expecting "
```

```
                        + outputColumns.length + " columns but get "
 + parts.length);
            }
            int index = 0;
            for(int i = 0; i &lt; parts.length; i++){
                // only parse data in columns indexed by output
indexes
                if (index &lt; outputIndexes.length && i ==
outputIndexes[index]){
                    switch (outputColumns[index].getType()) {
                        case STRING:
                            record.setString(index, parts[i]);
                            break;
                        case BIGINT:
                            record.setBigint(index, Long.parseLong(
parts[i]));
                            break;
                        case BOOLEAN:
                            record.setBoolean(index, Boolean.
parseBoolean(parts[i]));
                            break;
                        case DOUBLE:
                            record.setDouble(index, Double.
parseDouble(parts[i]));
                            break;
                        case DATETIME:
                        case DECIMAL:
                        case ARRAY:
                        case MAP:
                        Default:
                            throw new IllegalArgumentException("Type
 " + outputColumns[index].getType() + " not supported for now.");
                    }
                    index++;
                }
            }
        }
        return record;
    }
    /**
     * Read next line from underlying input streams.
     * @return The next line as String object. If all of the
contents of input
     * streams has been read, return null.
     */
    private String readNextLine() throws IOException {
        if (firstRead) {
            firstRead = false;
            // the first read, initialize things
            currentReader = moveToNextStream();
            if (currentReader == null) {
                // empty input stream set
                return null;
            }
        }
        while (currentReader ! = null) {
            String line = currentReader.readLine();
            if (line ! = null) {
                return line;
            }
            currentReader = moveToNextStream();
        }
        return null;
    }
```

```
    private BufferedReader moveToNextStream() throws IOException {
        InputStream stream = inputs.next();
        if (stream == null) {
            return null;
        } else {
            return new BufferedReader(new InputStreamReader(stream
));
        }
    }
}
```

詳細については、「*OSS* 非構造化データへのアクセス」をご参照ください。

- *StorageHandler* の 定義

```
package com.aliyun.odps.examples.unstructured.text;
import com.aliyun.odps.udf.Extractor;
import com.aliyun.odps.udf.OdpsStorageHandler;
import com.aliyun.odps.udf.Outputer;
public class TextStorageHandler extends OdpsStorageHandler {
    @Override
    public Class&lt;? extends Extractor&gt; getExtractorClass() {
        return TextExtractor.class;
    }
    @Override
    public Class&lt;? extends Outputer> getOutputerClass() {
        return TextOutputer.class;
    }
}
```

テーブルを読み取る必要がない場合は、エクストラクターのインターフェイスを指定する必要はありません。

- コンパイルとパッケージ

カスタムコードをパッケージにコンパイルして **MaxCompute** にアップロードします。**jar** パッケージの名前が **"odps-TextStorageHandler.jar"** の場合は、**MaxCompute** にアップロードします。

```
add jar odps-TextStorageHandler.jar;
```

- 外部テーブルの作成

組み込み **StorageHandler** を使用する場合と同様に、外部テーブルを作成する必要がありますが、今回はカスタム **StorageHandler** を使用してデータを外部テーブルに出力するように指定する必要がある点が異なります。

```
CREATE EXTERNAL TABLE IF NOT EXISTS output_data_txt_external
(
vehicleId int,
recordId int,
patientId int,
calls int,
locationLatitute double,
locationLongtitue double,
recordTime string,
```

```
direction string
)
STORED BY 'com.aliyun.odps.examples.unstructured.text.TextStorag
eHandler'
WITH SERDEPROPERTIES(
    'delimiter'='|'
    [,'odps.properties.rolearn'='${roleran}'])
LOCATION 'oss://${endpoint}/${bucket}/${userfilePath}/'
USING 'odps-TextStorageHandler.jar';
```



注：

`odps.properties.rolearn` プロパティが必要な場合、詳細については、「*OSS* 非構造化データへのアクセス」の **STS** モードの権限付与の「カスタム権限付与」をご参照ください。このプロパティが不要な場合は、「ワンクリック権限付与」を参照するか、**location** の先頭に平文の **AK** を使用します。

- *INSERT* を用いた非構造化ファイルの外部テーブルへの書き込み

  storagehandler をカスタマイズして **OSS** ストレージパスに外部テーブルの関連付けを作成した後で、組み込み **storagehandler** と同じ方法で、外部テーブルに対して標準 **SQL insert override/insert into** を実行して両方のデータを **OSS** に出力できます。

```
INSERT OVERWRITE|INTO TABLE &lt;external_tablename&gt; [PARTITION (
partcol1=val1, partcol2=val2 ...)]
Select_statement
FROM &lt;from_tablename&gt;
[WHERE where_condition];
```

  insert 操作が成功すると、組み込み **StorageHandler** として、**OSS** に対応する **LOCATION** パス *.odps* フォルダで一連のファイルが生成されます。

8.6 Table Store データへのアクセス

このドキュメントでは、**Table Store** から **MaxCompute** コンピューティング環境にデータをインポートする方法を説明します。 これにより、複数のデータソース間のシームレスな接続が可能になります。

Table Store は、**Alibaba Cloud** の **Apsara** 分散ファイルシステム上に構築された **NoSQL** データベースサービスです。ユーザーは、膨大な量の構造化データをリアルタイムで保存し、アクセスすることができます。 **Table Store** の詳細については、「*Table Store* の概要」をご参照ください。

MaxCompute と **TableStore** は、**2** つの独立したビッグデータコンピューティングサービスとストレージサービスです。 したがって、これら **2** つのサービスの間のネットワークが有効であることを確認する必要があります。 **MaxCompute** のパブリッククラウドサービスが **Table Store**

に保存されているデータにアクセスする場合、**Table Store** のプライベートネットワークアドレスを使用することを推奨します。通常、ホスト名の末尾に **'ots-internal.aliyuncs.com'** を付けたものです。例: `tablestore://odps-ots-dev.cn-shanghai.ots-internal.aliyuncs.com`

他のドキュメントで、*OSS* 非構造化データへのアクセスの使い方を説明しました。

Table Store と **MaxCompute** はどちらも独自の型システムを持っています。 **Table Store** と **MaxCompute** はどちらも独自のデータ型システムを持っています。 **MaxCompute** で **Table Store** データを処理する場合、データ型の関連付けは次のとおりです。

MaxCompute 型	TableStore 型
STRING	STRING
BIGINT	INTEGER
DOUBLE	Double
BOOLEAN	BOOLEAN
BINARY	BINARY

STS モードでの権限付与

MaxCompute から **Table Store** データにアクセスするには、安全な権限付与手段が必要です。 この問題に関して、**MaxCompute** は **Alibaba Cloud Resource Access Management (RAM)** と **Token Service (STS)** を統合して安全なデータアクセスを実装します。

次の 2 つの方法で権限を付与できます。

- **MaxCompute** と **Table Store** の所有者が同じアカウントの場合は、**Alibaba Cloud** アカウントで直接ログインし、ここをクリックして権限を付与します。
- カスタム権限付与

  1. まず、**RAM** コンソールで、**Table Store** アクセス権限を **MaxCompute** に付与する必要があります。

     RAM コンソールにログイン (**MaxCompute** と **Table Store** が同じアカウントでない場合は、**Table Store** アカウントでログインして権限を付与する必要があります) し、ロール **AliyunODPSDefaultRole** を作成します。
  2. ポリシーの内容を次のように設定します。

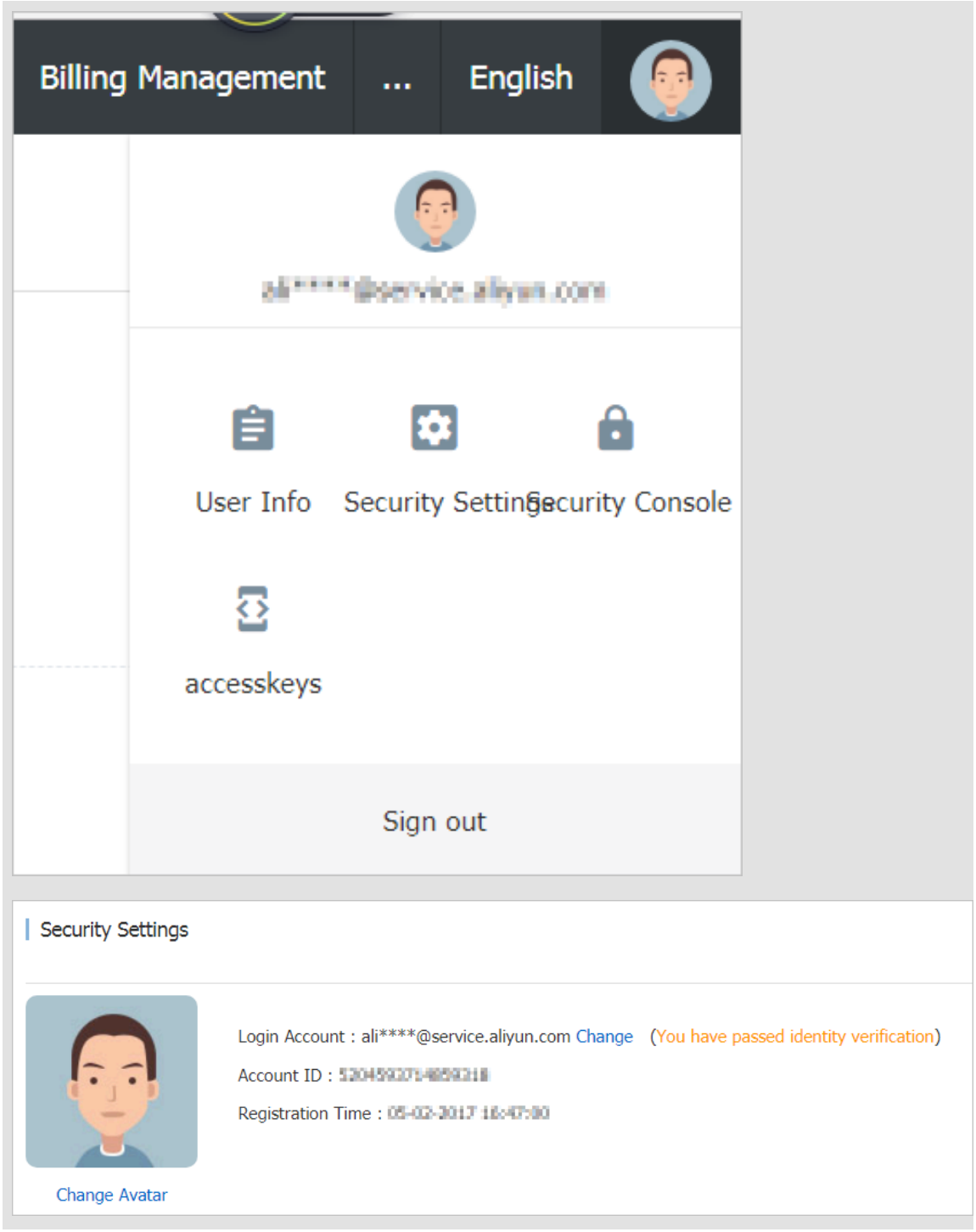
```
--if MaxCompute and Table Store are same account
{
"Statement": [
{
 "Action": "sts:AssumeRole",
```

```
 "Effect": "Allow",
 "Principal": {
   "Service": [
     "odps.aliyuncs.com"
   ]
 }
}
],
"Version": "1"
}
--if MaxCompute and Table Store are not the same account
{
"Statement ":[
{
 "Action": "sts:AssumeRole",
 "Effect": "Allow",
 "Principal": {
   "Service": [
     "MaxCompute's Owner cloud account UID@odps.aliyuncs.com"
   ]
 }
}
],
"Version": "1"
}
```



注：

右上隅にあるアバターをクリックして、[課金管理] ページを開き、アカウント **UID** を確認します。



3. このロールの認証ポリシー **AliyunODPSRolePolicy** を編集します。

```
{
"Version": "1",
"Statement ":[
{
 "Action ":[
   "ots:ListTable",
   "ots:DescribeTable",
   "ots:GetRow",
   "ots:PutRow",
   "ots:UpdateRow",
   "ots:DeleteRow",
   "ots:GetRange",
```

```
    "ots:BatchGetRow",
    "ots:BatchWriteRow",
    "ots:ComputeSplitPointsBySize"
  ],
  "Resource": "*",
  "Effect": "Allow"
}
]
}
--You can also customize other permissions
```

4. このロールに **AliyunODPSRolePolicy** 権限を付与します。

外部テーブルの作成

MaxCompute では、外部テーブルを作成し、**Table Store** テーブルデータ記述を **MaxCompute** メタシステムに追加した後、**Table Store** データを処理できるようになります。次の例は、**MaxCompute** の **Table Store** アクセスで使用される概念と操作を示しています。

次の文を使用して、外部テーブルを作成します。

```
DROP TABLE IF EXISTS ots_table_external;
CREATE EXTERNAL TABLE IF NOT EXISTS ots_table_external
(
odps_orderkey bigint,
odps_orderdate string,
odps_custkey bigint,
odps_orderstatus string,
odps_totalprice double
)
STORED BY 'com.aliyun.odps.TableStoreStorageHandler' -- (1)
WITH SERDEPROPERTIES ( -- (2)
'tablestore.columns.mapping'=':o_orderkey,:o_orderdate,o_custkey,
o_orderstatus,o_totalprice', -- ①
'tablestore.table.name'='ots_tpch_orders' -- ②
'odps.properties.rolearn'='acs:ram::xxxxx:role/aliyunodpsdefaultrole'
 --③
)
LOCATION 'tablestore://odps-ots-dev.cn-shanghai.ots-internal.aliyuncs.
com'; --  (3)
```

文の説明を以下に示します。

- com.aliyun.odps.TableStoreStorageHandler は、**Table Store** データを処理するための **MaxCompute** の組み込み **StorageHandler** です。 これは、**MaxCompute** と **Table Store** 間の相互作用を定義します。 関連ロジックは **MaxCompute** によって実装されています。
- SERDEPROPERITES は、パラメーターオプションを提供するインターフェイスです。TableStoreStorageHandler を使用する際、*tablestore.columns.mapping* と

tablestore.table.name と *odps.properties.rolearn* から 2 つオプションを指定する必要があります。

1. *filepathtablestore.columns.mapping optionfilepath*: MaxCompute がアクセスしようとしている Table Store テーブルの列を照会するのに必要です。
   - 列名の先頭の `:` は、Table Store の主キーを示します。 この例では、`:o_orderkey` と `:o_orderdate` が主キー列で、それ以外はすべて属性列です。
   - Table Store は最大 4 つの主キーをサポートします。 主キーは、STRING、INTEGER、および BINARY データ型をサポートします。 最初の主キーはパーティションキーです。
   - マッピング関係を指定するとき、指定した Table Store テーブルのすべての主キーを指定する必要がありますが、すべての属性列を指定する必要はなく、MaxCompute を使用してアクセスする必要がある属性列のみを指定します。
2. *filepathtablestore.table.namefilepath*: アクセスする必要がある Table Store テーブルの名前。 誤った Table Store テーブル名 (存在しないテーブルなど) を指定した場合、システムはエラーを報告します。 MaxCompute は、指定された名前で新しい Table Store テーブルを作成しません。
3. *odps.properties.rolearn*: RAM の AliyunODPSDefaultRole にある Arn 情報。 RAM コンソールの [ロールの詳細] から取得できます。

・ LOCATION 句: Table Store インスタンス名、エンドポイントなどを指定するための特定の情報。 ここでの Table Store データへの安全なアクセスは、前述の RAM/STS 権限付与の前提に基づいています。

作成した外部テーブル構造を表示する場合、次の文を実行します。

```
desc extended <table_name>;
```

返された情報の "Extended Info" には、StorageHandler や Location などの外部テーブル情報が含まれています。

外部テーブルを使用したテーブルデータへのアクセス

外部テーブルを作成した後、Table Store データを MaxCompute エコシステムに挿入できます。MaxCompute SQL 構文を使用して、次のように Table Store データにアクセスできます。

```
SELECT odps_orderkey, odps_orderdate, SUM(odps_totalprice) AS
sum_total
FROM ots_table_external
WHERE odps_orderkey > 5000 AND odps_orderkey < 7000 AND odps_orderdate
 >= '1996-05-03' AND odps_orderdate < '1997-05-01'
```

```
GROUP BY odps_orderkey, odps_orderdate
HAVING sum_total> 400000.0;
```

MaxCompute SQL 構文を使用する場合、アクセスした**Table Store** の詳細はすべて **MaxCompute** で処理されます。 これには、列名の選択も含まれます。 たとえば、前述の **SQL** 文で使用されている列名 **(odps_orderkey** や **odps_totalprice** など) は、**Table Store** で使用されている元の主キー名 **(o_orderkey)** や属性列名 **(o_totalprice)** ではありません。 これは、外部テーブルの作成時に使用された **DDL** 文で、マッピングが既に実行されているためです。 外部テーブルを作成するときに元の **Table Store** の主キー/列名を保持できます。

1 つのデータセットに対して複数の計算を実行する場合、毎回 **Table Store** からデータをリモートで読み取るのではなく、必要なデータをすべて **MaxCompute** にインポートして、**MaxCompute** (内部) テーブルを作成することができます。 例**:**

```
CREATE TABLE internal_orders AS
SELECT odps_orderkey, odps_orderdate, odps_custkey, odps_totalprice
FROM ots_table_external
Where fig> 5000;
```

internal_orders は、**MaxCompute** 内部テーブルのすべての機能を備えた **MaxCompute** テーブルです **(**効率的に圧縮された列ストレージデータ形式、完全な内部マクロデータ、統計情報など**)**。 また、データは **MaxCompute** に保存されるため、外部の **Table Store** データにアクセスする場合よりもアクセス速度が速くなります。 これは、頻繁に計算されるホットスポットデータに特に適しています。

MaxCompute データの Table Store へのエクスポート

 注**:**

MaxCompute は外部 **Table Store** テーブルを直接作成しません。したがって、**Table Store** テーブルにデータを出力する前に、このテーブルが既に作成されていることを確認する必要があります **(**作成されていない場合、システムがエラーを報告します**)**。

上記の操作では、**MaxCompute** と **Table Store** テーブル **ots_tpch_orders** を接続するための外部テーブル **ots_table_external** が作成され、内部 **MaxCompute** テーブル **internal_orders** にデータを保存しました。 これで、処理したデータを **internal_orders** から **Table Store** に書き込むことができます。外部テーブルに対して、以下の `INSERT OVERWITE TABLE` 操作を実行します。

```
INSERT OVERWRITE TABLE ots_table_external
SELECT odps_orderkey, odps_orderdate, odps_custkey, CONCAT(odps_custk
ey, 'SHIPPED'), CEIL(odps_totalprice)
```

```
FROM internal_orders;
```

 注：

ODPS テーブル自体のデータに特定の順序がある場合 (主キーに従って**1** 度ソートする、など)、**OTS** テーブルへの書込み時に **OTS** パーティションに負荷が集中します。これは、分散書き込みの特性をフルに活用できないことを意味します。 したがって、このような場合は、まず distribute by Rand () を介してデータを分散させることを推奨します。

```
INSERT OVERWRITE TABLE ots_table_external
SELECT odps_orderkey, odps_orderdate, odps_custkey, CONCAT(odps_custk
ey, 'SHIPPED'), CEIL(odps_totalprice)
FROM (SELECT * FROM internal_orders DISTRIBUTE BY rand()) t;
```

Table Store は **KV** データの **NoSQL** ストレージメディアであるので、**MaxCompute** からのデータ出力は対応する主キーを持つ行にのみ影響します。 この例では、出力は対応する **dps_orderkey** + **odps_orderdate** 主キー値を持つ行内のデータにのみ影響します。 また、**Table Store** 行では、外部テーブル **(ots_table_external)** の作成中に指定された属性列のみが更新されます。 外部テーブルに表示されないデータ列は変更されません。

 注：

- **MaxCompute** 内のデータを一度に **4 MB** を超えて **OTS** に書き込むことはできません。それ以外の場合は、サイズの大きすぎるデータを削除して書き戻す必要があります。 この時点でエラーが発生する可能性があります。

  ```
  ODPS-0010000:System internal error - Output to TableStore failed
  with exception:
  TableStore BatchWrite request id XXXXX failed with error code
  OTSParameterInvalid and message:The total data size of BatchWrite
  Row request exceeds the limit
  ```

- データを一括で、またはブランチ単位で書き込むのは、**1** 回の操作です。 詳細は、「*BatchWriteRow*」をご参照ください。 したがって、一括書き込みデータの量が大きすぎる場合は、ブランチ書き込みも可能です。
- データを一括で書き込む場合、重複する行がないことに注意してください。そうしないと、エラーが報告される可能性があります。

  ```
  Errorcode: FIG, errormessage: the input parameter is invalid
  ```

  詳細については、「*BatchWriteRow* を使用して一度に *100* 個のデータを送信すると、*OTSParameterInvalid* エラーが報告される」をご参照ください。

9 Lightning

9.1 Lightning の概要

MaxCompute Lightning は **MaxCompute** が提供する対話型のクエリサービスです。**PostgreSQL** のプロトコルと構文に基づいて **MaxCompute** プロジェクトへ容易に接続できます。 このサービスを使用すると、標準 **SQL** やよく使われているツールを使用して、**MaxCompute** プロジェクト データを迅速にクエリおよび分析することができます。

Tablueu や **FineReport** などの主要 **BI** ツールを使用して、**MaxCompute** プロジェクトに簡単に接続し、**BI** 分析やアドホッククエリを実行できます。 **MaxCompute Lightning** のクイッククエリ機能を使用すると、プロジェクトテーブルデータを **API** にカプセル化することでサービスを提供し、データの移行なしにさまざまなアプリケーションシナリオをサポートできます。

MaxCompute Lightning は、サーバーレスコンピューティングサービスを提供します。 インフラストラクチャは必要なく、料金はクエリに対してのみ発生します。

主な機能

- **PostgreSQL** プロトコル互換

  MaxCompute Lightning では、**PostgreSQL** プロトコル互換の **JDBC (Java Database Connectivity)** インターフェイスや、**ODBC (Open Database Connectivity)** インタフェイスが提供されます。 **PostgreSQL** データベースをベースにしたツールやアプリケーションは、デフォルトのドライバを使用して **MaxCompute** プロジェクトに簡単に接続できます。簡単な接続により、**MaxCompute** プロジェクトデータの分析にさまざまな **PostgreSQL** ツールが使用できます。
- 性能の向上

  MaxCompute テーブルのクイッククエリは、小規模なデータセットやクエリの同時実行性が高い場合に特に最適化されており、通常のレポートやサービス **API** など、さまざまなアプリケーションシナリオをサポートします。
- 統合された権限管理

  MaxCompute Lightning は、**MaxCompute** プロジェクト用に設計されたプロダクトで、**MaxCompute** プロジェクトへのアクセスを提供します。 このサービスは、**MaxCompute** プロジェクトと同じアクセス制御システムを共有します。 これにより、ユーザーはアクセスが許可されているデータのみをクエリできます。

・ すぐに使える機能と、クエリに基づく料金

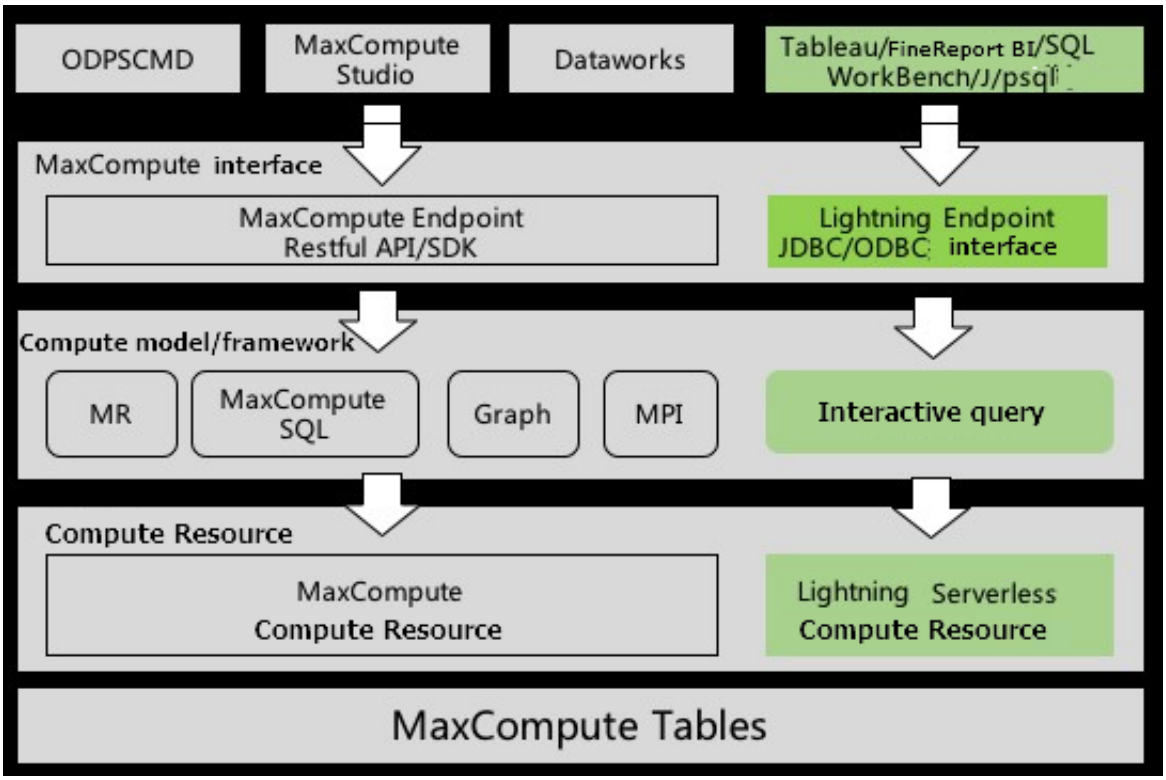
MaxCompute Lightning では、既存の **MaxCompute** コンピューティングリソースに基づいて、サーバーレスコンピューティングサービスが提供されます。 クエリを実行するには、**MaxCompute Lightning** を使用して **MaxCompute** プロジェクトへの接続を確立するだけです。

MaxCompute Lightning のリソースを設定、管理、保守する必要はありません。**MaxCompute Lightning** を使用すると、各クエリで処理されるデータ量に対してのみコストが発生します。

システムアーキテクチャ

MaxCompute Lightning では、**PostgreSQL** ドライバを使用して、エンドポイント、クライアント、またはアプリケーションを **JDBC** や **ODBC** インターフェイスに接続する方法を提供します。 これにより、**MaxCompute** プロジェクトの統合アクセス制御システム内で安全なデータアクセスが可能になります。

JDBC または **ODBC** インターフェイスで接続、送信されるクエリタスクは、**MaxCompute Lightning** のサーバーレスコンピューティングリソースを使用してクエリサービスの品質を保証します。



シナリオ

- アドホッククエリ

  小さなデータセット (100 GB 未満) のクエリは、MaxCompute テーブルを低レイテンシで簡単にクエリできるように最適化されています。 MaxCompute データを ADS (AnalyticDB)、RDS (Relational Database Service) や、その他のシステムにインポートする必要がないため、必要なリソースと管理コストが削減されます。

  このシナリオには、クエリ用の柔軟なデータオブジェクト、複雑なロジック、迅速なクエリ、クエリロジックの簡単な調整、および 1 分以内の低レイテンシクエリ要件などの特徴があります。 ユーザーは多くの場合、SQL スキルを習得し、クエリ分析に使い慣れたクライアントツールを使用したいデータアナリストです。

- レポートと分析

  分析レポートは、ETL (Extract-Transform-Load) プロセスで統合された MaxCompute プロジェクトデータに基づいて生成されます。 レポートは、定期的なチェックのためマネージャーとビジネスユーザーに提供されます。

  このシナリオには、クエリ対象のデータオブジェクトは通常集約されたデータである、という特長があります。 クエリ対象のデータオブジェクトは、小さいデータセットに含まれます。 クエリは、固定された簡単なクエリロジックに基づいています。 このシナリオでは低レイテンシという要件があります。 ほとんどのクエリでは、レイテンシは 5 秒以内です。 クエリのレイテンシ時間は、データ量とクエリの複雑さによって大きく異なります。

- オンラインアプリケーション

  MaxCompute プロジェクトデータは、オンラインアプリケーションをサポートするために RESTful API にカプセル化できます。

  このシナリオでは、MaxCompute Lightning は高速クエリエンジンとして機能し、MaxCompute テーブルデータが 手作業による介入を最小限とした API サービスとして提供されます。 これは Alibaba Cloud DataWorks のデータサービスコンポーネントを統合することによって可能になります。

9.2 制約と制限

ここでは、**MaxCompute Lightning** サービスを使用する際の制約と制限について説明します。

DDL/DML の制約と制限

MaxCompute Lightning は、**MaxCompute** テーブルに対する **Select** クエリのみをサポートし、**MaxCompute** テーブルに対する **UPDATE**、**CREATE**、**DELETE**、および **INSERT** 操作はサポートしません。

クエリの制約と制限

- パーティションテーブルをクエリするときに、最大 **1,024** のスキャンパーティションをクエリできます。
- 現在、ビューを作成または使用することはできません。
- 現在、精度データ型を持つ **MAP**、**ARRAY**、**TINYINT**、**BINARY**、**TIMESTAMP**、および **DECIMAL** はサポートされていません。
- 各クエリで、**1** つのテーブルに対して最大 **1 TB** のデータをスキャンできます。
- 送信されるクエリ文のサイズは **100 KB** を超えることはできません。
- クエリタイムアウト期間は **1** 時間です。

UDF の制約と制限

- **MaxCompute** を使用して作成されたユーザー定義関数 **(UDF)** は、**MaxCompute Lightning** では使用できません。
- *PostgreSQL* のユーザー定義関数を、**MaxCompute Lightning** で作成または使用することはできません。
- **MaxCompute** の組み込み関数は、現時点でサポートされていません。

クエリ同時実行性の制約

MaxCompute Lightning では、**MaxCompute** プロジェクトに対して最大 **20** の同時実行クエリがサポートされています。

9.3 クイックスタート

9.3.1 ガイドの説明

このトピックでは、特定の **MaxCompute** プロジェクトのテーブルを表示する方法や **BI** 分析を実行する方法など、主要なサードパーティツールを使用して **MaxCompute Lightning** サービスにアクセスする方法について説明します。

9.3.2 前提条件

MaxCompute の有効化とプロジェクトの作成

MaxCompute Lightning を使用するには、**MaxCompute** が有効化され、プロジェクトが作成されている必要があります。

MaxCompute を有効化していない場合、最初にサービスを有効化します。 詳しくは、「*MaxCompute の有効化*」をご参照ください。 次に、**MaxCompute** プロジェクトを作成します。

テーブルの作成とデータのインポート

プロジェクトにテーブルが作成され、データが読み込まれます。 詳しくは、「*MaxCompute クイックスタート*」をご参照ください。

アカウント情報の取得

MaxCompute プロジェクトのアクセス **ID** とアクセスキーを取得済みである必要があります。

Alibaba Cloud Web サイトにログインし、コンソールをクリックすると、**AccessKey** ページが表示されます。 **RAM** ユーザーに **AccessKey** を表示する権限が付与されていない場合は、プライマリアカウントの所有者に連絡してください。 **RAM** ユーザーにプロジェクトテーブルを表示する権限が付与されている必要があります。

9.3.3 接続用クライアントツールの準備

MaxCompute Lightning は **PostgreSQL** インターフェイスと互換性があり、**PostgreSQL** データベース接続用のクライアントツールを利用できます。

このチュートリアルでは、*Tableau Desktop* **BI** ツールを使用します。 **Tableau** 公式 **Web** サイトから関連ツールをダウンロードしてください。

SQL Workbench/J、**PSQL**、**FineReport BI**、**MicroStrategy BI** ツールなど、他の一般的に使用されるクライアントツールは、**PostgreSQL** データベースと同じ方法で **MaxCompute Lightning** に接続できます。

9.3.4 サービスへのアクセスと分析の実行

サービスへの接続が成功した後、特定の **MaxCompute** プロジェクトのデータテーブルを表示し、**BI** 分析を行うことができます。

1. サーバーへの接続を確立するときに、**PostgreSQL** をクリックします。

   Tableau Desktop を起動します。 左側のナビゲーションウィンドウで、**[Connection]** > **[To Servers]** > **[More]** > **[PostgreSQL]** の順にクリックします。

2. サービス接続とユーザー認証の情報を入力します。

パラメーター	説明
Server	特定のリージョンの **MaxCompute Lightning** エンドポイントを **[Server]** フィールドに入力します。 たとえば、中国 (上海) リージョンのエンドポイントとして、`lightning.cn-shanghai.maxcompute.aliyun.com` を入力します。
Port	**443**
Database	**MaxCompute** プロジェクト名
ID Verification	ユーザー名とパスワード
Username/ Password	ユーザーの **Access Key ID** または **Access Key Secret**
SSL connection	**[SSL]** 接続チェックボックスを選択します。

3. プロジェクトテーブル情報を取得してデータソースまたはモデルを作成します。

   連絡先情報を設定して **Tableau Desktop** にログインすると、接続された **MaxCompute** プロジェクトのテーブルが読み込まれます。 必要に応じてテーブルを選択し、データモデルとチャートを作成できます。

   次の図は、必要なディメンションとメジャーに基づいて作成されたチャートの例を示しています。

これで、**Tableau Desktop** を使用して **MaxCompute Lightning** にアクセスできるようになります。 接続された **MaxCompute** プロジェクトのデータに対して **BI** 分析を実行できます。

 注:

パフォーマンスを向上させるため、**Tableau** でサポートされている **TDC** ファイルを使用して **Lightning** データソースへの接続をカスタマイズすることを推奨します。 詳細は、「*Tableau Desktop*」をご参照ください。

9.4 アクセスドメイン名

MaxCompute Lightning は、対応するリージョンの **MaxCompute Lightning** サービスにアクセスできるリージョン固有のエンドポイントを提供します。

次の表は、各リージョンとパブリッククラウドネットワーク環境における **MaxCompute Lightning** サービスの接続ステータスを示しています。

表 9-1 : 各リージョンでのサービス接続ステータス (外部ネットワーク)

リージョン	サービスステータス	外部ネットワークのエンドポイント
中国 (杭州)	ベータ	**lightning.cn-hangzhou.maxcompute.aliyun.com**
中国 (上海)	ベータ	**lightning.cn-shanghai.maxcompute.aliyun.com**
中国 (北京)	ベータ	**lightning.cn-beijing.maxcompute.aliyun.com**
中国 (香港)	ベータ	**lightning.cn-hongkong.maxcompute.aliyun.com**
シンガポール	ベータ	**lightning.ap-southeast-1.maxcompute.aliyun.com**
他のリージョン	未稼働	-

表 9-2 : 各リージョンでのサービス接続ステータス (クラシックネットワーク)

リージョン	サービスステータス	クラシックネットワークのエンドポイント
中国 (杭州)	ベータ	**lightning.cn-hangzhou.maxcompute.aliyun-inc.com**
中国 (上海)	ベータ	**lightning.cn-shanghai.maxcompute.aliyun-inc.com**
中国 (北京)	ベータ	**lightning.cn-beijing.maxcompute.aliyun-inc.com**
中国 (香港)	ベータ	**lightning.cn-hongkong.maxcompute.aliyun-inc.com**
シンガポール	ベータ	**lightning.ap-southeast-1.maxcompute.aliyun-inc.com**
他のリージョン	未稼働	-

表 9-3 : 各リージョンでのサービス接続ステータス (VPC ネットワーク)

リージョン	サービスステータス	**VPC** のエンドポイント
中国 (杭州)	ベータ	**lightning.cn-hangzhou.maxcompute.aliyun-inc.com**
中国 (上海)	ベータ	**lightning.cn-shanghai.maxcompute.aliyun-inc.com**

リージョン	サービスステータス	VPC のエンドポイント
中国 (上海)		**lightning.cn-shanghai-finance.maxcompute.aliyun-inc.com**
中国 (北京)	ベータ	**lightning.cn-beijing.maxcompute.aliyun-inc.com**
シンガポール	ベータ	**lightning.ap-southeast-1.maxcompute.aliyun-inc.com**
他のリージョン	未稼働	-

9.5 JDBC インターフェイスを使用したアクセスサービス

MaxCompute Lightning クエリエンジンは **PostgreSQL 8.2** に基づいており、現在は既存の **MaxCompute** テーブルに対する **SELECT** クエリのみをサポートしています。 詳細は、「*Queries*」と「*Functions and Operators*」をご参照ください。

MaxCompute プロジェクトにデータが追加されていない場合、または既存のデータを処理する必要がある場合、*MaxCompute* ヘルプドキュメントをご参照ください。 *MaxCompute* クライアント、または *DataWorks* を使用して、データオブジェクトを作成および処理するための **MaxCompute** プロジェクトにアクセスできます。

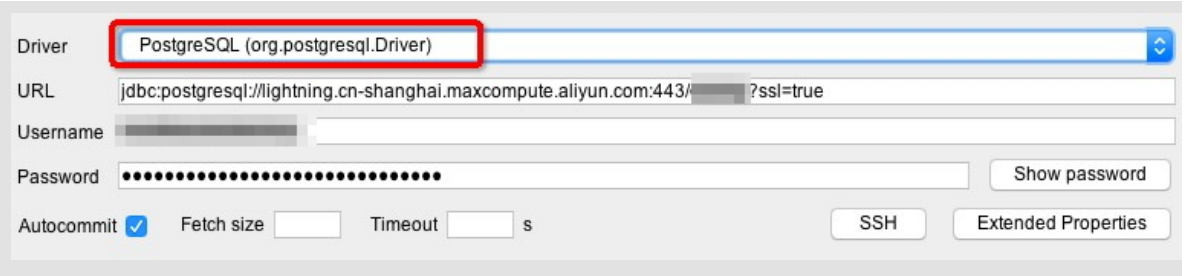
9.5.1 JDBC ドライバー

MaxCompute は **PostgreSQL** プロトコルと完全互換の **JDBC** インターフェイスを提供します。 **JDBC** インターフェイスを使用して **SQL** クライアントツールを **MaxCompute Lightning** サービスに接続できます。

PostgreSQL の公式 **Web** サイトの **JDBC**ドライバー、または **MaxCompute Lightning** 用に最適化された他のドライバーを使用して、**MaxCompute Lightning** にアクセスできます。

1. **PostgreSQL** 公式 **Web** サイトから *JDBC* ドライバーをダウンロードできます。

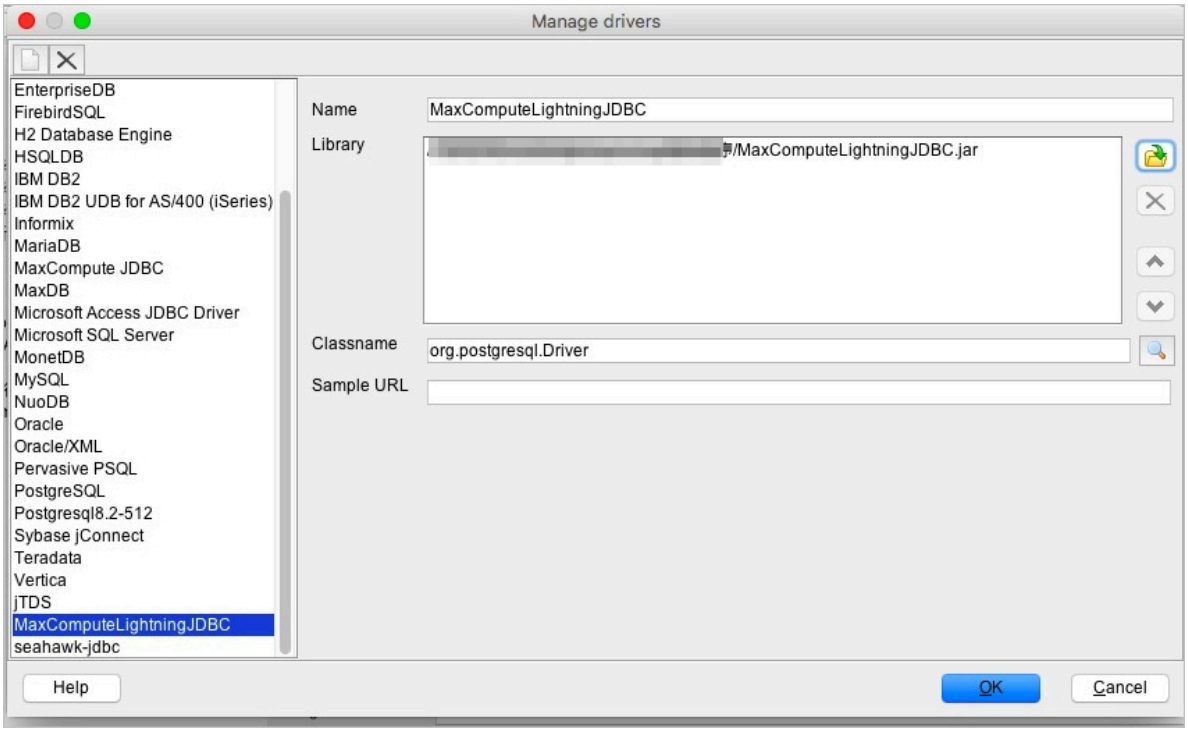
> 注：
> 多くのクライアントツールには既に **PostgreSQL** データベースドライバーが組み込まれており、組み込みドライバーを使用できます。 組み込まれていない場合、**PostgreSQL** 公式 **Web** サイトから必要なドライバーをダウンロードできます。 例として、**SQL Workbench/J** クライアントを取り上げます。 接続を作成するとき、**PostgreSQL** 公式ドライバーを選択できます。



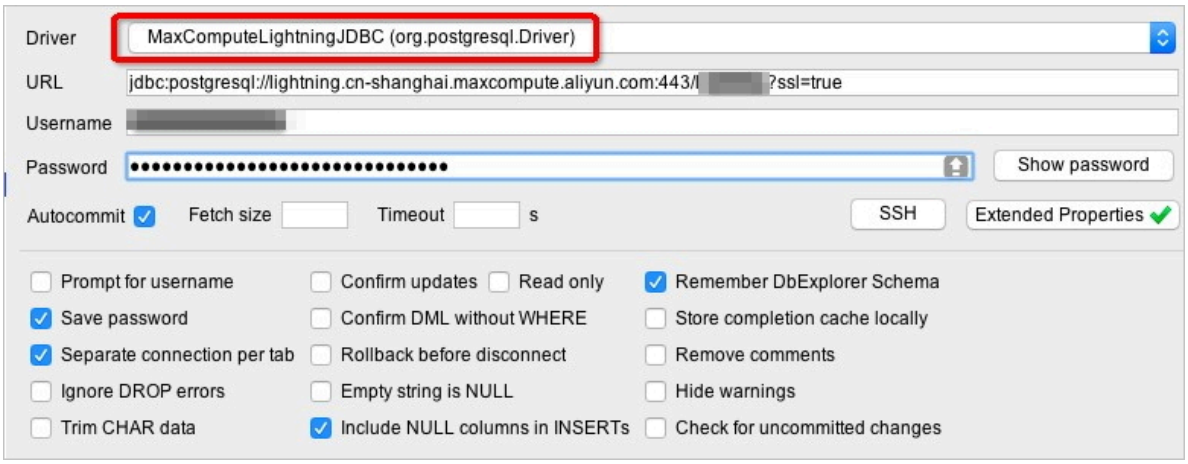
2. **Alibaba Cloud MaxCompute Lightning** によって最適化された *JDBC ドライバー*をダウンロードできます。

   MaxCompute Lightning JDBC ドライバーは、**MaxComputeLightningJDBC.jar** ファイルとして保存されます。 例として、**SQL Workbench/J** クライアントを取り上げます。

[Driver Management] メニューで、**MaxCompute Lightning JDBC** ドライバーエントリを追加します。



接続を作成するとき、ドライバーリストから追加したばかりの **[MaxCompute Lightning JDBC ドライバー]** をクリックします。



9.5.2 JDBC 接続の設定

SQL クライアントツールを **MaxCompute** プロジェクトに接続するには、**MaxCompute** プロジェクト用の **JDBC URL** が必要です。

JDBC URL の形式は次のとおりです。

```
jdbc:postgresql://endpoint:port/database
```

次の表は接続パラメーターを示します。

パラメーター	値	説明
endpoint	リージョン内の **MaxCompute Lightning** のアクセスドメイン名	詳細は、「アクセスドメイン名」をご参照ください。 たとえば、`lightning.cn-shanghai.maxcompute.aliyun.com` を使用して外部ネットワーク経由で上海リージョンサービスにアクセスします。
port	**443**	-
database	**MaxCompute** プロジェクト名	-
User	ユーザーの **Access Key ID**	-
password	ユーザーの **Access Key Secret**	-
ssl	**true**	**MaxCompute Lightning** サーバーはデフォルトで **SSL** 保護が有効になっているため、**SSL** 接続を使用する必要があります。
prepareThreshold	**0**	オプションです。 **JDBC PrepareStatement** 関数を使用するときは、`prepareThreshold = 0` に設定することを推奨します。

例: `jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/myproject`

MaxCompute プロジェクトへの接続を確立する前に、ユーザー、パスワード、および **SSL** 接続パラメーターを指定する必要があります。

JDBC URL にパラメーターを追加して、**MaxCompute** プロジェクトに接続することもできます。 例

```
jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/
myproject? ssl=true& prepareThreshold=0&user=xxx&password=yyy
```

注

- **lightning.cn-shanghai.maxcompute.aliyun.com:** 中国 (上海) リージョンのエンドポイント。
- **Myproject:** アクセスする **MaxCompute** プロジェクト名。
- **SSL = true: SSL** 接続のアプリケーション。
- **xxx:** ユーザーの **Access Key ID**。
- **yyy:** ユーザーの **Access Key Secret**。

9.5.3 共通ツールを使用したサービスへのアクセス

ここでは、**MaxCompute Lightning** へのアクセス方法を説明するため、例として、**SQL Workbench/J**、**PSQL**、および **Tableau BI** ツールなどの主要クライアントツールを使用します。 他の一般的に使用されているツールは、**PostgreSQL** データベースに接続するのと同じ方法で **MaxCompute Lightning** に接続できます。

Alibaba Cloud Quick BI

1. **Quick BI** コンソールにログインし、左側のナビゲーションウィンドウで **[Data source]** をクリックします。
2. データソース管理ページで、右上隅にある **[Create data source]** をクリックします。
3. クラウドデータベースまたは外部データソースで **[PostgreSQL]** をクリックし、データソースを追加します。
4. 表示されたダイアログボックスで、**MaxCompute Lightning** の接続情報を入力します。 次に、接続をテストします。

パラメーター	説明
Database address	**MaxCompute Lightning** のリージョンのエンドポイントを入力します。 パブリックネットワーク、クラシックネットワーク、または **VPC** ネットワークのエンドポイントを入力できます。
Database	アクセスする **MaxCompute** プロジェクト名を入力します。
Schema	**MaxCompute** プロジェクト名
User name/ Password	ユーザーの **Access Key ID/Access Key Secret.**

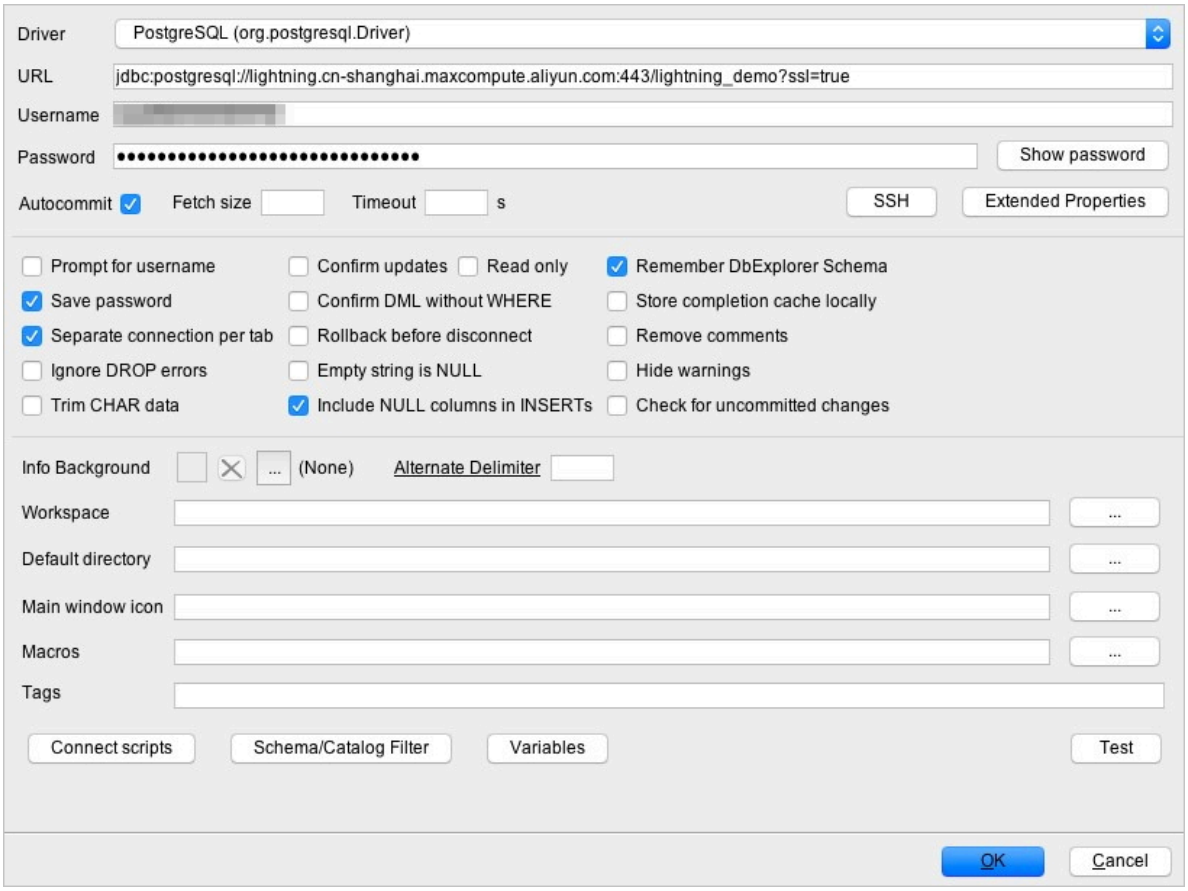
SQL Workbench/J

SQL Workbench/J は、広く使用されている無料のクロスプラットフォーム **SQL** クエリツールです。 このツールでは **PostgreSQL** ドライバーを使用して **MaxCompute Lightning** に接続できます。

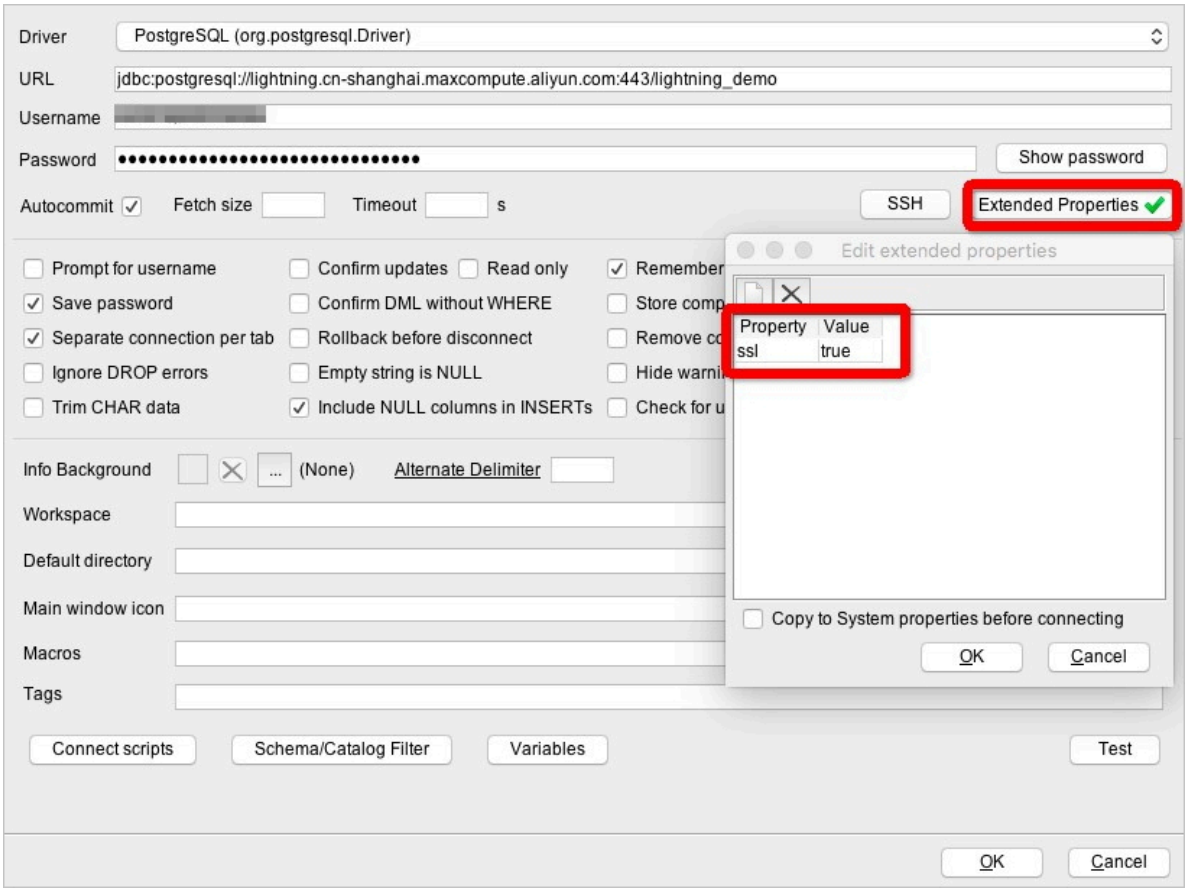
1. **SQL Workbench/J** をダウンロードしてインストールします。

2. **SQL Workbench/J** を起動し、データベース接続を確立します。

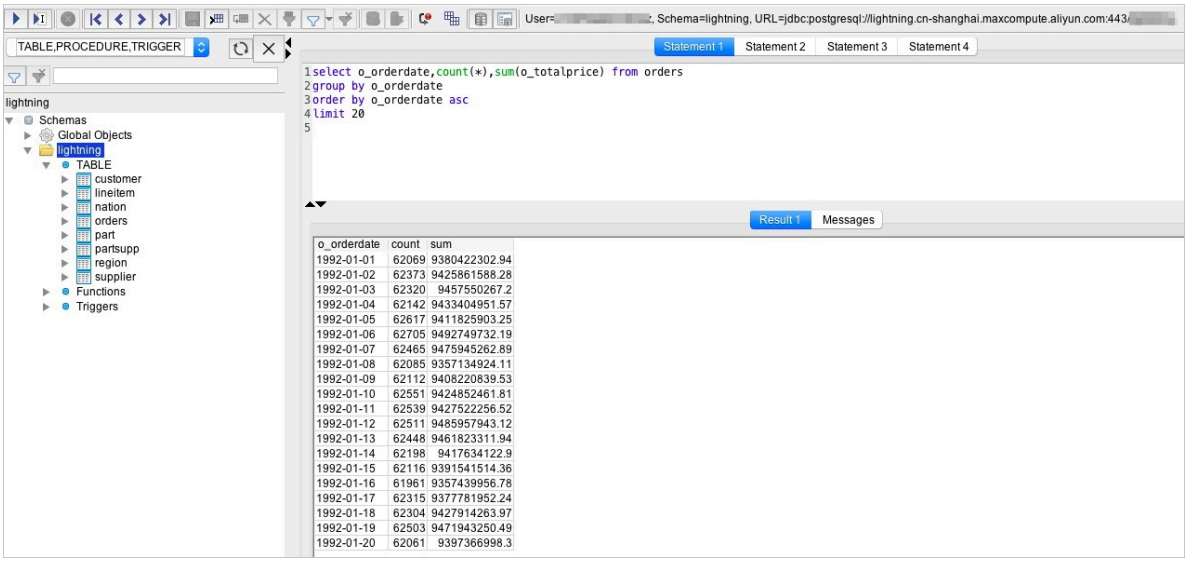
   PostgreSQLドライバーを選択し、**SQL Workbench/J** をプロジェクトの **MaxCompute Lightning URL** に接続します。 ユーザーの **Access Key ID** と **Access Key Secret** を入力する必要があります。



   もう一つの方法として、表示されたダイアログボックスで **[Extended Properties]** をクリックして **[ssl]** を **true** に設定することもできます。



3. **SQL Workbench/J** が **MaxCompute Lightning** に接続された後、**SQL Workbench/J** ワークスペースでテーブルデータを表示、クエリ、および分析することができます。



psql

psql はコマンドを使用してクエリを実行できる **PostgreSQL** の対話型ターミナルです。**PostgreSQL** データベースがローカル **PC** にインストールされるとき、**psql** のクライアントがデフォルトでインストールされます。

psql コマンドを使用して、**psql** を **MaxCompute Lightning** に接続できます。 接続の構文は **PostgreSQL** データベース接続の構文と同じです。

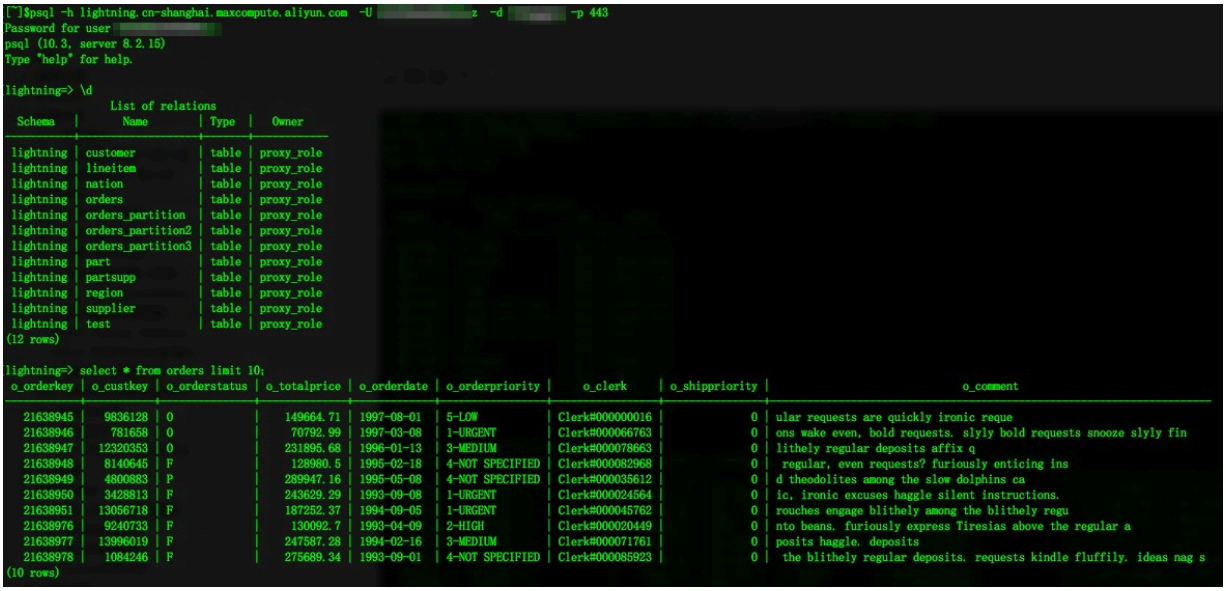
```
psql -h <endpoint> -U <userid> -d <databasename> -p <port>
```

パラメーターの説明

- **<endpoint>: MaxCompute Lightning** のエンドポイント。 詳細は、「アクセスドメイン名」をご参照ください。
- **<userid>: Access Key ID**。
- **<databasename>: MaxCompute** プロジェクト名。
- **<port>: 443**

コマンドが実行された後、コマンドプロンプトで **<userid>** パスワード **(Access Key Secret)** を入力してください。

例



注：

デフォルトで、**psql** の **SSL** 接続が推奨されます。

Tableau Desktop

BI ツールを起動し、**[PostgreSQL data source]** をクリックし、接続を設定します。

接続を設定するとき、**[SSL Connection]** チェックボックスをクリックします。

Tableau Desktop にログインした後、ビジュアル分析用のチャートを作成できます。

注：

パフォーマンスを向上させるため、**Tableau** でサポートされる **TDC** ファイルを使用して、**Lightning** データソースへの接続をカスタマイズすることを推奨します。 手順

1. 次の **xml** コンテンツを **postgresql.tdc** ファイルとして保存します。

```
<?xml version='1.0' encoding='utf-8' ?>
<connection-customization class='postgres' enabled='true' version='
8.10'>
<vendor name='postgres'/>
<driver name='postgres'/>
<customizations>
<customization name='CAP_CREATE_TEMP_TABLES' value='no' />
<customization name='CAP_STORED_PROCEDURE_TEMP_TABLE_FROM_BUFFER'
value='no' />
<customization name='CAP_CONNECT_STORED_PROCEDURE' value='no' />
<customization name='CAP_SELECT_INTO' value='no' />
<customization name='CAP_SELECT_TOP_INTO' value='no' />
<customization name='CAP_ISOLATION_LEVEL_SERIALIZABLE' value='yes
' />
<customization name='CAP_SUPPRESS_DISCOVERY_QUERIES' value='yes' />
<customization name='CAP_SKIP_CONNECT_VALIDATION' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_SUPPRESS_EXPLICIT_COMMIT
' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_SUPPRESS_AUTO_COMMIT'
value='yes' />
<customization name='CAP_ODBC_REBIND_SKIP_UNBIND' value='yes' />
<customization name='CAP_FAST_METADATA' value='no' />
<customization name='CAP_ODBC_METADATA_SUPPRESS_SELECT_STAR' value
='yes' />
<customization name='CAP_ODBC_METADATA_SUPPRESS_EXECUTED_QUERY'
value='yes' />
<customization name='CAP_ODBC_UNBIND_AUTO' value='yes' />
<customization name='SQL_TXN_CAPABLE' value='0' />
<customization name='CAP_ODBC_CURSOR_FORWARD_ONLY' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_COMMIT_INVALIDATES
_PREPARED_QUERY' value='yes' />
</customizations>
</connection-customization>
```

2. ファイルを *\My Documents\My Tableau Repository\Datasources* ディレクトリに保存します。 **Tableau Server** の場合、**Windows** では、*C:\ProgramData\Tableau\Tableau Server\data\tabsvc\vizqlserver\Datasources* に保存し、**Linux** では、*/var/opt/tableau/tableau_server/data/tabsvc/vizqlserver/Datasources/* に保存します。
3. **Tableau** を再度開き、**PostgreSQL** データソースを使用して **MaxCompute Lightning** サービスに接続します。 **tdc** ファイルのカスタムデータソースの詳細については、*Tableau 公式ドキュメント*をご参照ください。

FineReport

1. **FineReport** を起動し、**[Server]** > **[Define database connection]** の順にクリックします。

2. **JDBC** 接続を追加します。

   設定は次のとおりです。

パラメーター	説明
Database	**Postgre**
Driver	**FineReport** に統合されている **org.postgresql.Driver**
URL	`jdbc:postgresql://<MaxCompute Lightning Endpoint>:443/<Project_Name>? ssl=true&prepareThreshold=0` 例: `jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/lightning_demo? ssl=true&prepareThreshold=0`
User name/ Password	**Access Key ID** と **Access Key Secret**

9.6 SQL リファレンス

公式の **PostgreSQL** 関数に基づいて、**MaxCompute Lightning** は、以下のビルトイン関数を追加します。

クエリ構文

MaxCompute Lightning クエリエンジンは **PostgreSQL 8.2** に基づいており、現在は既存の **MaxCompute** テーブルに対する **SELECT** クエリのみをサポートしています。 クエリ構文の詳細については、『*PostgreSQL ドキュメント*』をご参照ください。

関数

MaxCompute Lightning クエリエンジンは、ビルトイン関数をサポートする **PostgreSQL 8.2** に基づいています。詳細については、『*PostgreSQL ドキュメント*』をご参照ください。

公式の **PostgreSQL** 関数に基づいて、**MaxCompute Lightning** は以下のビルトイン関数を追加します。

- **MAX_PT**

  コマンド形式

  ```
  max_pt(table_full_name)
  ```

  コマンド説明

  パーティションテーブルの場合、この関数は、アルファベット順にソートされたパーティションテーブルのレベル **1** パーティションの最大値を戻し、パーティションの下には対応するデータファイルがあります。

  パラメーター説明

  table_full_name: テーブル名を指定するために使用される文字列型 **(prj.src** などのプロジェクト名を含む必要があります**)**、このテーブルへの読み取りアクセス権が必要です。

  戻り値

  最大のレベル **1** パーティションの値を返します。

  例

  tbl がパーティションテーブルで、対応するパーティションが次のようになっていて、どちらもデータファイルを含んでいると仮定します。

  ```
  pt =‘20120901’
  pt =‘20120902’
  ```

  その後、パーティション **max_pt** は次の文で **'20120902'** の値を返し、**MaxCompute SQL** 文は **pt = '20120902'** パーティションの下のデータを読み取ります。

  ```
  select * from tbl where pt=max_pt('myproject.tbl');
  ```

9.7 タスクの表示

MaxCompute Lightning はシステムビュー **stv_recents** を提供します。 ビューをクエリすることで、現在のユーザーが実行しているすべてのクエリタスクを表示できます。

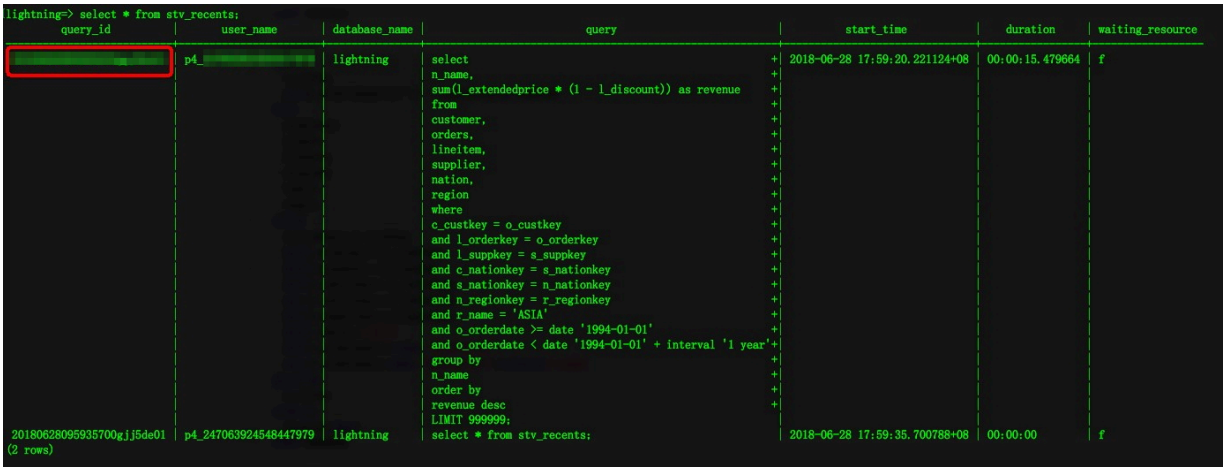
実行中のクエリの表示

クエリコマンドを実行して、クエリ **ID**、ユーザー名、クエリ **SQL** 文、開始時刻、期間、待機中のリソースなど、関連情報を見ることができます。 注**: "t"** は、クエリタスクがまだ実行されておらず、リソースを待機していることを示します。 **"f"** は、リソースが獲得され、クエリタスクが実行されていることを示します。

次のクエリコマンドを実行します。

```
select * from stv_recents;
```

次の図は、コマンド出力例を示します。



実行中のクエリのキャンセル

stv_recents テーブルをクエリすることで、実行中のクエリに関する情報を取得できます。 実行中のクエリをキャンセルするには、次のクエリコマンドを実行します。

```
select cancel('query_id');
```

かっこ内は、実行中のクエリの **query_id** です。



9.8 Lightning に関する FAQ

ここでは、**MaxCompute Lightning** のアプリケーションに関するよくある問題の体系化に役立ちます。

- **Q:** テーブルを作成していないときに **MaxCompute Lightning** を使用してデータをクエリする場合はどのようにすればよいですか。

  A: DataWorks または **odpscmd** クライアントツールを使用して **MaxCompute** プロジェクト用のテーブルを作成し、データをアップロードする必要があります。 **MaxCompute Lightning** を使用してプロジェクトにアクセスし、プロジェクト内のテーブルをクエリすることができます。

- **Q:** クエリできるデータ量の制限はどれくらいですか。 **MaxCompute Lightning** で処理でき、かつ優れたパフォーマンスを発揮できるクエリデータ量の制限はどれくらいですか 。

  A: 現在、各クエリで **1** つのテーブルにつき、最大 **1 TB**のデータをスキャンできます。 クエリ対象のデータが少ないほど、クエリのパフォーマンスはよい結果が得られます。

   注：
  スキャンするテーブルデータは **100 GB**を超えないようにすることを推奨します。 データ量の増加に伴い、クエリのパフォーマンスは徐々に低下します。 クエリのデータが**100 GB**を超える場合、パフォーマンスを向上させるには、**MaxCompute SQL** を推奨します。

- **Q**：分析のために **BI** ツールを使用してパーティションテーブルをドラッグしたとき、次のエラーメッセージが発生した場合、どのようにすればよいですか。`ERROR: AXF Exception : specified partitions count in odps table: <project_name.table_name> is: xxx, exceeds the limitation of xxx, please add stricter partition filter.`

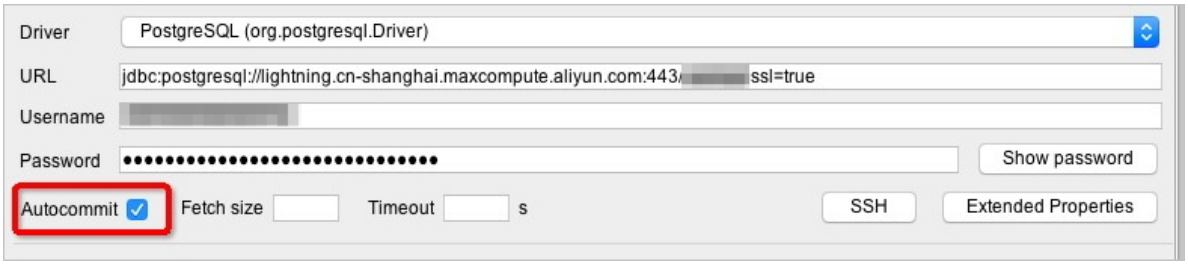
  A: MaxCompute Lightning は、**1** つのパーティションテーブルのパーティション数を制限して、クエリが効率的に実行されるようにします。 各クエリで、**1** つのテーブルに対して最大 **1,024** のパーティションをスキャンできます。 一部の **BI** ツールでは、ドラッグアンドドロップ方式を使用して分析するテーブルを選択できます。 この方法では、分析前にパーティション設定を指定することはできません。 このため、スキャンされるパーティションの数が制限を超え、**MaxCompute Lightning** からエラーが報告される可能性があります。 分析前に、クエリするテーブルの処理を推奨します。 パーティションテーブルを非パーティションテーブルに変換するか、またはパーティション数を **1,024** 未満に減らすことができます。

- **Q**：**MaxCompute Lightning** 接続中に `ERROR: SSL required` が表示されるのはなぜですか。

  A: MaxCompute Lightning では **SSL** 接続が必要なので、ユーザーは **SSL** 接続を使用する必要があります。 クライアントツールを使用している場合、**[SSL connection]** チェックボックスをオンにします。 クライアントツールで **[SSL connection]** を選択できない場合、**SSL** パラメータを **JDBC URL** に追加できます。 **JDBC URL** には、プロジェクトが属するリージョンのエンドポイントと接続先プロジェクト名を入力する必要があります。例: `jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/myproject? ssl=true`

- **Q: SQL Workbench/J** クライアントを使用したクエリ中に次のエラーメッセージが発生した場合、どうすればよいですか。`Error:current transaction is aborted,commands ignored until end of transaction block`

  A: クライアントの **[Autocommit]** チェックボックスをオンにしてください。



10 Graph

10.1 概要

MaxCompute Graph は反復グラフ計算のための処理フレームワークです。 **MaxCompute Graph** ジョブは、グラフを使用してモデルを構築します。 グラフは **Vertex** と **Edge** で構成されており、値を含みます。

MaxCompute Graph は以下のグラフ編集操作をサポートします。

- **Vertex** または **Edge** の値の変更
- **Vertex** の追加と削除
- **Edge** の追加と削除



注：

Vertex と **Edge** を編集する場合、両者の関係を維持しなければなりません。

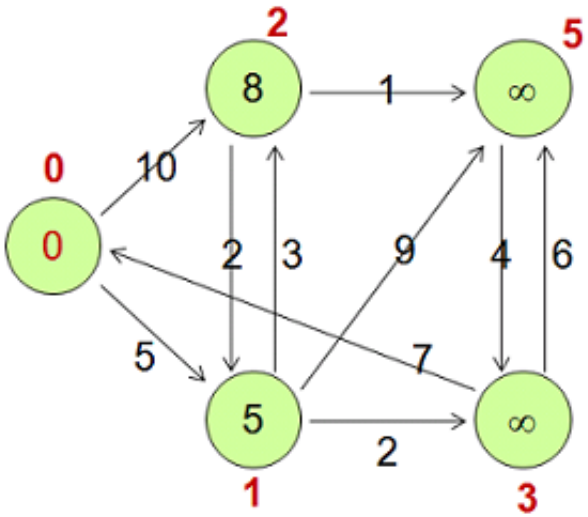
このプロセスは、反復グラフ編集と進化を実行した後に最終的な結果を出力します。 典型的なアプリケーションには、*PageRank*、*SSSP アルゴリズム*、*Kmeans アルゴリズム*などがあります。 **MaxCompute Graph** が提供するインターフェイスである **Java SDK** を使用して、グラフ計算プログラムをコンパイルします。

グラフのデータ構造

MaxCompute Graph が処理するグラフは **Vertex** と **Edge** で構成される有向グラフでなければなりません。 **MaxCompute** は **2** 次元のストレージ構造のみを提供するため、グラフデータを **2** 次元テーブルに変換して **MaxCompute** に保存する必要があります。

グラフ計算と分析の間に、カスタム **Graphloader** を使用して、**MaxCompute Graph** エンジン上の **2** 次元のテーブルデータを **Vertex** と **Edge** に変換します。 サービスシナリオに基づいてどのようにグラフデータを **2** 次元のテーブルに変換するかを決定できます。 サンプルコードでは、テーブル書式はさまざまなグラフデータ構造に対応します。

Vertex 構造は、< **ID, Value, Halted, Edges** > と表現でき、それぞれ順に **ID (vertex ID)**、**Value (**値**)**、**Halted (**ステータス。反復を終了するかどうか**)**、および **Edges (Edge** セット。**Vertex** を起点とするすべての **Edge** のリスト**)** を示します。**Edge** 構造は、< **DestVertexID, Value** > と表現でき、それぞれ宛先の **DestVertexID (Vertex)** と**Value (**値**)** を示します。



たとえば、上のグラフは次の **Vertex** で構成されています。

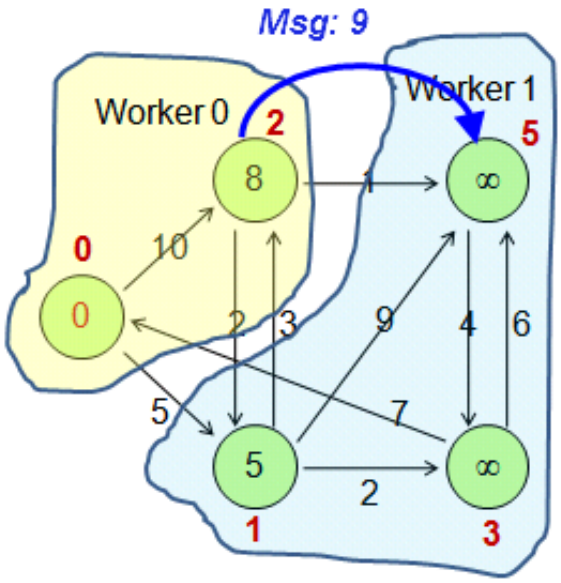
Vertex	**<ID, Value, Halted, Edges>**
v0	**<0, 0, false, [<1, 5 >, <2, 10 >] >**
v1	**<1, 5, false, [<2, 3>, <3, 2>, <5, 9>]>**
v2	**<2, 8, false, [<1, 2>, <5, 1 >]>**
v3	**<3, Long.MAX_VALUE, false, [<0, 7>, <5, 6>]>**
v5	**<5, Long.MAX_VALUE, false, [<3, 4 >]>**

Graph プログラムのロジック

グラフのロード

フレームワークはカスタム **GraphLoader** を呼び出し、入力テーブルのレコードを **Vertex** または **Edge** に変換します。

分散アーキテクチャ: フレームワークはカスタム **Partitioner** を呼び出し、**Vertix** をパーティション化しそれぞれに対応する **Worker** へ送信します。 デフォルトのパーティション化ロジック: **vertex ID** のハッシュ値を計算し **Worker** 数に対しモジュロ演算を行います。



たとえば、上の図で **Worker** 数は **2** であるとします。**Iv0** と **v2** は **ID** を **2** で割った余りが **0** であるため **Worker 0** に割り当てられ、**v1**、**v3**、**v5** は **ID** を **2** で割った余りが **1** となるので **Worker 1** に割り当てられます。

反復計算

- 反復のことをスーパーステップと呼びます。 これは、すべての未停止状態 **(halted** の値が **false)** の **Vertex**、またはメッセージを受け取った **Vertex** (メッセージを受け取った後、停止状態の **Vertex** は自動的に有効化される) を走査し、**compute (ComputeContext context, Iterable messages)** メソッドを呼び出します。
- 実装された **compute (ComputeContext context, Iterable messages)** メソッドでは、以下の手順に従います。
  - 直前のスーパーステップから現在の **Vertex** に送信されたメッセージを処理します。
  - 必要に応じてグラフを編集します。
    - ■ **Vertex** と **Edge** の値の変更
    - ■ 特定の **Vertex** にメッセージ送信
    - ■ **Vertex** または **Edge** の追加と削除
  - **Aggregator** を使用してグローバル情報を更新するための情報を収集します。
  - 現在の **Vertex** を停止または未停止のステータスに設定します。
  - 反復の間、フレームワークは対応する **Worker** へは非同期でメッセージを送信し、次のスーパーステップでは一切の介入なしでメッセージを処理します。

反復の終了

以下のいずれかの条件に一致した場合、反復を終了します。

- すべての **vertex** が停止状態 **(Halted** の値が **true)** で、新たなメッセージが生成されていない場合
- 反復の最大数に達した場合
- **Aggregator** の終了メソッドが **true** を返した場合

擬似コードは次のとおりりです。

```
// 1. load
for each record in input_table {
  GraphLoader.load();

// 2. setup
WorkerComputer.setup();
for each aggr in aggregators {
  aggr.createStartupValue();

for each v in vertices {
  v.setup();

// 3. superstep
for (step = 0; step < max; step ++) {
  for each aggr in aggregators {
    aggr.createInitialValue();

  for each v in vertices {
     v.compute();


// 4. cleanup
for each v in vertices {
  v.cleanup();

WorkerComputer.cleanup();
```

10.2 Aggregator

ここでは **Aggregator** の実装と関連 **API** について説明し、**KmeansClustering** を例に **Aggregator** の使用方法を説明します。

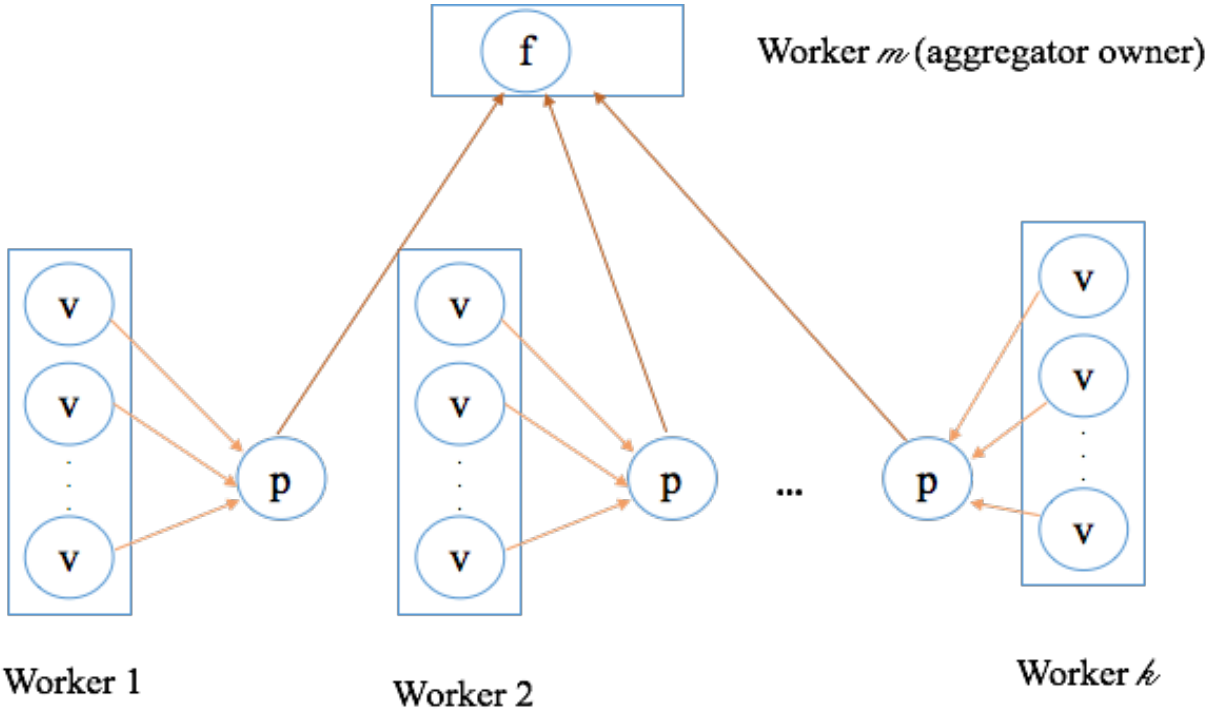
MaxCompute Graph では、**Aggregator** はグローバル情報の収集と処理を手助けします。**MaxCompute Graph** では、**Aggregator** はグローバル情報の要約と処理に使用されます。

Aggregator の実装

Aggregator のロジックは次の **2** つの部分に分かれます。

- **1** つは、分散モードのすべての **Worker** 上で実行されます。
- もう **1** つは、単一頂点モードで **AggregatorOwner** が配置された **Worker** 上で実行されます。

すべての **Worker** 上で実行される操作は、初期値と部分集計の生成が含まれています。 部分集計の結果は、**AggregatorOwner** が配置されている **Worker** に送信されます。**Worker** は、一般的な **Worker** によって送信された部分集計オブジェクトを集計してグローバル集計結果を取得し、反復を終了するかどうかを判断します。次の図に示すように、グローバル集計結果は、次の反復のためのスーパーステップの次のラウンドにわたり、すべての **Worker** に送信されます。



Aggregator の API

Aggregator は、ユーザー実装のための **5** つの **API** を提供します。 次のセクションでは、**5** つの **API** の呼び出し時期と用途について説明します。

- **createStartupValue(context)**

  この **API** はすべての **Worker** で一度だけ実行されます。 すべてのスーパーステップが始まる前に呼び出され、通常 **AggregatorValue** の初期化に使用されます。 最初のスーパーステップの反復 (スーパーステップは **0**) では、**API** で初期化された **AggregatorValue** オブジェクトは **WorkerContext.getLastAggregatedValue()** または **ComputeContext.getLastAggregatedValue()** の呼び出しにより取得できます。

- **createInitialValue(context)**

  この **API** は各スーパーステップが開始されるとすべての **Worker** で呼び出されます。 現在の反復の **AggregatorValue** を初期化するために使用されます。通常、前回の反復の結果は **WorkerContext.getLastAggregatedValue()** によって取得され、部分的な初期化が実行されます。

- **aggregate(value, item)**

  この **API** はすべての **Worker** で実行されます。 前述の **2** つの **API** はフレームワークによって自動的に呼び出されるのに対し、この **API** は**ComputeContext#aggregate(item)** の明示的な呼び出しによってトリガーされます。 この **API** は部分集計を実行するために使用されます。 最初のパラメーター値は、**Worker** が現在のスーパーステップで集計した結果を示します。 初期値は **createInitialValue** によって返されるオブジェクトです。 **2** 番目のパラメーターは、ユーザーコードが **ComputeContext#aggregate(item)** を呼び出すときに送信されます。 この **API** では、**item** は通常、集計値を更新するために使用されます。 すべての集計演算が実行された後、取得された値は、**Worker** の部分集計結果になります。 その後、結果はフレームワークによって **AggregatorOwner** が配置されている **Worker** に送信されます。

- **merge(value, partial)**

  この **API** は、**AggregatorOwner** が配置された **Worker** によって実行されます。 **Worker** の部分集計結果をマージしてグローバル集計オブジェクトを取得するために使用されます。 集計と同様に、**value** は集計結果を示し、**partial** は集計対象のオブジェクトを示します。 **partial** は値を更新するために使用されます。

  たとえば **3** つの **Worker w0**、**w1**、**w2** が存在し、集計結果を **p0**、**p1**、**p2** とします。**p1**、**p0**、**p2** の順番で **AggregatorOwner** が配置された **Worker** へ送信された場合、マージ順序は次のようになります。

  1. **merge(p1, p0)** が最初に実行され、**p1** と **p0** が **p1’** として集計されます。
  2. **merge(p1’, p2)** が実行され、**p1’** と **p2** は集計されて、このスーパーステップでのグローバル集計結果である **p1’’** となります。

  前の例では、**1** つの **Worker** しか存在しない場合、**merge()** 操作の実行は不要であることを示しています。 つまり、**merge()** は呼び出されません。

- **terminate(context, value)**

  AggregatorOwner が配置された **Worker** が **merge()** を実行した後、フレームワークは **terminate(context, value)** を呼び出し、最後の処理を実行します。 **2** 番目のパラメーターの値は **merge()** により取得されたグローバル集計結果を示します。 グローバル集計結果はこのメソッド内でさらに変更可能です。 **terminate()** が実行されると、フレームワークは次のスーパーステップのためにグローバル集計オブジェクトをすべての **Worker** に送信します。 **terminate()** の特別な機能は、**true** が返されると、ジョブ全体の反復が終了するというものです。 それ以外の場合、反復は継続されます。 機械学習シナリオでは、通常、収束後に **true** が返されてジョブの終了が決定されます。

KmeansClustering 例

次のセクションでは、典型的な **KmeansClustering** を例として、**Aggregator** の使用方法を説明します。 次のセクションでは、**KmeansClustering** を例として、**Aggregator** の使用方法を説明します。

 注：
完全なコードは、付属ファイル **Kmeans** で提供されています。 ここでは、コードは次の順序で解決されます。

- **GraphLoader** セクション

  GraphLoader: GraphLoader の部分は、入力テーブルをロードし、グラフの頂点または辺に変換するために使用されます。 入力テーブルのデータの各行はサンプルであり、サンプルは頂点を構成し、**VertexValue** はサンプルを格納するために使用されます。

  最初に、書き込み可能なクラス **KmeansValue** が **VertexValue** タイプとして定義されます。

```
public static class KmeansValue implements Writable {
  DenseVector sample;
  public KmeansValue() {

  public KmeansValue(DenseVector v) {
    this.sample = v;

  @Override
  public void write(DataOutput out) throws IOException {
    wirteForDenseVector(out, sample);

  @Override
  public void readFields(DataInput in) throws IOException {
    sample = readFieldsForDenseVector(in);
```

  KmeansValue: DenseVector オブジェクトは、サンプルを格納するために **KmeansValue** にカプセル化されています。 **DenseVector** 型は *matrix-toolkits-java*. **wirteForDenseVector()** を継承し、 **readFieldsForDenseVector()** はシリアル化と逆シリアル化に使用されます。 詳細については、付属ファイル **Kmeans** 内の完全なコードをご参照ください。

  カスタム **KmeansReader** コードは次のとおりです。

```
public static class KmeansReader extends
                                GraphLoader<LongWritable, KmeansValu
e, NullWritable, NullWritable> {
  @Override
  public void load(
      LongWritable recordNum,
      WritableRecord record,
      MutationContext<LongWritable, KmeansValue, NullWritable,
NullWritable> context)
```

```
      throws IOException {
    KmeansVertex v = new KmeansVertex();
    v.setId(recordNum);
    int n = record.size();
    DenseVector dv = new DenseVector(n);
    for (int i = 0; i < n; i++) {
      dv.set(i, ((DoubleWritable)record.get(i)).get());

    v.setValue(new KmeansValue(dv));
    context.addVertexRequest(v);
```

KmeansReader では、データの各行 (レコード) が読み込まれるときに頂点が作成されます。**recordNum** は頂点 **ID** として使用され、レコードの内容は **DenseVector** オブジェクトに変換され、**VertexValue** にカプセル化されます。

- **Vertex**

  カスタム **KmeansVertex** コード: ロジックについては、各反復で管理されるサンプルに対して部分集計が実行されます。 ロジックについての詳細は、次のセクションの「**Aggregator**」の実装をご参照ください。

```
public static class KmeansVertex extends
                                  Vertex<LongWritable, KmeansValue,
NullWritable, NullWritable> {
  @Override
  public void compute(
  ComputeContext<LongWritable, KmeansValue, NullWritable, NullWritab
le> context,
  Iterable<NullWritable> messages) throws IOException {
    context.aggregate(getValue());
```

- **Aggregator**

  Kmeans 全体の主なロジックは **Aggregator** に集計されます。カスタム **KmeansAggrValue** は、集計と送信の対象となるコンテンツの管理に使用されます。

```
public static class KmeansAggrValue implements Writable {
  DenseMatrix centroids;
  DenseMatrix sums; // used to recalculate new centroids
  DenseVector counts; // used to recalculate new centroids
  @Override
  public void write(DataOutput out) throws IOException {
    wirteForDenseDenseMatrix(out, centroids);
    wirteForDenseDenseMatrix(out, sums);
    wirteForDenseVector(out, counts);

  @Override
  public void readFields(DataInput in) throws IOException {
    centroids = readFieldsForDenseMatrix(in);
    sums = readFieldsForDenseMatrix(in);
    counts = readFieldsForDenseVector(in);
```

KmeansAggrValue では **3** つのオブジェクトが管理されています。**centroids** は既存の **K** 中心点を示します。 サンプルが **m** ディメンションの場合、**centroids** は **K x m** のマトリクスです。**sums** は **centroids** と同じサイズのマトリクスで、各要素は特定の中心点に最も近いサンプルの特定のディメンションの合計を記録します。 たとえば、**sums(i, j)** は、中心点 **i** に最も近いサンプルのディメンション **j** の合計を示します。

counts は各中心点に最も近いサンプル数を記録する **K** ディメンションベクトルです。**sums** と **counts** は主な集計内容である新しい中心点を計算するために一緒に使用されます。

次に **KmeansAggregator** がカスタム **Aggregator** 実装に使用されます。 前述の **API** の順に実装について説明します。

1. **createStartupValue()** を実行します。

```
public static class KmeansAggregator extends Aggregator<KmeansAggr
Value> {
public KmeansAggrValue createStartupValue(WorkerContext context)
throws IOException {
KmeansAggrValue av = new KmeansAggrValue();
byte[] centers = context.readCacheFile("centers");
String lines[] = new String(centers).split("\n");
int rows = lines.length;
int cols = lines[0].split(",").length; // assumption rows >= 1
av.centroids = new DenseMatrix(rows, cols);
av.sums = new DenseMatrix(rows, cols);
av.sums.zero();
av.counts = new DenseVector(rows);
av.counts.zero();
for (int i = 0; i < lines.length; i++) {
  String[] ss = lines[i].split(",");
  for (int j = 0; j < ss.length; j++) {
    av.centroids.set(i, j, Double.valueOf(ss[j]));


return av;
```

   このメソッドでは、**KmeansAggrValue** オブジェクトが初期化され、初期中心点がリソースファイルセンターから読み込まれ、値が重心に与えられます。**sums** と **counts** の初期値は **0** です。

2. **createInitialValue()** の実行します。

```
@Override
public void aggregate(KmeansAggrValue value, Object item)
  throws IOException {
DenseVector sample = ((KmeansValue)item).sample;
// find the nearest centroid
int min = findNearestCentroid(value.centroids, sample);
// update sum and count
for (int i = 0; i < sample.size(); i ++) {
  value.sums.add(min, i, sample.get(i));
```

```
value.counts.add(min, 1.0d);
```

この **createInitialValue()** メソッドでは、サンプルアイテムと最短のユークリッド距離にある中心点のインデックスを見つけるために **findNearestCentroid()** が呼び出されます。 次に、各ディメンションが合計に加算され、カウントの値がプラス **1** になります。 **findNearestCentroid()** の実装方法の詳細は、付属ファイル **Kmeans** をご参照ください。

前述の **3** つの関数は部分集計を実装するためにすべての **Worker** で実行されます。 以下に **AggregatorOwner** が配置された **Worker** で実行されるグローバル集計関連操作を説明します。

1. **merge()** を実行します。

```
@Override
public void merge(KmeansAggrValue value, KmeansAggrValue partial)
  throws IOException {
value.sums.add(partial.sums);
value.counts.add(partial.counts);
```

マージ処理の実装ロジックは各 **Workder** で集計された合計とカウントの値を追加します。

2. **terminate()** を実行します。

```
@Override
public boolean terminate(WorkerContext context, KmeansAggrValue
value)
   throws IOException {
 // Calculate the new means to be the centroids (original sums)
 DenseMatrix newCentriods = calculateNewCentroids(value.sums, value.
counts, value.centroids);
 // print old centroids and new centroids for debugging
 System.out.println("\nsuperstep: " + context.getSuperstep() +
     "\nold centriod:\n" + value.centroids + " new centriod:\n" +
newCentriods);
 boolean converged = isConverged(newCentriods, value.centroids, 0.
05d);
 System.out.println("superstep: " + context.getSuperstep() + "/"
     + (context.getMaxIteration() - 1) + " converged: " + converged
);
 if (converged || context.getSuperstep() == context.getMaxIteration
() - 1) {
   // converged or reach max iteration, output centriods
   for (int i = 0; i < newCentriods.numRows(); i++) {
     Writable[] centriod = new Writable[newCentriods.numColumns()];
     for (int j = 0; j < newCentriods.numColumns(); j++) {
       centriod[j] = new DoubleWritable(newCentriods.get(i, j));

     context.write(centriod);

   // true means to terminate iteration
   return true;

 // update centriods
 value.centroids.set(newCentriods);
```

```
// false means to continue iteration
return false;
```

terminate() では、**calculateNewCentroids()** が合計とカウントに基づいて呼び出され、平均値を計算して新しい中心点を取得します。 次に、新しい中心点と古い中心点との間のユークリッド距離に基づいて **isConverged()** が呼び出され、中心点が収束しているかどうかが判定されます。 収束または反復の回数が上限のしきい値に達すると、新しい中心点が出力され、**true** が返されて反復が終了します。 それ以外の場合は、中心点が更新され、反復を続けるために **false** が返されます **calculateNewCentroids()** および **isConverged()** の実装方法の詳細については、付属ファイルをご参照ください。

- **main()** メソッド

  main() メソッドは、**Graph** ジョブの作成、関連する設定の実行、およびジョブの送信に使用されます。 コードは以下のとおりです。

```
public static void main (final String [] args) throws IOException{
                                if (args.length < 2)
    printUsage();
  GraphJob job = new GraphJob();
  job.setGraphLoaderClass(KmeansReader.class);
  job.setRuntimePartitioning(false);
  job.setVertexClass(KmeansVertex.class);
  job.setAggregatorClass(KmeansAggregator.class);
  job.addInput(TableInfo.builder().tableName(args[0]).build());
  job.addOutput(TableInfo.builder().tableName(args[1]).build());
  // default max iteration is 30
  job.setMaxIteration(30);
  if (args.length >= 3)
    job.setMaxIteration(Integer.parseInt(args[2]));
  long start = System.currentTimeMillis();
  job.run();
  System.out.println("Job Finished in "
      + (System.currentTimeMillis() - start) / 1000.0 + " seconds");
```

 注：

job.setRuntimePartitioning(false) が **false** に設定されている場合、各 **Worker** によってロードされたデータは **Partitioner** に基づいてパーティション化されません。 つまり、データをロードする人がそれを維持します。

まとめ

ここでは、**MaxCompute Graph** の **Aggregator** の機能、**API** の意味、および **KmeansClustering** の例を紹介しています。 まとめると、**Aggregator** は以下のように実装できます。

1. 各 **Worker** は **AggregatorValue** 生成のための起動中に **createStartupValue** を実行します。

2. 各 **Worker** は、各反復が現在のラウンドで **AggregatorValue** を初期化する前に **createInitialValue** を実行します。
3. 反復では、各頂点は **context.aggregate()** を使用して**aggregate()** を実行し、**Worker** で部分的な反復を実装します。
4. 各 **Worker** は、**AggregatorOwner** が配置されている **Worker** に部分的な反復結果を送信します。
5. **AggregatorOwner** が配置されている **Worker** は、グローバル集計を実装するために複数回マージを実行します。
6. **AggregatorOwner** が配置されている **Worker** は、グローバル集計結果処理を終了し、反復を終了するかどうかを決定します。

付属ファイル

Kmeans

10.3 制限

MaxComputer Graph の制限は以下のとおりです。

- 各ジョブは最大 **256** 個のリソースを参照できます。 テーブルまたはアーカイブは **1** つの単位 **(1** つのリソース) として扱われます。
- **1** つのジョブで参照されるリソースの総バイト数は **512 MB** を超えることはできません。 各ジョブは最大 **512 MB** バイトのリソースを参照できます。
- **1** ジョブの入力数は **1,024** を超えることはできません。また、入力テーブルの数は **64** を超えることはできません。 **1** ジョブの出力数は **256** を超えることはできません。
- ラベル長は最大 **256** 文字で、文字、数字、アンダースコア **(_)**、シャープ **(#)**、ピリオド **(.)**、ハイフン **(-)**を含む特殊文字を含むことができます。 複数出力に指定されたラベルは、**null** または空の文字列にすることができません。
- 各ジョブは最大 **64** のカスタムカウンタを持つことが可能です。 グループ名とカウンタ名には最大 **100** 文字を使用できます。 名前にシャープ **(#)** を含めることはできません。
- **1** ジョブの **Worker** 数はフレームワークによって計算されます。 最大数は **1,000** です。 しきい値を超えた場合、例外が発生します。
- **1 Worker** は デフォルトで **200** の **CPU** リソースを占有します。 範囲は **50**～**800** です。
- **1 Worker** はデフォルトで **4096 MB** のメモリを占有します。 範囲は **256 MB**～**12 GB** です。
- **Worker** が繰り返しリソースを読み込むためのしきい値は **64** です。

- 分割サイズを設定できますが、デフォルトサイズは **64 MB** です。 範囲は **0** < 分割サイズ <= **(9223372036854775807** >> **20)** です。
- **MaxCompute Graph** プログラムでは、クラスター内で実行される **GraphLoader/Vertex/Aggregator** は **Java** サンドボックスにより制限されます。 **Graph** ジョブの主要プログラムは制限されません。 制限の詳細については、「*Java* サンドボックス」をご参照ください。

10.4 機能の概要

ジョブの実行

MaxCompute コンソールには、**MaxCompute Graph** ジョブを実行するための **jar** コマンドが提供されています。 これらのコマンドは、*MapReduce* の *jar* コマンドを実行するのと同じ方法で使用されます。

ここでは、これらのコマンドを説明します。

```
Usage: jar [<GENERIC_OPTIONS>] <MAIN_CLASS> [ARGS]
                -conf <configuration_file> Specify an application
configuration file
                -classpath <local_file_list> classpaths used to run
mainClass
                -D <name>=<value> Property value pair, which is used
to run mainClass
                -local Run job in local mode
                -resources <resource_name_list> file/table resources
used in graph, separated by command
```

< **GENERIC_OPTIONS**> には、以下のパラメーターを指定できます (すべてオプション)。

- **-conf** < **configuration file** >: **JobConf** 設定ファイルを指定します。
- **-classpath** < **local_file_list** >: ローカル実装のクラスパスを示します。 これは主に **main** 関数を含む **jar** パッケージを指定するのに使用されます。

  main 関数と **Graph** ジョブは、通常同じパッケージに記述されます。たとえば、**Single Source Shortest Path (SSSP)** パッケージなどです。 したがって、サンプルコード内の **-resources** と **-classpath** パラメータには、**jar** パッケージが含まれています。 違いは **-resources** が **Graph** ジョブの値を参照し分散環境で実行されるのに対し、**-classpath** は **main** 関数を参照しローカル環境で実行されます。 指定された **jar** パッケージパスはローカルファイルパスでもあります。 パッケージ名はシステムのデフォルトのファイル区切り文字を使用して分割されます。 一般に、区切り文字は **Windows** システムではセミコロン **(;)**、**Linux** システムではカンマ **(,)** です。
- **-D** < **prop_name** > = < **prop_value** >: ローカル実装のための < **mainClass** > の **Java** 属性を指定します。 複数の属性を定義できます。

- **-local:** ローカルモードで **Graph** ジョブを実行します。主にプログラムのデバッグに使用されます。
- **-resources <resource_name_list >: Graph** ジョブ実行のために使用されるリソース命令文を示します。一般に、**Graph** ジョブが配置されるリソース名は **resource_name_list** に指定されます。 **Graph** ジョブ内で他の **MaxCompute** リソースを読み込む場合、そのリソース名を **resource_name_list** に追加する必要があります。リソース名はカンマ **(,)** で区切ります。 プロジェクト間でリソースを使用する場合、**PROJECT_NAME/resources/** を前につける必要があります。 たとえば、 `-resources otherproject/resources/resfile;` とします。

また、**Graph** ジョブの **main** 関数を実行すると、**MaxCompute** コンソールからジョブを送信するのではなく、**MaxCompute** にジョブが直接送信されます。 次のセクションでは、例として *PageRank アルゴリズム*を使用します。

```
public static void main(String[] args) throws Exception {
                    if (args.length < 2)
    printUsage();
  Account account = new AliyunAccount(accessId, accessKey);
  Odps odps = new Odps(account);
  odps.setEndpoint(endPoint);
  odps.setDefaultProject(project);
  SessionState ss = SessionState.get();
  ss.setOdps(odps);
  ss.setLocalRun(false);
  String resource = "mapreduce-examples.jar";
  GraphJob job = new GraphJob();
  // Add the JAR file in use and other files to class cache resource,
corresponding to resources specified by -libjars in the JAR command
  job.addCacheResourcesToClassPath(resource);
  job.setGraphLoaderClass(PageRankVertexReader.class);
  job.setVertexClass(PageRankVertex.class);
  job.addInput(TableInfo.builder().tableName(args[0]).build());
  job.addOutput(TableInfo.builder().tableName(args[1]).build());
  // default max iteration is 30
  job.setMaxIteration(30);
  if (args.length >= 3)
    job.setMaxIteration(Integer.parseInt(args[2]));
  long startTime = System.currentTimeMillis();
  job.run();
  System.out.println("Job Finished in "
      + (System.currentTimeMillis() - startTime) / 1000.0
      + " seconds");
```

入出力

入出力の形式はカスタマイズできません。

次の例はジョブ入力を定義する方法です。 複数の入力がサポートされています。

```
GraphJob job = new GraphJob();
job.addInput(TableInfo.builder().tableName(“tblname”).build()); //
Table as input
```

```
job.addInput(TableInfo.builder().tableName(“tblname”).partSpec("pt1=a/
pt2=b").build()); //Shard as input
//Read-only columns col2 and col0 of the input table. In the load()
method of GraphLoader, column col2 is obtained by record.get(0), and
the sequence is the same
job.addInput(TableInfo.builder().tableName(“tblname”).partSpec("pt1=a/
pt2=b").build(), new String[]{"col2", "col0"});
```



注：

- ジョブ入力の定義の詳細は、**GraphJob** の **addInput()** メソッドの説明をご参照ください。フレームワークは入力テーブルのレコードを読み取り、カスタム **GraphLoader** へ送信してデータをロードします。
- 制限事項: 現在、シャードフィルタリング条件はサポートされていません。 詳細については、「アプリケーションの制限」をご参照ください。

次の例はジョブ出力を定義する方法です。 複数のジョブ出力がサポートされています。各出力はラベルでマークされます。

```
GraphJob job = new GraphJob();
//If the output table is a shard table, the last level of shards must
be provided
job.addOutput(TableInfo.builder().tableName("table_name").partSpec("
pt1=a/pt2=b").build());
// Parameter true indicates overwriting shards specified by tableinfo
, that is, the meaning of INSERT OVERWRITE. Parameter false indicates
the meaning of INSERT INTO
job.addOutput(TableInfo.builder().tableName("table_name").partSpec("
pt1=a/pt2=b").lable("output1").build(), true);
```



注：

- ジョブ出力の定義の詳細については、**GraphJob** の **addOutput()** メソッドの説明をご参照ください。
- **Graph** ジョブが実行されると、**WorkerContext** の **write()** メソッドによりレコードが出力テーブルに書き込まれます。 前のセクションの **"output1"** のように複数出力に対してラベルを指定する必要があります。
- 詳細については、「アプリケーションの制限」をご参照ください。

リソースの読み取り

- **Graph** プログラムへリソースを追加する

  jar コマンドに加えて、以下の **GraphJob** メソッドを使用して **Graph** が読み取るリソースを指定することができます。

  ```
  void addCacheResources(String resourceNames)
  ```

```
void addCacheResourcesToClassPath(String resourceNames)
```

- **Graph** プログラムでリソースを使用する

  Graph プログラムのリソースを読み取るには、次の手順に従います。

```
public byte[] readCacheFile(String resourceName) throws IOException;
                                    public Iterable<byte[]>
readCacheArchive(String resourceName) throws IOException;
                                    public Iterable<byte[]>
readCacheArchive(String resourceName, String relativePath)throws
IOException;
                                    public Iterable<WritableRecord>
readResourceTable(String resourceName);
public BufferedInputStream readCacheFileAsStream(String resourceName
) throws IOException;
public Iterable<BufferedInputStream> readCacheArchiveAsStream(String
 resourceName) throws IOException;
public Iterable<BufferedInputStream> readCacheArchiveAsStream(String
 resourceName, String relativePath) throws IOException;
```

  注：

  - 通常、リソースの読み取りには **WorkerComputer** の **setup()** メソッド (**Worker Value** に保存)、取得には **getWorkerValue()** メソッドが使用されます。
  - 全体的なメモリ消費を減らすには、前述のストリーム **API** を使用してリソースの読み取りと処理が同時に実行されるようにします。
  - 詳細については、「アプリケーションの制限」をご参照ください。

10.5 SDK の概要

Maven ユーザーは、*Maven* データベース内で **odps-sdk-graph** を検索し、 必要な **SDK** を入手できます (異なるバージョンを入手可能)。 設定情報は以下のとおりです。

```
<dependency>
    <groupId>com.aliyun.odps</groupId>
    <artifactId>odps-sdk-graph</artifactId>
    <version>0.20.7</version>
</dependency>
```

主要なインターフェイス	説明
GraphJob	**GraphJob** は **JobConf** から継承され、**MaxCompute Graph** ジョブの定義、送信、管理に使用されます。
Vertex	**vertex** は、**id**、**value**、**halted**、および **edges** を含む属性によって定義されたノードです。**vertex** は **GraphJob** の **setVertexClass** インターフェイスによって実装されます。

主要なインターフェイス	説明
Edge	**Edge** は **destVartexId** 属性と **value** 属性を含む、グラフの **edge** の抽象化です。隣接するテーブルはグラフデータ構造として使用され、**vertex** の外向き辺は **vertex** の **edges** に保存されます。
GraphLoader	**GraphLoader** はグラフをロードするために使用されます。**GraphLoader** は **GraphJob** の **setGraphLoaderClass** インターフェイスによって実装されます。
VertexResolver	**VertexResolver** はグラフのトポロジーを変更するために競合処理ロジックをカスタマイズするために使用されます。**GraphJob** の **setLoadingVertexResolverClass** と **setComputingVertexResolverClass** インターフェイスはグラフロード中および反復計算中にグラフテクノロジーの変更のための競合処理ロジックを提供します。
Partitioner	**Partitioner** はグラフを分割し、計算を分解するために使用されます。**Partitioner** は **GraphJob** の **setPartitionerClass** インターフェイスを使用することで実装されます。 **HashPartitioner** はデフォルトで使用されます。つまり、**vertex ID** のハッシュ値が計算され、モジュロ演算が **Worker** の数に対して実行されます。
WorkerComputer	**WorkerComputer** は **Worker** が起動時および終了時にカスタムロジックを実行することを許可します。 **WorkerComputer** は **GraphJob** の **setWorkerComputerClass** インターフェイスを使用することで実装されます。
Aggregator	**setAggregatorClass(Class …)** は **1** つまたは複数の **Aggregator** を定義します。
Combiner	**setCombinerClass** は **Combiner** を設定します。
Counters	カウンターを示します。 ジョブ実行ロジックでは、**WorkerContext** インターフェイスはカウンターを取得し、カウントを実行するために使用できます。 フレームワークは自動的に結果を集計します。
WorkerContext	コンテキストオブジェクトを示します。 グラフテクノロジーの変更、メッセージの送信、結果の書き込み、リソースの読み込みなど、フレームワークが提供する機能をカプセル化します。

10.6 開発とデバッグ

MaxCompute は、**Graph** 開発プラグインを提供していません。 ただし、**Eclipse** ベースの **MaxCompute Graph** プログラムを開発することはできます。 開発プロセスは以下のとおりです。

1. Graph のコードをコンパイルし、ローカルデバッグを使用して基本テストを実行します。

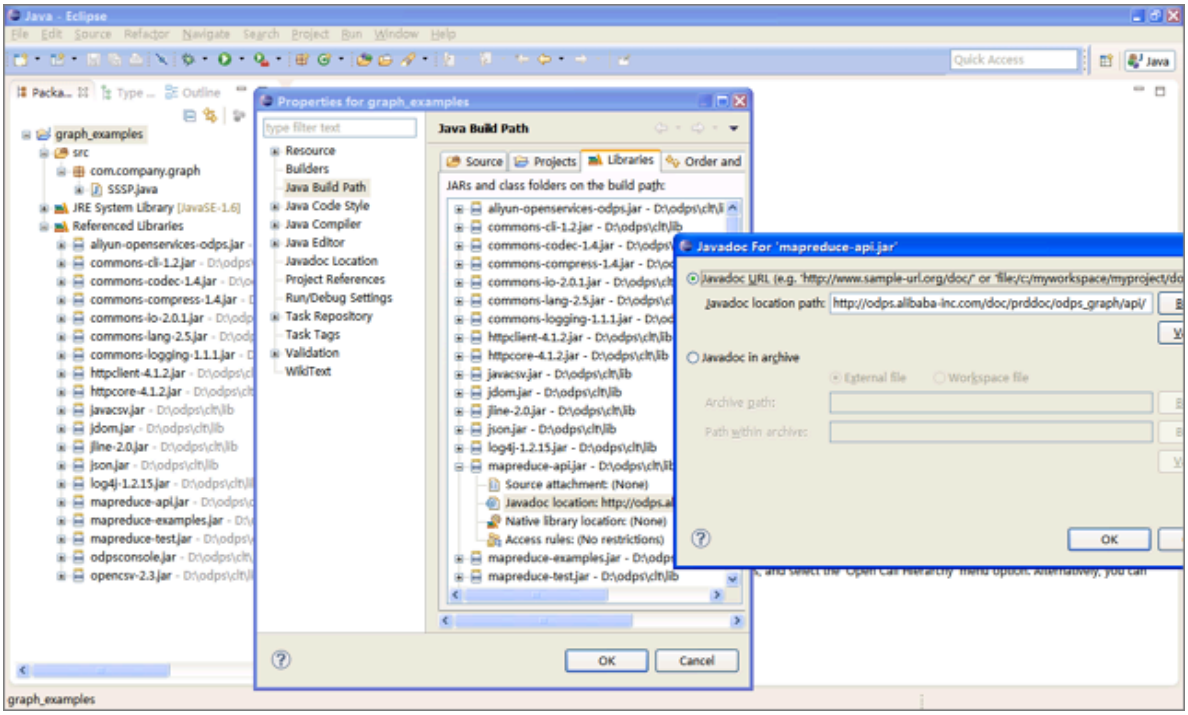
2. クラスタのデバッグを実行し、結果を確認します。

例

このセクションでは、*SSSP* アルゴリズムを例として使用し、 **Eclipse** を使用した **Graph** プログラムの開発とデバッグの方法について説明します。

手順

1. **Java** プロジェクトを作成します (例：**graph_examples**)。
2. **MaxCompute** クライアントの **lib** ディレクトリ内の **jar** パッケージを **Eclipse** プロジェクトのビルドパスに追加します。 次の図は、設定済みの **Eclipse** プロジェクトを示します。



3. **MaxCompute Graph** プログラムを開発します。

   実際の開発プロセスでは、例 (*SSSP* など) をコピーしてから変更することがよくあります。 この例では、パッケージパスのみが **package com.aliyun.odps.graph.example** に変更されています。

4. パッケージをコンパイルし、ビルドします。

   Eclipse 環境で、ソースコードディレクトリ (**src** ディレクトリ) を右クリックし、**[Export]** > **[Java]** > **[jar file]** の順に選択し、 **jar** パッケージを生成します。 ターゲットの **jar** パッケージを保存するパスを選択します。 たとえば、`D:\\odps\\clt\\odps-graph-example-sssp.jar` です。

5. **MaxCompute** コンソールを使用して、**SSSP** を実行します。 関連する操作の詳細については、「クイックスタート」の「*Graph の実行*」をご参照ください。

 注：

関連する開発手順の詳細については、「*Graph* 開発プラグイン」の説明をご参照ください。

ローカルデバッグ

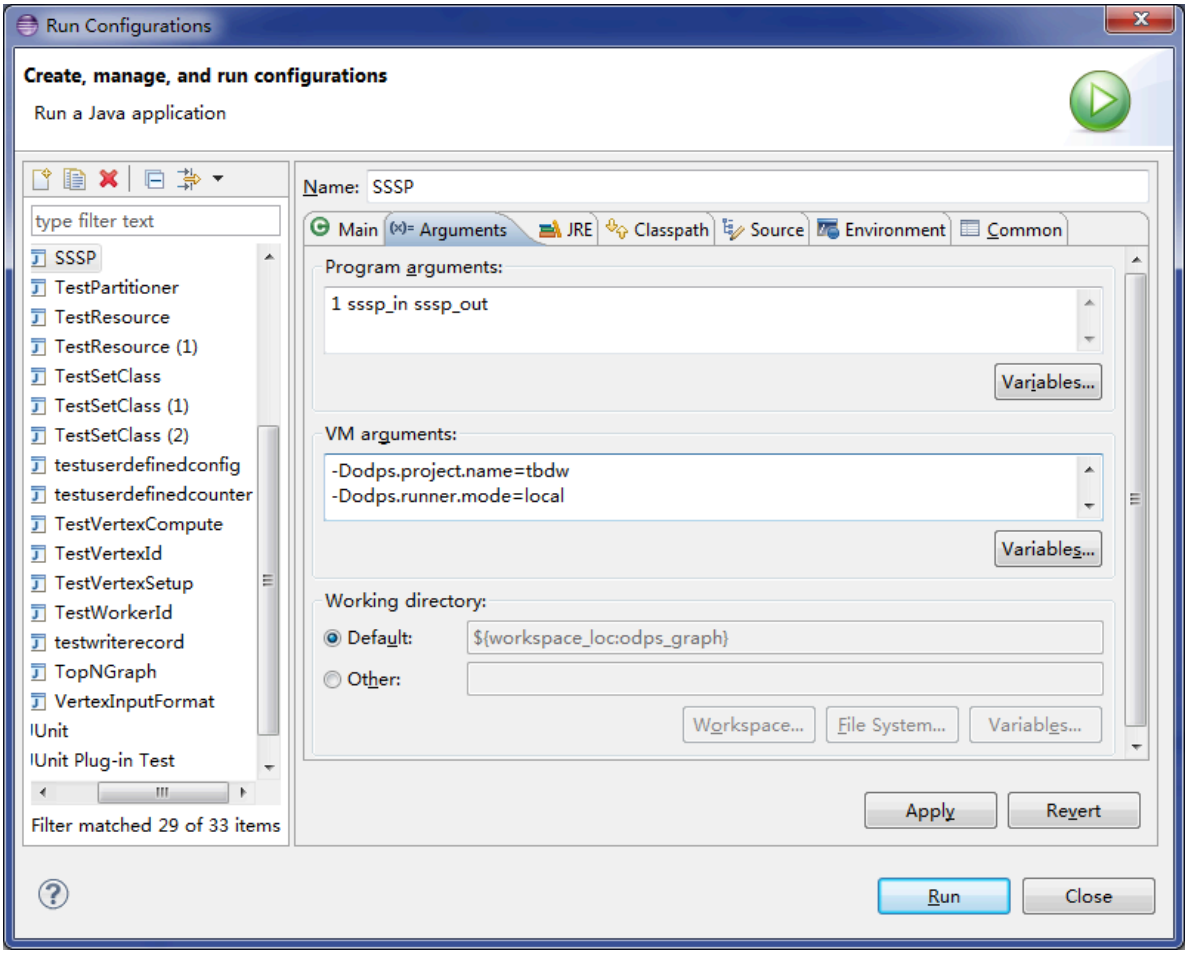
MaxCompute Graph はローカルデバッグモードをサポートします。 **Eclipse** を使用してブレークポイントデバッグを実行します。

手順

1. **maven** パッケージ **odps-graph-local** をダウンロードします。
2. **Eclipse** プロジェクトを選択し、**Graph** ジョブのメインプログラムファイル **(main** 関数を含む) を右クリックし、**[Run As]** > **[Run Configurations]** の順に実行して実行パラメーターを設定します。
3. **[Arguments]** タブページで、**Program**引数を、メインプログラムの入力パラメーターとして `1 sssp_in sssp_out` に設定します。
4. **[Arguments]** タブページで、**VM** 引数を以下のとおり設定します。

```
-Dodps.runner.mode=local
-Dodps.project.name=<project.name>
-Dodps.end.point=<end.point>
-Dodps.access.id=<access.id>
```

```
-Dodps.access.key=<access.key>
```



5. **MapReduce** がローカルモード **(odps.end.point** の値が指定されていない**)** の場合、**warehouse** に **sssp_in** と **sssp_out** テーブルを作成し、**sssp_in** のデータを追加する必要があります。 入力データは以下のとおりです。

```
1,"2:2,3:1,4:4"
2,"1:2,3:2,4:1"
3,"1:1,2:2,5:1"
4,"1:4,2:1,5:1"
5,"3:1,4:1"
```

warehouse の詳細については、「*MapReduce のローカル実行*」をご参照ください。

6. **[Run]** をクリックします。



注：

MaxCompute クライアントの **conf/odps_config.ini** の設定を確認し、パラメーターを設定します。 上記パラメーターは共通で使用されます。 以下に、他のパラメーターについて説明します。

- **odps.runner.mode:** パラメーター値はローカルです。 このパラメーターはローカルデバッグ機能に必要です。
- **odps.project.name:** 必須です。 現在のプロジェクトを指定します。
- **odps.end.point:** 任意です。 現在の **MaxCompute** サービスのアドレスを指定します。 このパラメーターが指定されていないと、テーブルまたはリソースのメタデータが **warehouse** からのみ読み取られ、アドレスが存在しない場合に例外が発生します。このパラメーターが指定されていると、データは最初に **warehouse** から読み取られ、次に、アドレスが存在しない場合は、リモート **MaxCompute** から読み取られます。
- **odps.access.id: MaxCompute** サービスに接続するための **ID** を示します。 このパラメーターは **odps.end.point** が指定されている場合にのみ有効です。
- **odps.access.key: MaxCompute** サービスに接続するためのキーを示します。 このパラメーターは **odps.end.point** が指定されている場合にのみ有効です。
- **odps.cache.resources:** 使用中のリソースリストを指定します。 このパラメーターは **jar** コマンドの **-resources** と同じ効果があります。
- **odps.local.warehouse:** ローカルの **warehouse** パスを示します。 このパラメーターが定されていなれば、デフォルトで **./warehouse** に設定されています。

Eclipse のローカルで **SSSP** デバッグが実行された後、以下の情報が出力されます。

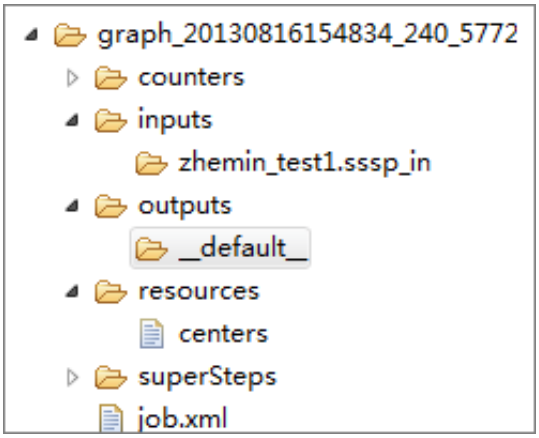
```
Counters: 3
         com.aliyun.odps.graph.local.COUNTER
                 TASK_INPUT_BYTE=211
                 TASK_INPUT_RECORD=5
                 TASK_OUTPUT_BYTE=161
                 TASK_OUTPUT_RECORD=5
 graph task finish
```

 注：

上の例では、**sssp_in** および **sssp_out** テーブルがローカルの **warehouse** に存在している必要があります。 **sssp_in** および **sssp_out** テーブルの詳細については、「クイックスタート」の「*Graph* の実行」をご参照ください。

ローカルジョブの一時ディレクトリ

以下の図で示すように、ローカルデバッグを実行するたびに **Eclipse** プロジェクトディレクトリに一時ディレクトリが作成されます。



ローカルで実行される **Graph** ジョブの一時ディレクトリは以下のディレクトリとファイルを含みます。

- **counters:** 実行中のジョブのカウント情報を保存します。
- **inputs:** ジョブの入力データを保存します。 データは優先的にローカルの **warehouse** から取得されます。 データがローカルに存在しない場合、**MaxCompute SDK** はデータをサーバーから **(odps.end.point** が設定されている場合) 読み取ります。 入力はデフォルトで **10** データレコードを読み取ります。 このしきい値は `-Dodps.mapred.local.record.limit` パラメーターで最大 **10,000** まで変更可能です。
- **outputs:** ジョブの出力データを保存します。 ローカル **warehouse** が出力テーブルの場合、出力結果データはジョブ実行後にローカル **warehouse** の対応するテーブルを上書きします。
- **resources:** ジョブが使用するリソースを保存します。 **inputs** 同様、データは優先的にローカル **warehouse** から取得されます。 データがローカルに存在しない場合、データは **MaxCompute SDK** を使用してサーバーから **(odps.end.point** が設定されている場合) 読み取られます。
- **job.xml:** ジョブの設定を示します。
- **superstep:** 反復の度にメッセージ永続化に関する情報を保存します。

> 注 :
> ローカルデバッグ中に詳細ログの出力が必要な場合、以下の **log4j** 設定ファイルが **src** ディレクトリ `log4j.properties_odps_graph_cluster_debug` に配置されている必要があります。

クラスタデバッグ

ローカルデバッグ後、テストのためにクラスタへジョブを送信します。

手順

1. **MaxCompute** クライアントを設定します。

2. `add jar /path/work.jar -f;` コマンドを実行し、**jar** パッケージを更新します。
3. **jar** コマンドを実行してジョブを実行し、実行ログと結果データを確認します。



注：
クラスタで **Graph** を実行する方法の詳細については、「クイックスタート」 の「*Graph の実行*」をご参照ください。

パフォーマンスチューニング

次のセクションでは、**MaxCompute Graph** フレームワーク上での一般的なパフォーマンスチューニング方法を説明します。

ジョブパラメーターの設定

パフォーマンスに影響を与える **GraphJob** の設定には、以下のものがあります。

- **setSplitSize(long):** 入力テーブルの分割サイズを示します。 単位は **MB** です。 値は **0** より大きくなければならず、デフォルト値は **64** です。
- **setNumWorkers(int):** ジョブの **Worker** 数を指定します。 値の範囲は **[1,1000]** で、デフォルト値は **–1** です。 **Worker** 数はジョブの入力バイト数と分割サイズによって異なります。
- **setWorkerCPU(int): Map** の **CPU** リソースを示します。 **1** コア **CPU** は **100** リソースを含んでいます。 値の範囲は **[50, 800]** で、デフォルト値は **200** です。
- **setWorkerMemory(int): Map** のメモリリソースを示します。 単位は **MB** です。 値の範囲は **[256 MB, 12 GB]** で、デフォルト値は **4,096 MB** です。
- **setMaxIteration(int):** 反復の最大数を指定します。 デフォルト値は **–1** です。 値が **0** 以下の場合、反復の最大数はジョブ終了条件となりません。
- **setJobPriority(int):** ジョブの優先度を指定します。 値の範囲は **[0, 9]** で、デフォルト値は **9** です。 値が大きいほど、優先順位は低くなります。

全体的な処理能力を向上させるための追加アクションは以下のとおりです:

- **Worker** 数を増やすには、**setNumWorkers()** メソッドを使用することができます。
- 分割サイズを縮小し、ジョブがデータをロードする速度を上げるには、**setSplitSize()** メソッドを使用することができます。
- **Worker** の **CPU** またはメモリを増やします。
- 反復の最大数を設定します。 アプリケーションが結果の精度について高い要件を持たない場合は、反復の回数を減らしてプロセスをスピードアップさせることができます。

データのロード速度を上げるには、**setNumWorkers** と **setSplitSize** インターフェイスを一緒に使用します。 たとえば、**setNumWorkers** が **workerNum**、**setSplitSize** が **splitSize**、入力バイトの総数が **inputSize** とします。 分割数は、計算式**: splitNum** = **inputSize /splitSize** を使用して計算されます。 **workerNum** と **splitNum** の関係は以下のとおりです。

- **splitNum** == **workerNum** の場合、各 **Worker** は **1** つの分割をロードします。
- **splitNum** > **workerNum** の場合、各 **Worker** は **1** つ以上の分割をロードします。
- **splitNum** < **workerNum** の場合、各 **Worker** は **0** または **1** つの分割をロードします。

そのため、最初の **2** つの条件に一致する場合、**workerNum** と **splitSize** を調整することで高速データロードを実現できます。反復フェーズでは、 **workerNum** だけを調整する必要があります。

ランタイムパーティショニングを **false** に設定する場合、 **setSplitSize** を使用して **Worker** 数を制御することを推奨します。 **3** つめの条件については、一部の **Worker** の頂点の数が **0** であるかもしれません。**jar** コマンドの前に、 **setNumWorkers** と **setSplitSize** 同様の効果がある **set odps.graph.split.size=<m>; set odps.graph.worker.num=<n>;** を使用できます。

もう **1** つの共通するパフォーマンスの問題はデータスキューです。 たとえば、カウンターでは、ある **Worker** で処理される頂点と辺の数が、他の **Worker** で処理される量を大きく上回ることがあります。

データスキューは通常、キーに対応する頂点、辺またはメッセージの数が、他のキーを大きく上回る場合に発生します。 データ量の多いキーは、少数の **Worker** で処理され、結果的にこれらの **Worker** の実行時間は長くなります。

この問題を解決するには、以下のステップを推奨します。

- コンバイナを使用して、そのようなキーに対応する頂点のメッセージをローカルで集計し、送信メッセージ数を減らします。
- サービスロジックを改善します。

コンバイナを使用する

メッセージとネットワークデータトラフィック量を保存するメモリを縮小し、ジョブの実行時間を短縮するため、コンバイナを定義します。 詳しい情報は、**MaxCompute SDK** の 「コンバイナ」 をご参照ください。

データ入力量を減らす

データ量が大きい場合、ディスクのデータ読み取りが処理時間を増大させることがあります。
そのため、読み取るデータのバイト数を減らすことで全体的なスループット、つまりジョブのパフォーマンスを向上させることができます。次のいずれかの方法を使用できます。

- 入力データ量を減らす: 意思決定アプリケーションでは、データのサンプリング後にサブセットを処理することで取得できる結果は、全体の精度ではなく結果の精度にのみ影響します。そのため、特別なデータサンプリングを実行し、データを入力テーブルにインポートして処理することができます。
- 使用されていないフィールドの読み取りを回避する: **MaxCompute Graph** フレームワークの **Tableinfo** クラスは、全テーブルの読み取りやテーブルパーティションの読み取りではなく、特定の列 (列名配列を使用して送信される) の読み取りをサポートします。 これにより入力データ量が減り、ジョブのパフォーマンスが向上します。

組み込み jar **パッケージ**

デフォルトで、以下の **jar** パッケージは、**Graph** プログラムを実行する **JVM** にロードされます。 これらのリソースをアップロードしたり、コマンドラインで **-libjars** を実行する際にこれらの **jar** パッケージを持つ必要はありません。

- **commons-codec-1.3.jar**
- **commons-io-2.0.1.jar**
- **commons-lang-2.5.jar**
- **commons-logging-1.0.4.jar**
- **commons-logging-api-1.0.4.jar**
- **guava-14.0.jar**
- **json.jar**
- **log4j-1.2.15.jar**
- **slf4j-api-1.4.3.jar**
- **slf4j-log4j12-1.4.3.jar**
- **xmlenc-0.52.jar**

 注 :

JVM を実行するクラスパスでは、 上述のビルトイン **jar** パッケージはユーザーの **jar** パッケージより前に配置され、バージョンの競合が発生する可能性があります。 たとえば、プログラムで **commons-codec-1.5.jar** 内のクラスの関数を使用していても、この関数は **commons-codec-1.3.jar** には含まれていません。 実装メソッドが **commons-codec-1.3.jar** に存在するかどうかを確認するか、**MaxCompute** がサポート対象バージョンへアップデートするまでお待ちください。

10.7 例

10.7.1 SSSP

Dijkstra は有向グラフの **SSSP (Single Source Shortest Path)** を計算する典型的なアルゴリズムです。

重み付き有向グラフ **G=(V,E)** では、多くの経路がソース頂点 **s** からシンク頂点 **v** に向かっています。これらの経路では、辺の重みの合計が最小となる頂点が **s** から **v** の最短距離と呼ばれます。

アルゴリズムの基本概念は以下のとおりです。

- 初期化**:** ソース頂点 **s** から **s** 自身への距離はゼロ **(d[s] = 0)** で、他の頂点 **u** から **s** までの距離は無限大 **(d[u]=∞)** です。
- 反復**: u** から **v** への辺が存在する場合、**s** から **v** への最短距離は **d[v] = min(d[v], d[u] + weight(u, v))** として更新されます。 すべての頂点から **s** までの距離が 変化しなくなると、反復を終了します。

このアルゴリズムの基本概念は、**MaxCompute Graph** プログラムを使用するソリューションに適用可能なことを示しています。 各頂点はソース頂点までの現在の最短距離を保持します。 値が変わると、新たな値と辺の重みを含むメッセージが、隣接する頂点に対して送信されます。 次の反復では、受け取ったメッセージに基づいて隣接する頂点が現在の最短距離を更新します。 現在のすべての頂点の最短距離が変化しなくなると、反復を終了します。

コード例

SSSP のコードは以下のとおりです。

```
import java.io.IOException;

import com.aliyun.odps.io.WritableRecord;
import com.aliyun.odps.graph.Combiner;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.Edge;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.data.TableInfo;

public class SSSP {

  public static final String START_VERTEX = "sssp.start.vertex.id";

  public static class SSSPVertex extends
      Vertex<LongWritable, LongWritable, LongWritable, LongWritable> {

    private static long startVertexId = -1;

    public SSSPVertex() {
```

```
      this.setValue(new LongWritable(Long.MAX_VALUE));


    public boolean isStartVertex(
        ComputeContext<LongWritable, LongWritable, LongWritable,
LongWritable> context) {
      if (startVertexId == -1) {
        String s = context.getConfiguration().get(START_VERTEX);
        startVertexId = Long.parseLong(s);

      return getId().get() == startVertexId;


    @Override
    public void compute(
        ComputeContext<LongWritable, LongWritable, LongWritable,
LongWritable> context,
        Iterable<LongWritable> messages) throws IOException {
      long minDist = isStartVertex(context) ? 0 : Integer.MAX_VALUE;
      for (LongWritable msg : messages) {
        if (msg.get() < minDist) {
          minDist = msg.get();



      if (minDist < this.getValue().get()) {
        this.setValue(new LongWritable(minDist));
        if (hasEdges()) {
          for (Edge<LongWritable, LongWritable> e : this.getEdges()) {
            context.sendMessage(e.getDestVertexId(), new LongWritable(
minDist
                + e.getValue().get()));


      } else {
        voteToHalt();



    @Override
    public void cleanup(
        WorkerContext<LongWritable, LongWritable, LongWritable,
LongWritable> context)
        throws IOException {
      context.write(getId(), getValue());



  public static class MinLongCombiner extends
      Combiner<LongWritable, LongWritable> {

    @Override
    public void combine(LongWritable vertexId, LongWritable combinedMe
ssage,
        LongWritable messageToCombine) throws IOException {
      if (combinedMessage.get() > messageToCombine.get()) {
        combinedMessage.set(messageToCombine.get());




  public static class SSSPVertexReader extends
```

```
      GraphLoader<LongWritable, LongWritable, LongWritable, LongWritab
le> {

    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, LongWritable, LongWritable,
LongWritable> context)
        throws IOException {
      SSSPVertex vertex = new SSSPVertex();
      vertex.setId((LongWritable) record.get(0));
      String[] edges = record.get(1).toString().split(",");
      for (int i = 0; i < edges.length; i++) {
        String[] ss = edges[i].split(":");
        vertex.addEdge(new LongWritable(Long.parseLong(ss[0])),
            new LongWritable(Long.parseLong(ss[1])));


      context.addVertexRequest(vertex);




  public static void main(String[] args) throws IOException {
    if (args.length < 2) {
      System.out.println("Usage: <startnode> <input> <output>");
      System.exit(-1);


    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(SSSPVertexReader.class);
    job.setVertexClass(SSSPVertex.class);
    job.setCombinerClass(MinLongCombiner.class);

    job.set(START_VERTEX, args[0]);
    job.addInput(TableInfo.builder().tableName(args[1]).build());
    job.addOutput(TableInfo.builder().tableName(args[2]).build());

    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

SSSP のソースコードの詳細を以下に示します。

- **19** 行目**: SSSPVertex** を定義します。
  - 頂点の値は、この頂点からソース頂点 **startVertexId** までの現在の最短距離を示します。
  - **compute()** メソッドは、反復計算式 **d[v] = min(d[v], d[u] + weight(u, v))** を使用して頂点の値を更新します。
  - **cleanup()** メソッドは、頂点とソース頂点までの最短距離を結果テーブルに書き込みます。

- **58** 行目**:** 頂点の値が変化しない場合、**voteToHalt()** が呼び出され、頂点が停止状態に変わったことをフレームワークに通知します。 すべての頂点が 停止状態に変わると、この計算は終了します。
- **70** 行目**: MinLongCombiner** を定義し、同一頂点に送信されるメッセージを組み合わせてパフォーマンスを最適化し、メモリの使用を減らします。
- **83** 行目**: SSSPVertexReader** クラスを定義し、グラフをロードし、 テーブルの各レコードを頂点に分解します。 レコードの最初の列は頂点 **ID** で、**2** 列目は **2:2**、**3:1**、**4:4** のように、頂点を始点とするすべての辺のセットを保存します。
- **106** 行目**:** メインプログラム **(main** 関数**)** を実行し、**GraphJob** を定義し、 **Vertex/GraphLoader/Combiner** および入出力テーブルの実装を指定します。

10.7.2 PageRank

PageRank は **Web** ページのランキングを計算する典型的なアルゴリズムです。 入力有向グラフ **G** において、頂点は **Web** ページを示します。 **Web** ページ **A** と **B** の間にリンクが存在する場合、**A** と **B** を結ぶ辺が存在します。

アルゴリズムの基本概念は以下のとおりです。

- 初期化**:** 頂点の値は **PageRank** のランク値 **(double** 型**)** を示します。 初期段階では、すべての頂点の値は **1/TotalNumVertices** です。
- 反復計算式**: PageRank(i) = 0.15/TotalNumVertices + 0.85 x sum**。 **sum** は、**PageRank(j)/out_degree(j)** の合計を示します。**j** は頂点 **i** を指すすべての頂点を示します。

このアルゴリズムの基本概念は、**MaxCompute Graph** プログラムを使用するソリューションに適用可能なことを示しています。 各頂点 **j** は **PageRank** の値を保持します。 **PageRank(j)/out_degree(j)** は、反復の度に、隣接する頂点 **(**投票用**)** に送信されます。 次の反復では、各頂点は反復計算式を使って **PageRank** 値を再計算します。

サンプルコード

```
import java.io.IOException;

import org.apache.log4j.Logger;

import com.aliyun.odps.io.WritableRecord;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
```

```
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.io.Writable;

public class PageRank {

  private final static Logger LOG = Logger.getLogger(PageRank.class);

  public static class PageRankVertex extends
      Vertex<Text, DoubleWritable, NullWritable, DoubleWritable> {

    @Override
    public void compute(
        ComputeContext<Text, DoubleWritable, NullWritable, DoubleWrit
able> context,
        Iterable<DoubleWritable> messages) throws IOException {
      if (context.getSuperstep() == 0) {
        setValue(new DoubleWritable(1.0 / context.getTotalNumVertices
()));
      } else if (context.getSuperstep() >= 1) {
        double sum = 0;
        for (DoubleWritable msg : messages) {
          sum += msg.get();

        DoubleWritable vertexValue = new DoubleWritable(
            (0.15f / context.getTotalNumVertices()) + 0.85f * sum);
        setValue(vertexValue);

      if (hasEdges()) {
        context.sendMessageToNeighbors(this, new DoubleWritable(
getValue()
            .get() / getEdges().size()));



    @Override
    public void cleanup(
        WorkerContext<Text, DoubleWritable, NullWritable, DoubleWrit
able> context)
        throws IOException {
      context.write(getId(), getValue());



  public static class PageRankVertexReader extends
      GraphLoader<Text, DoubleWritable, NullWritable, DoubleWritable>
 {

    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<Text, DoubleWritable, NullWritable, DoubleWrit
able> context)
        throws IOException {
      PageRankVertex vertex = new PageRankVertex();
      vertex.setValue(new DoubleWritable(0));
      vertex.setId((Text) record.get(0));
      System.out.println(record.get(0));

      for (int i = 1; i < record.size(); i++) {
        Writable edge = record.get(i);
        System.out.println(edge.toString());
        if (!( edge.equals(NullWritable.get()))) {
```

```
          vertex.addEdge(new Text(edge.toString()), NullWritable.get
());


      LOG.info("vertex edgs size: "
          + (vertex.hasEdges() ? vertex.getEdges().size() : 0));
      context.addVertexRequest(vertex);




  private static void printUsage() {
    System.out.println("Usage: <in> <out> [Max iterations (default 30
)]");
    System.exit(-1);


  public static void main(String[] args) throws IOException {
    if (args.length < 2)
      printUsage();

    GraphJob job = new GraphJob();

    job.setGraphLoaderClass(PageRankVertexReader.class);
    job.setVertexClass(PageRankVertex.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());

    // default max iteration is 30
    job.setMaxIteration(30);
    if (args.length >= 3)
      job.setMaxIteration(Integer.parseInt(args[2]));

    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

PageRank のソースコードの詳細を以下に示します。

- **23** 行目**: PageRankVertex** を定義します。
  - 頂点の値は、現在の頂点 **(Web** ページ**)** の **PageRank** 値を示します。
  - **compute()** メソッド は、反復計算式 `PageRank(i) = 0.15/TotalNumVertices + 0.85 x sum` を使用して頂点の値を更新します。
  - **cleanup()** メソッドは、頂点と **PageRank** 値を結果テーブルに書き込みます。
- **55** 行目**: PageRankVertexReader** クラスを定義し、グラフをロードし、テーブルの各レコードを頂点に分解します。 レコードの最初の列は開始頂点で、他の列は到達頂点です。
- **88** 行目**:** メインプログラム **(main** 関数**)** を実行し、**GraphJob** を定義し、**Vertex/GraphLoader**、反復最大数 **(**デフォルトは **30)** および 入出力テーブルの実装を指定します。

10.7.3 K-means

K-means アルゴリズムは、典型的なクラスタリングアルゴリズムです。

これは、空間内の **k** 個の頂点を中心として使用し、それらに最も近い頂点をグループ化することによってクラスタリングを実行します。 クラスタリングの中心値は、最適なクラスタリング結果が得られるまで、反復しながら連続的に更新されます。

サンプルの集合を k 個のクラスに分割するため、アルゴリズムは以下のように動作します。

1. **k** 個のクラスの中心の初期値を選択します。
2. 任意のサンプルから **k** 個の中心までの距離を反復 **i** で 計算し、サンプルを最も近い中心のクラスにグループ化します。
3. 平均や他の方法を使ってクラスの中心値を更新します。
4. すべての k 個のクラスター中心について、反復後に更新された値が変化しないままであるか、しきい値よりも小さい場合、反復は終了します。 そうでない場合、反復は継続します。

サンプルコード

K-means クラスタリングアルゴリズムのコードは以下のとおりです。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.logging.log4j.Logger;

import com.aliyun.odps.io.WritableRecord;
import com.aliyun.odps.graph.Aggregator;
Import com. aliyun. ODPS. graph. computercontext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.io.Tuple;
import com.aliyun.odps.io.Writable;

public class Kmeans {
  private final static Logger LOG = Logger.getLogger(Kmeans.class);

  public static class KmeansVertex extends
  Vertex<Text, Tuple, NullWritable, NullWritable> {

    @ Override
    public void compute(
    ComputeContext<Text, Tuple, NullWritable, NullWritable> context,
    Iterable<NullWritable> messages) throws IOException {
      context.aggregate(getValue());
    }
```

```
  public static class KmeansVertexReader extends
  GraphLoader<Text, Tuple, NullWritable, NullWritable> {
    @Override
    public void load(LongWritable recordNum, WritableRecord record,
    MutationContext<Text, Tuple, NullWritable, NullWritable> context)
        throws IOException {
      KmeansVertex vertex = new KmeansVertex();
      vertex.setId(new Text(String.valueOf(recordNum.get())));
      vertex.setValue(new Tuple(record.getAll()));
      context.addVertexRequest(vertex);



  public static class KmeansAggrValue implements Writable {

    Tuple centers = new Tuple();
    Tuple sums = new Tuple();
    Tuple counts = new Tuple();

    @ Override
    public void write(DataOutput out) throws IOException {
      centers.write(out);
      sums.write(out);
      counts.write(out);


    @Override
    public void readFields(DataInput in) throws IOException {
      centers = new Tuple();
      centers.readFields(in);
      sums = new Tuple();
      sums.readFields(in);
      counts = new Tuple();
      counts.readFields(in);


    @Override
    public String toString () {
      return "centers " + centers.toString() + ", sums " + sums.
toString()
          + ", counts " + counts.toString();



public static class KmeansAggregator extends Aggregator<KmeansAggr
Value> {

    @SuppressWarnings("rawtypes")
    @Override
    public KmeansAggrValue createInitialValue(WorkerContext context)
        throws IOException {
      KmeansAggrValue aggrVal = null;
      if (context.getSuperstep() == 0) {
        aggrVal = new KmeansAggrValue();
        aggrVal.centers = new Tuple();
        aggrVal.sums = new Tuple();
        aggrVal.counts = new Tuple();

        byte[] centers = context.readCacheFile("centers");
```

```
        String lines[] = new String(centers).split("\n");

for (int i = 0; i < lines.length; i++) {
          String[] ss = lines[i].split(",");
          Tuple center = new Tuple();
          Tuple sum = new Tuple();
          for (int j = 0; j < ss.length; ++j) {
            center.append(new DoubleWritable(Double.valueOf(ss[j].trim
()));
            sum.append(new DoubleWritable(0.0));

          LongWritable count = new LongWritable(0);
          aggrVal.sums.append(sum);
          aggrVal.counts.append(count);
          aggrVal.centers.append(center);

      } else{
        aggrVal = (KmeansAggrValue) context.getLastAggregatedValue(0);


      return aggrVal;


    @Override
    Public void aggregate (glasvalue, object item ){
      int min = 0;
      double mindist = Double.MAX_VALUE;
      Tuple point = (Tuple) item;

for (int i = 0; i < value.centers.size(); i++) {
        Tuple center = (Tuple) value.centers.get(i);
        // use Euclidean Distance, no need to calculate sqrt
        double dist = 0.0d;
        for (int j = 0; j < center.size(); j++) {
          double v = ((DoubleWritable) point.get(j)).get()
              - ((DoubleWritable) center.get(j)).get();
          dist += v * v;

        if (dist < mindist) {
          mindist = dist;
          min = i;



      // update sum and count
      Tuple sum = (Tuple) value.sums.get(min);
      for (int i = 0; i < point.size(); i++) {
        DoubleWritable s = (DoubleWritable) sum.get(i);
        s.set(s.get() + ((DoubleWritable) point.get(i)).get());

      LongWritable count = (LongWritable) value.counts.get(min);
      count.set(count.get() + 1);


    @Override
    public void merge(KmeansAggrValue value, KmeansAggrValue partial)
 {
    for (int i = 0; i < value.sums.size(); i++) {
        Tuple sum = (Tuple) value.sums.get(i);
        Tuple that = (Tuple) partial.sums.get(i);
        for (int j = 0; j < sum.size(); j++) {
          DoubleWritable s = (DoubleWritable) sum.get(j);
          s.set(s.get() + ((DoubleWritable) that.get(j)).get());
```

```
for (int i = 0; i < value.counts.size(); i++) {
        LongWritable count = (LongWritable) value.counts.get(i);
        count.set(count.get() + ((LongWritable) partial.counts.get(i
)).get());



    @SuppressWarnings("rawtypes")
    @Override
    public boolean terminate(WorkerContext context, KmeansAggrValue
value)
        throws IOException {

      // compute new centers
      Tuple newCenters = new Tuple(value.sums.size());
      for (int i = 0; i < value.sums.size(); i++) {
        Tuple sum = (Tuple) value.sums.get(i);
        Tuple newCenter = new Tuple(sum.size());
        LongWritable c = (LongWritable) value.counts.get(i);
        for (int j = 0; j < sum.size(); j++) {

          DoubleWritable s = (DoubleWritable) sum.get(j);
          double val = s.get() / c.get();
          newCenter.set(j, new DoubleWritable(val));

          // reset sum for next iteration
          s.set(0.0d);

        // reset count for next iteration
        c.set(0);
        newCenters.set(i, newCenter);


      // update centers
      Tuple oldCenters = value.centers;
      value.centers = newCenters;

      LOG.info("old centers: " + oldCenters + ", new centers: " +
newCenters);

      // compare new/old centers
      boolean converged = true;
      for (int i = 0; i < value.centers.size() && converged; i++) {
        Tuple oldCenter = (Tuple) oldCenters.get(i);
        Tuple newCenter = (Tuple) newCenters.get(i);
        double sum = 0.0d;
        for (int j = 0; j < newCenter.size(); j++) {
          double v = ((DoubleWritable) newCenter.get(j)).get()
              - ((DoubleWritable) oldCenter.get(j)).get();
          sum += v * v;

        double dist = Math.sqrt(sum);
        LOG.info("old center: " + oldCenter + ", new center: " +
newCenter
            + ", dist: " + dist);
        // converge threshold for each center: 0.05
        converged = dist < 0.05d;


      if (converged || context.getSuperstep() == context.getMaxIter
ation() - 1) {
        // converged or reach max iteration, output centers
```

```
        for (int i = 0; i < value.centers.size(); i++) {
          context.write(((Tuple) value.centers.get(i)).toArray());

        // true means to terminate iteration
        return true;


      // false means to continue iteration
      return false;



  private static void printUsage() {
  System. out. println ("Usage: <in> <out> [Max iterations (default 30
)] ");
    System.exit(-1);


  public static void main (final String [] args) throws IOException{
  if (args.length < 2)
      printUsage();

    GraphJob job = new GraphJob();

    job.setGraphLoaderClass(KmeansVertexReader.class);
    job.setRuntimePartitioning(false);
    job.setVertexClass(KmeansVertex.class);
    job.setAggregatorClass(KmeansAggregator.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());

    // default max iteration is 30
    job.setMaxIteration(30);
    if (args.length >= 3)
      job.setMaxIteration(Integer.parseInt(args[2]));

    long start = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - start) / 1000.0 + " seconds");
```

以下は、**K-means** のソースコードについての説明です。

- **26** 行目**: KmeansVertex** を定義します。 **compute ()** メソッドの実装はシンプルです。 コンテキストオブジェクトの **aggregate ()** メソッドを呼び出します。 次に、現在の頂点の値 **(**タプル 型で、ベクトルで表現される**)** を送信します。
- **38** 行目**: KmeansVertexReader** クラスを定義し、グラフを読み込み、テーブル内の各レコードを頂点と見なします。 頂点 **ID** は関係ありません。送信された **recordNum** は **ID** として使用されます。 頂点の値は、レコードのすべての列から構成されるタプルです。

- **83** 行目**: KmeansAggregator** を定義します。 このクラスは、**K-means** アルゴリズムの主なロジックをカプセル化します。ここで、
  - **createInitialValue** は、反復ごとに初期値を作成します **(k** クラス 中心点**)**。 最初の反復 **(superstep** が **0)** では、値は中心点の初期値です。 それ以外の場合は、値は最後の反復が終了したときの新しい中心点です。
  - **aggregate ()** メソッドは、各頂点から異なるクラスの中心までの距離を計算し、最も近い中心のクラスとしてその頂点を分類し、そのクラスの合計とカウントを更新します。
  - **merge ()** メソッドは、各 **Worker** によって収集された合計とカウントを組み合わせます。
  - **terminate ()** メソッドは、各クラスの合計とカウントに基づいて新しい中心点を計算します。 新しい中心点と古い中心点の間の距離がしきい値より小さいか、または反復回数が上限値に達すると、反復は終了します **(false** が返されます**)**。 最終的な中心点が結果テーブルに書き込まれます。
- **236** 行目**:** メインプログラム **(main** 関数**)** を実行し、**GraphJob** を定義し、そして **Vertex/GraphLoader/Aggregator** の実装を指定します。 最大反復数 **(**デフォルトは **30)**、および入力テーブルと出力テーブル。
- **243** 行目**: job.setRuntimePartitioning (false)** を指定します。 **K-means** アルゴリズムでは、グラフの読み込み中に頂点を分散させる必要はありません。 **RuntimePartitioning** が **false** に設定されると、グラフ読み込みのパフォーマンスが向上します。

10.7.4 2 部マッチング

2 部グラフとは、すべてのグラフの頂点を **2** つの集合に分割でき、各辺に対応する **2** つの頂点がそれぞれ **2** つの集合に属することを意味します。 **2** 部グラフ **G** において、**M** はそのサブグラフの **1** つです。 **M** の辺集合のいずれかの **2** つの辺が 同じ頂点に隣接されていない場合、**M** はマッチングと呼ばれます。 通常、**2** 部グラフマッチングは、需給関係が明確なシナリオでの情報マッチングに使用されます。

アルゴリズムの基本概念は以下のとおりです。

- 左側の最初の頂点からマッチングしていない頂点を選択して、増加道を検索します。
- マッチングしていない頂点が見つかった場合、検索は成功です。
- 道の情報は更新されます。 マッチングする辺の数が **1** つ増えた場合、検索は停止します。
- 増加道が見つからない場合、検索はこの頂点から開始されなくなります。

サンプルコード

2 部マッチング アルゴリズムのコードは以下のとおりです。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.util.Random;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.io.WritableRecord;
public class BipartiteMatching {
  private static final Text UNMATCHED = new Text("UNMATCHED");
  public static class TextPair implements Writable {
    public Text first;
    public Text second;
    public TextPair() {
      first = new Text();
      second = new Text();

    public TextPair(Text first, Text second) {
      this.first = new Text(first);
      this.second = new Text(second);

    @ Override
    public void write(DataOutput out) throws IOException {
      first.write(out);
      second.write(out);

    @ Override
    public void readFields(DataInput in) throws IOException {
      first = new Text();
      first.readFields(in);
      second = new Text();
      second.readFields(in);

    @ Override
    public String toString () {
      return first + ": " + second;


  public static class BipartiteMatchingVertexReader extends
      GraphLoader<Text, TextPair, NullWritable, Text> {
    @ Override
    public void load(LongWritable recordNum, WritableRecord record,
        MutationContext<Text, TextPair, NullWritable, Text> context)
        throws IOException {
      BipartiteMatchingVertex vertex = new BipartiteMatchingVertex();
      vertex.setId((Text) record.get(0));
      vertex.setValue(new TextPair(UNMATCHED, (Text) record.get(1)));
      String[] adjs = record.get(2).toString().split(",");
      for (String adj : adjs) {
        vertex.addEdge(new Text(adj), null);
```

```
      context.addVertexRequest(vertex);


  public static class BipartiteMatchingVertex extends
  Vertex <Text, TextPair, NullWritable, Text> {
    private static final Text LEFT = new Text("LEFT");
    private static final Text RIGHT = new Text("RIGHT");
    private static Random rand = new Random();
    @ Override
    public void compute (
    ComputeContext<Text, TextPair, NullWritable, Text> context,
        Iterable messages) throws ioexception {
      if (isMatched()) {
        voteToHalt();
        return;

      switch ((int) context.getSuperstep() % 4) {
      case 0:
        if (isLeft()) {
          context.sendMessageToNeighbors(this, getId());

        break;
      case 1:
        if (isRight()) {
          Text luckyLeft = null;
          for (Text message : messages) {
            if (luckyLeft == null) {
              luckyLeft = new Text(message);
            } else{
              if (rand.nextInt(1) == 0) {
                luckyLeft.set(message);



          if (luckyLeft ! = null) {
            context.sendMessage(luckyLeft, getId());


        break;
      case 2:
        if (isLeft()) {
          Text luckyRight = null;
          for (Text msg : messages) {
            if (luckyRight == null) {
              luckyRight = new Text(msg);
            } else{
              if (rand.nextInt(1) == 0) {
                luckyRight.set(msg);



          if (luckyRight ! = null) {
            setMatchVertex(luckyRight);
            context.sendMessage(luckyRight, getId());


        break;
      case 3:
        if (isRight()) {
          for (Text msg : messages) {
            setMatchVertex(msg);


        break;
```

```
    @ Override
    public void cleanup(
    WorkerContext<Text, TextPair, NullWritable, Text> context)
        throws IOException {
      context.write(getId(), getValue().first);

    private boolean isMatched() {
      return ! getValue().first.equals(UNMATCHED);

    private boolean isLeft() {
      return getValue().second.equals(LEFT);

    private boolean isRight() {
      return getValue().second.equals(RIGHT);

    private void setMatchVertex(Text matchVertex) {
      getValue().first.set(matchVertex);


  private static void printUsage() {
    System.err.println("BipartiteMatching <input> <output> [maxIterati
on]");

  public static void main (final String [] args) throws IOException{
    if (args.length < 2) {
      printUsage();

    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(BipartiteMatchingVertexReader.class);
    job.setVertexClass(BipartiteMatchingVertex.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    int maxIteration = 30;
    if (args.length > 2) {
      maxIteration = Integer.parseInt(args[2]);

    job.setMaxIteration(maxIteration);
    job.run();
```

10.7.5 強連結成分

有向グラフでは、任意の頂点から始まり、辺を通ってグラフ内のすべての頂点に到達する場合、強連結グラフと呼ばれます。 極大な頂点数を持つ強連結サブグラフは強連結成分と呼ばれます。アルゴリズムは彩色並列アルゴリズムに基づいています。

各頂点は以下の成分を含みます。

- **colorID:** 順方向走査中の頂点 **v** の色 を保存します。 計算が終了した後、同じ **colorID** を持つ頂点は、**1** つの強連結成分に属します。
- **transposeNeighbors:** 入力グラフの転置グラフに頂点 **v** の隣接頂点 **ID** を保存します。

アルゴリズムは以下の **4** ステップから構成されます。

- 転置グラフの生成**:** 2 つのスーパーステップから構成されます。 各頂点は、対応する出力辺をもつ隣接頂点に **ID** を送信します。 次のスーパーステップで、これらの **ID** は **transposeNeighbors** 値として保存されます。
- **Trim:** 1 つのスーパーステップから構成されます。 入力辺または出力辺が **1** つだけの各頂点は、それ自身の **ID** として **colorID** を設定し、ステータスを非アクティブにします。 頂点に送信される後続の信号は無視されます。
- 順方向走査**:** 頂点は、スタートアップとスリープの **2** つのサブプロセス (スーパーステップ) から構成されます。 スタートアップフェーズでは、各頂点は **colorID** を自身の **ID** として設定し、その **ID** を 対応する出力辺を持つ隣接頂点へ送信します。 スリープフェーズでは、頂点は受信した最大の **colorID** でそれ自身の **colorID** を更新し、**colorID** が収束するまで **colorID** を送信します。 **colorID** が収束したとき、 マスタープロセスは、グローバルオブジェクトを逆方向走査に設定します。
- 逆方向走査**:** スタートアップとスリープの **2** つのサブプロセスが含まれています。 スタートアップフェーズでは、**ID** が **colorID** と同じ頂点がその **ID** を 転置グラフの隣接頂点に送信し、ステータスを非アクティブにします。 頂点に送られる後続の信号は無視されます。 各スリープステップで、各頂点はその **colorID** と一致するシグナルを受け取り、 転置グラフで **colorID** を送信してから、ステータスを非アクティブにします。 このステップの終了後にアクティブな頂点が存在する場合、プロセスはトリムステップに戻ります。

サンプルコード

強連結成分のコードは以下のとおりです。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.Aggregator;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.BooleanWritable;
import com.aliyun.odps.io.IntWritable;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.Tuple;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.io.WritableRecord;

 * Definition from Wikipedia:
 * In the mathematical theory of directed graphs, a graph is said
 * to be strongly connected if every vertex is reachable from every
 * other vertex. The strongly connected components of an arbitrary
 * directed graph form a partition into subgraphs that are themselves
 * Strictly connected.
```

```
 * Algorithms with four phases as follows.
 * 1. Transpose Graph Formation: Requires two supersteps. In the first
 * superstep, each vertex sends a message with its ID to all its
outgoing
 * neighbors, which in the second superstep are stored in transposeN
eighbors.

 * 2. Trimming: Takes one superstep. Every vertex with only in-coming
or
 * only outgoing edges (or neither) sets its colorID to its own ID and
 * becomes inactive. Messages subsequently sent to the vertex are
ignored.

 * 3. Forward-Traversal: There are two sub phases: Start and Rest. In
the
 * Start phase, each vertex sets its colorID to its own ID and
propagates
 * its ID to its outgoing neighbors. In the Rest phase, vertices
update
 * their own colorIDs with the minimum colorID they have seen, and
propagate
 * their colorIDs, if updated, until the colorIDs converge.
 * Set the phase to Backward-Traversal when the colorIDs converge.

 * 4. Backward-Traversal: We again break the phase into Start and Rest
.
 * In Start, every vertex whose ID equals its colorID propagates its
ID to
 * the vertices in transposeNeighbors and sets itself inactive.
Messages
 * subsequently sent to the vertex are ignored. In each of the Rest
phase supersteps,
 * each vertex receiving a message that matches its colorID: (1)
propagates
 * its colorID in the transpose graph; (2) sets itself inactive.
Messages
 * subsequently sent to the vertex are ignored. Set the phase back to
Trimming
 * if not all vertex are inactive.

 * http://ilpubs.stanford.edu:8090/1077/3/p535-salihoglu.pdf

public class StronglyConnectedComponents {
  public final static int STAGE_TRANSPOSE_1 = 0;
  public final static int STAGE_TRANSPOSE_2 = 1;
  public final static int STAGE_TRIMMING = 2;
  public final static int STAGE_FW_START = 3;
  public final static int STAGE_FW_REST = 4;
  public final static int STAGE_BW_START = 5;
  public final static int STAGE_BW_REST = 6;

   * The value is composed of component id, incoming neighbors,
   * active status and updated status.

  public static class MyValue implements Writable {
    LongWritable sccID;// strongly connected component id
    Tuple inNeighbors; // transpose neighbors
    BooleanWritable active; // vertex is active or not
    BooleanWritable updated; // sccID is updated or not
    public MyValue() {
      this.sccID = new LongWritable(Long.MAX_VALUE);
      this.inNeighbors = new Tuple();
      this.active = new BooleanWritable(true);
```

```
      this.updated = new BooleanWritable(false);

    public void setSccID(LongWritable sccID) {
      this.sccID = sccID;

    public LongWritable getSccID() {
      return this.sccID;

    public void setInNeighbors(Tuple inNeighbors) {
      this.inNeighbors = inNeighbors;

    public Tuple getInNeighbors() {
      return this.inNeighbors;

    public void addInNeighbor(LongWritable neighbor) {
      this.inNeighbors.append(new LongWritable(neighbor.get()));

    public boolean isActive() {
      return this.active.get();

    public void setActive(boolean status) {
      this.active.set(status);

    public boolean isUpdated() {
      return this.updated.get();

    public void setUpdated(boolean update) {
      this.updated.set(update);

    @Override
    public void write(DataOutput out) throws IOException {
      this.sccID.write(out);
      this.inNeighbors.write(out);
      this.active.write(out);
      this.updated.write(out);

    @Override
    public void readFields(DataInput in) throws IOException {
      this.sccID.readFields(in);
      this.inNeighbors.readFields(in);
      this.active.readFields(in);
      this.updated.readFields(in);

    @Override
    public String toString() {
      StringBuilder sb = new StringBuilder();
      sb.append("sccID: " + sccID.get());
      sb.append(" inNeighbores: " + inNeighbors.toDelimitedString
(','));
      sb.append(" active: " + active.get());
      sb.append(" updated: " + updated.get());
      return sb.toString();


  public static class SCCVertex extends
  Vertex <LongWritable, MyValue, NullWritable, LongWritable> {
    public SCCVertex() {
      this.setValue(new MyValue());

    @Override
    public void compute(
    ComputeContext < LongWritable, MyValue, NullWritable, LongWritable
> context,
    Iterable <LongWritable> msgs) throws IOException {
```

```
      // Messages sent to inactive vertex are ignored.
      if (! this.getValue().isActive()) {
        this.voteToHalt();
        return;

      int stage = ((SCCAggrValue)context.getLastAggregatedValue(0)).
getStage();
      switch (stage) {
      case STAGE_TRANSPOSE_1:
        context.sendMessageToNeighbors(this, this.getId());
        break;
      case STAGE_TRANSPOSE_2:
        for (LongWritable msg: msgs) {
          this.getValue().addInNeighbor(msg);

      case STAGE_TRIMMING:
        this.getValue().setSccID(getId());
        if (this.getValue().getInNeighbors().size() == 0 ||
            this.getNumEdges() == 0) {
          this.getValue().setActive(false);

        break;
      case STAGE_FW_START:
        this.getValue().setSccID(getId());
        context.sendMessageToNeighbors(this, this.getValue().getSccID
());
        break;
      case STAGE_FW_REST:
        long minSccID = Long.MAX_VALUE;
        for (LongWritable msg : msgs) {
          if (msg.get() < minSccID) {
            minSccID = msg.get();


        if (minSccID < this.getValue().getSccID().get()) {
          this.getValue().setSccID(new LongWritable(minSccID));
          context.sendMessageToNeighbors(this, this.getValue().
getSccID());
          this.getValue().setUpdated(true);
        } else {
          this.getValue().setUpdated(false);

        break;
      case STAGE_BW_START:
        if (this.getId().equals(this.getValue().getSccID())) {
          for (Writable neighbor : this.getValue().getInNeighbors().
getAll()) {
            context.sendMessage((LongWritable)neighbor, this.getValue
().getSccID());

          this.getValue().setActive(false);

        break;
      case STAGE_BW_REST:
        this.getValue().setUpdated(false);
        for (LongWritable msg : msgs) {
          if (msg.equals(this.getValue().getSccID())) {
            for (Writable neighbor : this.getValue().getInNeighbors().
getAll()) {
              context.sendMessage((LongWritable)neighbor, this.
getValue().getSccID());

            this.getValue().setActive(false);
            this.getValue().setUpdated(true);
```

```
            break;


        break;

      context.aggregate(0, getValue());

    @Override
    public void cleanup(
        WorkerContext<LongWritable, MyValue, NullWritable, LongWritab
le> context)
        throws IOException {
      context.write(getId(), getValue().getSccID());



   * The SCCAggrValue maintains global stage and graph updated and
active status.
   * updated is true only if one vertex is updated.
   * active is true only if one vertex is active.

  public static class SCCAggrValue implements Writable {
    IntWritable stage = new IntWritable(STAGE_TRANSPOSE_1);
    BooleanWritable updated = new BooleanWritable(false);
    BooleanWritable active = new BooleanWritable(false);
    public void setStage(int stage) {
      this.stage.set(stage);

    public int getStage() {
      return this.stage.get();

    public void setUpdated(boolean updated) {
      this.updated.set(updated);

    public boolean getUpdated() {
      return this.updated.get();

    public void setActive(boolean active) {
      this.active.set(active);

    public boolean getActive() {
      return this.active.get();

    @ Override
    public void write(DataOutput out) throws IOException {
      this.stage.write(out);
      this.updated.write(out);
      this.active.write(out);

    @ Override
    public void readFields(DataInput in) throws IOException {
      this.stage.readFields(in);
      this.updated.readFields(in);
      this.active.readFields(in);



   * The job of SCCAggregator is to schedule global stage in every
superstep.

  public static class SCCAggregator extends Aggregator<SCCAggrValue> {
    @SuppressWarnings("rawtypes")
    @ Override
```

```
    public SCCAggrValue createStartupValue(WorkerContext context)
throws IOException {
      return new SCCAggrValue();

    @SuppressWarnings("rawtypes")
    @ Override
    public SCCAggrValue createInitialValue(WorkerContext context)
        throws IOException {
      return (SCCAggrValue) context.getLastAggregatedValue(0);

    @ Override
    public void aggregate(SCCAggrValue value, Object item) throws
IOException {
      MyValue v = (MyValue)item;
      if ((value.getStage() == STAGE_FW_REST || value.getStage() ==
STAGE_BW_REST)
          && v.isUpdated()) {
        value.setUpdated(true);

      // only active vertex invoke aggregate()
      value.setActive(true);

    @ Override
    public void merge(SCCAggrValue value, SCCAggrValue partial)
        throws IOException {
      boolean updated = value.getUpdated() || partial.getUpdated();
      value.setUpdated(updated);
      boolean active = value.getActive() || partial.getActive();
      value.setActive(active);

    @SuppressWarnings("rawtypes")
    @ Override
    public boolean terminate(WorkerContext context, SCCAggrValue value
)
        throws IOException {
      // If all vertices is inactive, job is over.
      if (! value.getActive()) {
        return true;

      // state machine
      switch (value.getStage()) {
      case STAGE_TRANSPOSE_1:
        value.setStage(STAGE_TRANSPOSE_2);
        break;
      case STAGE_TRANSPOSE_2:
        value.setStage(STAGE_TRIMMING);
        break;
      case STAGE_TRIMMING:
        value.setStage(STAGE_FW_START);
        break;
      case STAGE_FW_START:
        value.setStage(STAGE_FW_REST);
        break;
      case STAGE_FW_REST:
        if (value.getUpdated()) {
          value.setStage(STAGE_FW_REST);
        } else {
          value.setStage(STAGE_BW_START);

        break;
      case STAGE_BW_START:
        value.setStage(STAGE_BW_REST);
        break;
      case STAGE_BW_REST:
```

```
        if (value.getUpdated()) {
          value.setStage(STAGE_BW_REST);
        } else {
          value.setStage(STAGE_TRIMMING);

        break;

      value.setActive(false);
      value.setUpdated(false);
      return false;


  public static class SCCVertexReader extends
  GraphLoader<LongWritable, MyValue, NullWritable, LongWritable> {
    @ Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, MyValue, NullWritable,
LongWritable> context)
    throws IOException {
      SCCVertex vertex = new SCCVertex();
      vertex.setId((LongWritable) record.get(0));
      String[] edges = record.get(1).toString().split(",");
      for (int i = 0; i < edges.length; i++) {
        try {
          long destID = Long.parseLong(edges[i]);
          vertex.addEdge(new LongWritable(destID), NullWritable.get
());
        } catch(NumberFormatException nfe) {
          System.err.println("Ignore " + nfe);


      context.addVertexRequest(vertex);


  public static void main(String[] args) throws IOException {
    if (args.length < 2) {
      System.out.println("Usage: <input> <output>");
      System.exit(-1);

    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(SCCVertexReader.class);
    job.setVertexClass(SCCVertex.class);
    job.setAggregatorClass(SCCAggregator.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

10.7.6 連結成分

2 つの頂点間に道がある場合、**2** つの頂点が連結されていることを意味します。 無向グラフ **G** 内の任意の **2** つの頂点が連結されている場合、 **G** は連結グラフと呼ばれます。 そうでない場合、**G** は非連結グラフと呼ばれます。 多数の頂点を有する連結サブグラフは連結成分と呼ばれます。

このアルゴリズムは、各頂点の連結成分メンバーを計算し、最小の頂点 **ID** を含む頂点値の連結成分を出力します。 最小の頂点 **ID** は、辺に沿って連結成分の全ての頂点に送信されます。

サンプルコード

連結成分のコードは以下のとおりです。

```
import java.io.IOException;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.graph.examples.SSSP.MinLongCombiner;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.WritableRecord;

 * Compute the connected component membership of each vertex and
output
 * each vertex which's value containing the smallest id in the
connected
 * component containing that vertex.

 * Algorithm: propagate the smallest vertex id along the edges to all
 * vertices of a connected component.


public class ConnectedComponents {
  public static class CCVertex extends
  Vertex<LongWritable, LongWritable, NullWritable, LongWritable> {
    @Override
    public void compute(
    ComputeContext<LongWritable, LongWritable, NullWritable,
LongWritable> context,
    Iterable<LongWritable> msgs) throws IOException {
      if (context.getSuperstep() == 0L) {
        this.setValue(getId());
        context.sendMessageToNeighbors(this, getValue());
        return;

      long minID = Long.MAX_VALUE;
      for (LongWritable id : msgs) {
      if (id.get() < minID) {
          minID = id.get();


      if (minID < this.getValue().get()) {
        this.setValue(new LongWritable(minID));
        context.sendMessageToNeighbors(this, getValue());
      } else {
        this.voteToHalt();



     * Output Table Description:

     * | Field | Type | Comment |
```

```
     * | v | bigint | vertex id |
     * | minID | bigint | smallest id in the connected component |


    @Override
    public void cleanup(
    WorkerContext<LongWritable, LongWritable, NullWritable, LongWritab
le> context)
        throws IOException {
      context.write(getId(), getValue());



   * Input Table Description:

   * | Field | Type | Comment |

   * | v | bigint | vertex id |
   * | es | string | comma separated target vertex id of outgoing
edges |


   * Example:
   * For graph:
   * 1 ----- 2

   * 3 ----- 4
   * Input table:

   * | v | es |

   * | 1 | 2,3 |
   * | 2 | 1,4 |
   * | 3 | 1,4 |
   * | 4 | 2,3 |


  public static class CCVertexReader extends
  GraphLoader<LongWritable, LongWritable, NullWritable, LongWritable>
 {
    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, LongWritable, NullWritable,
LongWritable> context)
    throws IOException {
      CCVertex vertex = new CCVertex();
      vertex.setId((LongWritable) record.get(0));
      String[] edges = record.get(1).toString().split(",");
      for (int i = 0; i < edges.length; i++) {
        long destID = Long.parseLong(edges[i]);
        vertex.addEdge(new LongWritable(destID), NullWritable.get());

      context.addVertexRequest(vertex);


  public static void main(String[] args) throws IOException {
  if (args.length < 2) {
  System.out.println("Usage: <input> <output>");
      System.exit(-1);

    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(CCVertexReader.class);
```

```
    job.setVertexClass(CCVertex.class);
    job.setCombinerClass(MinLongCombiner.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

10.7.7 トポロジカルソート

有向辺 **(u, v)** では、**u** < **v** を満たすすべての頂点の順序をトポロジカル順序と呼びます。 トポロジカルソートは、有向グラフのトポロジカル順序を計算するために使用されるアルゴリズムです。

具体的なアルゴリズム

1. グラフから入力辺がない頂点を見つけて出力
2. グラフから頂点とすべての出力辺を削除
3. すべての頂点が出力されるまで、上記の手順を繰り返す

サンプルコード

トポロジカル順序付けアルゴリズムのコードは以下のとおりです。

```
import java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.Aggregator;
import com.aliyun.odps.graph.Combiner;
import com.aliyun.odps.graph.ComputeContext;
Import com. aliyun. ODPS. graph. graphjob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.BooleanWritable;
import com.aliyun.odps.io.WritableRecord;
public class TopologySort {
  private final static Log LOG = LogFactory.getLog(TopologySort.class
);
  public static class TopologySortVertex extends
  Vertex<LongWritable, LongWritable, NullWritable, LongWritable> {
    @Override
    public void compute(
    ComputeContext<LongWritable, LongWritable, NullWritable,
LongWritable> context,
    Iterable<LongWritable> messages) throws IOException {
      // in superstep 0, each vertex sends message whose value is 1 to
 its
      // neighbors
```

```
      if (context.getSuperstep() == 0) {
        if (hasEdges()) {
          context.sendMessageToNeighbors(this, new LongWritable(1L));

      } else if (context.getSuperstep() >= 1) {
        // compute each vertex's indegree
        long indegree = getValue().get();
        for (LongWritable msg : messages) {
          indegree += msg.get();

        setValue(new LongWritable(indegree));
        if (indegree == 0) {
          voteToHalt();
          if (hasEdges()) {
            context.sendMessageToNeighbors(this, new LongWritable(-1L
));

          context.write(new LongWritable(context.getSuperstep()),
getId());
          LOG.info("vertex: " + getId());

        context.aggregate(new LongWritable(indegree));



  public static class TopologySortVertexReader extends
  GraphLoader<LongWritable, LongWritable, NullWritable, LongWritable>
 {
    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, LongWritable, NullWritable,
LongWritable> context)
        throws IOException {
      TopologySortVertex vertex = new TopologySortVertex();
      vertex.setId((LongWritable) record.get(0));
      vertex.setValue(new LongWritable(0));
      String[] edges = record.get(1).toString().split(",");
      for (int i = 0; i < edges.length; i++) {
        long edge = Long.parseLong(edges[i]);
        if (edge >= 0) {
          vertex.addEdge(new LongWritable(Long.parseLong(edges[i])),
              NullWritable.get());


      LOG.info(record.toString());
      context.addVertexRequest(vertex);


  public static class LongSumCombiner extends
  Combiner<LongWritable, LongWritable> {
    @Override
    public void combine(LongWritable vertexId, LongWritable combinedMe
ssage,
        LongWritable messageToCombine) throws IOException {
      combinedMessage.set(combinedMessage.get() + messageToCombine.get
());


  public static class TopologySortAggregator extends
  Aggregator<BooleanWritable> {
    @SuppressWarnings("rawtypes")
    @Override
```

```
    public BooleanWritable createInitialValue(WorkerContext context)
        throws IOException {
      return new BooleanWritable(true);

    @Override
    public void aggregate(BooleanWritable value, Object item)
        throws IOException {
      boolean hasCycle = value.get();
      boolean inDegreeNotZero = ((LongWritable) item).get() == 0 ?
false : true;
      value.set(hasCycle && inDegreeNotZero);

    @Override
    public void merge(BooleanWritable value, BooleanWritable partial)
        throws IOException {
      value.set(value.get() && partial.get());

    @SuppressWarnings("rawtypes")
    @Override
    public boolean terminate(WorkerContext context, BooleanWritable
value)
        throws IOException {
      if (context.getSuperstep() == 0) {
        // since the initial aggregator value is true, and in
superstep we don't
        // do aggregate
        return false;

      return value.get();


  public static void main(String[] args) throws IOException {
    if (args.length ! = 2) {
    System.out.println("Usage : <inputTable> <outputTable>");
      System.exit(-1);

    // The input table is in the format of
    // 0 1, 2
    // 1 3
    // 2 3
    // 3 -1
    // The first column is vertexid, and the second column is the
 destination vertexid of the vertex edge. If the value is -1, the
vertex does not have any outbound edge
    // The output table is in the format of
    // 0 0
    // 1 1
    // 1 2
    // 2 3
    // The first column is the supstep value, in which the topological
 sequence is hidden. The second column is vertexid
    // TopologySortAggregator is used to determine if the graph has
loops
    // If the input graph has a loop, the iteration ends when the
indegree of vertices in the active state is not 0
    // You can use records in the input and output tables to determine
 if the graph has loops
    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(TopologySortVertexReader.class);
    job.setVertexClass(TopologySortVertex.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    job.setCombinerClass(LongSumCombiner.class);
    job.setAggregatorClass(TopologySortAggregator.class);
```

```
    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

10.7.8 線形回帰

統計学では、線形回帰は 2 つ以上の変数間の依存関係を決定するために使用される統計分析方法です。 離散予測を処理する分類アルゴリズムとは異なり、

回帰アルゴリズムは連続値型を予測できます。 線形回帰アルゴリズムは、損失関数をサンプルセットの最小二乗誤差の合計として定義します。 これは、重みベクトルを計算するための損失関数を最小にします。

一般的な方法は、勾配降下法です。

1. 重みベクトルを初期化して、降下速度と反復 (または反復収束条件) を与えます。
2. 各サンプルの最小二乗誤差を計算します。
3. 最小二乗誤差の合計を計算し、降下速度に基づいて重みを更新します。
4. 収束するまで繰り返しを繰り返します。

サンプルコード

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.Aggregator;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.Tuple;
import com.aliyun.odps.io.Writable;
import com.aliyun.odps.io.WritableRecord;

 * LineRegression input: y,x1,x2,x3,......

public class LinearRegression {
  public static class GradientWritable implements Writable {
    Tuple lastTheta;
    Tuple currentTheta;
    Tuple tmpGradient;
    LongWritable count;
    DoubleWritable lost;
    @Override
    public void readFields(DataInput in) throws IOException {
      lastTheta = new Tuple();
```

```
      lastTheta.readFields(in);
      currentTheta = new Tuple();
      currentTheta.readFields(in);
      tmpGradient = new Tuple();
      tmpGradient.readFields(in);
      count = new LongWritable();
      count.readFields(in);
      /* update 1: add a variable to store lost at every iteration */
      lost = new DoubleWritable();
      lost.readFields(in);

    @Override
    public void write(DataOutput out) throws IOException {
      lastTheta.write(out);
      currentTheta.write(out);
      tmpGradient.write(out);
      count.write(out);
      lost.write(out);


  public static class LinearRegressionVertex extends
  Vertex<LongWritable, Tuple, NullWritable, NullWritable> {
    @Override
    public void compute(
    ComputeContext<LongWritable, Tuple, NullWritable, NullWritable>
context,
    Iterable<NullWritable> messages) throws IOException {
      context.aggregate(getValue());


  public static class LinearRegressionVertexReader extends
  GraphLoader<LongWritable, Tuple, NullWritable, NullWritable> {
    @Override
    public void load(LongWritable recordNum, WritableRecord record,
    MutationContext<LongWritable, Tuple, NullWritable, NullWritable>
context)
        throws IOException {
      LinearRegressionVertex vertex = new LinearRegressionVertex();
      vertex.setId(recordNum);
      vertex.setValue(new Tuple(record.getAll()));
      context.addVertexRequest(vertex);


  public static class LinearRegressionAggregator extends
  Aggregator<GradientWritable> {
    @SuppressWarnings("rawtypes")
    @Override
    public GradientWritable createInitialValue(WorkerContext context)
        throws IOException {
      if (context.getSuperstep() == 0) {
        /* set initial value, all 0 */
        GradientWritable grad = new GradientWritable();
        grad.lastTheta = new Tuple();
        grad.currentTheta = new Tuple();
        grad.tmpGradient = new Tuple();
        grad.count = new LongWritable(1);
        grad.lost = new DoubleWritable(0.0);
        int n = (int) Long.parseLong(context.getConfiguration()
            .get("Dimension"));
            for (int i = 0; i < n; i++) {
          grad.lastTheta.append(new DoubleWritable(0));
          grad.currentTheta.append(new DoubleWritable(0));
          grad.tmpGradient.append(new DoubleWritable(0));
```

```
        return grad;
      } else
        return (GradientWritable) context.getLastAggregatedValue(0);

    public static double vecMul(Tuple value, Tuple theta) {
      /* perform this partial computing: y(i)-hθ(x(i)) for each sample
 */
      /* value denote a piece of sample and value(0) is y */
      double sum = 0.0;
      for (int j = 1; j < value.size(); j++)
        sum += Double.parseDouble(value.get(j).toString())
            * Double.parseDouble(theta.get(j).toString());
      Double tmp = Double.parseDouble(theta.get(0).toString()) + sum
          - Double.parseDouble(value.get(0).toString());
      return tmp;

    @Override
    public void aggregate(GradientWritable gradient, Object value)
        throws IOException {

       * perform on each vertex--each sample i：set theta(j) for each
sample i
       * for each dimension

      double tmpVar = vecMul((Tuple) value, gradient.currentTheta);

       * update 2:local worker aggregate(), perform like merge() below
. This
       * means the variable gradient denotes the previous aggregated
value

      gradient.tmpGradient.set(0, new DoubleWritable(
          ((DoubleWritable) gradient.tmpGradient.get(0)).get() +
tmpVar));
      gradient.lost.set(Math.pow(tmpVar, 2));

       * calculate (y(i)-hθ(x(i))) x(i)(j) for each sample i for each
       * dimension j

       for (int j = 1; j < gradient.tmpGradient.size(); j++)
        gradient.tmpGradient.set(j, new DoubleWritable(
            ((DoubleWritable) gradient.tmpGradient.get(j)).get() +
tmpVar
                * Double.parseDouble(((Tuple) value).get(j).toString
())));

    @Override
    public void merge(GradientWritable gradient, GradientWritable
partial)
        throws IOException {
      /* perform SumAll on each dimension for all samples.
      Tuple master = (Tuple) gradient.tmpGradient;
      Tuple part = (Tuple) partial.tmpGradient;
      for (int j = 0; j < gradient.tmpGradient.size(); j++) {
        DoubleWritable s = (DoubleWritable) master.get(j);
        s.set(s.get() + ((DoubleWritable) part.get(j)).get());

      gradient.lost.set(gradient.lost.get() + partial.lost.get());

    @SuppressWarnings("rawtypes")
    @Override
    public boolean terminate(WorkerContext context, GradientWritable
gradient)
        throws IOException {
```

```
       * 1. calculate new theta 2. judge the diff between last step
and this
       * step, if smaller than the threshold, stop iteration

      gradient.lost = new DoubleWritable(gradient.lost.get()
          / (2 * context.getTotalNumVertices()));

       * we can calculate lost in order to make sure the algorithm is
running on
       * the right direction (for debug)

      System.out.println(gradient.count + " lost:" + gradient.lost);
      Tuple tmpGradient = gradient.tmpGradient;
      System.out.println("tmpGra" + tmpGradient);
      Tuple lastTheta = gradient.lastTheta;
      Tuple tmpCurrentTheta = new Tuple(gradient.currentTheta.size());
      System.out.println(gradient.count + " terminate_start_last:" +
lastTheta);
      double alpha = 0.07; // learning rate
      // alpha =
      // Double.parseDouble(context.getConfiguration().get("Alpha"));
      /* perform theta(j) = theta(j)-alpha*tmpGradient */
      long M = context.getTotalNumVertices();

       * update 3: add (/M) on the code. The original code forget this
 step

       for (int j = 0; j < lastTheta.size(); j++) {
        tmpCurrentTheta
            .set(
                J,
                new DoubleWritable(Double.parseDouble(lastTheta.get(j)
                    .toString())
                    - alpha
                    / M
                    * Double.parseDouble(tmpGradient.get(j).toString
()))));

      System.out.println(gradient.count + " terminate_start_current:"
          + tmpCurrentTheta);
      // judge if convergence is happening.
      double diff = 0.00d;
      for (int j = 0; j < gradient.currentTheta.size(); j++)
        diff += Math.pow(((DoubleWritable) tmpCurrentTheta.get(j)).get
()
            - ((DoubleWritable) lastTheta.get(j)).get(), 2);
      if (/*
      * Math.sqrt(diff) < 0.00000000005d ||
           */Long.parseLong(context.getConfiguration().get("Max_Iter_N
um")) == gradient.count
          .get()) {
        context.write(gradient.currentTheta.toArray());
        return true;

      gradient.lastTheta = tmpCurrentTheta;
      gradient.currentTheta = tmpCurrentTheta;
      gradient.count.set(gradient.count.get() + 1);
      int n = (int) Long.parseLong(context.getConfiguration().get("
Dimension"));

       * update 4: Important!!! Remember this step. Graph won't reset
the
```

```
       * initial value for global variables at the beginning of each
iteration

       for (int i = 0; i < n; i++) {
        gradient.tmpGradient.set(i, new DoubleWritable(0));

      return false;


  public static void main(String[] args) throws IOException {
    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(LinearRegressionVertexReader.class);
    job.setRuntimePartitioning(false);
    job.setNumWorkers(3);
    job.setVertexClass(LinearRegressionVertex.class);
    job.setAggregatorClass(LinearRegressionAggregator.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    job.setMaxIteration(Integer.parseInt(args[2])); // Numbers of
Iteration
    job.setInt("Max_Iter_Num", Integer.parseInt(args[2]));
    job.setInt("Dimension", Integer.parseInt(args[3])); // Dimension
    job.setFloat("Alpha", Float.parseFloat(args[4])); // Learning rate
    long start = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - start) / 1000.0 + " seconds");
```

10.7.9 三角形カウント

このアルゴリズムは、各頂点を通る三角形の数を計算するために使用されます。

アルゴリズムは次の手順で実装されます。

1. 各頂点は **ID** をすべての出力隣接頂点に送信します。
2. 入力および出力の隣接頂点を保存し、それらを出力の隣接頂点に送信します。
3. 各辺の端点交差の数を計算し、合計した後、結果をテーブルに出力します。
4. テーブル内の出力結果の合計を計算し、それを **3** で除算して三角形の数を得ます。

サンプルコード

三角形カウントアルゴリズムのコードは以下のとおりです。

```
import java.io.IOException;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.Edge;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.WorkerContext;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.NullWritable;
import com.aliyun.odps.io.Tuple;
import com.aliyun.odps.io.Writable;
```

```
import com.aliyun.odps.io.WritableRecord;

 * Compute the number of triangles passing through each vertex.

 * The algorithm can be computed in three supersteps:
 * I. Each vertex sends a message with its ID to all its outgoing
 * neighbors.
 * II. The incoming neighbors and outgoing neighbors are stored and
 * send to outgoing neighbors.
 * III. For each edge compute the intersection of the sets at
destination
 * vertex and sum them, then output to table.

 * The triangle count is the sum of output table and divide by three
since
 * each triangle is counted three times.


public class TriangleCount {
  public static class TCVertex extends
    Vertex<LongWritable, Tuple, NullWritable, Tuple> {
    @Override
    public void setup(
        WorkerContext<LongWritable, Tuple, NullWritable, Tuple>
context)
      throws IOException {
      // collect the outgoing neighbors
      Tuple t = new Tuple();
      if (this.hasEdges()) {
        for (Edge<LongWritable, NullWritable> edge : this.getEdges())
 {
          t.append(edge.getDestVertexId());
        }

      this.setValue(t);

    @Override
    public void compute(
        ComputeContext<LongWritable, Tuple, NullWritable, Tuple>
context,
        Iterable<Tuple> msgs) throws IOException {
      if (context.getSuperstep() == 0L) {
        // sends a message with its ID to all its outgoing neighbors
        Tuple t = new Tuple();
        t.append(getId());
        context.sendMessageToNeighbors(this, t);
      } else if (context.getSuperstep() == 1L) {
        // store the incoming neighbors
        for (Tuple msg : msgs) {
          for (Writable item : msg.getAll()) {
            if (! this.getValue().getAll().contains((LongWritable)item
)) {
              this.getValue().append((LongWritable)item);



        // send both incoming and outgoing neighbors to all outgoing
neighbors
        context.sendMessageToNeighbors(this, getValue());
      } else if (context.getSuperstep() == 2L) {
        // count the sum of intersection at each edge
        long count = 0;
        for (Tuple msg : msgs) {
          for (Writable id : msg.getAll()) {
```

```
            if (getValue().getAll().contains(id)) {
              count ++;



        // output to table
        context.write(getId(), new LongWritable(count));
        this.voteToHalt();



  public static class TCVertexReader extends
  GraphLoader<LongWritable, Tuple, NullWritable, Tuple> {
    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, Tuple, NullWritable, Tuple>
context)
    throws IOException {
      TCVertex vertex = new TCVertex();
      vertex.setId((LongWritable) record.get(0));
      String[] edges = record.get(1).toString().split(",");
      for (int i = 0; i < edges.length; i++) {
        try {
          long destID = Long.parseLong(edges[i]);
          vertex.addEdge(new LongWritable(destID), NullWritable.get
());
        } catch(NumberFormatException nfe) {
          System.err.println("Ignore " + nfe);


      context.addVertexRequest(vertex);


  public static void main(String[] args) throws IOException {
  if (args.length < 2) {
  System.out.println("Usage: <input> <output>");
      System.exit(-1);

    GraphJob job = new GraphJob();
    job.setGraphLoaderClass(TCVertexReader.class);
    job.setVertexClass(TCVertex.class);
    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addOutput(TableInfo.builder().tableName(args[1]).build());
    long startTime = System.currentTimeMillis();
    job.run();
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + "
seconds");
```

10.7.10 頂点の入力

サンプルコード

```
import java.io.IOException;
import com.aliyun.odps.conf.Configuration;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
```

```
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.VertexResolver;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.VertexChanges;
import com.aliyun.odps.graph.Edge;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.WritableComparable;
import com.aliyun.odps.io.WritableRecord;

 * The following example describes how to compile a graph job program
to load data of different types.  It mainly describes how GraphLoader
 * and VertexResolver are cooperated to build the graph.

 * A MaxCompute Graph job uses MaxCompute tables as the input. Assume
that a job has two tables as the input, one storing vertices and the
other storing edges.
 * The format of the table storing vertex information is as follows:

 * | VertexID | VertexValue |

 * | id0| 9|

 * | id1| 7|

 * | id2| 8|


 * The format of the table storing edge information is as follows:

 * | VertexID | DestVertexID| EdgeValue|

 * | id0| id1| 1|

 * | id0| id2| 2|

 * | id2| id1| 3|


 * The preceding two tables show that id0 has two outbound edges
 pointing to id1 and id2 respectively. id2 has an outbound edge
pointing to id1, and id1 has no outbound edges.

 * For data of this type, in GraphLoader::load(LongWritable, Record,
MutationContext),
 * MutationContext#addVertexRequest(Vertex) can be used to add
vertices to the graph, while
 * link MutationContext#addEdgeRequest(WritableComparable, Edge) can
be used to add edges to the graph. In
 * link VertexResolver#resolve(WritableComparable, Vertex, VertexChan
ges, boolean)
 * vertices and edges added in the load() method are combined to a
 vertex object, which is used as the return value and added to the
graph for calculation.


public class VertexInputFormat {
  private final static String EDGE_TABLE = "edge.table";

   * Resolve a record to vertices and edges. Each record indicates a
vertex or an edge according to its source.

   * Similar to com.aliyun.odps.mapreduce.Mapper#map,
```

```
   * enter a record to generate key-value pairs. The keys are vertex
IDs,
   * and the values are vertices or edges written based on the context
. These key-value pairs are summarized based on vertex IDs using
LoadingVertexResolver.

   * Note: Vertices or edges added here are requests sent based on the
 record content, and are not used in calculation. Only
   * vertices or edges added using VertexResolver participate in
calculation.

  public static class VertexInputLoader extends
  GraphLoader<LongWritable, LongWritable, LongWritable, LongWritable>
 {
    private boolean isEdgeData;

     * Configure VertexInputLoader.

     * @param conf
     * Indicates the configuration parameters of a job, which are
configured in the main GraphJob or set on the console.
     * @param workerId
     * Indicates the serial number of the operating Worker, which
starts from 0 and can be used to build a unique vertex ID.
     * @param inputTableInfo
     * Indicates information about the input table load to the current
 Worker, which can be used to determine the type of currently input
data, that is, the record format.

    @Override
    public void setup(Configuration conf, int workerId, TableInfo
inputTableInfo) {
      isEdgeData = conf.get(EDGE_TABLE).equals(inputTableInfo.
getTableName());


     * Based on the record content, resolve corresponding edges and
send a request to add them to the graph.

     * @param recordNum
     * Indicates the record serial number, which starts from 1 and is
separately counted in each Worker.
     * @param record
     * Indicates the record in the input table. It contains three
columns, indicating the first vertex, last vertex, and edge weight.
     * @param context
     * Indicates the context, requesting to add resolved edges to the
graph.

    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
        MutationContext<LongWritable, LongWritable, LongWritable,
LongWritable> context)
        throws IOException {
      if (isEdgeData) {

         * Data is from the table that stores edge information.

         * 1. The first column indicates the first vertex ID.

        LongWritable sourceVertexID = (LongWritable) record.get(0);
```

```
         * 2. The second column indicates the last vertex ID.

        LongWritable destinationVertexID = (LongWritable) record.get(1
);

         * 3. The third column indicates the edge weight.

        LongWritable edgeValue = (LongWritable) record.get(2);

         * 4. Create an edge that consists of the last vertex ID and
edge weight.

         Edge<LongWritable, LongWritable> edge = new Edge<LongWritable
, LongWritable>(
            destinationVertexID, edgeValue);

         * 5. Send a request to add an edge to the first vertex.

        context.addEdgeRequest(sourceVertexID, edge);

        * 6. If each record indicates a bidirectional edge, repeat
steps 4 and 5. Edge<LongWritable, LongWritable> edge2 = new
        * Edge<LongWritable, LongWritable>( sourceVertexID, edgeValue
);
         * context.addEdgeRequest(destinationVertexID, edge2);

      } else {

         * Data comes from the table that stores vertex information.

         * 1. The first column indicates the vertex ID.

        LongWritable vertexID = (LongWritable) record.get(0);

         * 2. The second column indicates the vertex value.

        LongWritable vertexValue = (LongWritable) record.get(1);

         * 3. Create a vertex that consists of the vertex ID and
vertex value.

        MyVertex vertex = new MyVertex();

         * 4. Initialize the vertex.

        vertex.setId(vertexID);
        vertex.setValue(vertexValue);

         * 5. Send a request to add a vertex.

        context.addVertexRequest(vertex);



   * Summarize key-value pairs generated using GraphLoader::load(
LongWritable, Record, MutationContext), which is similar to
   * reduce in com.aliyun.odps.mapreduce.Reducer.  For the unique
vertex ID, all actions such as
   * adding/deleting vertices or edges on the ID is stored in
VertexChanges.
```

```
   * Note: Not only conflicting vertices or edges added by using the
load() method are called. (A conflict occurs when multiple same vertex
 objects or duplicate edges are added.)
   * All IDs requested to be generated using the load() method are
called.

  public static class LoadingResolver extends
  VertexResolver<LongWritable, LongWritable, LongWritable, LongWritab
le> {

     * Process a request about adding/deleting vertices or edges for
an ID.

     * VertexChanges has four APIs, which correspond to the four APIs
of MutationContext:
     * VertexChanges::getAddedVertexList() corresponds to
     * MutationContext::addVertexRequest(Vertex).
     * In the load() method, if vertex objects with the same ID are
requested to be added, such vertex objects are collected to the return
 list.
     * VertexChanges::getAddedEdgeList() corresponds to
     * MutationContext::addEdgeRequest(WritableComparable, Edge)
     * If edge objects with the same first vertex ID are requested to
be added, such edge objects are collected to the return list.
     * VertexChanges::getRemovedVertexCount() corresponds to
     * MutationContext::removeVertexRequest(WritableComparable)
     * If vertices with the same ID are requested to be deleted, the
number of total deletion requests is returned.
     * VertexChanges#getRemovedEdgeList() corresponds to
     * MutationContext#removeEdgeRequest(WritableComparable,
WritableComparable)
     * If edge objects with the same first vertex ID are requested to
be deleted, such edge objects are collected to the return list.

     * By processing ID changes, you can state whether the ID
participates in calculation using the return value. If the returned
vertex is not null,
     * the ID participates in subsequent calculation. If the returned
vertex is null, the ID does not participate in subsequent calculation.

     * @param vertexId
     * Indicates the ID of the vertex requested to be added or first
vertex ID of the edge requested to be added.
     * @param vertex
     * Indicates an existing vertex object. Its value is always null
in the data loading phase.
     * @param vertexChanges
     * Indicates the set of vertices or edges requested to be added/
deleted on the ID.
     * @param hasMessages
     * Indicates whether the ID has any input message. Its value is
always false in the data loading phase.

    @Override
    public Vertex<LongWritable, LongWritable, LongWritable, LongWritab
le> resolve(
        LongWritable vertexId,
        Vertex<LongWritable, LongWritable, LongWritable, LongWritable
> vertex,
        VertexChanges<LongWritable, LongWritable, LongWritable,
LongWritable> vertexChanges,
        boolean hasMessages) throws IOException {

       * 1. Obtain the vertex object for calculation.
```

```
      MyVertex computeVertex = null;
      if (vertexChanges.getAddedVertexList() == null
          || vertexChanges.getAddedVertexList().isEmpty()) {
        computeVertex = new MyVertex();
        computeVertex.setId(vertexId);
      } else {

         * Assume that each record indicates a unique vertex in the
table storing vertex information.

        computeVertex = (MyVertex) vertexChanges.getAddedVertexList().
get(0);


       * 2. Add the edge requested to be added to the vertex to the
vertex object. If data is duplicated, perform deduplication based on
the algorithm needs.

      if (vertexChanges.getAddedEdgeList() ! = null) {
      for (Edge<LongWritable, LongWritable> edge : vertexChanges
            .getAddedEdgeList()) {
          computeVertex.addEdge(edge.getDestVertexId(), edge.getValue
());



       * 3. Return the vertex object and add it to the final graph for
 calculation.

      return computeVertex;



   * Determine actions of the vertex that participates in calculation.


  public static class MyVertex extends
  Vertex<LongWritable, LongWritable, LongWritable, LongWritable> {

     * Write the vertex edge to the result table according to the
format of the input table.  Ensure that the format and data of the
input and output tables are the same.

     * @param context
     * Indicates the context during running.
     * @param messages
     * Indicates the input message.

    @Override
    public void compute(
    ComputeContext<LongWritable, LongWritable, LongWritable,
LongWritable> context,
    Iterable<LongWritable> messages) throws IOException {

       * Write the vertex ID and value to the result table storing
vertices.

      context.write("vertex", getId(), getValue());

       * Write the vertex edge to the result table storing edges.

      if (hasEdges()) {
      for (Edge<LongWritable, LongWritable> edge : getEdges()) {
```

```
          context.write("edge", getId(), edge.getDestVertexId(),
              edge.getValue());



       * Perform one round of iteration.

      voteToHalt();



   * @param args
   * @throws IOException

  public static void main(String[] args) throws IOException {
  if (args.length < 4) {
      throw new IOException(
      "Usage: VertexInputFormat <vertex input> <edge input> <vertex
output> <edge output>");


     * GraphJob is used to configure Graph jobs.

    GraphJob job = new GraphJob();

     * 1. Specify input graph data and the table storing edge data.

    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addInput(TableInfo.builder().tableName(args[1]).build());
    job.set(EDGE_TABLE, args[1]);

     * 2. Specify the data loading mode, resolve the record as edges.
Similar to the map, the generated key is the vertex ID, and the value
is the edge.

    job.setGraphLoaderClass(VertexInputLoader.class);

     * 3. Specify the data loading phase, and generate the vertex for
calculation.  Similar to reduce, edges generated by map are combined
to a vertex.

    job.setLoadingVertexResolverClass(LoadingResolver.class);

     * 4. Specify actions of the vertex that participates in
calculation.  The vertex.compute() method is used for each round of
iteration.

    job.setVertexClass(MyVertex.class);

     * 5. Specify the output table of the Graph job, and write the
calculation result to the result table.

    job.addOutput(TableInfo.builder().tableName(args[2]).label("vertex
").build());
    job.addOutput(TableInfo.builder().tableName(args[3]).label("edge
").build());

     * 6. Submit the job for execution.

    job.run();
```

10.7.11 辺の入力

サンプルコード

```
import java.io.IOException;
import com.aliyun.odps.conf.Configuration;
import com.aliyun.odps.data.TableInfo;
import com.aliyun.odps.graph.ComputeContext;
import com.aliyun.odps.graph.GraphJob;
import com.aliyun.odps.graph.GraphLoader;
import com.aliyun.odps.graph.Vertex;
import com.aliyun.odps.graph.VertexResolver;
import com.aliyun.odps.graph.MutationContext;
import com.aliyun.odps.graph.VertexChanges;
import com.aliyun.odps.graph.Edge;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.WritableComparable;
import com.aliyun.odps.io.WritableRecord;

 * The following example describes how to compile a graph job program
to load data of different types. It mainly describes how GraphLoader
 * and VertexResolver are cooperated to build the graph.

 * A MaxCompute Graph job uses MaxCompute tables as the input. Assume
that a job has two tables as the input, one storing vertices and the
other storing edges.
 * The format of the table storing vertex information is as follows:

  * | VertexID | VertexValue |

 * | id0| 9|

 * | id1| 7|

 * | id2| 8|


  * The format of the table storing edge information is as follows:

 * | VertexID | DestVertexID| EdgeValue|

 * | id0| id1| 1|

 * | id0| id2| 2|

 * | id2| id1| 3|


 * The preceding two tables show that id0 has two outbound edges
 pointing to id1 and id2 respectively. id2 has an outbound edge
pointing to id1, and id1 has no outbound edges.

 * For data of this type, in GraphLoader::load(LongWritable, Record,
MutationContext),
 * MutationContext#addVertexRequest(Vertex) can be used to add
vertices to the graph, while
 * link MutationContext#addEdgeRequest(WritableComparable, Edge) can
be used to add edges to the graph. In
 * link VertexResolver#resolve(WritableComparable, Vertex, VertexChan
ges, boolean)
```

```
 * vertices and edges added in the load() method are combined to a
 vertex object, which is used as the return value and added to the
graph for calculation.


public class VertexInputFormat {
  private final static String EDGE_TABLE = "edge.table";

   * Resolve a record to vertices and edges. Each record indicates a
vertex or an edge according to its source.
   * Similar to com.aliyun.odps.mapreduce.Mapper#map,
   * enter a record to generate key-value pairs. The keys are vertex
IDs,
   * and the values are vertices or edges written based on the context
. These key-value pairs are summarized based on vertex IDs using
LoadingVertexResolver.

   * Note: Vertices or edges added here are requests sent based on the
 record content, and are not used for calculation. Only
   * vertices or edges added using VertexResolver participate in
calculation.

  public static class VertexInputLoader extends
  GraphLoader<LongWritable, LongWritable, LongWritable, LongWritable>
 {
    private boolean isEdgeData;

     * Configure VertexInputLoader.

     * @param conf
     * Indicates the configuration parameters of a job, which are
configured in the main GraphJob or set on the console.
     * @param workerId
     * Indicates the serial number of the operating Worker, which
starts from 0 and can be used to build a unique vertex ID.
     * @param inputTableInfo
     * Indicates information about the input table loaded to the
current Worker, which can be used to determine the type of currently
input data, that is, the record format.

    @ Override
    public void setup(Configuration conf, int workerId, TableInfo
inputTableInfo) {
      isEdgeData = conf.get(EDGE_TABLE).equals(inputTableInfo.
getTableName());


     * Based on the record content, resolve corresponding edges and
send a request to add them to the graph.

     * @param recordNum
     * Indicates the record serial number, which starts from 1 and is
separately counted in each Worker.
     * @param record
     * Indicates the record in the input table. It contains three
columns, indicating the first vertex, last vertex, and edge weight.
     * @param context
     * Indicates the context, requesting to add resolved edges to the
graph.

    @Override
    public void load(
        LongWritable recordNum,
        WritableRecord record,
```

```
        MutationContext<LongWritable, LongWritable, LongWritable,
LongWritable> context)
        throws IOException {
      if (isEdgeData) {

         * Data comes from the table that stores edge information.

         * 1. The first column indicates the first vertex ID.

        LongWritable sourceVertexID = (LongWritable) record.get(0);

         * 2. The second column indicates the last vertex ID.

        LongWritable destinationVertexID = (LongWritable) record.get(1
);

         * 3. The third column indicates the edge weight.

        LongWritable edgeValue = (LongWritable) record.get(2);

         * 4. Create an edge that consists of the last vertex ID and
edge weight.

         Edge<LongWritable, LongWritable> edge = new Edge<LongWritable
, LongWritable>(
            destinationVertexID, edgeValue);

         * 5. Send a request to add an edge to the first vertex.

        context.addEdgeRequest(sourceVertexID, edge);

        * 6. If each record indicates a bidirectional edge, repeat
steps 4 and 5. Edge<LongWritable, LongWritable> edge2 = new
        * Edge<LongWritable, LongWritable>( sourceVertexID, edgeValue
);
         * context.addEdgeRequest(destinationVertexID, edge2);

      } else {

         * Data comes from the table that stores vertex information.

         * 1. The first column indicates the vertex ID.

        LongWritable vertexID = (LongWritable) record.get(0);

         * 2. The second column indicates the vertex value.

        LongWritable vertexValue = (LongWritable) record.get(1);

         * 3. Create a vertex that consists of the vertex ID and
vertex value.

        MyVertex vertex = new MyVertex();

         * 4. Initialize the vertex.

        vertex.setId(vertexID);
        vertex.setValue(vertexValue);

         * 5. Send a request to add a vertex.

        context.addVertexRequest(vertex);
```

```
   * Summarize key-value pairs generated using GraphLoader::load(
LongWritable, Record, MutationContext), which is similar to
   * reduce in com.aliyun.odps.mapreduce.Reducer.  For the unique
vertex ID, all actions such as
   * adding/deleting vertices or edges on the ID is stored in
VertexChanges.

   * Note: Not only conflicting vertices or edges added by using the
load() method are called. (A conflict occurs when multiple same vertex
 objects or duplicate edges are added.)
   * All IDs requested to be generated using the load() method are
called.

  public static class LoadingResolver extends
  VertexResolver<LongWritable, LongWritable, LongWritable, LongWritab
le> {

     * Process a request about adding/deleting vertices or edges for
an ID.

     * VertexChanges has four APIs, which correspond to the four APIs
of MutationContext:
     * VertexChanges::getAddedVertexList() corresponds to
     * MutationContext::addVertexRequest(Vertex).
     * In the load() method, if vertex objects with the same ID are
requested to be added, such vertex objects are collected to the return
 list.
     * VertexChanges::getAddedEdgeList() corresponds to
     * MutationContext::addEdgeRequest(WritableComparable, Edge)
     * If edge objects with the same first vertex ID are requested to
be added, such edge objects are collected to the return list.
     * VertexChanges::getRemovedVertexCount() corresponds to
     * MutationContext::removeVertexRequest(WritableComparable)
     * If vertices with the same ID are requested to be deleted, the
number of total deletion requests is returned.
     * VertexChanges#getRemovedEdgeList() corresponds to
     * MutationContext#removeEdgeRequest(WritableComparable,
WritableComparable)
     * If edge objects with the same first vertex ID are requested to
be deleted, such edge objects are collected to the return list.

     * By processing ID changes, you can state whether the ID
participates in calculation using the return value. If the returned
vertex is not null,
     * the ID participates in subsequent calculation. If the returned
vertex is null, the ID does not participate in subsequent calculation.

     * @param vertexId
     * Indicates the ID of the vertex requested to be added or first
vertex ID of the edge requested to be added.
     * @param vertex
     * Indicates an existing vertex object. Its value is always null
in the data loading phase.
     * @param vertexChanges
     * Indicates the set of vertices or edges requested to be added/
deleted on the ID.
     * @param hasMessages
     * Indicates whether the ID has any input message. Its value is
always false in the data loading phase.

    @Override
```

```
    public Vertex<LongWritable, LongWritable, LongWritable, LongWritab
le> resolve(
        LongWritable vertexId,
        Vertex<LongWritable, LongWritable, LongWritable, LongWritable
> vertex,
        VertexChanges<LongWritable, LongWritable, LongWritable,
LongWritable> vertexChanges,
        boolean hasMessages) throws IOException {

       * 1. Obtain the vertex object to participate in calculation.

      MyVertex computeVertex = null;
      if (vertexChanges.getAddedVertexList() == null
          || vertexChanges.getAddedVertexList().isEmpty()) {
        computeVertex = new MyVertex();
        computeVertex.setId(vertexId);
      } else {

         * Assume that each record indicates a unique vertex in the
table storing vertex information.

        computeVertex = (MyVertex) vertexChanges.getAddedVertexList().
get(0);


       * 2. Add the edge requested to be added to the vertex to the
vertex object. If data may be duplicate, perform deduplication based
on the algorithm needs.

      if (vertexChanges.getAddedEdgeList() ! = null) {
      for (Edge<LongWritable, LongWritable> edge : vertexChanges
            .getAddedEdgeList()) {
          computeVertex.addEdge(edge.getDestVertexId(), edge.getValue
());



       * 3. Return the vertex object and add it to the final graph for
 calculation.

      return computeVertex;



   * Determine actions of the vertex that participates in calculation.


  public static class MyVertex extends
  Vertex<LongWritable, LongWritable, LongWritable, LongWritable> {

     * Write the vertex edge to the result table according to the
format of the input table.  Ensure that the format and data of the
input and output tables are the same.

     * @param context
     * Indicates the context during running.
     * @param messages
     * Indicates the input message.

    @Override
    public void compute(
    ComputeContext<LongWritable, LongWritable, LongWritable,
LongWritable> context,
         Iterable<LongWritable> messages) throws IOException {
```

```
       * Write the vertex ID and value to the result table storing
vertices.

      context.write("vertex", getId(), getValue());

       * Write the vertex edge to the result table storing edges.

      if (hasEdges()) {
      for (Edge<LongWritable, LongWritable> edge : getEdges()) {
          context.write("edge", getId(), edge.getDestVertexId(),
              edge.getValue());



       * Perform one round of iteration.

      voteToHalt();



   * @param args
   * @throws IOException

  public static void main(String[] args) throws IOException {
  If (ARGs. Length <4 ){
      throw new IOException(
      "Usage: VertexInputFormat <vertex input> <edge input> <vertex
output> <edge output>");


     * GraphJob is used to configure Graph jobs.

    GraphJob job = new GraphJob();

     * 1. Specify input graph data and the table storing edge data.

    job.addInput(TableInfo.builder().tableName(args[0]).build());
    job.addInput(TableInfo.builder().tableName(args[1]).build());
    job.set(EDGE_TABLE, args[1]);

     * 2. Specify the data loading mode, resolve the record as edges.
Similar to the map, the generated key is the vertex ID, and the value
is the edge.

    job.setGraphLoaderClass(VertexInputLoader.class);

     * 3. Specify the data loading phase, and generate the vertex that
 participates in calculation.  Similar to reduce, edges generated by
map are combined to a vertex.

    job.setLoadingVertexResolverClass(LoadingResolver.class);

     * 4. Specify actions of the vertex that participates in
calculation.  The vertex.compute() method is used for each round of
iteration.

    job.setVertexClass(MyVertex.class);

     * 5. Specify the output table of the Graph job, and write the
calculation result to the result table.

    job.addOutput(TableInfo.builder().tableName(args[2]).label("vertex
").build());
```

```
    job.addOutput(TableInfo.builder().tableName(args[3]).label("edge
").build());

     * 6. Submit the job for execution.

    job.run();
```

11 ジョブ実行情報の表示

11.1 Logview

Logview は、**MaxCompute** ジョブの送信後、タスクを確認し、デバッグするためのツールです。

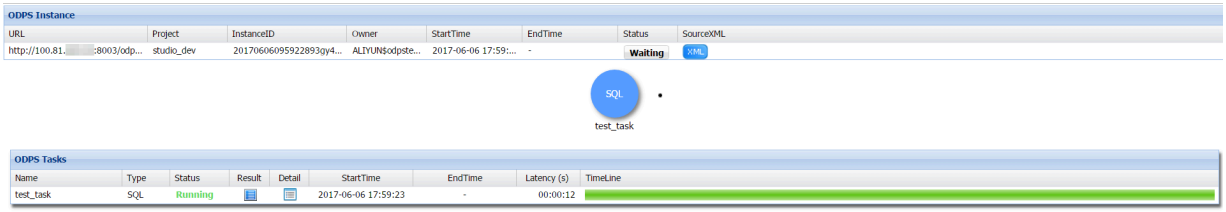
Logview を使用すると、ジョブに関する以下の詳細を確認できます。

- タスクの実行ステータス
- タスクの操作結果
- タスクの詳細および各ステップの進行状況

ジョブが **MaxCompute** に送信されると、**Logview** へのリンクが生成されます。 ブラウザ上で直接 **Logview** リンクを開き、各ジョブの情報を表示することができます。 **Logview** ページは **7** 日間有効です。

機能

以下に、特定の **Logview Web** ユーザーインターフェイスを使用して、各コンポーネントについて説明します。



Logview ホームページは 上部と下部の **2** つのセクションに分かれています。

- インスタンス情報
- タスク情報

インスタンス情報

Logview ホームページの上部は、**SQL** を生成するために送信された **MaxCompute** インスタンスです。 各 **SQL** のコミット後、一意の **ID** が生成されます。 レイテンシーは、実行所要時間のことです。他のページのレイテンシーも同様です。

4 つのステータスがあります。

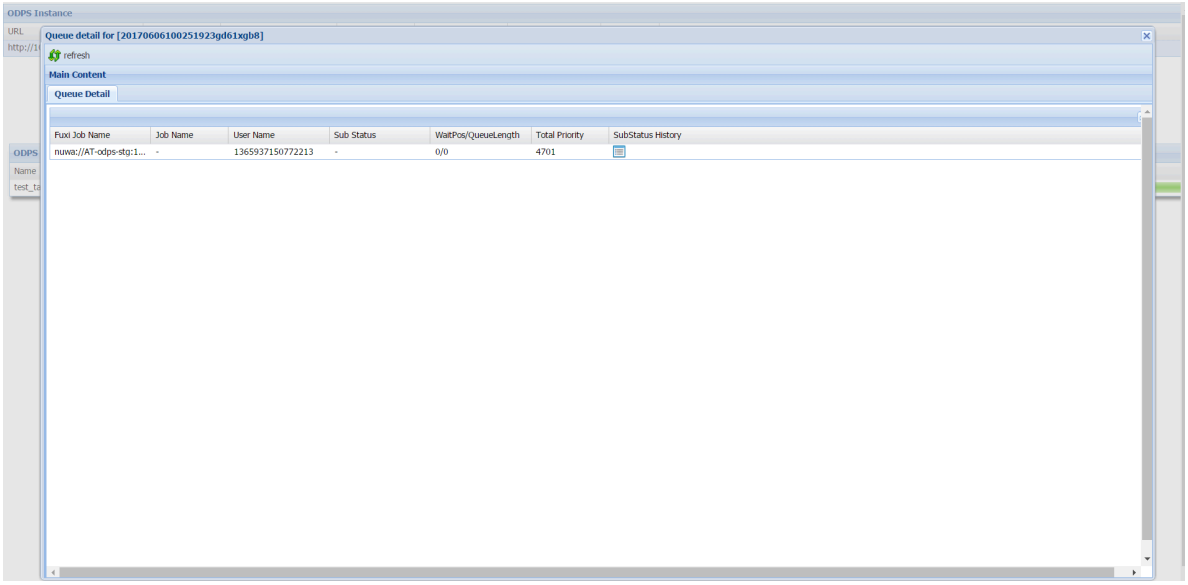
- **Waiting:** 現在のジョブが **MaxCompute** で処理中であり、**Fuxi** での実行がコミットされていないことを示します。

- **Waiting List: N** はジョブが **Fuxi** へ送信され、キューにつながれており、キュー内で **n** ビットであることを示します。
- **Running:** ジョブは **Fuxi** で実行中です。
- **Terminated:** ジョブはキュー情報無しで終了しました。

ジョブの未終了ステータスをクリックし、キューの詳細情報を確認します。

ステータスをクリックし、キューの詳細を確認します。

- **Sub status:** 現在のサブステータス情報です。
- **Waitpos:** キューイングの場所のことで、**0** は実行中、**(-)** は **Fuxi** へ到達していないことを示します。
- **Queuing length: Fuxi** のキューの長さの合計です。
- **Total priority:** システムがジョブ実行時間により判断する優先度です。
- **SubStatus history:** クリックすると、ジョブ実行の詳細な履歴を表示できます。 ステータスコード、ステータスの説明、開始時間、実行時間などを含みます。 **(**現在、一部のバージョンには履歴情報がありません**)**

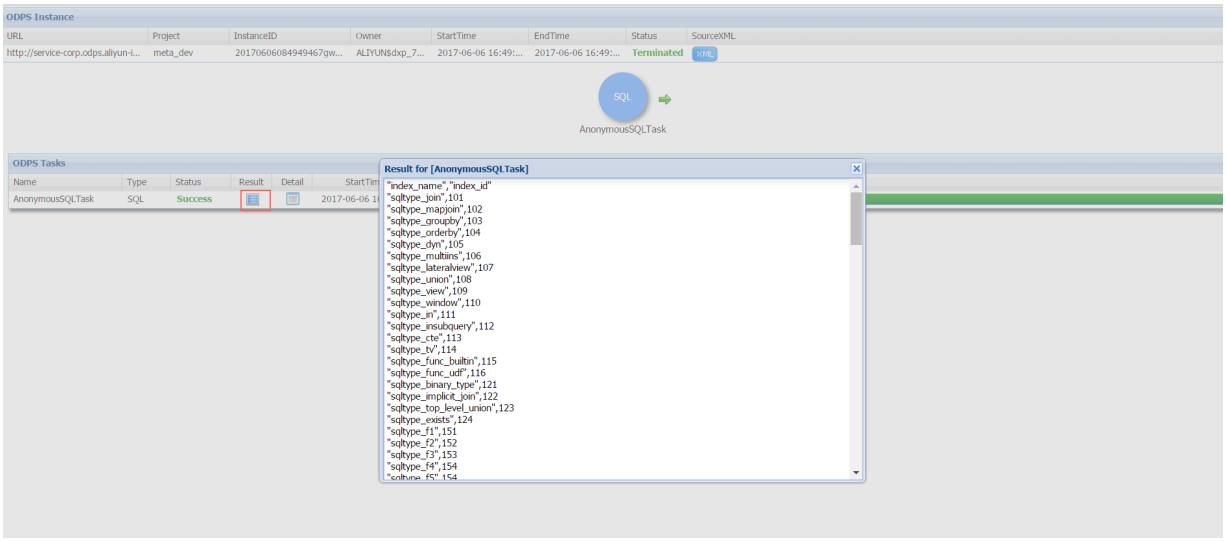


タスク情報

Logview ホームページの下部セクションは、タスク説明、結果の説明および他の詳細情報です。

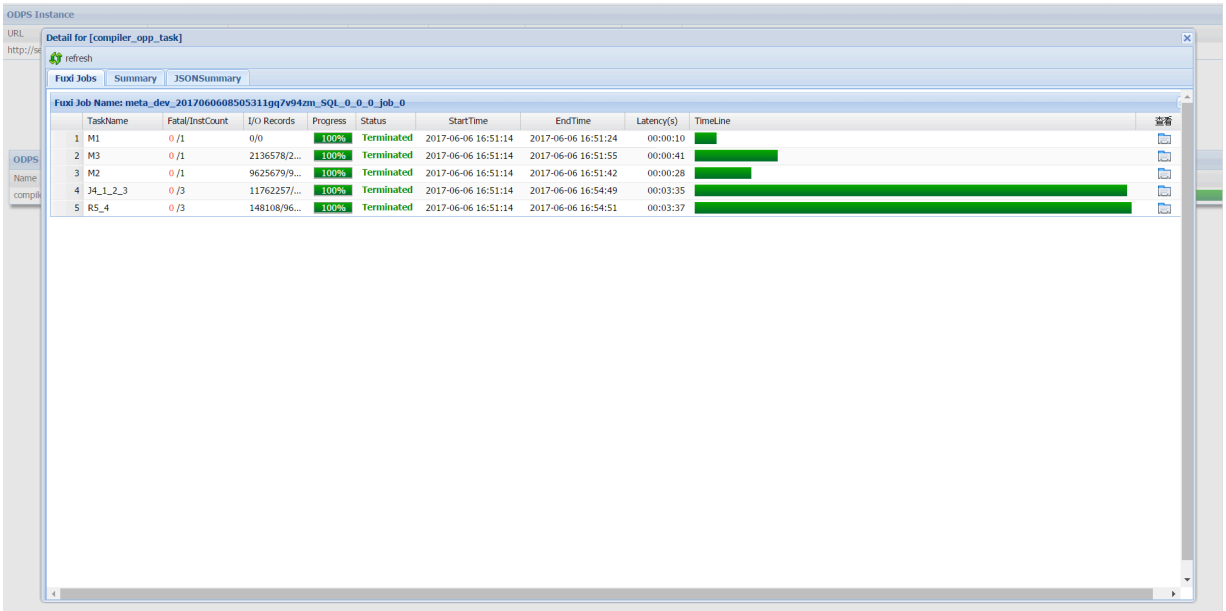
結果

ジョブの実行後、以下の図のように結果 **(select SQL** 結果など**)** が表示されます。



詳細

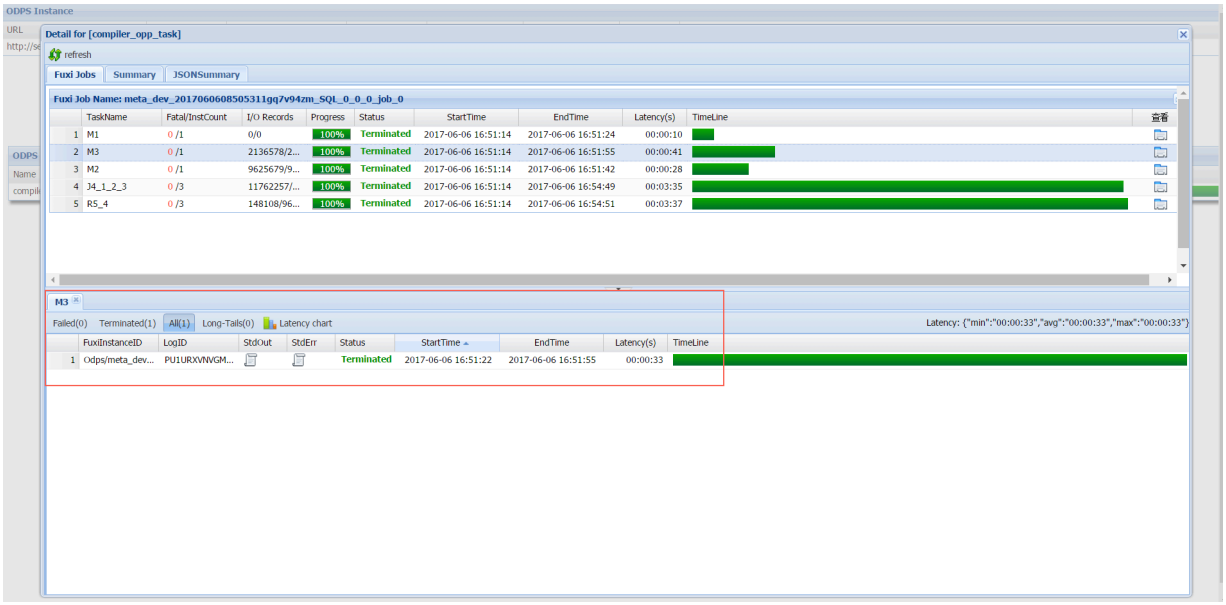
ジョブの実行後、詳細をクリックするとタスクの実行ステータスが表示されます。



- **MaxCompute** タスクは **1** つ以上の **Fuxi** ジョブで構成されます。 たとえば、**SQL** タスクが複雑な場合、**MaxCompute** は **Fuxi** に複数の **Fuxi** ジョブを送信します。
- 各 **Fuxi** ジョブは **1** つ以上の **Fuxi** タスクで構成されます。 シンプルな **MapReduce** は通常、**Map** と **Reduce** という **2** つの **Fuxi** タスクを生成します。 **2** つの **Fuxi** タスク名を順番に **M1** および **M2** として表示できます。 **SQL** が複雑な場合、**1** つ以上の **Fuxi** が上述の図で示されるようなタスクを生成する場合があります。
- 各タスクはタスク名を表示します。 たとえば、**M1** はマップタスクで、 **r54** の **4** は **J4** に依存していることを意味します。 最後の実行が完了するまで実行は開始されません。 同様に、**j4_1_2_3** は **join4** の開始条件が **M1**、**M2**、**M3** の **3** つのタスクに完全に依存していることを示します。

・ **I/O Records** は、このタスクの入出力のレコード数を表します。

いずれかの **Fuxi** タスクをクリックすると、 以下の図で示すような **Fuxi** インスタンスの内容が表示されます。



各 **Fuxi** タスクは 1 つ以上の **Fuxi** インスタンスで構成されます。 入力データレベルが大きい場合、**MaxCompute** はより多くのノードを起動してデータを処理します。 各ノードは **Fuxi** インスタンスです。 右側の **Fuxi** タスク列をダブルクリックして確認するか、行をダブルクリックして特定の **Fuxi** インスタンス情報を開きます。

ページの末尾に進むと、**Logview** は異なるステージのインスタンスでグループ化されます。 **failed** 列をクリックして、誤ったノードを表示します。

stdout と **stderr** 列では、標準出力と標準エラーメッセージが、出力される情報とともに確認できます。

Logview によるトラブルシューティング

・ タスクエラー

タスクエラーが発生した場合、**Logview** ページ上の結果にエラーを示すプロンプトメッセージが表示されます。 **Fuxi** インスタンスの **Stderr** 詳細ページを使用して、特定インスタンスのエラーに関する情報を確認します。

・ データスキュー

動作が遅い場合は、通常、すべての **Fuxi** インスタンス内の個々のインスタンスの中の特定の **Fuxi** タスクが原因です。 ロングテールは同一タスク内でのタスク割り当て不均等によって引

き起こされます。 タスクが実行された後、サマリータブで実行結果を確認できます。 各タスクの出力は以下のとおりです。

```
output records:
R2_1_Stg1: 199998999 (min: 22552459, max: 177446540, avg: 99999499)
```

上の図では、**min** と **max** に大きな差があります、これは、このステージでデータの偏りが発生していることを示しています。 つまり、高頻度のワードが **1** つ出現すると、このワードを結合する際に偏りが発生します。

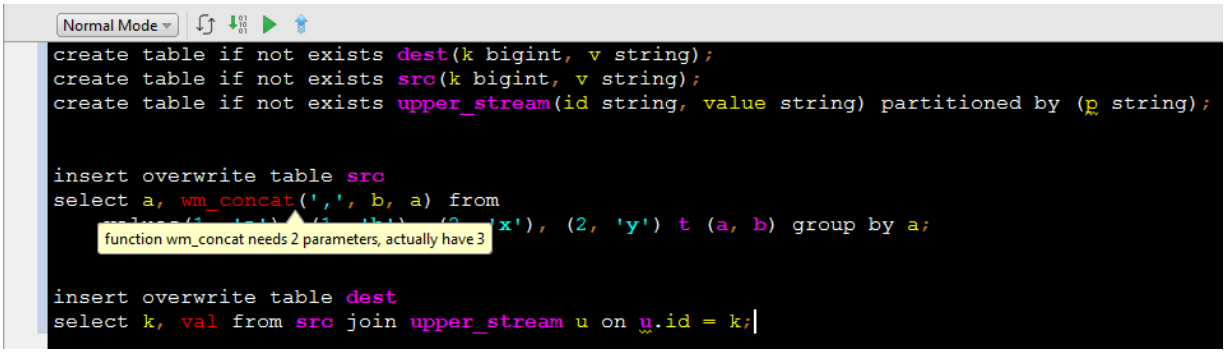
11.2 MaxCompute コンパイラ使用時のエラーと警告

MaxCompute コンパイラは、**SQL** を大幅に強化する **MaxCompute2.0** と呼ばれる次世代 **SQL** エンジンがベースとなっています。 言語コンパイルプロセスおよび言語表現能力を容易にします。 ここでは、コンパイラの高度な活用方法を紹介します。

コンパイラの利便性向上

MaxCompute コンパイラの利便性を向上させるには、**MaxCompute Studio** とともに使用することを推奨します。

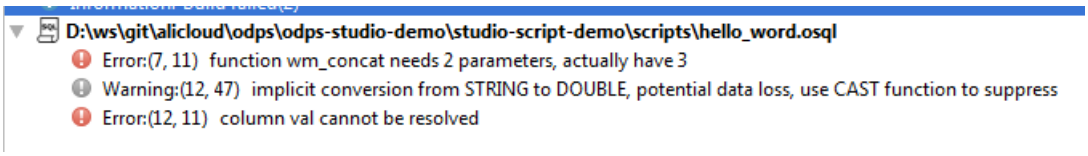
まず、**MaxCompute Studio** をインストールし、**MaxCompute** プロジェクトを追加してプロジェクトを作成した後に、新しい **MaxCompute** を作成します。 スクリプトは以下のとおりです。



上の図では、以下の問題が検出されています。

- **wm_concat** 関するのエラーが、最初の **insert** 文で見つかります。
- **MaxCompute** は **bigint** 型と **double** 型のデータを比較し、すべてのデータを **double** 型に変換します。 この文字列から **double** 型への変換は、 **SQL** 実行時にエラーが発生する可能性があります。 ただし、**MaxCompute** はこの操作を行うかどうか、ユーザーに警告を表示します。

マウスカーソルをエラーまたは警告プロンプトに直接ポイントし、特定のエラーまたは警告メッセージを表示します。 エラーを修正せず、そのままコミットすると、以下の図に示すように、**MaxCompute Studio** でブロックされます。



プロンプトに従い、以下のようにエラーと警告を修正してください。

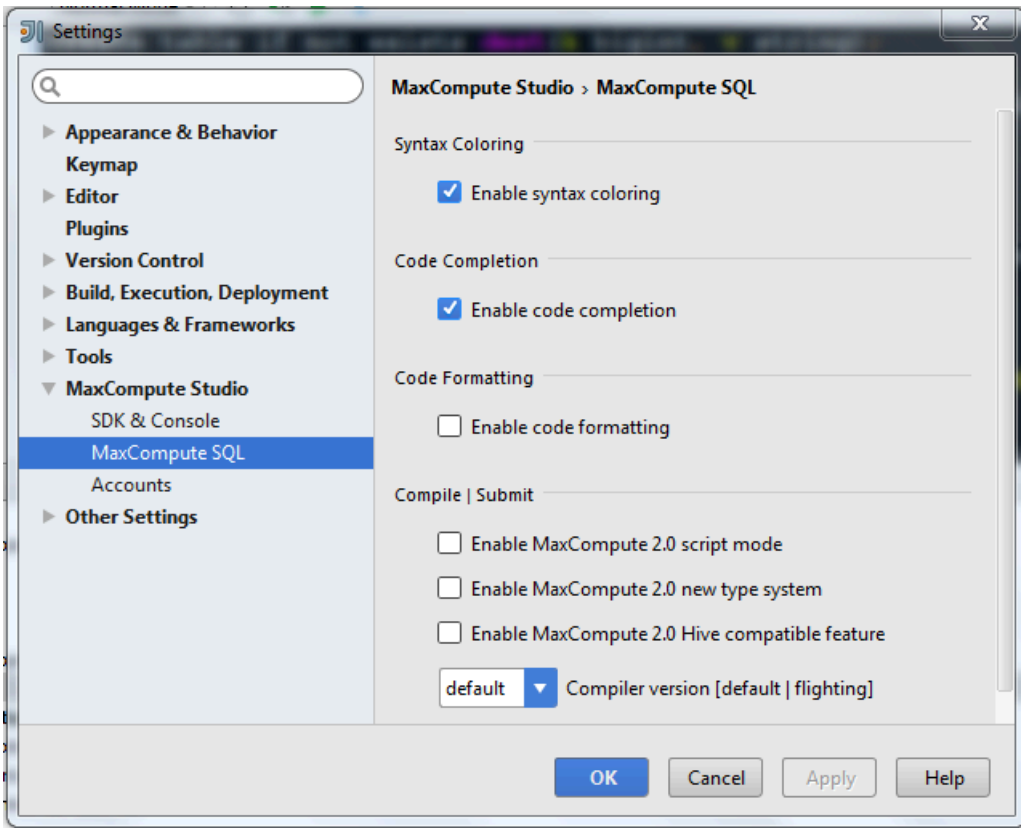
```
create table if not exists dest(k bigint, v string);
create table if not exists src(k bigint, v string);
create table if not exists upper_stream(id string, value string) partitioned by (dt string);


insert overwrite table src
select a, wm_concat(',', b) from
    values(1, 'a'), (1, 'b'), (2, 'x'), (2, 'y') t (a, b) group by a;


insert overwrite table dest
select k, value from src join upper_stream u on u.id = string(k);
```

修正後、スクリプトを再度送信すると、問題なく実行できるようになります。

以下の図に示すように、**MaxCompute Studio** を使用して、すべての警告をエラーとして設定することもできます。



上述のように設定すれば、どのようなエラーも誤って見逃すことがありません。

スクリプトを送信する前に **MaxCompute Studio** を使用することを推奨します。静的コンパイル用のチェックが行われるので、警告をエラーとして設定することを推奨します。 スクリプトの送信前に警告を修正することで時間とリソースを節約することができます。 さらに、エラースクリプトが送信されると、ヘルススコア計算に送られます。 これにより、今後のタスクの優先度が下がります。 また、今後の未修正の警告もヘルスシステムに取り込まれます。 つまり、**MaxCompute** コンパイラ と **Studio** は、品質が低下することはありません。

多くのシナリオにおいて、暗黙の型変換は安全ではないという警告を受け取るかもしれません。ただし、この変換が必要であれば、**cast (xxx As)** を使用することで、警告は表示されなくなります。**MaxCompute** またはコンパイラを使用して問題を解決します。