# Alibaba Cloud E-MapReduce

## 開発ガイド

Document Version20200426

# 目次

1 E-MapReduce での Python の使用方法........................................ 1
2 準備........................................................................ 3
2.1 開発準備........................................................................ 3
2.2 OSS URI を設定してE-MapReduce を使用........................................ 3
2.3 サンプルプロジェクトの使用........................................................ 4
3 Spark........................................................................ 16
3.1 準備........................................................................ 16
3.2 パラメーター説明........................................................................ 18
3.3 OSS ファイルの操作........................................................................ 21
3.4 Spark を使用して OSS へアクセス........................................ 22
3.5 スパークにおける MaxCompute の利用........................................ 23
3.6 Spark Streaming を使用して MQ データを消費........................................ 25
3.7 Spark で Table Store データを消費........................................ 26
3.8 Spark Streaming を使用して MNS データを消費........................................ 27
3.9 Spark を使用して HBase にデータを書き込む........................................ 28
3.10 Spark Streaming を使用して Kafka データを処理........................................ 29
3.11 Spark を使用して MySQL にデータを書き込む........................................ 30
3.12 Spark-Submit パラメーターの設定........................................ 31
4 Spark Streaming SQL........................................ 38
4.1 はじめに........................................................................ 38
4.2 キーワード........................................................................ 38
4.3 ストリーミングクエリ........................................................................ 38
4.3.1 ストリーミングクエリ設定........................................ 38
4.3.2 ジョブテンプレート........................................ 40
4.4 DML の概要........................................................................ 42
4.4.1 DML 概要........................................................................ 42
4.5 クエリ概要........................................................................ 43
4.5.1 SELECT 文........................................................................ 43
4.5.2 WHERE 文........................................................................ 43
4.5.3 GROUP BY 文........................................................................ 44
4.5.4 JOIN 文........................................................................ 44
4.5.5 UNION ALL 文........................................................................ 45
4.6 ウィンドウ関数........................................................................ 46
4.6.1 概要........................................................................ 46
4.6.2 タンブリングウィンドウ........................................ 47
4.6.3 スライディングウィンドウ........................................ 47
4.7 データソース........................................................................ 48
4.7.1 概要........................................................................ 48
4.7.2 Kafka データテーブルの説明........................................ 49

4.7.3 LogHub データテーブルの説明........50
4.7.4 HBase データテーブルの説明........51
4.7.5 JDBC データテーブルの説明........53
4.7.6 Table Store データテーブルの説明........55
4.7.7 Druid データテーブルの説明........56
4.7.8 Redis データテーブルの説明........58
**5 Hadoop........61**
5.1 パラメーター説明........61
5.2 MapReduce ジョブの作成と実行........62
5.3 Hive ジョブの作成と実行........70
5.4 Pig ジョブの作成および実行........74
5.5 Hadoop ストリーミングジョブの作成........76
5.6 Hive で Table Store データを処理........77
5.7 MR での Table Store データの処理........79
**6 HBase クラスターを作成して HBase ストレージサービスを使用........83**
**7 HBase のバックアップ........88**
**8 E-MapReduce クラスター運用ガイド........90**

# 1 E-MapReduce での Python の使用方法

E-MapReduce (EMR) 2.0.0 以降で Python を使用できます。 このページでは、Python のインストール方法について説明します。

**Python 2.7**

EMR 2.0.0 以降で Python 2.7 を使用できます。

デフォルトで Python は usr/local/Python-2.7.11/ ディレクトリにインストールされます。NumPy が含まれます。

**Python 3.6**

EMR 2.10.0 以降および 3.10.0 以降で Python 3.6.4 を使用できます。 デフォルトで Python 3.6.4 は /usr/bin/python3.6ディレクトリにインストールされます。 以下のコマンドを使用して、Python 3 がインストールされているかどうかを確認できます。

```
[root@emr-header-1 ~]# python36
```

pip3 ツールはデフォルトではプリインストールされていません。 必要に応じてツールをインストールできます。

デフォルトでは、EMR 2.10.0 または EMR 3.10.0 より前のバージョンでは Python 3 を使用できません。 次のように Python 3 をダウンロードしてインストールする必要があります。

1. こちらのリンク から Python 3 のインストレーションパッケージをダウンロードします。
2. ダウンロードしたファイルを展開し、Python 3 をインストールします

```
tar zxvf Python-3.6.4.tgz
cd Python-3.6.4 ./configure --prefix=/usr/local/Python-3.6.4
make && make install
ln -s /usr/local/Python-3.6.4/bin/python3.6 /bin/python3
ln -s /usr/local/Python-3.6.4/bin/pip3 /bin/pip3
```

3. Python 3 のインストール結果を確認します。

```
[root@emr-header-1 bin]# python3
```

```
Python 3.6.4 (default, Mar 12 2018, 14:03:26)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linuxType “help”, “copyright”, “credits
” or “license” for more information.
```

```
[root@emr-header-1 bin]# pip3 -V
pip 9.0.1 from /usr/local/Python-3.6.4/lib/python3.6/site-packages (python 3.6)
```

前述のコマンド情報が表示された場合は、Python 3 のインストールは成功です。

# 2 準備

## 2.1 開発準備

本ページでは、E-MapReduce 開発に必要な準備について説明します。

- すでに Alibaba Cloud サービスを有効にし、AccessKey ID と AccessKey Secret を作成していることを前提とします。 ID は AccessKey ID、KEY はAccessKey Secret を表します。 OSS サービスを有効にしていない、または OSS サービスについて不明な点がある場合は、OSS のプロダクトページにログインし、ヘルプをご確認ください。
- Spark、Hadoop、Hive、Pig について基本的に理解している方を前提とします。 このページでは、Spark、Hadoop、Hive、および Pig の開発手法については説明しません。 詳細は、Apache Web サイトをご参照ください。
- Alibaba Cloud Elastic MapReduce (E-MapReduce) 開発コンポーネントについて基本的に理解している方を前提とします。 プロジェクト推進に役立つオープンソースコミュニティ、問題のフィードバック、バグ修正、および新しいコンポーネントなどをお知らせください。

## 2.2 OSS URI を設定してE-MapReduce を使用

このページでは、OSS URI を設定して E-MapReduce を使用する方法について説明します。

### OSS URI

E-MapReduce を使用するときは、2 種類の OSS URI を使用できます。

- ネイティブ URI : oss://[accessKeyId:accessKeySecret@]bucket[.endpoint]/object/path

  この URI は、ジョブ内で入出力データソースを指定する場合に使用します。hdfs:// と同様の URI です。 OSS データの操作時に、AccessKey ID、AccessKey Secret、およびエンドポイントを設定できます。 または、URI に AccessKey ID、AccessKey Secret、およびエンドポイントを指定することもできます。
- 参照 URI : ossref://bucket/object/path

  E-MapReduce ジョブの設定でのみ有効で、ジョブの実行に必要なリソースを指定する場合に使用します。

oss や ossref などのプレフィックスをスキームと呼びます。 スキームの URI との違いには特にご注意ください。

:

現在、サポートされている操作は標準ストレージタイプの OSS だけです。

- E-MapReduce はマルチパートモードを使用して大規模ファイルを OSS にアップロードします。 ジョブが中断されても、結果データは一部 OSS に残ります。 手動で削除する必要があります。 ここでの手順は HDFS を使用する場合と同じです。 ただし、E-MapReduce はマルチパートモードを使用して大規模ファイルをアップロードする点が異なります。 ファイルフラグメントは OSS フラグメント管理にアップロードされます。 そのため、OSS ファイル管理に残っているジョブファイルを削除し、OSS フラグメント管理に残っているファイルフラグメントを消去する必要があります。 削除しないでいると、データストレージの料金が課金されます。
- 上記の手動クリーンアップ以外に、期限切れのフラグメントが自動的に削除されるよう、フラグメントのライフサイクルを設定する対応を取ることもできます。 詳細は、「OSS ファイルのライフサイクル管理 (Lifecycle management of OSS files)」をご参照ください。

## 2.3 サンプルプロジェクトの使用

このサンプルプロジェクトは、MapReduce、Pig、Hive、および Spark のサンプルコードを含む、完全、コンパイル可能、実行可能なプロジェクトです。

### サンプルプロジェクト

オープンソースプロジェクトをご確認ください。 詳細は以下のとおりです。

- MapReduce

  WordCount : 単語数をカウントします。
- Hive

  sample.hive : テーブルを簡単に検索します。
- Pig

  sample.pig : Pig が処理する OSS データインスタンス

- Spark
  - SparkPi : Pi を計算します。
  - SparkWordCount : 単語数をカウントします。
  - LinearRegression : 線形回帰
  - OSSSample : OSS のサンプル
  - ODPSSample: ODPS サンプル
  - MNSSample: MNS サンプル
  - LoghubSample: Loghub サンプル

**依存関係**

- テストデータ (データディレクトリの下)
  - The_Sorrows_of_Young_Werther.txt: WordCount (MapReduce/Spark) の入力データとして使用します。
  - patterns.txt: WordCount (MapReduce) ジョブの文字をフィルタリングします。
  - u.data: sample.hive スクリプトのテストテーブルのデータ
  - abalone: 線形回帰アルゴリズムのテストデータ
- JAR 依存関係 (lib ディレクトリの下)

  tutorial.jar: sample.pig ジョブに必要な JAR 依存関係

**準備**

このプロジェクトはいくつかのテストデータを提供します。 OSS にアップロードするだけで使用できます。 MaxCompute、MNS、ONS、Log Service など他のサンプルの場合は、以下のとおりデータを準備する必要があります。

- (オプション) Log Service を有効にします。 『Log Service ユーザーガイド』をご参照ください。
- (オプション) MaxCompute のプロジェクトとテーブルを作成します。 「MaxCompute プロジェクトの作成 (Create a MaxCompute project)」 および 『MaxCompute クイックスタート』をご参照ください。
- (オプション) ONS を使用します。 「メッセージキューのクイックスタート (Quick Start for Message Queue)」ご参照ください。
- (オプション) MNS を使用します。 Message Service コンソールの使用に関するページをご参照ください。

### 概念

- OSSURI: oss://accessKeyId:accessKeySecret@bucket.endpoint/a/b/c.txt. 入力データソースと出力データソースを指定します。 OSSURI は、hdfs:// などの URL と類似しています。
- AccessKey ID と AccessKey Secret の組み合わせは、Alibaba Cloud API にアクセスするうえで鍵となります。 取得するにはこちらをクリックします。

### クラスターでジョブを実行

- Spark
  - SparkWordCount：

    ```
    spark-submit --class SparkWordCount examples-1.0-SNAPSHOT-shaded.jar<
    inputPath>
               <outputPath> <numPartition>
    ```

    パラメーターの記述は以下のとおりです。

    - inputPath：データ入力パス
    - outputPath：データ出力パス
    - numPartition：入力データの RDD パーティションの数
  - SparkPi: `spark-submit --class SparkPi examples-1.0-SNAPSHOT-shaded.jar`
  - OSSSample:

    ```
    spark-submit --class OSSSample examples-1.0-SNAPSHOT-shaded.jar <inputPath>
               <numPartition>
    ```

    パラメーターの記述は以下のとおりです。

    - inputPath：データ入力パス
    - numPartition：データ RDD パーティション数を入力します。
  - ONSSample:

    ```
    spark-submit --class ONSSample examples-1.0-SNAPSHOT-shaded.jar <accessKeyI
    d>
    ```

```
        <accessKeySecret> <consumerId> <topic> <subExpression> <parallelism>
```

パラメーターの記述は以下のとおりです。

- accessKeyId: Alibaba Cloud AccessKey ID.
- accessKeySecret: Alibaba Cloud AccessKey Secret
- consumerId: See Consumer ID description.
- topic : 各メッセージキューにはトピックがあります。
- subExpression: See Message filtering.
- parallelism : キュー内のメッセージを消費するレシーバー数を指定します。

- ODPSSample:

```
spark-submit --class ODPSSample examples-1.0-SNAPSHOT-shaded.jar <
accessKeyId>
        <accessKeySecret> <envType> <project> <table> <numPartitions>
```

パラメーターの記述は以下のとおりです。

- accessKeyId : Alibaba Cloud AccessKey ID
- accessKeySecret: Alibaba Cloud AccessKey Secret.
- envType : 0 はパブリックネットワーク、1 はプライベートネットワークを示します。ローカルデバッグには 0、E-MapReduce での実行には 1 を選択します。
- project: 「ODPS クイックスタート (ODPS Quick Start)」をご参照ください。
- numPartition: 入力データの RDD パーティションの数

- MNSSample:

```
spark-submit --class MNSSample examples-1.0-SNAPSHOT-shaded.jar <
queueName>
        <accessKeyId> <accessKeySecret> <endpoint>
```

パラメーターの記述は以下のとおりです。

- queueName: キュー名 コア概念をご参照ください。
- accessKeyId: Alibaba Cloud AccessKey ID
- accessKeySecret: Alibaba Cloud AccessKey Secret
- endpoint: キューデータにアクセスするためのアドレス

- LoghubSample:

```
spark-submit --class LoghubSample examples-1.0-SNAPSHOT-shaded.jar <sls
project> <sls
        logstore> <loghub group name> <sls endpoint> <access key id> <access
key secret> <batch
```

```
            interval seconds>
```

パラメーターの記述は以下のとおりです。

- sls project: Log Service プロジェクト名
- sls logstore: Logstore 名
- loghub group name: ジョブで Logstore データを消費するグループの名前 必要に応じて名前を指定できます。 sls project と sls store の値が同じ場合、同じグループ名のジョブは sls store 内のデータを共同で消費します。 グループ名が異なるジョブは、sls store 内のデータを個別に消費します。
- sls endpoint: 「サービスエンドポイント (Service endpoint)」をご参照ください。
- accessKeyId: Alibaba Cloud AccessKey ID
- accessKeySecret: Alibaba Cloud AccessKey Secret
- batch interval seconds: Spark Streaming ジョブ間の間隔 (単位：秒)

- LinearRegression:

```
spark-submit --class LinearRegression examples-1.0-SNAPSHOT-shaded.jar <
inputPath>
            <numPartitions>
```

パラメーターの記述は以下のとおりです。

- inputPath: データ入力パス
- numPartition: 入力データの RDD パーティションの数

- MapReduce
  - WordCount:

```
hadoop jar examples-1.0-SNAPSHOT-shaded.jar WordCount
            -Dwordcount.case.sensitive=true <inputPath> <outputPath> -skip <
patternPath>
```

パラメーターの記述は以下のとおりです。

- inputPath: データ入力パス
- outputPath: データ出力パス
- patternPath: フィルタリングする文字を含むファイル。 data/patterns.txt を使用できます。

- Hive
  - `hive -f sample.hive -hiveconf inputPath=<inputPath>`

    パラメーターの記述は以下のとおりです。

    - inputPath: データ入力パス
- Pig
  - ```
    pig -x mapreduce -f sample.pig -param tutorial=<tutorialJarPath> -param
            input=<inputPath> -param result=<resultPath>
    ```

    パラメーターの記述は以下のとおりです。

    - tutorialJarPath: JAR 依存関係。 lib/tutorial.jar を使用できます。
    - inputPath: データ入力パス
    - resultPath: データ出力パス

:

- E-MapReduce で作業している場合は、テストデータと JAR 依存関係を OSS にアップロードします。 パスルールは、上記のとおり OSSURI の定義に従います。
- クラスター内で使用する場合は、ローカルに格納できます。

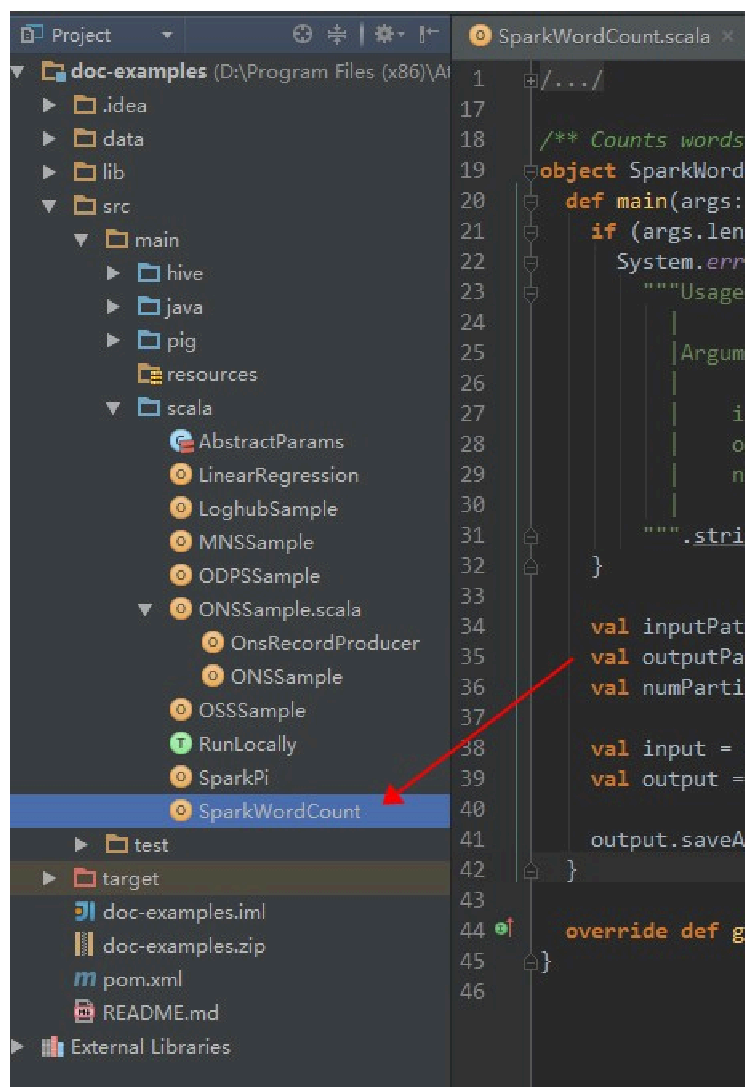
**ローカルに実行**

ここでは、Spark プログラムをローカルで実行して、OSS など Alibaba Cloud のデータソースにアクセスする方法について説明します。 プログラムをローカルでデバッグして実行する場合、特に Windows を使用している場合は、IntelliJ IDEA や Eclipse など何らかの開発ツールを使用するよう推奨します。 開発ツールを使用しない場合は、Windows マシン上で Hadoop および Spark 実行時環境を設定する必要があります。
    ```

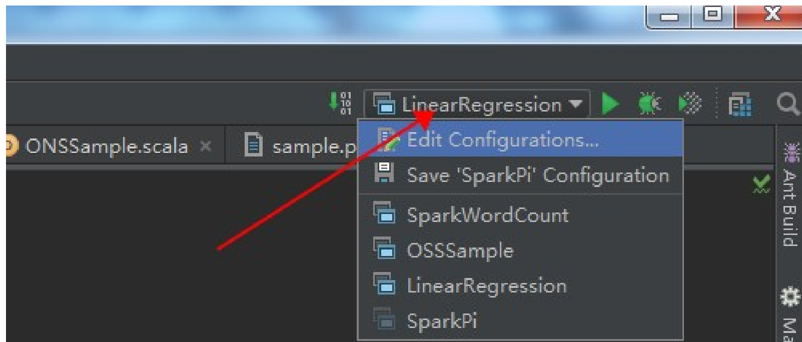
- IntelliJ IDEA
  - 準備

    IntelliJ IDEA、Maven、IntelliJ IDEA 用の Maven プラグイン、IntelliJ IDEA 用の Scala および Scala プラグインをインストールします。
  - 開発プロセス

    **1.** ダブルクリックして SparkWordCount.scala を入力します。

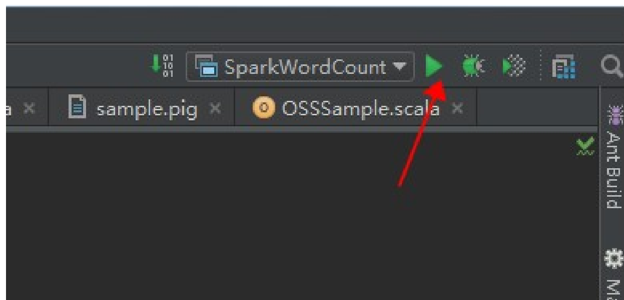



    **2.** 方向ボタンをクリックしてジョブ設定ページに入ります (次の図を参照)。




3. SparkWordCount を選択し、必要なジョブパラメーターをジョブパラメーターボックスに入力します。

4. **[OK]** をクリックします。
5. [実行] をクリックしてジョブを実行します。

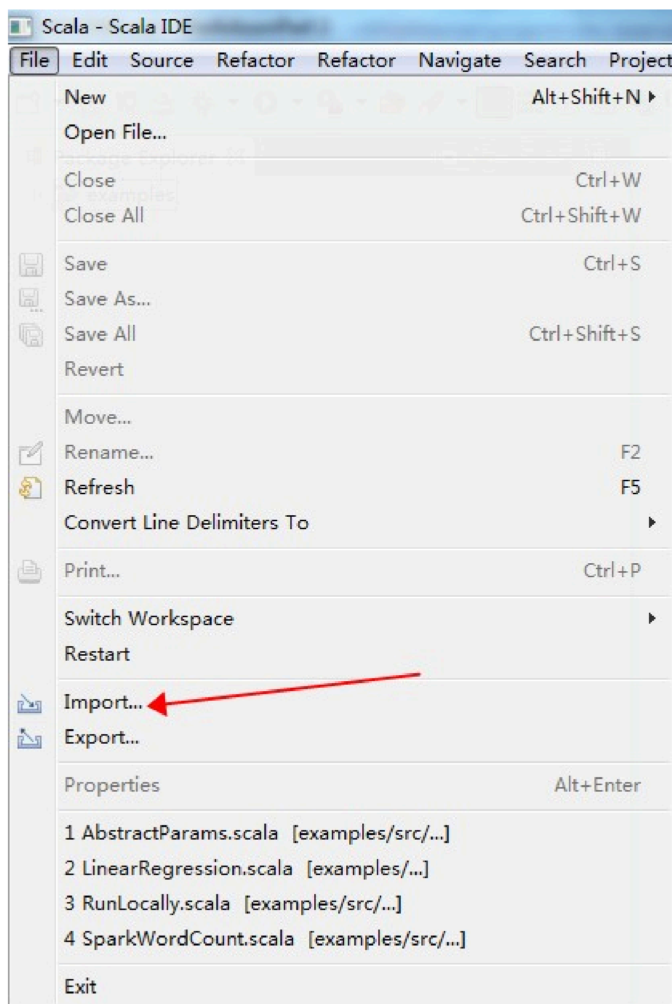


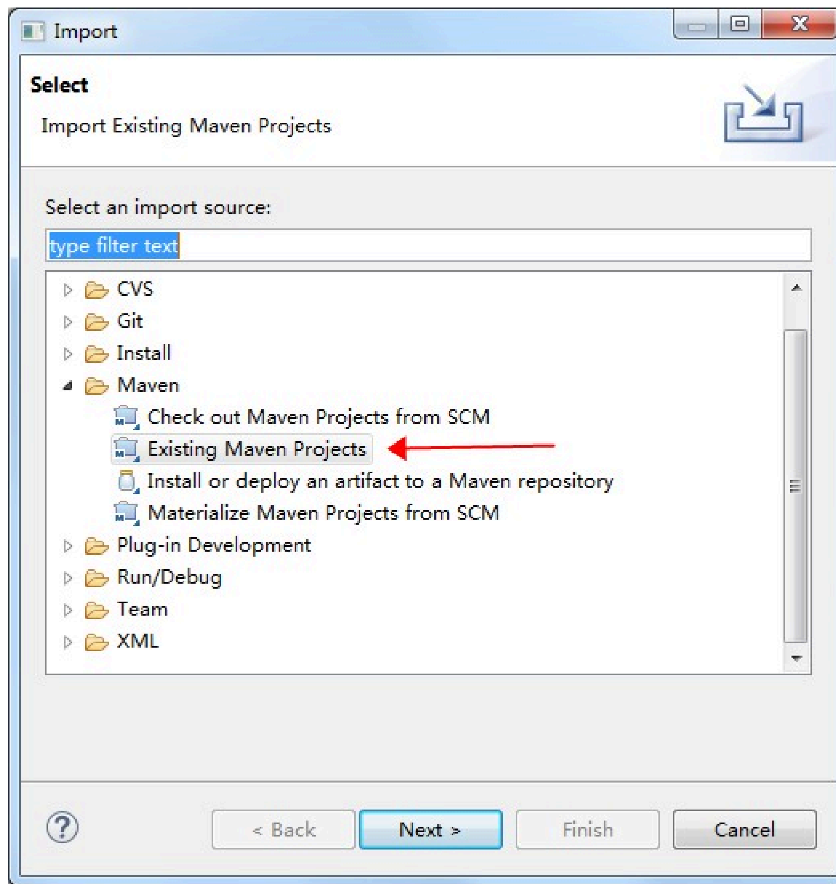

6. ジョブログを閲覧します。

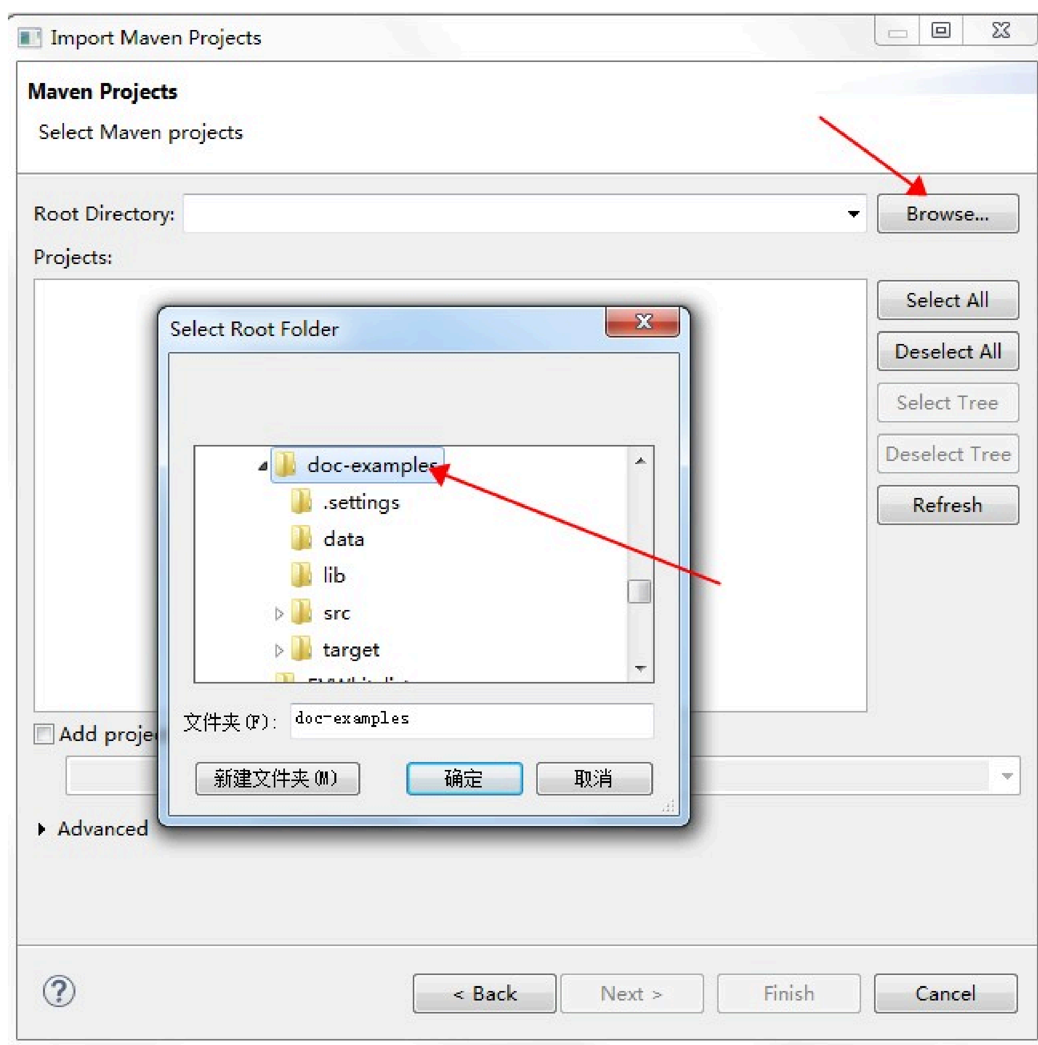
- Scala IDE for Eclipse
  - 準備

    Scala IDE for Eclipse、Maven、および Eclipse 用の Maven プラグインをインストールします。
  - 開発プロセス

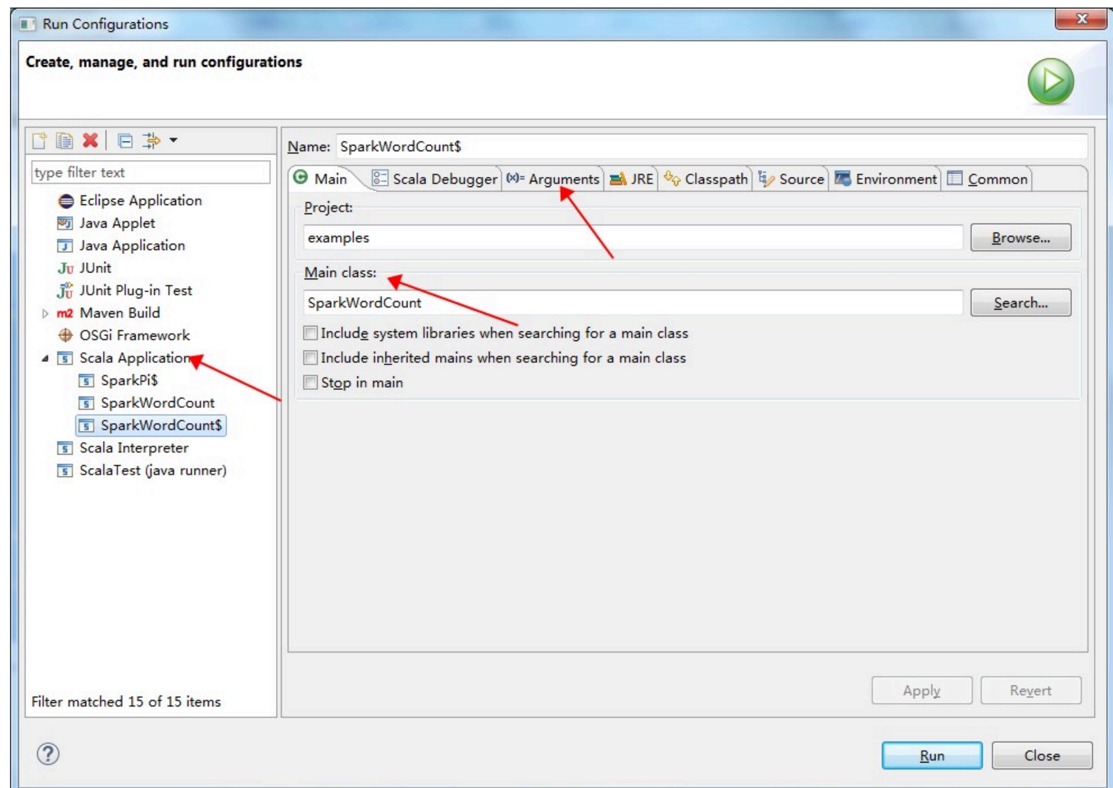
    1. 次の図のとおりプロジェクトをインポートします。




Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
CVS
Git
Install
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Plug-in Development
Run/Debug
Team
XML
< Back
Next >
Finish
Cancel




2. The shortcut for Run as Maven ビルドでの実行時のショートカットは Alt + Shift + X, M です。プロジェクト名を右クリックして **[Run As]** > **[Maven ビルド]** をクリックすることもできます。
3. コンパイル後に実行するジョブを右クリックし、**[設定の実行]** を選択して設定ページに入ります。
4. 設定ページで **[Scalaアプリケーション]** を選択してジョブのメインクラスとパラメーターを設定します (次の図を参照)。




5. **[実行]** をクリックします。
6. コンソールの出力ログを閲覧します (次の図を参照)。

# 3 Spark

## 3.1 準備

このページでは、Spark 開発の準備方法について説明します。

### E-MapReduce SDK のインストール

以下のどちらかの方法を使用して E-MapReduce SDK をインストールできます。

- 方法 1 : Eclipse で JAR パッケージを直接使用 以下のステップを実行します。
  1. Mavenリポジトリの E-MapReduce に必要な依存関係をダウンロードします。
  2. 必要なJARファイルをプロジェクトディレクトリにコピーします。 (ジョブを実行するクラスターで使用している Spark のバージョンに基づいて SDK バージョン 2.10 または 2.11 を選択します。 Spark1.x にはバージョン 2.10、Spark2.x には 2.11 が推奨されます)。
  3. Eclipse でプロジェクト名を右クリックし、 **[プロパティ]** > **[Java** ビルドパス**]** > **[JAR** の追加**]** をクリックします。
  4. ダウンロードした SDK を選択します。
  5. 上記のステップを完了したら、OSS、Log Service、MNS、ONS、Table Store、および MaxCompute でデータを読み書きできます。
- 方法 2 : Maven プロジェクトを編集して、以下のとおり依存関係を追加します。

```
<! -- Support for OSS data sources -->
   <dependency>
      <groupId>com.aliyun.emr</groupId>
      <artifactId>emr-core</artifactId>
      <version>1.4.1</version>
   </dependency>
   <! -- Support for Table Store data sources -->
   <dependency>
      <groupId>com.aliyun.emr</groupId>
      <artifactId>emr-tablestore</artifactId>
      <version>1.4.1</version>
   </dependency>
   <! -- Support for Message Service (MNS), Open Notification Service (ONS), Log 
Service, MaxCompute data sources (Spark 1.x environment) -->
   <dependency>
      <groupId>com.aliyun.emr</groupId>
      <artifactId>emr-mns_2.10</artifactId>
      <version>1.4.1</version>
   </dependency>
   <dependency>
      <groupId>com.aliyun.emr</groupId>
      <artifactId>emr-logservice_2.10</artifactId>
      <version>1.4.1</version>
   </dependency>
   <dependency>
```

```
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-maxcompute_2.10</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-ons_2.10</artifactId>
        <version>1.4.1</version>
    </dependency>
    <! -- Support for MNS, ONS, Log Service, MaxCompute data sources (spark 2.x
environment) -->
    <dependency>
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-mns_2.11</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-logservice_2.11</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-maxcompute_2.11</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>com.aliyun.emr</groupId>
        <artifactId>emr-ons_2.11</artifactId>
        <version>1.4.1</version>
    </dependency>
```

### ローカルで Spark コードをデバッグ

spark.hadoop.mapreduce.job.run-local は、Spark が OSS を読み書きするためのデバッグに必要なシナリオ用のプロパティです。 他のシナリオではデフォルト設定を使用します。

OSS の読み書き用に Spark をデバッグする必要がある場合は、spark.hadoop.mapreduce.job.run-local を true に設定して SparkConf オブジェクトを設定します。 以下のサンプルコードをご参照ください。

```
val conf = new SparkConf().setAppName(getAppName).setMaster("local[4]")
   conf.set("spark.hadoop.fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssFileSystem")
   conf.set("spark.hadoop.mapreduce.job.run-local", "true")
   val sc = new SparkContext(conf)
   val data = sc.textFile("oss://...")
   println(s"count: ${data.count()}")
```

### サードパーティの依存関係の説明

E-MapReduce が Alibaba Cloud データソース (OSS および MaxCompute など) にアクセスできるようにするには、必要なサードパーティの依存関係をインストールする必要があります。

pom ファイルを参照して必要なサードパーティの依存関係を追加または削除します。

#### OSS 出力ディレクトリ設定

パラメーター **fs.oss.buffer.dirs** をローカルの hadoop 設定ファイルに設定します。 パラメーター値はローカルディレクトリパスです。 パラメーター値が設定されていない場合、Spark 実行のプロセス中に OSS データが書き込まれると、NULL ポインタ例外が発生します。

#### データのクリーンアップ

Spark ジョブが失敗しても、生成されたデータは自動的には削除されません。 ジョブが失敗した場合は、OSS 出力ディレクトリをチェックして、ファイルが存在するかどうか確認します。 サブミットされていないフラグメントについては、OSS フラグメント管理もチェックする必要があります。 見つかったフラグメントを削除します。

#### PySpark の使用

Python を使用して Spark ジョブを作成する場合は、環境の設定について「注意事項」をご参照ください。

## 3.2 パラメーター説明

Spark コード内のパラメーターの説明

Spark コードでは、以下のパラメータを設定できます。

| プロパティ | デフォルト | 説明 |
| --- | --- | --- |
| spark.hadoop.fs.oss.accessKeyId | なし | (省略可能) OSS へのアクセスに必要な AccessKey ID |
| spark.hadoop.fs.oss.accessKeySecret | なし | (省略可能) OSS へのアクセスに必要な AccessKey Secret |
| spark.hadoop.fs.oss.securityToken | なし | (省略可能) OSS へのアクセスに必要な STS トークン |
| spark.hadoop.fs.oss.endpoint | なし | (省略可能) OSS へのアクセスに使用するエンドポイント |
| spark.hadoop.fs.oss.multipart.thread.number | 5 | アップロード部分 - コピー操作に OSS が使用するスレッド (同時実行) の数 |
| spark.hadoop.fs.oss.copy.simple.max.byte | 134217728 | 一般的な API を使用して OSS のバケット間でファイルをコピーする場合のファイルサイズ制限 |

| プロパティ | デフォルト | 説明 |
|---|---|---|
| spark.hadoop.fs.oss.multipart.split.max.byte | 67108864 | 一般的な API を使用して OSS のバケット間でファイルをコピーする場合のファイルマルチパート制限 |
| spark.hadoop.fs.oss.multipart.split.number | 5 | 一般的な API を使用して OSS のバケット間でファイルをコピーする場合のマルチパートファイルの数 デフォルトの数は使用されているスレッド (同時実行) の数と同じです。 |
| spark.hadoop.fs.oss.impl | com.aliyun.fs.oss.nat.NativeOssFileSystem | OSS のネイティブファイルシステムの実装クラス |
| spark.hadoop.fs.oss.buffer.dirs | /mnt/disk1,/mnt/disk2,... | OSS のローカルの一時ディレクトリ デフォルトではクラスター内のデータディスクを使用します。 |
| spark.hadoop.fs.oss.buffer.dirs.exists | false | OSS 一時ディレクトリが存在するかどうか |
| spark.hadoop.fs.oss.client.connection.timeout | 50000 | OSS クライアント接続のタイムアウト時間 (単位：ミリ秒) |
| spark.hadoop.fs.oss.client.socket.timeout | 50000 | OSS クライアントソケットのタイムアウト時間 (単位：ミリ秒) |
| spark.hadoop.fs.oss.client.connection.ttl | -1 | クライアントの Time-to-Live の値 |
| spark.hadoop.fs.oss.connection.max | 1024 | 許可される最大接続数 |
| spark.hadoop.job.runlocal | false | データソースが OSS で、Spark コードをローカルで実行してデバッグする必要がある場合は、このパラメーターを true に設定します。 その必要がない場合は、このパラメーターを false に設定します。 |
| spark.logservice.fetch.interval.millis | 200 | データを LogHub から取得するレシーバーの間隔 |

| プロパティ | デフォルト | 説明 |
| --- | --- | --- |
| spark.logservice.fetch.inOrder | true | データが分割された後に Shard データを順番に消費するかどうか |
| spark.logservice.heartbeat.interval.millis | 30000 | データ消費プロセスのハートビート間隔 (単位：ミリ秒) |
| spark.mns.batchMsg.size | 16 | 一括してフェッチする MNS メッセージの数。最大 16 です。 |
| spark.mns.pollingWait.seconds | 30 | MNS キューが空の場合のポーリング待機時間 |
| spark.hadoop.io.compression.codec.snappy.native | false | Snappy ファイルが標準 Snappy 形式かどうか。デフォルトでは、Hadoop は Hadoop で編集される Snappy ファイルを認識します。 |

**Smart Shuffle 最適化構成**

Smart Shuffle は、EMR バージョン 3.16.0 で Spark SQL によって提供されるシャッフル実装です。 Spark SQL の実行効率を向上させるため、大量のシャッフルデータを持つクエリを処理します。 Smart Shuffle の起動後、シャッフル出力データはエフェメラルディスクに格納されません。 データは格納されず事前にネットワークを介してリモートノードに伝送されます。 同じパーティション内のデータは同じノードに伝送され、同じファイルに格納されます。 現在、Smart Shuffle には以下の制限があります。

- Smart Shuffle はタスク実行の原子性を保証できません。 タスクの実行が失敗した場合は、すべての Spark ジョブを再起動する必要があります。
- Smart Shuffle は現在、外部での Smart Shuffle の実装を提供しておらず、動的リソース割り当てもサポートしていません。
- 投機的タスクはサポートされていません。
- Smart Shuffle は Adaptive Execution と互換性がありません。

Smart Shuffle 関連の設定は以下のとおりです。 Spark 設定ファイルまたは spark-submit パラメーターを使用して Smart Shuffle を有効にできます。

| パラメーター | デフォルト | 説明 |
| --- | --- | --- |
| spark.shuffle.manager | sort | spark.shuffle.manager = org.apache.spark.shuffle.sort.SmartShuffleManager または spark.shuffle.manager=smart を設定することにより、Spark Session で Smart Shuffle 機能を有効にできるシャッフルメソッド |
| spark.shuffle.smart.spill.memorySizeForceSpillThreshold | 128m | Smart Shuffle を有効にする場合、各シャッフルタスクに使用されるメモリにはしきい値があります。 しきい値に達すると、シャッフルデータがパーティションに従ってネットワークを介して対応するリモートノードに送信されます。 |
| spark.shuffle.smart.transfer.blockSize | 1m | Smart Shuffle が有効になっているときにネットワーク転送に使用されるキャッシュのサイズ |

## 3.3 OSS ファイルの操作

このページでは、OSS SDK を使用する際に起こりうる問題と推奨される方法について説明します。

**OSS SDK の使用上の問題**

OSS SDK を使用して Spark ジョブまたは Hadoop ジョブの OSS ファイルを直接操作できない場合は、OSS SDK が依存している http-client-4.4.x バージョンと Spark または Hadoop の実行環境の http-client バージョンとの間で競合していることが原因です。 当該操作が必要な場合は、まず依存関係の競合を解決する必要があります。 実際、Spark と Hadoop はすでに E-MapReduce の OSS とシームレスに互換性があり、HDFS を使用していれば OSS ファイルを操作できます。

- 現在の E-MapReduce 環境は MetaService をサポートしており、AccessKey なしで E-MapReduce 環境の OSS データにアクセスできます。 AccessKey を使用する以前の方法は引

き続きサポートされます。 OSS を使用する場合は、内部ネットワークエンドポイントが優先されます。

- ローカルでテストするときは、ローカルマシンから OSS データにアクセスできるよう、OSS のパブリックネットワークエンドポイントを使用する必要があります。

完全なエンドポイントの一覧の詳細は、「OSS エンドポイント (OSS endpoints)」をご参照ください。

#### 推奨される方法 (AccessKey を使用しない方法)

OSS ディレクトリの下のファイルを照会するには、以下の方法を使用するよう推奨します。

```
[Scala]
  import org.apache.hadoop.conf.Configuration
  import org.apache.hadoop.fs.{ Path, FileSystem}
  val dir = "oss://bucket/dir"
  val path = new Path(dir)
  val conf = new Configuration()
  conf.set("fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssFileSystem")
  val fs = FileSystem.get(path.toUri, conf)
  val fileList = fs.listStatus(path)
  ...
  [Java]
  import org.apache.hadoop.conf.Configuration;
  import org.apache.hadoop.fs.Path;
  import org.apache.hadoop.fs.FileStatus;
  import org.apache.hadoop.fs.FileSystem;
  String dir = "oss://bucket/dir";
  Path path = new Path(dir);
  Configuration conf = new Configuration();
  conf.set("fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssFileSystem");
  FileSystem fs = FileSystem.get(path.toUri(), conf);
  FileStatus[] fileList = fs.listStatus(path);
  ...
```

## 3.4 Spark を使用して OSS へアクセス

E-MapReduce は、MetaServiceをサポートしており、AccessKey なしで E-MapReduce 環境の OSS データにアクセスできます。 AccessKey とエンドポイントを使用する以前の方法もサポートされます。 OSS エンドポイントには必ず内部 IP アドレスを使用します。 完全なエンドポイント一覧の詳細は、「OSS エンドポイント (OSS endpoints)」をご参照ください。

#### Spark への OSS アクセス許可

以下の例は、Spark が AccessKey なしで OSS からデータを読み取り、処理したデータを OSS に書き戻す方法を示しています。

```
val conf = new SparkConf().setAppName("Test OSS")
   val sc = new SparkContext(conf)
   val pathIn = "oss://bucket/path/to/read"
   val inputData = sc.textFile(pathIn)
```

```
    val cnt = inputData.count
    println(s"count: $cnt")
    val outputPath = "oss://bucket/path/to/write"
    val outpuData = inputData.map(e => s"$e has been processed.")
    outpuData.saveAsTextFile(outputPath)
```

### 付録

完全なサンプルコードについては、以下をご参照ください。

- Spark への OSS アクセス許可

## 3.5 スパークにおける MaxCompute の利用

このページでは、E-MapReduce SDK を使用して Spark で MaxCompute データを読み書きする方法について説明します。

### Spark への MaxCompute アクセス許可

1. OdpsOps オブジェクトを初期化します。 Spark では、MaxCompute のデータ操作は OdpsOps クラスを使用して実行されます。 OdpsOps オブジェクトを作成するには、以下のステップを実行します。

```
import com.aliyun.odps.TableSchema
    import com.aliyun.odps.data.Record
    import org.apache.spark.aliyun.odps.OdpsOps
    import org.apache.spark.{ SparkContext, SparkConf}
    object Sample {
      def main(args: Array[String]): Unit = {
        // == Step-1 ==
        val accessKeyId = "<accessKeyId>"
        val accessKeySecret = "<accessKeySecret>"
        // Take the internal network address as an example
        val urls = Seq("http://odps-ext.aliyun-inc.com/api", "http://dt-ext.odps.aliyun-
inc.com")
        val conf = new SparkConf().setAppName("Test Odps")
        val sc = new SparkContext(conf)
        val odpsOps = OdpsOps(sc, accessKeyId, accessKeySecret, urls(0), urls(1))
        // A part of the calling code is shown as follows
        // == Step-2 ==
        ...
        // == Step-3 ==
        ...
      }
      // == Step-2 ==
      // Method definition 1
      // == Step-3 ==
      // Method definition 2
```

```
    }
```

2. MaxCompute からテーブルデータを Spark に読み込みます。 以下のとおり、OdpsOps オブジェクトの readTable メソッドを使用することで MaxCompute テーブルを Spark に読み込み RDD を作成できます。

```
// == Step-2 ==
        val project = <odps-project>
        val table = <odps-table>
        val numPartitions = 2
        val inputData = odpsOps.readTable(project, table, read, numPartitions)
        inputData.top(10).foreach(println)
        // == Step-3 ==
        ...
```

上記のコードでは、MaxCompute テーブルのデータを解析して事前処理するため以下のとおり読み取り関数を定義する必要があります。

```
def read(record: Record, schema: TableSchema): String = {
          record.getString(0)
        }
```

この関数により、MaxCompute テーブルの先頭の列を Spark 実行時環境に読み込みます。

3. Spark の結果データを MaxCompute テーブルに保存します。 OdpsOps オブジェクトの saveToTable メソッドを使用して、Spark RDD を MaxCompute に保存できます。

```
val resultData = inputData.map(e => s"$e has been processed.")
        odpsOps.saveToTable(project, table, dataRDD, write)
```

上記のコードでは、MaxCompute テーブルに書き込む前に以下のとおりデータ事前処理用の書き込み関数を定義する必要があります。

```
def write(s: String, emptyReord: Record, schema: TableSchema): Unit = {
          val r = emptyReord
          r.set(0, s)
        }
```

この関数により、各行の RDD データを対応する MaxCompute テーブルの先頭の列に書き込みます。

4. パーティションテーブルパラメーターの表記

SDK は、MaxCompute パーティションテーブルの読み書きをサポートしています。 テーブルの標準命名規則は partition_column_name=partition_name (複数のパーティションをコンマで区切ります) です。 パーティション列 pt と ps があるとします。

- pt が 1 のパーティションのテーブルデータを読み取ります。
- pt が 1、ps が 2 のパーティションのテーブルデータを読み取ります。

**付録**

完全なサンプルコードについては、以下をご参照ください。Spark への MaxCompute アクセス許可

## 3.6 Spark Streaming を使用して MQ データを消費

このページでは、Spark Streaming を使用して MQ データを消費し、各バッチの単語数を計算する方法について説明します。

### Spark を使用して MQ にアクセス

以下の例に、Spark Streaming を使用して MQ データを消費し、各バッチの単語数をカウントする方法を示します。

```
val Array(cId, topic, subExpression, parallelism, interval) = args
    val accessKeyId = "<accessKeyId>"
    val accessKeySecret = "<accessKeySecret>"
    val numStreams = parallelism.toInt
    val batchInterval = Milliseconds(interval.toInt)
    val conf = new SparkConf().setAppName("Test ONS Streaming")
    val ssc = new StreamingContext(conf, batchInterval)
    def func: Message => Array[Byte] = msg => msg.getBody
    val onsStreams = (0 until numStreams).map { i =>
      println(s"starting stream $i")
      OnsUtils.createStream(
        ssc,
        cId,
        topic,
        subExpression,
        ccessKeyId,
        accessKeySecret,
        StorageLevel.MEMORY_AND_DISK_2,
        func)
    }
    val unionStreams = ssc.union(onsStreams)
    unionStreams.foreachRDD(rdd => {
      rdd.map(bytes => new String(bytes)).flatMap(line => line.split(" "))
        .map(word => (word, 1))
        .reduceByKey(_ + _).collect().foreach(e => println(s"word: ${e._1}, cnt: ${e._2}"))
    })
    ssc.start()
    ssc.awaitTermination()
```

**付録**

完全なサンプルコードについては、以下をご参照ください。Spark への ONS アクセス許可

# 3.7 Spark で Table Store データを消費

このページでは、Spark で Table Store データを消費する方法について説明します。

### Spark への Table Store アクセス許可

- テーブルを準備します。

  pet という名前のテーブルを作成します。 名前列をプライマリキーフィールドに設定します。

| 名前 | 飼い主 | 種類 | 性別 | 誕生日 | 命日 |
| --- | --- | --- | --- | --- | --- |
| ふわふわ | ハロルド | ネコ | 雌 | 1993-02-04 | |
| クローズ | グウェン | ネコ | 雄 | 1994-03-17 | |
| バフィ | ハロルド | 犬 | 雌 | 1989-05-13 | |
| ファング | ベニー | 犬 | 雄 | 1990-08-27 | |
| クッパ | ダイアン | 犬 | 雄 | 1979-08-31 | 1995-07-29 |
| チャーピー | グウェン | 鳥 | 雌 | 1998-09-11 | |
| ウィスラー | グウェン | 鳥 | | 1997-12-09 | |
| スリム | ベニー | ヘビ | 雄 | 1996-04-29 | |
| パフボール | ダイアン | ハムスター | 雌 | 1999-03-30 | |

- 以下の例に、Spark が Table Store データを消費する方法を示します。

```
private static RangeRowQueryCriteria fetchCriteria() {
    RangeRowQueryCriteria res = new RangeRowQueryCriteria("pet");
    res.setMaxVersions(1);
    List<PrimaryKeyColumn> lower = new ArrayList<PrimaryKeyColumn>();
    List<PrimaryKeyColumn> upper = new ArrayList<PrimaryKeyColumn>();
    lower.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MIN));
    upper.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MAX));
    res.setInclusiveStartPrimaryKey(new PrimaryKey(lower));
    res.setExclusiveEndPrimaryKey(new PrimaryKey(upper));
    return res;
}
public static void main(String[] args) {
    SparkConf sparkConf = new SparkConf().setAppName("RowCounter");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);
    Configuration hadoopConf = new Configuration();
    JavaSparkContext sc = null;
    try {
        sc = new JavaSparkContext(sparkConf);
        Configuration hadoopConf = new Configuration();
        TableStore.setCredential(
                hadoopConf,
                new Credential(accessKeyId, accessKeySecret, securityToken));
        Endpoint ep = new Endpoint(endpoint, instance);
        TableStore.setEndpoint(hadoopConf, ep);
        TableStoreInputFormat.addCriteria(hadoopConf, fetchCriteria());
        JavaPairRDD<PrimaryKeyWritable, RowWritable> rdd = sc.newAPIHadoopRDD(
```

```
                hadoopConf, TableStoreInputFormat.class,
                PrimaryKeyWritable.class, RowWritable.class);
        System.out.println(
            new Formatter().format("TOTAL: %d", rdd.count()).toString());
    } finally {
        if (sc ! = null) {
            sc.close();
        }
    }
}
```

**付録**

完全なサンプルコードについては、以下をご参照ください。Spark への Table Store アクセス許可

## 3.8 Spark Streaming を使用して MNS データを消費

このページでは、Spark Streaming を使用して MNS 内のデータを消費し、各バッチの単語数を計算する方法について説明します。

**Spark への MNS アクセス許可**

サンプルコードは以下のとおりです。

```
val conf = new SparkConf().setAppName("Test MNS Streaming")
    val batchInterval = Seconds(10)
    val ssc = new StreamingContext(conf, batchInterval)
    val queuename = "queuename"
    val accessKeyId = "<accessKeyId>"
    val accessKeySecret = "<accessKeySecret>"
    val endpoint = "http://xxx.yyy.zzzz/abc"
    val mnsStream = MnsUtils.createPullingStreamAsRawBytes(ssc, queuename,
accessKeyId, accessKeySecret, endpoint,
      StorageLevel.MEMORY_ONLY)
    mnsStream.foreachRDD( rdd => {
      rdd.map(bytes => new String(bytes)).flatMap(line => line.split(" "))
        .map(word => (word, 1))
        .reduceByKey(_ + _).collect().foreach(e => println(s"word: ${e._1}, cnt: ${e._2}"))
    })
    ssc.start()
    ssc.awaitTermination()
```

**Spark Streaming を MetaService と併用**

上記の例では、AccessKey を明示的に API に渡しています。 E-MapReduce SDK 1.3.2 以降、Spark Streaming は AccessKey なしで MetaService を使用して MNS データを処理できます。 詳細は、E-MapReduce SDK の MnsUtils クラスの説明をご参照ください。

```
MnsUtils.createPullingStreamAsBytes(ssc, queueName, endpoint, storageLevel)
```

```
MnsUtils.createPullingStreamAsRawBytes(ssc, queueName, endpoint, storageLevel)
```

### 付録

完全なサンプルコードについては、以下をご参照ください。Spark への MNS アクセス許可

## 3.9 Spark を使用して HBase にデータを書き込む

このページでは、Spark が HBase にデータを書き込む方法について説明します。 注記：

:

コンピューティングクラスターは、HBase クラスターと同じセキュリティグループに属している必要があります。 別のグループの場合、ネットワークに接続できません。 E-MapReduce でクラスターを作成するときは、必ず HBase クラスターが配置されているセキュリティグループを選択します。

### Spark への HBase アクセス許可

以下のコードを使用します。

```
object ConnectionUtil extends Serializable {
   private val conf = HBaseConfiguration.create()
   conf.set(HConstants.ZOOKEEPER_QUORUM,"ecs1,ecs1,ecs3")
   conf.set(HConstants.ZOOKEEPER_ZNODE_PARENT, "/hbase")
   private val connection = ConnectionFactory.createConnection(conf)
   def getDefaultConn: Connection = connection
  }
  //Create data streaming unionStreams
  unionStreams.foreachRDD(rdd => {
   rdd.map(bytes => new String(bytes))
    .flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
    .mapPartitions {words => {
     val conn = ConnectionUtil.getDefaultConn
     val tableName = TableName.valueOf(tname)
     val t = conn.getTable(tableName)
     try {
      words.sliding(100, 100).foreach(slice => {
       val puts = slice.map(word => {
        println(s"word: $word")
        val put = new Put(Bytes.toBytes(word. _1 + System.currentTimeMillis()))
        put.addColumn(COLUMN_FAMILY_BYTES, COLUMN_QUALIFIER_BYTES,
         System.currentTimeMillis(), Bytes.toBytes(word._2))
        put
       }).toList
       t.put(puts)
      })
     } finally {
      t.close()
     }
     Iterator.empty
    }}.count()
   })
  ssc.start()
```

```
ssc.awaitTermination()
```

#### 付録

完全なサンプルコードについては、以下をご参照ください。

- Spark への HBase アクセス許可

## 3.10 Spark Streaming を使用して Kafka データを処理

このページでは、E-MapReduce の Hadoop クラスターで Spark Streaming ジョブを実行し、Kafka クラスターのデータを処理する方法について説明します。

#### プログラミングリファレンス

E-MapReduce の Hadoop クラスターと Kafka クラスターは完全にオープンソースのソフトウェアをベースとしています。 このため、開発時に対応する公式ドキュメントを参照できます。

- Spark の公式ドキュメント：streaming-kafka-integration
- Spark の公式ドキュメント：structured-streaming-kafka-integration
- E-MapReduce-demo :Github

#### Spark Streaming ジョブへの Kerberos Kafka クラスターアクセス許可

E-MapReduce を使用すると、Kerberos 認証をベースとした Kafka クラスターを作成できます。Hadoop クラスター内のジョブは、2 通りの方法で Kerberos Kafka クラスターにアクセスできます。

- 非 Kerberos Hadoop クラスター：Kafka クラスターの Kerberos 認証用に kafka_client_jaas.conf ファイルを提供します。
- Kerberos Hadoop クラスター：Kerberos クラスターをベースとしたクロスドメイン通信。Hadoop クラスターの Kerberos 認証用に kafka_client_jaas.conf ファイルを提供します。

どちらの方法でも、ジョブ実行時に Kerberos 認証用の kafka_client_jaas.conf ファイルを提供することが求められす。

kafka_client_jaas.conf のファイル形式は以下のとおりです。

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    serviceName="kafka"
    keyTab="/path/to/kafka.keytab"
    principal="kafka/emr-header-1.cluster-12345@EMR. 12345. COM";
```

```
};
```

keytab ファイルの取得方法の詳細は、『『ユーザーガイド』』の 「**Kerberos 認証 (Kerberos authentication)**」をご参照ください。

#### Spark Streaming ジョブへの Kerberos Kafka クラスターアクセス許可

Spark Streaming ジョブを実行して Kerberos Kafka クラスターにアクセスするときは、spark-submit コマンドラインパラメーターで必要に応じ kafka_client_jaas.conf ファイルと kafka.keytab ファイルを提供できます。

```
spark-submit --conf spark.driver.extraJavaOptions=-Djava.security.auth.login.config={{
PWD}}/kafka_client_jaas.conf --conf spark.executor.extraJavaOptions=-Djava.security.
auth.login.config={{PWD}}/kafka_client_jaas.conf --files /local/path/to/kafka_client_jaas
.conf,/local/path/to/kafka.keytab --class xx.xx.xx.KafkaSample --num-executors 2 --
executor-cores 2 --executor-memory 1g --master yarn-cluster xxx.jar arg1 arg2 arg3
```

kafka_client_jaas.conf ファイル内で、keytabファイルのパスは相対パスにする必要があります。 パスは必ず以下の形式で設定します。

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    serviceName = "kafka"
    keyTab = "kafka.keytab"
    principal="kafka/emr-header-1.cluster-12345@EMR. 12345。 COM ";
};
```

## 3.11 Spark を使用して MySQL にデータを書き込む

このページでは、E-MapReduce の Hadoop クラスターで Spark ジョブを実行して単語数を計算し、その結果を MySQL に書き込む方法について説明します。 Spark Streaming ジョブにより、Spark ジョブと類似する方法で MySQL にデータを書き込みます。

#### Spark への MySQL アクセス許可

以下のコードを使用します。

```
val input = getSparkContext.textFile(inputPath, numPartitions)
    input.flatMap(_.split(" ")).map(x => (x, 1)).reduceByKey(_ + _)
      .mapPartitions(e => {
        var conn: Connection = null
        var ps: PreparedStatement = null
        val sql = s"insert into $tbName(word, count) values (?, ?)"
        try {
          conn = DriverManager.getConnection(s"jdbc:mysql://$dbUrl:$dbPort/$dbName",
dbUser, dbPwd)
          ps = conn.prepareStatement(sql)
          e.foreach(pair => {
            ps.setString(1, pair._1)
            ps.setLong(2, pair._2)
```

```
          ps.executeUpdate()
        })

        ps.close()
        conn.close()
      } catch {
        case e: Exception => e.printStackTrace()
      } finally {
        if (ps != null) {
          ps.close()
        }
        if (conn != null) {
          conn.close()
        }
      }
    Iterator.empty
  }).count()
```

**付録**

完全なサンプルコードについては、以下をご参照ください。 Spark への RDS データ書き込み許可

## 3.12 Spark-Submit パラメーターの設定

このページでは、E-MapReduce で spark-submit パラメーターを設定する方法について説明します。

**クラスター設定**

- ソフトウェア設定

  E-MapReduce V1.1.0

  - Hadoop V2.6.0
  - Spark V1.6.0
- ハードウェア設定
  - マスターノード
    - 8 コア、16 GB メモリ、500 GB 記憶域スペース (ウルトラディスク)
    - 1 台
  - ワーカーノード
    - 8 コア、16 GB メモリ、500 GB 記憶域スペース (ウルトラディスク)
    - 10 台
  - 合計 : 8 コア 16 GB (ワーカー) x 10 + 8 コア 16 GB (マスター)

:

ジョブがサブミットされると、CPU とメモリのリソースのみが計算されます。 したがって、ディスクサイズは合計リソースに含まれません。

- YARN の利用可能なリソースの合計数 : 12 コア、12.8 GB (ワーカー) x 10

:

デフォルトでは、YARN で利用可能なコア = コア数 x 1.5、Yarn で利用可能なメモリ = マシンメモリ x 0.8 です。

**ジョブのサブミット**

クラスターが作成されたら、ジョブをサブミットできます。 まず、E-MapReduce でジョブを作成する必要があります。 次の図にパラメーターを示します。

SPARK FJ-210DFFD36EC33416 Content: Run Stop Save Job Settings Help

```
1  --class org.apache.spark.examples.SparkPi --master yarn --deploy-mode client --driver-memory 4g
   --num-executors 2 --executor-memory 2g --executor-cores 2
   /opt/apps/spark-1.6.0-bin-hadoop2.6/lib/spark-examples*.jar 10
```

Command (Reference Only)

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn --deploy-mode client --driver-memo
ry 4g --num-executors 2 --executor-memory 2g --executor-cores 2 /opt/apps/spark-1.6.0-bin-hadoop2.6/li
b/spark-examples*.jar 10
```

上の図のジョブは公式の Spark サンプルパッケージを直接使用しているので、 お客様は独自の JAR パッケージをアップロードする必要はありません。

パラメーターの記述は以下のとおりです。

```
--class org.apache.spark.examples.SparkPi --master yarn --deploy-mode client --driver
-memory 4g --num-executors 2 --executor-memory 2g --executor-cores 2 /opt/apps/
spark-1.6.0-bin-hadoop2.6/lib/spark-examples*.jar 10
```

パラメーターの説明は以下のとおりです。

| パラメーター | 例 | 説明 |
| --- | --- | --- |
| class | org.apache.spark.examples.SparkPi | ジョブのメインクラス |
| master | yarn | E-MapReduce は YARN モードを使用します。 値を yarn に設定します。 |

| パラメーター | 例 | 説明 |
| --- | --- | --- |
| | yarn-client | master パラメーターを yarn に、deploy-mode パラメーターを client に設定するのと同等です。 このパラメーターを設定した場合は、deploy-mode パラメーターを設定する必要はありません。 |
| | yarn-cluster | master パラメーターを yarn 、deploy-mode パラメーターを client に設定するのと同等です。 このパラメーターを設定した場合は、deploy-mode パラメーターを設定する必要はありません。 |
| deploy-mode | client | クライアントモードは、ジョブの AM がマスターノードで実行されることを示します。 このパラメーターを設定する場合は、マスターパラメーターも yarn に設定する必要があります。 |
| | cluster | クラスターモードは、AM がワーカーノードの 1 つでランダムに実行されることを示します。 このパラメーターを設定する場合、マスターパラメーターも yarn に設定する必要があります。 |
| driver-memory | 4g | ドライバーに割り当てるメモリ。 割り当てられたメモリは、各ノードのメモリの合計サイズを超えることはできません。 |
| num-executors | 2 | 作成する実行プログラムの数 |
| executor-memory | 2g | 各実行プログラムに割り当てるメモリの最大量。 割り当てられたメモリは、各ノードの使用可能なメモリの最大数を超えることはできません。 |

| パラメーター | 例 | 説明 |
| --- | --- | --- |
| executor-cores | 2 | 各実行プログラムが使用するスレッドの数で、各実行プログラムが同時に実行できるタスクの最大数と同等です。 |

**リソース計算**

以下の表に、さまざまなモードおよび設定で実行されているジョブが使用するリソースを示します。

- yarn-client モードのリソース計算

| ノード | リソースタイプ | リソース量 (結果は上記の例をベースに計算されます) |
| --- | --- | --- |
| Master | Core | 1 core |
| | Memory | driver-memory = 4 GB |
| Worker | Core | num-executors × executor-cores = 4 cores |
| | Memory | num-executors × executor-memory = 4 GB |

  - ジョブの主プログラム (ドライバプログラム) は、マスターノード上で実行されます。 --driver-memory パラメーターで指定の通り、ジョブ設定に準じて 4 GB のメモリはメインプログラムに割り振られます。 メインプログラムは、割り当てられたメモリのすべてを使用しない場合もあります。
  - --num-executs パラメーターで指定の通り、2 つの実行プログラムがワークノードで開始されます。 各実行プログラムは 2 GB のメモリ (--executor-memory パラメーターで指定) で割り当てられ、最大 2 つの同時タスク (--executor-cores パラメーターで指定) をサポートします。
- yarn-cluster モードのリソース計算

| ノード | リソースタイプ | リソース量 (結果は上記の例をベースに計算されます) |
| --- | --- | --- |
| Master | N/A | ジョブ情報の同期化を担当し、リソースの消費が少ない、小さなクライアントプログラム |

| ノード | リソースタイプ | リソース量 (結果は上記の例をベースに計算されます) |
| --- | --- | --- |
| Worker | Core | num-executors × executor-cores + spark.driver.cores = 5 cores |
| | Memory | num-executors × executor-memory + driver-memory = 8 GB |

注：

デフォルトでは、spark.driver.cores の値は 1 です。 1 より大きい値に設定できます。

**リソース使用量の最適化**

- yarn-client モード

  yarn-client モードで大規模なジョブがあり、より多くのクラスターのリソースを使用する場合は、以下の設定をご参照ください。

```
--master yarn-client --driver-memory 5g --num-executors 20 --executor-memory 4g
 --executor-cores 4
```

：

- Spark は、設定したメモリ値に加えて、375 MB または 7% (いずれか高い方) のメモリを割り当てます。

- メモリをコンテナに割り当てるとき、YARN は最も近い整数のギガバイトに切り上げます。 ここでのメモリ値は 1 GB の倍数である必要があります。

上記のリソース式をベースとする場合：

- マスターのリソース量は以下のとおりです。
  - Cores: 1
  - Memory : 6 GB (5 GB + 375 MB、6 GB に切り上げ)
- ワーカーのリソース量は以下のとおりです。
  - Core: 20 × 4 = 80 cores
  - Memory : 20 x 5 GB (4 GB + 375 MB、5 GB に切り上げ) = 100 GB

リソース計算結果によると、ジョブに割り振られたリソース量がクラスターのリソースの合計量を超えることはありません。 この規則に従うと、他のリソース割り当て設定を使用できます。

```
--master yarn-client --driver-memory 5g --num-executors 40 --executor-memory 1g
 --executor-cores 2
```

```
--master yarn-client --driver-memory 5g --num-executors 15 --executor-memory 4g
 --executor-cores 4
```

```
--master yarn-client --driver-memory 5g --num-executors 10 --executor-memory 9g
 --executor-cores 6
```

理論的には、上記の式を使用して計算されたリソースの総量が、クラスターのリソースの総量を超えないようにする必要があります。 ただし、生産シナリオでは、オペレーティング システム、HDFS ファイル システム、E-MapReduce サービスもコアリソースとメモリリソースを使用する場合があります。 使用可能なコアリソースとメモリー リソースがない場合、ジョブのパフォーマンスが低下するか、ジョブが失敗します。

通常、executor-cores パラメーターはクラスターコアの数と同じ値に設定されます。 この値が高すぎると、期待通りにパフォーマンスが向上せず CPU が頻繁に切り替わります。

- yarn-cluster モード

  yarn-cluster モードでは、ドライバプログラムはワーカーノードで実行されます。 ワーカーノードのリソースプール内のリソースが使用されます。 このクラスターのリソースをより多く使用する場合は、以下の設定をご使用ください。

  ```
  --master yarn-cluster --driver-memory 5g --num-executors 15 --executor-memory 4g
   --executor-cores 4
  ```

**推奨設定**

- メモリを非常に大きな値に設定する場合は、ガベージコレクションによるオーバーヘッドに注意を払う必要があります。 通常は、実行プログラムに 64 GB 以下のメモリを割り当てることを推奨します。
- HDFS 読み取り/書き込みジョブを実行中の場合は、データの読み取りや書き込み用に、各実行プログラムの同時実行ジョブの数を 5 以下に設定することを推奨します。
- OSS 読み取り/書き込みジョブを実行中の場合は、すべての ECS インスタンスの帯域幅を使用できるようにするため、実行プログラムを異なる ECS インスタンスに分散するよう推奨します。 たとえば、10 個の ECS インスタンスがある場合は、num-executor を 10 に設定し、適切なメモリと同時実行ジョブの数を設定できます。
- ジョブで使用するコードがスレッドセーフではない場合は、executor-cores 設定時に同時実行でジョブエラーが発生するかどうかモニタリングする必要があります。 エラーが発生する場合は、executor-cores を 1 に設定するよう推奨します。

# 4 Spark Streaming SQL

## 4.1 はじめに

EMR-3.21.0 から、プレビュー版の Spark Streaming SQL が提供され、SQL を使用したストリーミング分析ジョブの開発がサポートされます。 Spark Streaming SQL は、Spark Structured Streaming に基づいて開発されています。 すべての構文関数と制限は、Spark Structured Streaming の機能の説明に従います。

## 4.2 キーワード

このトピックでは、Spark Streaming SQL で使用する一般的なキーワードと、これらのキーワードの使用方法について説明します。

**一般的なキーワードタイプ**

| キーワードタイプ | キーワード |
| --- | --- |
| DDL | CREATE TABLE、CREATE TABLE AS SELECT |
| DML | INSERT INTO |
| SELECT 句 | SELECT FROM、WHERE、GROUP BY、JOIN |

**使用法**

キーワードをフィールド名として使用する場合、キーワードをバッククォート (`) で囲みます。例：`value`

## 4.3 ストリーミングクエリ

## 4.3.1 ストリーミングクエリ設定

このトピックでは、ストリーミングクエリ設定の基本概念と関連パラメーターについて説明します。

**クエリ設定**

ストリーミングクエリに Spark SQL を使用する前に、次の 2 つの概念を理解する必要があります。

- データソース設定：テーブルの定義。
- クエリインスタンス設定：各ストリーミングクエリを実行するためのパラメーター設定。

テーブルの定義には、Kafka データソースの接続アドレスやトピック名など、データソース設定のみが含まれます。 テーブル内の複数の非ビジネスクエリを同時に実行できます。 したがって、テーブルの定義には、特定のクエリインスタンスを実行するための設定を含めることはできません。

各クエリインスタンスは個別に設定する必要があります。 クエリ SQL 文への不必要な変更を減らすため、クエリインスタンスごとに **query name** を設定できます。 **query name** を使用することで、各クエリインスタンスを実行するためのパラメーターを設定できます。 クエリインスタンスのパラメーターは、 SET 構文を使用して設定します。 詳細は、「設定パラメーター」をご参照ください。

クエリ設定に関する規則：各クエリインスタンスの **query name** は、最も近い SQL SET 文で使用できます。 例を 2 つ示します。

- 例 1

```
SET streaming.query.name=one_test_job

-- query 1
INSERT INO tb_test_1 SELECT ...

-- query 2
INSERT INO tb_test_2 SELECT ...

-- The names of queries 1 and 2 are both one_test_job. However, this case is invalid
because the name of each query instance must be unique.
```

- 例 2

```
SET streaming.query.name=one_test_job_1

SET streaming.query.name=one_test_job_2

-- query 1
CREATE TABLE tb_test_1 AS SELECT ...

-- The name of query 1 is one_test_job_2.
```

クエリインスタンスの文には、次のものがあります。

- INSERT INTO ...
- CREATE TABLE ... AS SELECT ...

**設定パラメーター**

| パラメーター | 対応する **DataFrame API** | **SQL 文の形式** | 説明 | 必須 |
| --- | --- | --- | --- | --- |
| queryName | writeStream.queryName(...) | SET streaming.query.name=$queryName | 各ストリーミングクエリの名前。 異なるクエリインスタンスのパラメーターは、クエリ名で区別されます。 | はい |
| option | writeStream.option(...) | SET spark.sql.streaming.query.options.$queryName.$optionName=$optionValue | checkpointLocation：checkpoint のディレクトリ。 | はい |
| | | | カスタムオプション。 | いいえ |
| outputMode | writeStream.outputMode(...) | SET spark.sql.streaming.query.outputMode.$queryName=$outputMode | クエリ結果の出力モード。 デフォルト値：append。 | いいえ |
| trigger | writeStream.trigger(...) | SET spark.sql.streaming.query.trigger.$queryName=$triggerType | クエリを実行するタイミングを制御するトリガー。 デフォルト値：ProcessingTime。<br> **注：**<br>このパラメーターには、ProcessingTime のみを設定できます。 | いいえ |
| | | SET spark.sql.streaming.query.trigger.intervalMs.$queryName=$intervalMs | クエリバッチ間の間隔。 単位 ：ミリ秒。 デフォルト値：0。 | いいえ |

## 4.3.2 ジョブテンプレート

このトピックでは、Spark SQL を使用してストリーミングジョブを開発する方法について説明します。

**クエリ文ブロック**

SQL クエリ文でジョブ関連パラメーターを適切に表すことは困難です。 したがって、必須パラメーターを設定するには、SQL クエリ文の前に SET 文を追加する必要があります。 重要なパ

ラメーターは、**streaming.query.name** です。 各 SQL クエリは、一意の **streaming.query.name** に関連付ける必要があります。 このクエリ文に基づき、各 SQL クエリに他のパラメーター (checkpoint など) を設定できます。 規則に基づき、SET streaming.query.name 文を各 SQL クエリの先頭に追加する必要があります。 そうしないと、クエリ時にエラーが返される場合があります。 有効なクエリ文ブロックは次のとおりです。

```
SET streaming.query.name=${queryName};

queryStatement
```

### ジョブテンプレート

```
-- dbName: the name of the database where a table is to be created.
CREATE DATABASE IF NOT EXISTS ${dbName};
USE ${dbName};

-- Create a Log Service table.
-- slsTableName: the name of the Log Service table.
-- logProjectName: the name of the Log Service project.
-- logStoreName: the name of the Logstore in Log Service.
-- accessKeyId: the AccessKey ID provided by Alibaba Cloud.
-- accessKeySecret: the AccessKey secret provided by Alibaba Cloud.
-- endpoint: the endpoint of the Logstore in Log Service. For more information,
 see #unique_27.
CREATE TABLE IF NOT EXISTS ${slsTableName}
USING loghub
OPTIONS (
sls.project = '${logProjectName}',
sls.store = '${logStoreName}',
access.key.id = '${accessKeyId}',
access.key.secret = '${accessKeySecret}',
endpoint = '${endpoint}');

-- Create an HDFS table and define the column fields in the table.
-- hdfsTableName: the name of the HDFS table.
-- location: the storage path of data. Both HDFS and OSS paths are supported.
-- Supported data formats: CSX, JSON, ORC, and Parquet. The default format is Parquet.
CREATE TABLE IF NOT EXISTS ${hdfsTableName} (col1 dataType[, col2 dataType])
USING PARQUET
LOCATION '${location}';

-- Define some parameters for running each streaming query. Such parameters include:
-- streaming.query.name: the name of the streaming query job.
-- spark.sql.streaming.checkpointLocation.${queryName}: the directory of the checkpoint
 for the streaming query job.
SET streaming.query.name=${queryName};
SET spark.sql.streaming.query.options.${queryName}.checkpointLocation=
${checkpointLocation};
-- The following parameters are optional and can be defined as required:
-- outputMode: the output mode of the query result. Default value: append.
-- trigger: the trigger controlling the moment where the query is executed. Default value:
 ProcessingTime. Currently, this parameter can only be set to ProcessingTime.
-- trigger.intervalMs: the interval between query batches. Unit: milliseconds. Default
 value: 0.
-- SET spark.sql.streaming.query.outputMode.${queryName}=${outputMode};
SET spark.sql.streaming.query.trigger.${queryName}=ProcessingTime;
SET spark.sql.streaming.query.trigger.intervalMs.${queryName}=30;

INSERT INTO ${hdfsTableName}
```

```
SELECT col1, col2
FROM ${slsTableName}
WHERE ${condition}
```

**パラメーター**

次の表に、主要なパラメーターを示します。

| パラメーター | 説明 | デフォルト値 |
| --- | --- | --- |
| streaming.query.name | クエリの名前。 | このパラメーターは明示的に設定する必要があります。 |
| spark.sql.streaming.query.options.${queryName}.checkpointLocation | ストリーミングクエリジョブの checkpoint のディレクトリ。 | このパラメーターは明示的に設定する必要があります。 |
| spark.sql.streaming.query.outputMode.${queryName} | クエリ結果の出力モード。 | append |
| spark.sql.streaming.query.trigger.${queryName} | クエリを実行するタイミングを制御するトリガー。 このパラメーターには、ProcessingTime のみを設定できます。 | ProcessingTime |
| spark.sql.streaming.query.trigger.intervalMs.${queryName} | クエリバッチ間の間隔。 単位：ミリ秒。 | 0 |

# 4.4 DML の概要

## 4.4.1 DML 概要

このトピックでは、Spark ストリーミング SQL で INSERT INTO 文を使用する方法について説明します。

**構文**

```
INSERT INTO tbName[(columnName[,columnName]*)]
queryStatement;
```

**例**

```
INSERT INTO LargeOrders
SELECT * FROM Orders WHERE units > 1000;
```

**注意**

- Spark ストリーミング SQL では、クエリに単独の SELECT 文を使用できません。 SELECT 文は、CTAS 文と一緒に使用するか、INSERT INTO 文に含める必要があります。

- 1 つのジョブの場合、SQL ファイルには複数のデータ操作言語 (DML) 操作、および複数のデータソースとターゲットを含めることができます。

# 4.5 クエリ概要

## 4.5.1 SELECT 文

SELECT 文は、テーブルからデータを抽出します。

**構文**

```
SELECT [ DISTINCT ]
{ * | projectItem [, projectItem ]* }
FROM tableExpression;
```

**サブクエリ**

通常、SELECT 文は複数のテーブルからデータを読み取ります。例：SELECT column_1, column_2 ... FROM table_name など。 SELECT 文は、別の SELECT 文からデータを読み取ることもできます。これはサブクエリと呼ばれます。

:

サブクエリはエイリアスを使用する必要があります (次の例を参照)。

```
INSERT INTO result_table
SELECT * FROM(
  SELECT t.a,
    sum(t.b) AS sum_b
  FROM t
  GROUP BY t.a) t1
WHERE  t1.sum_b > 100;
```

## 4.5.2 WHERE 文

WHERE 文は、SELECT 文で生成されたデータを絞り込みます。

**構文**

```
SELECT [ ALL | DISTINCT ]
{ * | projectItem [, projectItem ]* }
FROM tableExpression
```

```
[ WHERE booleanExpression ];
```

## 4.5.3 GROUP BY 文

GROUP BY 文は、1 つ以上の列を基準にして結果セットをグループ化します。

**構文**

```
SELECT [ DISTINCT ]
{ * | projectItem [, projectItem ]* }
FROM tableExpression
[ GROUP BY { groupItem [, groupItem ]* } ];
```

**例**

```
SELECT Customer, SUM(OrderPrice) FROM xxx
GROUP BY Customer;
```

## 4.5.4 JOIN 文

Spark SQL は、バッチ処理データとストリーミングデータの結合、バッチ処理データの結合、ストリーミングデータの結合をサポートしています。 JOIN 文は、従来のバッチ処理用の JOIN 文と同じセマンティクスを備えています。

**構文**

```
tableReference [, tableReference ]* | tableexpression
[ joinType ] JOIN tableexpression [ joinCondition ];
```

**制限**

ストリーミングデータを結合する場合、一部の結合タイプはサポートされないことに注意してください。 詳細は、Spark 公式ドキュメントをご参照ください。 次の表に、一部の結合タイプを示します。

| 左テーブル | 右テーブル | 結合タイプ | サポートの有無 |
| --- | --- | --- | --- |
| ストリーム | 静的 | 内部 | サポート。非ステートフル。 |
| | | 左外部 | サポート。非ステートフル。 |
| | | 右外部 | 未サポート。 |
| | | 完全外部 | 未サポート。 |
| 静的 | ストリーム | 内部 | サポート。非ステートフル。 |
| | | 左外部 | 未サポート。 |

| 左テーブル | 右テーブル | 結合タイプ | サポートの有無 |
| --- | --- | --- | --- |
| | | 右外部 | サポート。非ステートフル。 |
| | | 完全外部 | 未サポート。 |
| ストリーム | ストリーム | 内部 | サポート。 オプションで、状態のクリーンアップのため、両側にウォーターマーク、および時間の制約を指定できます。 |
| | | 左外部 | 条件付きサポート。 正しい結果を得るため、右側にウォーターマーク、および時間の制約を指定する必要があります。 オプションで、すべての状態のクリーンアップのため、左側にウォーターマークを指定できます。 |
| | | 右外部 | 条件付きサポート。 正しい結果を得るため、左側にウォーターマーク、および時間の制約を指定する必要があります。 オプションで、すべての状態のクリーンアップのため、右側にウォーターマークを指定できます。 |
| | | 完全外部 | 未サポート。 |

## 4.5.5 UNION ALL 文

UNION ALL 文は、2 つのデータストリームを結合します。 2 つのデータストリームのフィールドは、フィールドタイプとシーケンスが同じでなければなりません。

**構文**

```
select_statement
UNION ALL
```

```
select_statement;
```

**例**

```
SELECT
    a,
    sum(b),
FROM
    (SELECT * from tb_1
    UNION ALL
    SELECT * from tb_2
    )t
 GROUP BY a;
```

# 4.6 ウィンドウ関数

## 4.6.1 概要

このトピックでは、Spark Streaming SQL でサポートされるウィンドウ関数、ウィンドウタイプ、時間属性について説明します。

**ウィンドウ関数**

ウィンドウ関数は、特定のウィンドウの集計をサポートしています。 たとえば、ある Web ページに過去 1 分間にアクセスしたユーザー数を計算するには、過去 1 分間のデータを収集するウィンドウを定義し、このウィンドウでデータを計算します。 Spark Streaming SQL は、次の 2 つのウィンドウタイプをサポートしています。

- タンブリングウィンドウ
- スライディングウィンドウ

**時間属性**

Spark SQL は、イベント時間属性を使用したウィンドウ内のデータ集計をサポートします。

通常、イベント時間属性は、データレコードが作成された時間を示します。 この属性はスキーマで提供されます。

**注意**

時間ウィンドウを使用するクエリの場合、window.start と window.end で指定されたウィンドウの開始時間と終了時間を含むウィンドウ列が自動生成されます。

## 4.6.2 タンブリングウィンドウ

このトピックでは、Spark Streaming SQL でタンブリングウィンドウ関数を使用する方法について説明します。

#### タンブリングウィンドウとは

タンブリングウィンドウを使用して、指定したサイズのウィンドウに各要素を割り当てることができます。 通常、タンブリングウィンドウのサイズは固定で、互いにオーバーラップすることはありません。 たとえば、5 分間のタンブリングウィンドウが定義されている場合、期間に基づいて要素は [0:00, 0:05)、[0:05, 0:10)、[0:10, 0:15) のウィンドウに割り当てられます。

#### 構文

```
GROUP BY TUMBLING (colName, windowDuration)
```

#### 例

```
SELECT avg(inv_quantity_on_hand) qoh
FROM kafka_inventory
GROUP BY TUMBLING (inv_data_time, interval 1 minute)
```

## 4.6.3 スライディングウィンドウ

このトピックでは、Spark Streaming SQL でスライディングウィンドウ関数を使用する方法について説明します。

#### スライディングウィンドウとは

スライディングウィンドウは、ホップウィンドウとも呼ばれます。 タンブリングウィンドウとは異なり、スライドウィンドウは互いにオーバーラップすることが可能です。 スライディングウィンドウには、windowDuration と slideDuration の 2 つのパラメーターがあります。 slideDuration パラメーターは、各スライドのステップサイズを示します。 windowDuration パラメーターは、ウィンドウサイズを示します。

- slideDuration パラメーターの値が windowDuration パラメーターの値よりも小さい場合、ウィンドウは互いにオーバーラップし、各要素は複数のウィンドウに割り当てられます。
- slideDuration パラメーターの値が windowDuration パラメーターの値と等しい場合、ウィンドウはタンブリングウィンドウと同じです。

#### 構文

```
GROUP BY HOPPING (colName, windowDuration, slideDuration)
```

#### 例

```
SELECT avg(inv_quantity_on_hand) qoh
```

```
FROM kafka_inventory
GROUP BY HOPPING (inv_data_time, interval 1 minute, interval 30 second)
```

# 4.7 データソース

## 4.7.1 概要

このトピックでは、Spark SQL でサポートされるデータソースの種類と、データソースのデータの処理方法について説明します。

### サポートされているデータソース

| データソース | テーブル作成時に定義されたスキーマ | テーブル作成時に定義されていないスキーマ | 読み取り | 書き込み |
| --- | --- | --- | --- | --- |
| **Kafka** | # | # | # | # |
| **LogHub** | # | なし | # | # |
| **Table Store** | なし | # | なし | # |
| **HBase** | なし | # | なし | # |
| **JDBC** | なし | # | なし | # |
| **Druid** | なし | # | なし | # |
| **Redis** | なし | # | なし | # |

### データソースのデータの処理方法

Spark SQL はコマンドラインまたはワークフローを使用して、データソースのデータを処理できます。

- コマンドライン
  - プリコンパイルされたデータソース JAR パッケージをダウンロードします。

    JAR パッケージには LogHub、Table Store、HBase、JDBC、Redis データソースの実装と、関連する依存パッケージが含まれます。 JAR パッケージを 1 回ダウンロードすると、これらすべてのデータソースを使用できます。 Kafka と Druid データソースのパッケージ

は、この JAR パッケージに含まれておらず、将来的に追加されます。 詳細は、「リリースノート」をご参照ください。

- インタラクティブな開発には、**streaming-sql** コマンドラインを使用します。

```
[hadoop@emr-header-1 ~]# streaming-sql --master yarn-client --jars emr-datasources_shaded_2.11-${version}.jar --driver-class-path emr-datasources_shaded_2.11-${version}.jar
```

- -f または -e パラメーターを使用して、SQL 文を送信することもできます。
- Spark SQL を終了せずにストリーミングジョブを長時間実行する必要がある場合、**nohup** コマンドを使用して HUP (ハングアップ) シグナルを無視します。

• ワークフロー

詳細は、「#unique_42」をご参照ください。

## 4.7.2 Kafka データテーブルの説明

このトピックでは、Kafka データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName[(columnName dataType [,columnName dataType]*)]
USING kafka
OPTIONS(propertyName=propertyValue[,propertyName=propertyValue]*);
```

### 設定パラメーター

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| kafka.schema.registry.url | Kafka Schema Registry の URL。 | はい |
| subsribe | 関連付けられた Kafka トピックの名前。 | はい |
| kafka.bootstrap.servers | Kafka クラスターの接続アドレス。 | はい |
| kafka.schema.record.name | Avro の RecordName パラメーターの定義。 | いいえ |
| kafka.schema.record.namespace | Avro の RecordNameSpace パラメーターの定義。 | いいえ |

### テーブルスキーマ

Kafka データテーブルを作成するとき、 必要に応じて、データテーブルのフィールドを定義するかどうかを選択できます。

| | トピックのスキーマが **Kafka Schema Registry** に定義されている場合のシステムの処理 | トピックのスキーマが **Kafka Schema Registry** に定義されていない場合のシステムの処理 |
|---|---|---|
| データテーブルのフィールドを定義しない。 | トピックスキーマに基づいてテーブルスキーマが定義されます。 | テーブルの作成に失敗します。 |
| データテーブルのフィールドを定義する。 | テーブルスキーマがトピックスキーマと互換性があるかどうかのチェックが行われます。 互換性がない場合、エラーが返されます。 互換性がある場合、カスタムフィールドタイプに基づいてテーブルスキーマが定義されます。 | Kafka データの Avro データスキーマが Kafka Schema Registry に自動的に登録されます。 これには、 **kafka.schema.record.name** フィールドと **kafka.schema.record.namespace** フィールドを指定する必要があります。 |

Kafka Schema Registry にトピックのスキーマを定義する場合、次の制限が適用されます。

- すべてのフィールドは、null 許容型でなければなりません。 たとえば、次のコードの f1 フィールドのタイプは null です。

```
{"type":"record","name":"myrecord","namespace":"test","fields":[{"name":"f1","type":["
string","null"]}]}
```

## 4.7.3 LogHub データテーブルの説明

このトピックでは、LogHub データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName(columnName dataType [,columnName dataType]*)
USING loghub
OPTIONS(propertyName=propertyValue[,propertyName=propertyValue]*);
```

### 設定パラメーター

| パラメーター | 説明 | 必須 |
|---|---|---|
| sls.project | Log Service プロジェクトの名前。 | はい |
| sls.store | Logstore の名前。 | はい |
| access.key.id | Alibaba Cloud によって提供される AccessKey ID。 | はい |

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| access.key.secret | Alibaba Cloud によって提供される AccessKey Secret 。 | はい |
| endpoint | Log Service API のエンドポイント。 | はい |

### テーブルスキーマ

LogHub データテーブルを作成するとき、 データテーブルのフィールドを明示的に定義する必要があります。 たとえば、有効なテーブル作成文は次のとおりです。

```
spark-sql> CREATE TABLE loghub_table_test
         > USING loghub
         > OPTIONS
         > (...)
```

これ以外の場合、次のエラーが返されます。

```
java.lang.IllegalArgumentException: requirement failed: Unable to infer the schema. The
 schema
specification is required to create the table. ;
```

有効なテーブル作成文は次のとおりです。

```
spark-sql> CREATE TABLE loghub_table_test(content string)
         > USING loghub
         > OPTIONS
         > (...)

spark-sql> DESC loghub_table_test;
content  string  NULL
Time taken: 0.436 seconds, Fetched 1 row(s)
```

## 4.7.4 HBase データテーブルの説明

このトピックでは、HBase データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName
USING hbase
OPTIONS(propertyName=propertyValue[,propertyName=propertyValue]*);
```

### 設定パラメーター

- | パラメーター | 説明 | 必須 |
  | --- | --- | --- |
  | catalog | JSON 形式の HBase データテーブルのフィールドの記述。 | はい |

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| hbaseConfiguration | JSON 形式の HBase 設定。たとえば、HBase 接続アドレスを '{"hbase.zookeeper.quorum":"a.b.c.d:2181"}' に設定します。 | はい |

- catalog 設定例を次に示します。

```
{
 "table":{"namespace":"default", "name":"table1"},
 "rowkey":"key",
 "columns":{
  "col0":{"cf":"rowkey", "col":"key", "type":"string"},
  "col1":{"cf":"cf1", "col":"col1", "type":"boolean"},
  "col2":{"cf":"cf2", "col":"col2", "type":"double"},
  "col3":{"cf":"cf3", "col":"col3", "type":"float"},
  "col4":{"cf":"cf4", "col":"col4", "type":"int"},
  "col5":{"cf":"cf5", "col":"col5", "type":"bigint"},
  "col6":{"cf":"cf6", "col":"col6", "type":"smallint"},
  "col7":{"cf":"cf7", "col":"col7", "type":"string"},
  "col8":{"cf":"cf8", "col":"col8", "type":"tinyint"}
 }
}
```

上記の例では、HBase table1 のスキーマを記述しています。 この例では、rowkey フィールドが key に設定され、関連情報が列 1 から 8 に定義されています。

**:**

列 0 で cf フィールドを rowkey に設定する必要があります。

## テーブルスキーマ

HBase データテーブルを作成するとき、データテーブルのフィールドを明示的に定義する必要はありません。 たとえば、有効なテーブル作成文を次に示します。

```
spark-sql> CREATE DATABASE IF NOT EXISTS default;
spark-sql> USE default;
spark-sql> DROP TABLE IF IF EXISTS hbase_table_test;
spark-sql> CREATE TABLE hbase_table_test
        > USING hbase
        > OPTIONS(
        > catalog='{"table":{"namespace":"default","name":"test"},"rowkey":"key", "columns
":{"key":{"cf":"rowkey", "col":"key", "type":"string"},"data":{"cf":"info", "col":"data", "type
":"string"}}}',
        > hbaseConfiguration='{"hbase.zookeeper.quorum":"a.b.c.d:2181"}');

spark-sql> DESC hbase_table_test;
key string  NULL
data  string  NULL
```

```
Time taken: 0.436 seconds, Fetched 2 row(s)
```

## 4.7.5 JDBC データテーブルの説明

このトピックでは、JDBC データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName
USING jdbc2
OPTIONS(propertyName=propertyValue[,propertyName=propertyValue]*);
```

:

上記の構文では、jdbc2 を使用しています。

### 設定パラメーター

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| url | データベースの URL。 | はい |
| driver | データベース接続の確立に使用されるドライバー。 このパラメーターを "com.mysql.jdbc.Driver"eper.quorum":"a.b.c.d:2181"}' に設定できます。 | はい |
| dbtable | データテーブルの名前。 | はい |
| user | データベースへの接続に使用されるユーザー名。 | はい |
| password | データベースへの接続に使用されるパスワード。 | はい |
| batchsize | 一度にデータベースに更新されるデータの数。 このパラメーターは、データがデータベースに書き込まれるときに有効になります。 | いいえ |
| isolationLevel | トランザクション分離レベル。 デフォルト値：READ_UNCOMMITTED。 | いいえ |

- トランザクション分離レベル

| トランザクション分離レベル | ダーティリード | ノンリピータブルリード | ファントムリード |
| --- | --- | --- | --- |
| READ_UNCOMMITTED | はい | はい | はい |
| READ_COMMITTED | いいえ | はい | はい |
| REPEATABLE_READ | いいえ | いいえ | はい |
| SERIALIZABLE | いいえ | いいえ | いいえ |
| NONE | - | - | - |

### テーブルスキーマ

JDBC データテーブルを作成するとき、データテーブルのフィールドを明示的に定義する必要はありません。 たとえば、有効なテーブル作成文を次に示します。

```
spark-sql> CREATE DATABASE IF NOT EXISTS default;
spark-sql> USE default;
spark-sql> DROP TABLE IF EXISTS rds_table_test;
spark-sql> CREATE TABLE rds_table_test
         > USING jdbc2
         > OPTIONS (
         > url="jdbc:mysql://rm-bp11*********i7w9.mysql.rds.aliyuncs.com:3306/default?
useSSL=true",
         > driver="com.mysql.jdbc.Driver",
         > dbtable="test",
         > user="root",
         > password="thisisapassword",
         > batchsize="100",
         > isolationLevel="NONE");

spark-sql> DESC rds_table_test;
id  int NULL
name  string  NULL
Time taken: 0.413 seconds, Fetched 2 row(s)
```

### データ書き込み

データベースへのデータの書き込み方法を表すには、次の追加の SQL 文を作成する必要があります。

```
spark-sql> SET streaming.query.${queryName}.sql=insert into `test` (`id`,`name`) values
(?, ?) ;
spark-sql> SET ...
```

```
spark-sql> INSERT INTO rds_table_test SELECT ...
```

## 4.7.6 Table Store データテーブルの説明

このトピックでは、Table Store データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName
USING tablestore
OPTIONS(propertyName=propertyValue[,propertyName=propertyValue]*);
```

### 設定パラメーター

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| access.key.id | Alibaba Cloud によって提供される AccessKey ID。 | はい |
| access.key.secret | Alibaba Cloud によって提供される AccessKey Secret 。 | はい |
| endpoint | Table Store API のエンドポイント。 | はい |
| table.name | Table Store データテーブルの名前。 | はい |
| instance.name | Table Store インスタンスの名前。 | はい |
| batch.update.size | 一度に Table Store に更新されるデータの数。 このパラメーターは、データがデータベースに書き込まれるときに有効になります。 デフォルト値は 0 です。これは、データが 1 つずつ更新されることを示します。 パラメーターを小さな数に設定することは推奨しません。 | いいえ |
| catalog | JSON 形式の Table Store データテーブルのフィールドの記述。 | はい |

- catalog 設定例を次に示します。

```
{"columns":{
  "col0":{"cf":"cf0", "col":"col0", "type":"string"},
  "col1":{"cf":"cf1", "col":"col1", "type":"boolean"},
  "col2":{"cf":"cf2", "col":"col2", "type":"double"},
```

```
  "col3":{"cf":"cf3", "col":"col3", "type":"float"},
  "col4":{"cf":"cf4", "col":"col4", "type":"int"},
  "col5":{"cf":"cf5", "col":"col5", "type":"bigint"},
  "col6":{"cf":"cf6", "col":"col6", "type":"smallint"},
  "col7":{"cf":"cf7", "col":"col7", "type":"string"},
  "col8":{"cf":"cf8", "col":"col8", "type":"tinyint"}
}
```

上記の例では、Table Store table1 のスキーマを記述しています。 この例では、関連情報が列 1 から 8 に定義されています。

#### テーブルスキーマ

Table Store データテーブルを作成するとき、データテーブルのフィールドを明示的に定義する必要はありません。 たとえば、有効なテーブル作成文を次に示します。

```
spark-sql> CREATE DATABASE IF NOT EXISTS default;
spark-sql> USE default;
spark-sql> DROP TABLE IF EXISTS ots_table_test;
spark-sql> CREATE TABLE ots_table_test
         > USING tablestore
         > OPTIONS(
         > endpoint="http://xxx.cn-hangzhou.vpc.ots.aliyuncs.com",
         > access.key.id="yHiu*******BG2s",
         > access.key.secret="ABctuw0M***************iKKljZy",
         > table.name="test",
         > instance.name="myInstance",
         > batch.update.size = "100"、
         > catalog='{"columns":{"pk":{"col":"pk","type":"string"},"data":{"col":"data","type":"
string"}}}');

spark-sql> DESC ots_table_test;
pk  string  NULL
data  string  NULL
Time taken: 0.501 seconds, Fetched 2 row(s)
```

## 4.7.7 Druid データテーブルの説明

このトピックでは、Druid データテーブルの使用方法について説明します。

#### 構文

```
create table tbName
using druid
options(propertyKey=propertyValue[, propertyKey=propertyValue]*);
```

#### 設定パラメーター

| パラメーター | 説明 | 必須 |
|---|---|---|
| curator.connect | ZooKeeper のホストとポート。例：emr-header-1: 2181 | はい |
| curator.max. retries | ZooKeeper への接続に失敗した時に再接続を試行する最大回数。 デフォルト値： 5。 | いいえ |

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| curator.retry.base.sleep | ZooKeeper への接続に失敗した時に再接続を試行する初期間隔。 デフォルト値：100。 単位：ミリ秒。 | いいえ |
| curator.retry.max.sleep | ZooKeeper への接続に失敗した時に再接続を試行する最大間隔。 デフォルト値：3000。 単位：ミリ秒。 | いいえ |
| index.service | インデックスサービスオーバーロードノードのサービス名。 例：druid/overlord | はい |
| data.source | Druid に書き込まれるデータのデータソースの名前。 | はい |
| discovery.path | ZooKeeper 内の Druid のディスカバリパス。 デフォルト値：/druid/discovery。 | いいえ |
| firehouse | firehose。 例：druid:firehose:%s | はい |
| rollup.aggregators | JSON 形式の Tranquility のロールアップアグリゲーター。 例：<br>`{\"metricsSpec\":[{\"type\":\"count\",\"name\":\"count\"},`<br>`{\"type\":\"doubleSum\",\"fieldName\":\"value\",\"name\":\"sum\"},`<br>`{\"type\":\"doubleMin\",\"fieldName\":\"value\",\"name\":\"min\"},`<br>`{\"type\":\"doubleMax\",\"fieldName\":\"value\",\"name\":\"max\"}]}`<br>ここで、**metricsSpec** は固定値です。 | はい |
| rollup.dimensions | Druid に書き込まれるデータのディメンション。 | はい |
| rollup.query.granularities | ロールアップの粒度。 例：minute | はい |
| tuning.window.period | Tranquility のチューニングウィンドウ期間。 デフォルト値：PT10M | いいえ |
| tuning.segment.granularity | Tranquility のチューニングセグメントの粒度。 デフォルト値：DAY | いいえ |
| tuning.partitions | Tranquility のチューニングパーティションの数。 デフォルト値：1 | いいえ |
| tuning.replications | Tranquillity のチューニングレプリケーション回数。 デフォルト値：1 | いいえ |
| tuning.warming.period | Tranquility のチューニング初期化期間。 デフォルト値：0 | いいえ |
| timestampSpec.column | データが Druid に書き込まれるときのタイムスタンプ列の名前。 デフォルト値：timestamp | いいえ |

| パラメーター | 説明 | 必須 |
|---|---|---|
| timestampSpec.format | データが Druid に書き込まれるときのタイムスタンプ列の形式。 デフォルト値：iso | いいえ |

### テーブルスキーマ

Druid データテーブルを作成するとき、データテーブルのフィールドを明示的に定義する必要はありません。 たとえば、有効なテーブル作成文を次に示します。

```
create table druid_test_table
using druid
options(
curator.connect="${ZooKeeper-host}:${ZooKeeper-port}}",
index.service="druid/overlord",
data.source="test_source",
discovery.path="/druid/discovery",
firehouse="druid:firehose:%s",
rollup.aggregators="{\"metricsSpec\":[{\"type\":\"count\",\"name\":\"count\"},
                         {\"type\":\"doubleSum\",\"fieldName\":\"value\",\"name\":\"sum
\"},
                         {\"type\":\"doubleMin\",\"fieldName\":\"value\",\"name\":\"min
\"},
                         {\"type\":\"doubleMax\",\"fieldName\":\"value\",\"name\":\"max
\"}]}",
rollup.dimensions="timestamp,metric,userId",
rollup.query.granularities="minute",
tuning.segment.granularity="FIVE_MINUTE",
tuning.window.period="PT5M",
timestampSpec.column="timestamp",
timestampSpec.format="posix");
```

## 4.7.8 Redis データテーブルの説明

このトピックでは、Redis データテーブルの使用方法について説明します。

### 構文

```
CREATE TABLE tbName[(columnName dataType [,columnName dataType]*)]
USING redis
```

```
OPTIONS(propertyKey=propertyValue[, propertyKey=propertyValue]*);
```

#### 設定パラメーター

| パラメーター | 説明 | 必須 |
|---|---|---|
| **table** | Redis に書き込まれるデータのすべてのキーに追加されるプレフィックス。 キーの形式は、${table}:${key.column} です。 この形式の ${table} はプレフィックスを示し、${key.column} は key.column パラメーターで指定されたキーを示します。 | はい |
| **redis.save.mode** | 書き込むデータが Redis に既に存在する場合の処理方法。 有効な値：append (書き込むデータを既存のデータに追加する)、overwrite (既存のデータを上書きする)、errorifexists (例外をスローする)、ignore (既存のデータを破棄する)。 デフォルト値：append | いいえ |
| **model** | データの保存形式。 有効な値：hash と binary。 デフォルト値：hash。 | いいえ |
| **filter.keys.by.type** | model パラメーターで指定された形式で保存されていないデータを除外するかどうか。 デフォルト値：false。 | いいえ |
| **key.column** | Redis にデータを書き込むためのキーとして使用される行の列。 デフォルトでは、universally unique identifier (UUID) がキーとして自動生成されます。 | いいえ |

| パラメーター | 説明 | 必須 |
| --- | --- | --- |
| **ttl** | キーの有効期間 (TTL)。 このパラメーターを設定しない場合、データはデフォルトで永続的に保存されます。 このパラメーターを設定する場合、パラメーター値は有効期間を示します。 単位：秒。 | いいえ |
| **max.pipeline.size** | パイプラインでサポート可能なバルクデータ書き込み操作の最大数。 デフォルト値：100 | いいえ |
| **host** | Redis に接続するホスト。 デフォルト値：localhost | いいえ |
| **port** | Redis に接続するポート。 デフォルト値: 6379 | いいえ |
| **dbNum** | データを保存する Redis のデータベース番号。 デフォルト値： 0 | いいえ |

**テーブルスキーマ**

Redis データテーブルを作成するとき、 データテーブルのフィールドを明示的に定義する必要があります。 たとえば、有効なテーブル作成文を次に示します。

```
spark-sql> CREATE TABLE redis_test_table(`key0` STRING, `value0` STRING, `key1` STRING,
 `value1` STRING)
         > USING redis
         > OPTIONS(
         > table="test",
         > redis.save.mode="append",
         > model="hash",
         > filter.keys.by.type="false",
         > key.column="uuid",
         > max.pipeline.size="100",
         > host="localhot",
         > port="6379",
         > dbNum="0");
```

# 5 Hadoop

## 5.1 パラメーター説明

Hadoop コード内のパラメーターの説明

以下のパラメーターは、Hadoop コードで使用できます。

| プロパティ | デフォルト | 説明 |
| --- | --- | --- |
| fs.oss.accessKeyId | なし | (省略可能) OSS へのアクセスに使用する必須の AccessKey ID |
| fs.oss.accessKeySecret | なし | (省略可能) OSS へのアクセスに使用する必須の AccessKey Secret |
| fs.oss.securityToken | なし | (省略可能) OSS へのアクセスに使用する必須の STS トークン |
| fs.oss.endpoint | なし | (省略可能) OSS へのアクセスに使用するエンドポイント |
| fs.oss.multipart.thread.number | 5 | アップロード部分コピー時に OSS が使用するスレッド (同時実行) 数 |
| fs.oss.copy.simple.max.byte | 134217728 | 一般的な API を使用した OSS の内部コピーのファイル制限 |
| fs.oss.multipart.split.max.byte | 67108864 | 一般的な API を使用して OSS のバケット間でファイルをコピーする場合のファイルマルチパート制限 |
| fs.oss.multipart.split.number | 5 | OSS の一般的な API を使用中のバケット間でファイルをコピーする場合のファイルマルチパート制限。 デフォルトの制限はコピーで使用されるスレッド数と同じです。 |
| fs.oss.impl | com.aliyun.fs.oss.nat.NativeOssFileSystem | ネイティブ OSS ファイルシステムの実装クラス |

| プロパティ | デフォルト | 説明 |
| --- | --- | --- |
| fs.oss.buffer.dirs | /mnt/disk1,/mnt/disk2,... | OSS のローカルの一時ディレクトリ。 It uses a cluster data disk by default. |
| fs.oss.buffer.dirs.exists | false | OSS 一時ディレクトリが存在するかどうかを示します。 |
| fs.oss.client.connection.timeout | 50000 | OSS クライアントの接続タイムアウト (ミリ秒) |
| fs.oss.client.socket.timeout | 50,000 | OSS クライアントのソケットタイムアウト (ミリ秒)。 |
| fs.oss.client.connection.ttl | -1 | 接続存続時間 |
| fs.oss.connection.max | 1024 | 許可される最大接続数 |
| io.compression.codec.snappy.native | false | Snappy ファイルを標準としてマークするかどうかを示します。 デフォルトでは、Hadoop は Hadoop によって変更された Snappy ファイルを認識します。 |

## 5.2 MapReduce ジョブの作成と実行

このページでは、E-MapReduce クラスター上で MapReduce ジョブを作成して実行する方法について説明します。

### MapReduce での OSS 利用

MapReduce で OSS に対してデータを読み書きするには、以下のパラメータを設定します。

```
conf.set("fs.oss.accessKeyId", "${accessKeyId}");
    conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
    conf.set("fs.oss.endpoint","${endpoint}");
```

パラメーター説明 :

- ${accessKeyId}: お客様のアカウントの AccessKey ID
- ${accessKeySecret}: AccessKey ID に対応する AccessKey Secret
- ${endpoint}: OSS へのアクセスに使用するネットワーク。 クラスターが存在するリージョンによって異なり、対応する OSS もクラスターが存在するリージョンにある必要があります。

  特定の値の詳細は、 「OSS エンドポイント (OSS endpoints)」をご参照ください。

### ワードカウント

以下の例では、OSS からテキストを読み取り、ワードカウントを計算する方法について説明します。 手順は以下のとおりです。

1. プログラムの記述

Java コードを例とします。 Hadoop の公式 Web サイトの WordCount サンプルを以下のように変更します。コードに AccessKey ID と AccessKey Secret の設定を追加して、ジョブが OSS ファイルにアクセスする権限を持つようにインスタンスを変更します。

```
package org.apache.hadoop.examples;
 import java.io.IOException;
 import java.util.StringTokenizer;
 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 org.apache.hadoop.util.GenericOptionsParser;
 public class EmrWordCount {
  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(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
```

```
    conf.set("fs.oss.accessKeyId", "${accessKeyId}");
    conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
    conf.set("fs.oss.endpoint","${endpoint}");
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(EmrWordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

2. プログラムのコンパイル

   まず、JDK 環境と Hadoop 環境を設定してから、以下の操作を実行する必要があります。

```
mkdir wordcount_classes
 javac -classpath ${HADOOP_HOME}/share/hadoop/common/hadoop-common-2.6.
0.jar:${HADOOP_HOME}/share/hadoop/mapreduce/hadoop-mapreduce-client-core
-2.6.0.jar:${HADOOP_HOME}/share/hadoop/common/lib/commons-cli-1.2.jar -d
wordcount_classes EmrWordCount.java
 jar cvf wordcount.jar -C wordcount_classes .
```

3. ジョブの作成

   - 前のステップで準備した JAR ファイルを OSS にアップロードします。 詳細な操作は、OSS Web サイトにログインしてご確認ください。 OSS 上の JAR ファイルのパスが oss://emr/

jars/wordcount.jar、入力パスと出力パスは oss://emr/data/WordCount/Input および oss://emr/data/WordCount/Output であるとします。

- 以下のとおり E-MapReduce ジョブを作成します。

Create job

* Name : testing-OSS-wordcount

Length: 1 to 64 characters. Only Chinese characters, English letters, numbers '-', and '_' are allowed

* Type : Spark Hadoop Hive Pig Sqoop Spark SQL Shell

* Parameter : jar ossref://emr/jars/wordcount.jar org.apache.hadoop.examples.EmrWordCount oss://emr/data/WordCount/Input oss://emr/data/WordCount/Output

Select OSS path oss console Upload

* Actual execution : **hadoop** jar ossref://emr/jars/wordcount.jar org.apache.hadoop.examples.EmrWordCount oss://emr/data/WordCount/Input oss://emr/data/WordCount/Output

* Failure policy : Pause current execution plan / Continue execution of next job

OK Cancel

**4.** 実行プランの作成

E-MapReduce で実行プランを作成し、作成したジョブを実行プランに追加します。 ポリシーとして **[今すぐ実行]** を選択し、WordCount ジョブが選択したクラスターで実行されるようにします。

### Maven を使用して MR ジョブを管理

プロジェクトのサイズが大きくなると、管理はかなり複雑になります。 プロジェクトを管理するには、Maven または同様のソフトウェア管理ツールを使用するよう推奨します。 手順は以下のとおりです。

1. Maven のインストール

   まず、Maven がインストール済みであることを確認します。

2. プロジェクトフレームワークを生成します。

   プロジェクトのルートディレクトリ (プロジェクトのルートディレクトリを D:/workspace と仮定) で、以下のコマンドを実行します。

```
mvn archetype:generate -DgroupId=com.aliyun.emr.hadoop.examples -DartifactId=
wordcountv2 -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=
false
```

   Maven は自動的に空のサンプルプロジェクトを基本の pom.xml ファイルと App クラス (クラスパッケージパスは指定済みの groupId と同じ) を含む D:/workspace/wordcountv2 (指定済みの artifactId と同じ) に生成します。

3. Hadoop 依存関係の追加

   お気に入りの IDE でプロジェクトを開き、pom.xml ファイルを編集します。 以下の内容を依存関係に追加します。

```
<dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-mapreduce-client-common</artifactId>
        <version>2.6.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.6.0</version>
    </dependency>
```

4. プログラムの記述

   com.aliyun.emr.hadoop.examples パッケージの下の App クラスと同じディレクトリレベルに WordCount2.java という名前の新しいクラスを追加します。 コンテンツは以下のとおりです。

```
package com.aliyun.emr.hadoop.examples;
 import java.io.BufferedReader;
 import java.io.FileReader;
 import java.io.IOException;
 import java.net.URI;
 import java.util.ArrayList;
 import java.util.HashSet;
```

```
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;
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 org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.StringUtils;
public class WordCount2 {
  public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{
    static enum CountersEnum { INPUT_WORDS }
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private boolean caseSensitive;
    private Set<String> patternsToSkip = new HashSet<String>();
    private Configuration conf;
    private BufferedReader fis;
    @Override
    public void setup(Context context) throws IOException,
        InterruptedException {
      conf = context.getConfiguration();
      caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
      if (conf.getBoolean("wordcount.skip.patterns", true)) {
        URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
        for (URI patternsURI : patternsURIs) {
          Path patternsPath = new Path(patternsURI.getPath());
          String patternsFileName = patternsPath.getName().toString();
          parseSkipFile(patternsFileName);
        }
      }
    }
    private void parseSkipFile(String fileName) {
      try {
        fis = new BufferedReader(new FileReader(fileName));
        String pattern = null;
        while ((pattern = fis.readLine()) ! = null) {
          patternsToSkip.add(pattern);
        }
      } catch (IOException ioe) {
        System.err.println("Caught exception while parsing the cached file '"
            + StringUtils.stringifyException(ioe));
      }
    }
    @Override
    public void map(Object key, Text value, Context context
    ) throws IOException, InterruptedException {
      String line = (caseSensitive) ?
          value.toString() : value.toString().toLowerCase();
      for (String pattern : patternsToSkip) {
        line = line.replaceAll(pattern, "");
      }
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        Counter counter = context.getCounter(CountersEnum.class.getName(),
```

```
                    CountersEnum.INPUT_WORDS.toString());
                counter.increment(1);
            }
        }
    }
    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(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("fs.oss.accessKeyId", "${accessKeyId}");
       conf.set("fs.oss.accessKeySecret", "${accessKeySecret}");
        conf.set("fs.oss.endpoint","${endpoint}");
        GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
        String[] remainingArgs = optionParser.getRemainingArgs();
        if (!( remainingArgs.length ! = 2 || remainingArgs.length ! = 4)) {
            System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount2.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        List<String> otherArgs = new ArrayList<String>();
        for (int i=0; i < remainingArgs.length; ++i) {
            if ("-skip".equals(remainingArgs[i])) {
                job.addCacheFile(new Path(EMapReduceOSSUtil.buildOSSCompleteUri(
remainingArgs[++i], conf)).toUri());
                job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
            } else {
                otherArgs.add(remainingArgs[i]);
            }
        }
        FileInputFormat.addInputPath(job, new Path(EMapReduceOSSUtil.buildOSSCo
mpleteUri(otherArgs.get(0), conf)));
        FileOutputFormat.setOutputPath(job, new Path(EMapReduceOSSUtil.buildOSSCo
mpleteUri(otherArgs.get(1), conf)));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
 }
```

同じ WordCount2 のディレクトリにある EMapReduceOSSUtil クラスの以下のサンプルコードを参照します。

```
package com.aliyun.emr.hadoop.examples;
 import org.apache.hadoop.conf.Configuration;
 public class EMapReduceOSSUtil {
    private static String SCHEMA = "oss://";
```

```
    private static String AKSEP = ":";
    private static String BKTSEP = "@";
    private static String EPSEP = ".";
    private static String HTTP_HEADER = "http://";
    /**
     * complete OSS uri
     * convert uri like: oss://bucket/path  to  oss://accessKeyId:accessKeySecret@
bucket.endpoint/path
     * ossref do not need this
     *
     * @param oriUri original OSS uri
     */
    public static String buildOSSCompleteUri(String oriUri, String akId, String akSecret,
String endpoint) {
        if (akId == null) {
            System.err.println("miss accessKeyId");
            return oriUri;
        }
        if (akSecret == null) {
            System.err.println("miss accessKeySecret");
            return oriUri;
        }
        if (endpoint == null) {
            System.err.println("miss endpoint");
            return oriUri;
        }
        int index = oriUri.indexOf(SCHEMA);
        if (index == -1 || index ! = 0) {
            return oriUri;
        }
        int bucketIndex = index + SCHEMA.length();
        int pathIndex = oriUri.indexOf("/", bucketIndex);
        String bucket = null;
        if (pathIndex == -1) {
            bucket = oriUri.substring(bucketIndex);
        } else {
            bucket = oriUri.substring(bucketIndex, pathIndex);
        }
        StringBuilder retUri = new StringBuilder();
        retUri.append(SCHEMA)
                .append(akId)
                .append(AKSEP)
                .append(akSecret)
                .append(BKTSEP)
                .append(bucket)
                .append(EPSEP)
                .append(stripHttp(endpoint));
        if (pathIndex > 0) {
            retUri.append(oriUri.substring(pathIndex));
        }
        return retUri.toString();
    }
    public static String buildOSSCompleteUri(String oriUri, Configuration conf) {
        return buildOSSCompleteUri(oriUri, conf.get("fs.oss.accessKeyId"), conf.get("fs.
oss.accessKeySecret"), conf.get("fs.oss.endpoint"));
    }
    private static String stripHttp(String endpoint) {
        if (endpoint.startsWith(HTTP_HEADER)) {
            return endpoint.substring(HTTP_HEADER.length());
        }
        return endpoint;
    }
```

```
}
```

5. コードコンパイル、パッケージ化、およびアップロード

   プロジェクトディレクトリで、以下のコマンドを実行します。

   ```
   mvn clean package -DskipTests
   ```

   プロジェクトディレクトリのターゲットディレクトリに、ジョブの JAR パッケージである wordcountv2-1.0-SNAPSHOT.jar ファイルが表示されます。 JAR パッケージを OSS にアップロードします。

6. ジョブの作成

   以下のパラメーターを使用して E-MapReduce に新しいジョブを作成します。

   ```
   jar ossref://yourBucket/yourPath/wordcountv2-1.0-SNAPSHOT.jar com.aliyun.emr.
   hadoop.examples.WordCount2 -Dwordcount.case.sensitive=true oss://yourBucket/
   yourPath/The_Sorrows_of_Young_Werther.txt oss://yourBucket/yourPath/output -skip
    oss://yourBucket/yourPath/patterns.txt
   ```

   ここで、yourBucket は OSS バケットを表し、yourPath はバケット内のパスを表します。 必要に応じて設定します。 関連するリソースを処理するファイル、oss://yourBucket/yourPath/The_Sorrows_of_Young_Werther.txt および oss://yourBucket/yourPath/patterns.txt をダウンロード後、OSS に格納する必要があります。 ジョブに必要なリソースをダウンロードし、それらのリソースを OSS 内の対応するディレクトリに格納できます。

   ダウンロードするファイル: The_Sorrows_of_Young_Werther.txtpatterns.txt

7. 実行プランの作成および実行

   E-MapReduce で実行プランを作成し、それをジョブと関連付けてから実行プランを実行します。

## 5.3 Hive ジョブの作成と実行

このページでは、E-MapReduce クラスター上で Hive ジョブを作成し実行する方法について説明します。

### Hive で OSS を使用

Hive で OSS を読み書きするには、以下の文を使用して外部テーブルを作成します。

```
CREATE EXTERNAL TABLE eusers (
  userid INT)
```

```
LOCATION 'oss://emr/users';
```

ご利用のクラスターのバージョンが古く、上記のメソッドがサポートされていない、または別のアカウントの AccessKey を使用して別のリージョンの OSS インスタンスにアクセスする場合は、以下のステップを実行します。

```
CREATE EXTERNAL TABLE eusers (
 userid INT)
 LOCATION 'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/users';
```

パラメーターの説明：

- ${accessKeyId}: お客様のアカウントの AccessKeyId
- ${accessKeySecret} : AccessKeyId に対応する秘密鍵
- ${endpoint} : OSS へのアクセスに使用されるネットワーク。 クラスターが存在するリージョンによって異なります。 対応する OSS インスタンスは、クラスターが存在するリージョンにある必要があります。

  特定の値の詳細は、 「OSSエンドポイント (OSS endpoints)」をご参照ください。

### コンピューティングエンジンに Tez を使用

Tez は E-MapReduce バージョン 2.1.0 以降に実装されています。 Tez は、複雑な DAG スケジューリングジョブを最適化するためのコンピューティングフレームワークです。 多くのシナリオで Hive ジョブのスピードが大幅に向上しています。

実行エンジンとして Tez を設定することでジョブを最適化することができます。 以下のステップを実行します。

```
set hive.execution.engine=tez
```

- 例 1

  以下のステップを実行します。

  1. 1. 以下のスクリプトを記述し、hiveSample1.sql として保存してから OSS にアップロードします。

     ```
     USE DEFAULT;
      set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
      set hive.stats.autogather=false;
      DROP TABLE emrusers;
      CREATE EXTERNAL TABLE emrusers (
       userid INT,
       movieid INT,
       rating INT,
       unixtime STRING )
      ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '\t'
     ```

```
 STORED AS TEXTFILE
 LOCATION 'oss://${bucket}/yourpath';
SELECT COUNT(*) FROM emrusers;
SELECT * from emrusers limit 100;
SELECT movieid,count(userid) as usercount from emrusers group by movieid order
by usercount desc limit 50;
```

2. テスト用のデータリソース

   以下のリンクから Hive ジョブに必要なリソースをダウンロードして、それらのリソースを OSS インスタンス内の対応するディレクトリに格納できます。

   ダウンロードするリソース : 公開テストデータ

3. ジョブの作成

   以下の設定を使用して E-MapReduce に新しいジョブを作成します。

   ```
   -f ossref://${bucket}/yourpath/hiveSample1.sql
   ```

   この例では、${bucket} は OSS バケットを表し、yourPath はバケット内のパスを表します。 yourPath を Hive スクリプトが保存されている場所に置き換える必要があります。

4. 実行プランの作成と実行

   実行プランを作成します。 既存のクラスターと関連付けることも、必要に応じてクラスターを作成してプランをクラスターと関連付けることもできます。 ジョブを **[手動で実行]** で保存します。 前のページに戻り、**[今すぐ実行]** をクリックしてジョブを実行します。

- 例 2

  例として HiBench でスキャンします。

  以下のステップを実行します。

  1. 以下のスクリプトを記述します。

     ```
     USE DEFAULT;
      set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
      set mapreduce.job.maps=12;
      set mapreduce.job.reduces=6;
      set hive.stats.autogather=false;
      DROP TABLE uservisits;
      CREATE EXTERNAL TABLE uservisits (sourceIP STRING,destURL STRING,visitDate
     STRING,adRevenue DOUBLE,userAgent STRING,countryCode STRING,languageCo
     de STRING,searchWord STRING,duration INT ) ROW FORMAT DELIMITED FIELDS
     ```

```
TERMINATED BY ',' STORED AS SEQUENCEFILE LOCATION 'oss://${bucket}/sample-data/hive/Scan/Input/uservisits';
```

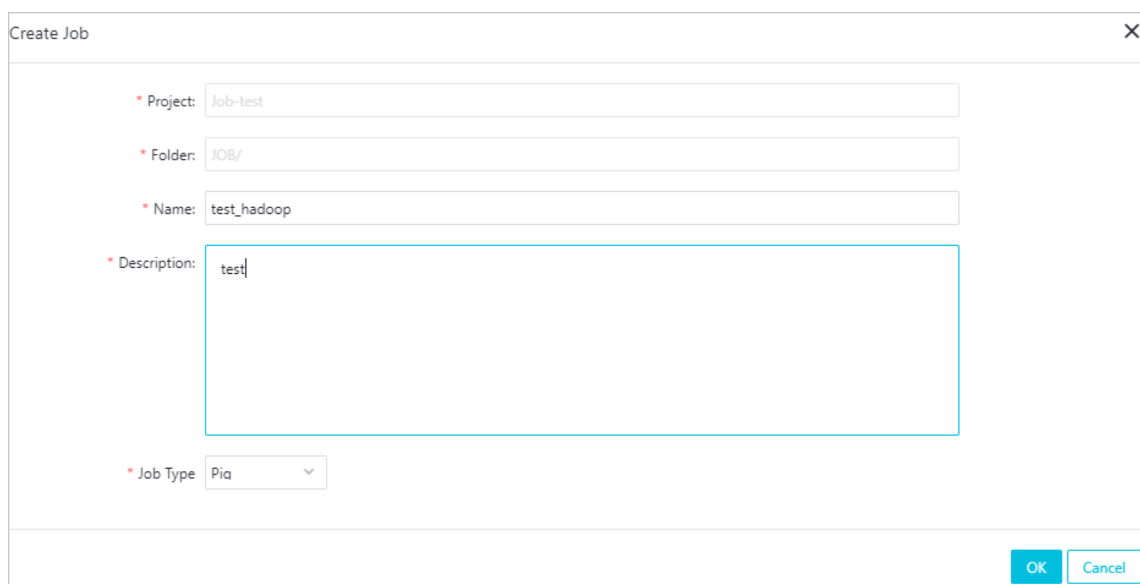
2. テストデータの準備

   以下のリンクからジョブの必須のリソースをダウンロードしそれらのリソースを OSS インスタンス内の対応するディレクトリに格納できます。

   ダウンロードするリソース：uservisits

3. ジョブの作成

   ステップ 1 で作成したスクリプトを OSS に格納します。 たとえば、ストレージパスが oss://emr/jars/scan.hive の場合は、以下のステップを実行して E-MapReduce にジョブを作成します。




4. 実行プランを作成し実行します。

   実行プランを作成します。 既存のクラスターと関連付けることも、必要に応じてクラスターを作成してプランをクラスターと関連付けることもできます。 ジョブを **[手動で実行]** で保存します。 前のページに戻り、**[今すぐ実行]** をクリックしてジョブを実行します。

## 5.4 Pig ジョブの作成および実行

このページでは、E-MapReduce クラスター上で Pig ジョブを作成し実行する方法について説明します。

### Pig で OSS を使用

OSS パスには以下の形式を使用します。

```
oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/${path}
```

パラメーターの説明 :

- ${accessKeyId} : お客様のアカウントの AccessKey ID
- ${accessKeySecret} : AccessKey ID に対応する AccessKey Secret
- ${bucket} : AccessKey ID に対応するバケット
- ${endpoint} : OSS へのアクセスに使用するネットワーク。 クラスターが存在するリージョンによって異なり、対応する OSS もクラスターが存在するリージョンにある必要があります。

  特定の値に関する詳細は、 「OSS エンドポイント (OSS endpoint)」をご参照ください。
- ${path} : バケット内のパス

Pig の script1-hadoop.pig を例に取り上げます。 Pig 内の tutorial.jar および excite.log.bz2 を OSS にアップロードします。 ファイルをアップロードする URL がそれぞれ oss://emr/jars/tutorial.jar および oss://emr/data/excite.log.bz2 であるとします。

以下のステップを実行します。

1. スクリプトの作成

   以下のとおり、JAR ファイルのパスおよびスクリプト内の 入力パスと出力パスを編集します。OSS パスの形式が oss://${accesskeyId}:${accessKeySecret}@${bucket}.${endpoint}/object/path であることにご注意ください。

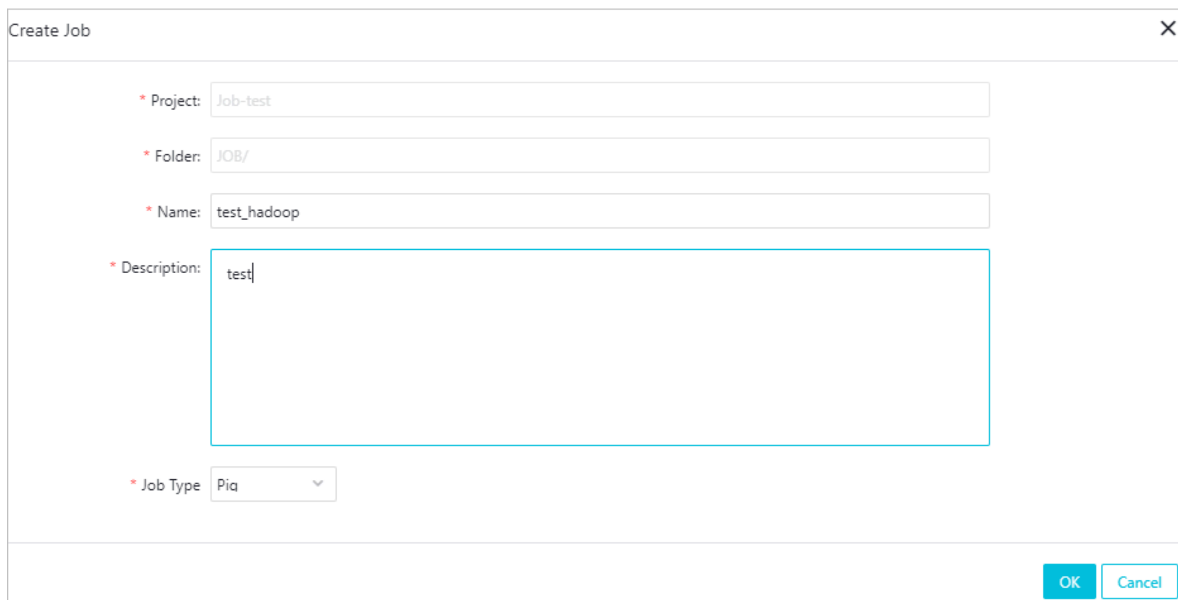
```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
* or more contributor license agreements.  NOTICE ファイルの表示
* distributed with this work for additional information
* regarding copyright ownership.  The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License.  You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
```

```
* limitations under the License.
*/
-- Query Phrase Popularity (Hadoop cluster)
-- This script processes a search query log file from the Excite search engine and finds
 search phrases that occur with particular high frequency during certain times of the
day.
-- Register the tutorial JAR file so that the included UDFs can be called in the script.
REGISTER oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/data/
tutorial.jar;
-- Use the  PigStorage function to load the excite log file into the #raw# bag as an
array of records.
-- Input: (user,time,query)
raw = LOAD 'oss://${AccessKeyId}:${AccessKeySecret}@${bucket}.${endpoint}/data/
excite.log.bz2' USING PigStorage('\t') AS (user, time, query);
-- Call the NonURLDetector UDF to remove records if the query field is empty or a URL.
clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);
-- Call the ToLower UDF to change the query field to lowercase.
clean2 = FOREACH clean1 GENERATE user, time,     org.apache.pig.tutorial.ToLower(
query) as query;
-- Because the log file only contains queries for a single day, we are only interested in
 the hour.
-- The excite query log timestamp format is YYMMDDHHMMSS.
-- Call the ExtractHour UDF to extract the hour (HH) from the time field.
houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time)
as hour, query;
-- Call the NGramGenerator UDF to compose the n-grams of the query.
ngramed1 = FOREACH houred GENERATE user, hour, flatten(org.apache.pig.tutorial.
NGramGenerator(query)) as ngram;
-- Use the  DISTINCT command to get the unique n-grams for all records.
ngramed2 = DISTINCT ngramed1;
-- Use the  GROUP command to group records by n-gram and hour.
hour_frequency1 = GROUP ngramed2 BY (ngram, hour);
-- Use the  COUNT function to get the count (occurrences) of each n-gram.
hour_frequency2 = FOREACH hour_frequency1 GENERATE flatten($0), COUNT($1) as
count;
-- Use the  GROUP command to group records by n-gram only.
-- Each group now corresponds to a distinct n-gram and has the count for each hour.
uniq_frequency1 = GROUP hour_frequency2 BY group::ngram;
-- For each group, identify the hour in which this n-gram is used with a particularly
high frequency.
-- Call the ScoreGenerator UDF to calculate a "popularity" score for the n-gram.
uniq_frequency2 = FOREACH uniq_frequency1 GENERATE flatten($0), flatten(org.
apache.pig.tutorial.ScoreGenerator($1));
-- Use the  FOREACH-GENERATE command to assign names to the fields.
uniq_frequency3 = FOREACH uniq_frequency2 GENERATE $1 as hour, $0 as ngram, $2
as score, $3 as count, $4 as mean;
-- Use the  FILTER command to move all records with a score less than or equal to 2.0.
filtered_uniq_frequency = FILTER uniq_frequency3 BY score > 2.0;
-- Use the  ORDER command to sort the remaining records by hour and score.
ordered_uniq_frequency = ORDER filtered_uniq_frequency BY hour, score;
-- Use the  PigStorage function to store the results.
-- Output: (hour, n-gram, score, count, average_counts_among_all_hours)
```

```
STORE ordered_uniq_frequency INTO 'oss://${AccessKeyId}:${AccessKeySecret}@${
bucket}.${endpoint}/data/script1-hadoop-results' USING PigStorage();
```

2. ジョブの作成

   ステップ 1 で作成したスクリプトを OSS に格納します。 たとえば、ストレージパスが oss://emr/jars/script1-hadoop.pig の場合は、以下のステップを実行して E-MapReduce にジョブを作成します。




3. 実行プランの作成および実行

   E-MapReduce に実行プランを作成し、作成した Pig ジョブを実行プランに追加します。 ポリシーとして **[今すぐ実行]**を選択し、script1-hadoop ジョブを選択したクラスターで実行できるようにします。

# 5.5 Hadoop ストリーミングジョブの作成

このページでは、Python を使用して Hadoop ストリーミングジョブを作成する方法について説明します。

### Python を使用して Hadoop ストリーミングジョブを作成

マッパーコードは以下のとおりです。

```
#! /usr/bin/env python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
```

```
        print '%s\t%s' % (word, 1)
```

レデューサーのコードは以下のとおりです。

```
#! /usr/bin/env python
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

マッパーコードが /home/hadoop/mapper.py、レデューサーコードが /home/hadoop/reducer.py で保存され、入力と出力のパスが HDFS ファイルシステム内でそれぞれ /tmp/input および /tmp/output であるとします。 E-MapReduce クラスター内で以下の Hadoop コマンドを送信します。

```
hadoop jar /usr/lib/hadoop-current/share/hadoop/tools/lib/hadoop-streaming-*.jar
 -file /home/hadoop/mapper.py -mapper mapper.py -file /home/hadoop/reducer.py -
reducer reducer.py -input /tmp/hosts -output /tmp/output
```

## 5.6 Hive で Table Store データを処理

このページでは、Hive で Table Store データを処理する方法について説明します。

### Hive を Table Store に接続

- データテーブルの準備

  pet という名前のテーブルを作成し、名前フィールドを主キーに設定します。

| 名前 | 飼い主 | 種 | 性別 | 誕生日 | 命日 |
| --- | --- | --- | --- | --- | --- |
| ふわふわ | ハロルド | ネコ | 雌 | 1993-02-04 | |
| クローズ | グウェン | ネコ | 雄 | 1994-03-17 | |
| バフィ | ハロルド | 犬 | 雌 | 1989-05-13 | |

| 名前 | 飼い主 | 種 | 性別 | 誕生日 | 命日 |
| --- | --- | --- | --- | --- | --- |
| ファング | ベニー | 犬 | 雄 | 1990-08-27 | |
| クッパ | ダイアン | 犬 | 雄 | 1979-08-31 | 1995-07-29 |
| チャーピー | グウェン | 鳥 | 雌 | 1998-09-11 | |
| ウィスラー | グウェン | 鳥 | | 1997-12-09 | |
| スリム | ベニー | ヘビ | 雄 | 1996-04-29 | |
| パフボール | ダイアン | ハムスター | 雌 | 1999-03-30 | |

- 以下に、Hive で Table Store データを処理するコーディング例を記述します。

1. コマンドライン

```
$ HADOOP_HOME=YourHadoopDir HADOOP_CLASSPATH=emr-tablestore-<version>.
jar:tablestore-4.1.0-jar-with-dependencies.jar:joda-time-2.9.4.jar bin/hive
```

2. 外部テーブルの作成

```
CREATE EXTERNAL TABLE pet
  (name STRING, owner STRING, species STRING, sex STRING, birth STRING, death
STRING)
  STORED BY 'com.aliyun.openservices.tablestore.hive.TableStoreStorageHandler'
  WITH SERDEPROPERTIES(
    "tablestore.columns.mapping"="name,owner,species,sex,birth,death")
  TBLPROPERTIES (
    "tablestore.endpoint"="YourEndpoint",
    "tablestore.access_key_id"="YourAccessKeyId",
    "tablestore.access_key_secret"="YourAccessKeySecret",
    "tablestore.table.name"="pet");
```

3. 外部テーブルにデータを挿入

```
INSERT INTO pet VALUES("Fluffy", "Harold", "cat", "f", "1993-02-04", null);
INSERT INTO pet VALUES("Claws", "Gwen", "cat", "m", "1994-03-17", null);
INSERT INTO pet VALUES("Buffy", "Harold", "dog", "f", "1989-05-13", null);
INSERT INTO pet VALUES("Fang", "Benny", "dog", "m", "1990-08-27", null);
INSERT INTO pet VALUES("Bowser", "Diane", "dog", "m", "1979-08-31", "1995-07-29
");
INSERT INTO pet VALUES("Chirpy", "Gwen", "bird", "f", "1998-09-11", null);
INSERT INTO pet VALUES("Whistler", "Gwen", "bird", null, "1997-12-09", null);
INSERT INTO pet VALUES("Slim", "Benny", "snake", "m", "1996-04-29", null);
INSERT INTO pet VALUES("Puffball", "Diane", "hamster", "f", "1999-03-30", null);
```

4. クエリデータ

```
> SELECT * FROM pet;
Bowser  Diane   dog     m       1979-08-31      1995-07-29
Buffy   Harold  dog     f       1989-05-13      NULL
Chirpy  Gwen    bird    f       1998-09-11      NULL
Claws   Gwen    cat     m       1994-03-17      NULL
Fang    Benny   dog     m       1990-08-27      NULL
Fluffy  Harold  cat     f       1993-02-04      NULL
Puffball        Diane   hamster f       1999-03-30      NULL
Slim    Benny   snake   m       1996-04-29      NULL
Whistler        Gwen    bird    NULL    1997-12-09      NULL
```

```
> SELECT * FROM pet WHERE birth > "1995-01-01";
Chirpy  Gwen    bird    f       1998-09-11      NULL
Puffball        Diane   hamster f       1999-03-30      NULL
Slim    Benny   snake   m       1996-04-29      NULL
Whistler        Gwen    bird    NULL    1997-12-09      NULL
```

**データ型変換**

| | TINYINT | SMALLINT | INT | BIGINT | FLOAT | DOUBLE | BOOLEAN | STRING | BINARY |
|---|---|---|---|---|---|---|---|---|---|
| INTEGER | サポートあり(精度は低下します) | Supported (with loss of precision) | Supported (with loss of precision) | サポートあり | サポートあり(精度は低下します) | サポートあり(精度は低下します) | | | |
| DOUBLE | サポートあり(精度は低下します) | サポートあり(精度は低下します) | サポートあり(精度は低下します) | サポートあり(精度は低下します) | サポートあり(精度は低下します) | サポートあり | | | |
| BOOLEAN | | | | | | | サポートあり | | |
| STRING | | | | | | | | サポートあり | |
| BINARY | | | | | | | | | サポートあり |

**付録**

完全なサンプルコードについては、

- 「Hive で Table Store データを処理 (Process Table Store data in Hive)」をご参照ください。

## 5.7 MR での Table Store データの処理

このページでは、MapReduce で Table Store データを処理する方法について説明します。

**MR を Table Store に接続**

- データテーブルの準備

  pet という名前のテーブルを作成し、名前フィールドを主キーに設定します。

| 名前 | 飼い主 | 種 | 性別 | 誕生日 | 命日 |
|---|---|---|---|---|---|
| ふわふわ | ハロルド | ネコ | 雌 | 1993-02-04 | |

| 名前 | 飼い主 | 種 | 性別 | 誕生日 | 命日 |
|---|---|---|---|---|---|
| クローズ | グウェン | ネコ | 雄 | 1994-03-17 | |
| バフィ | ハロルド | 犬 | 雌 | 1989-05-13 | |
| ファング | ベニー | 犬 | 雄 | 1990-08-27 | |
| クッパ | ダイアン | 犬 | 雄 | 1979-08-31 | 1995-07-29 |
| チャーピー | グウェン | 鳥 | 雌 | 1998-09-11 | |
| ウィスラー | グウェン | 鳥 | | 1997-12-09 | |
| スリム | ベニー | ヘビ | 雌 | 1996-04-29 | |
| パフボール | ダイアン | ハムスター | 雌 | 1999-03-30 | |

- 以下に、MR で Table Store データを処理するコーディング例を記述します。

```
public class RowCounter {
    public static class RowCounterMapper
      extends Mapper<PrimaryKeyWritable, RowWritable, Text, LongWritable> {
        private final static Text agg = new Text("TOTAL");
        private final static LongWritable one = new LongWritable(1);
        @Override public void map(PrimaryKeyWritable key, RowWritable value,
            Context context) throws IOException, InterruptedException {
            context.write(agg, one);
        }
    }
    public static class IntSumReducer
      extends Reducer<Text,LongWritable,Text,LongWritable> {
        @Override public void reduce(Text key, Iterable<LongWritable> values,
            Context context) throws IOException, InterruptedException {
            long sum = 0;
            for (LongWritable val : values) {
                sum += val.get();
            }
            context.write(key, new LongWritable(sum));
        }
    }
    private static RangeRowQueryCriteria fetchCriteria() {
        RangeRowQueryCriteria res = new RangeRowQueryCriteria("pet");
        res.setMaxVersions(1);
        List<PrimaryKeyColumn> lower = new ArrayList<PrimaryKeyColumn>();
        List<PrimaryKeyColumn> upper = new ArrayList<PrimaryKeyColumn>();
        lower.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MIN));
        upper.add(new PrimaryKeyColumn("name", PrimaryKeyValue.INF_MAX));
        res.setInclusiveStartPrimaryKey(new PrimaryKey(lower));
        res.setExclusiveEndPrimaryKey(new PrimaryKey(upper));
        return res;
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        job.setJarByClass(RowCounter.class);
        job.setMapperClass(RowCounterMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        job.setInputFormatClass(TableStoreInputFormat.class);
        TableStore.setCredential(job, accessKeyId, accessKeySecret, securityToken);
```

```
        TableStore.setEndpoint(job, endpoint, instance);
        TableStoreInputFormat.addCriteria(job, fetchCriteria());
        FileOutputFormat.setOutputPath(job, new Path(outputPath));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

- 以下に、MR で Table Store にデータを書き込むコーディング例を記述します。

```
public static class OwnerMapper
  extends Mapper<PrimaryKeyWritable, RowWritable, Text, MapWritable> {
    @Override public void map(PrimaryKeyWritable key, RowWritable row,
        Context context) throws IOException, InterruptedException {
        PrimaryKeyColumn pet = key.getPrimaryKey().getPrimaryKeyColumn("name");
        Column owner = row.getRow().getLatestColumn("owner");
        Column species = row.getRow().getLatestColumn("species");
        MapWritable m = new MapWritable();
        m.put(new Text(pet.getValue().asString()),
            new Text(species.getValue().asString()));
        context.write(new Text(owner.getValue().asString()), m);
    }
}
public static class IntoTableReducer
  extends Reducer<Text,MapWritable,Text,BatchWriteWritable> {
    @Override public void reduce(Text owner, Iterable<MapWritable> pets,
        Context context) throws IOException, InterruptedException {
        List<PrimaryKeyColumn> pkeyCols = new ArrayList<PrimaryKeyColumn>();
        pkeyCols.add(new PrimaryKeyColumn("owner",
                PrimaryKeyValue.fromString(owner.toString())));
        PrimaryKey pkey = new PrimaryKey(pkeyCols);
        List<Column> attrs = new ArrayList<Column>();
        for(MapWritable petMap: pets) {
            for(Map.Entry<Writable, Writable> pet: petMap.entrySet()) {
                Text name = (Text) pet.getKey();
                Text species = (Text) pet.getValue();
                attrs.add(new Column(name.toString(),
                        ColumnValue.fromString(species.toString())));
            }
        }
        RowPutChange putRow = new RowPutChange(outputTable, pkey)
            .addColumns(attrs);
        BatchWriteWritable batch = new BatchWriteWritable();
        batch.addRowChange(putRow);
        context.write(owner, batch);
    }
}
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, TableStoreOutputFormatExample.class.getName());
    job.setMapperClass(OwnerMapper.class);
    job.setReducerClass(IntoTableReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(MapWritable.class);
    job.setInputFormatClass(TableStoreInputFormat.class);
    job.setOutputFormatClass(TableStoreOutputFormat.class);
    TableStore.setCredential(job, accessKeyId, accessKeySecret, securityToken);
    TableStore.setEndpoint(job, endpoint, instance);
    TableStoreInputFormat.addCriteria(job, ...) ;
    TableStoreOutputFormat.setOutputTable(job, outputTable);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

```
}
```

## 付録

完全なサンプルコードについては、以下をご参照ください。

- MR は Table Store からデータを読み取ります。
- MR は Table Store にデータを書き込みます。

# 6 HBase クラスターを作成して HBase ストレージサービスを使用

このページでは、HBase クラスターを作成および設定し、HBase ストレージサービスを使用する方法について説明します。

HBase をより有効に活用するため、クラスターの作成時に以下の設定を使用するよう推奨します。

- パブリックネットワークからのアクセスを有効にします。
- HBase にアクセスするアプリケーションのサーバーが存在するゾーンを選択します。 ゾーンランダムに割り当てないようにしてください。
- マスターノードとスレーブノードを含む 4 つ以上のハードウェアノードを選択します。 E-MapReduce は、これらのノードに NameNode、DataNode、JournalNode、HMaster、RegionServer、ZooKeeper の各ロールを作成します。
- サーバーは 4 コア 16 GB および 8 コア 32 GB を選択します。 設定が低いと、HBase クラスターの安定性が損なわれる可能性があります。
- 費用対効果のため SSD ディスクタイプを選択してください。 使用頻度は低いが大量のストレージスペースが必要になるサービスのディスクの種類として、ベーシックディスクを選択できます。
- 必要に応じてデータ容量を設定します。
- HBase クラスターはメモリー拡張をサポートします。

**HBase の設定**

HBase クラスターの作成時には [クラスターの作成] ページの ソフトウェア設定機能を使用できます。 特定のシナリオに応じて、HBase のデフォルトパラメーターを最適化および変更できます。 以下の例をご参照ください。

```
{
  "configurations": [
    {
      "classification": "hbase-site",
      "properties": {
        "hbase.hregion.memstore.flush.size": "268435456",
        "hbase.regionserver.global.memstore.size": "0.5",
        "hbase.regionserver.global.memstore.lowerLimit": "0.6"
      }
    }
  ]
```

```
}
```

HBase クラスターのいくつかのデフォルト設定は以下のとおりです。

| キー | 値 |
| --- | --- |
| zookeeper.session.timeout | 180000 |
| hbase.regionserver.global.memstore.size | 0.35 |
| hbase.regionserver.global.memstore.lowerLimit | 0.3 |
| hbase.hregion.memstore.flush.size | 128 MB |

**HBase へのアクセス**

**:**

- ネットワークパフォーマンスを考慮すると、同じゾーン内の ECS インスタンスから E-MapReduce によって作成された HBase クラスターへのアクセスリクエストを開始するよう推奨します。
- HBase クラスターにアクセスする ECS インスタンスは、HBase クラスターと同じセキュリティグループに属している必要があります。 違うグループの場合、アクセスは失敗します。したがって、E-MapReduce で Hadoop、Spark、または Hive クラスターを作成し、作成したクラスターが HBase にアクセスする必要がある場合は、必ず HBase クラスターと同じセキュリティグループを選択してください。

E-MapReduce コンソールに HBase クラスターを作成したら、HBase ストレージサービスを使用できます。 手順は以下のとおりです。

1. マスター IP アドレスと ZooKeeper クラスターのアドレスを取得します。 次の図のとおり、E-MapReduce コンソールの [クラスターの概要] ページで、マスターノードの IP アドレスと ZooKeeper のアクセスアドレス (イントラネットの IP アドレス) を閲覧できます。

   パブリック IP アドレスを有効にしたマスターノードの詳細は、「マスターノードにログインする方法 (How to log on to a master node)」をご参照のうえ、HMaster の WEB UI ( localhost:16010) をブラウズしてください。
2. HBase Shell を使用して SSH 経由でクラスターのマスターノードに接続できます。 SSH を使用してクラスターのマスターノードにアクセスできます。 HDFS アカウントに切り替えて HBase Shell を使用してクラスターにアクセスします。 (HBase シェルの詳細は、 Apache HBase Web サイトをご参照ください)。

```
[root@emr-header-1 ~]# su hdfs
[hadoop@emr-header-1 root]$ hbase shell
HBase Shell; enter 'help<RETURN>' for a list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.1.1, r374488, Fri Aug 21 09:18:22 CST 2015
hbase(main):001:0>
```

3. (同じセキュリティグループ内の) 別の ECS ノードから HBase Shell を使用してクラスターにアクセスします。 Apache HBase Web サイトから HBase-1.x リソースをダウンロードします (ダウンロードリンク)。 リソースを抽出し、conf/hbase-site.xml を変更してから、以下のとおりクラスターの ZooKeeper アドレスを追加します。

```
<configuration>
    <property>
       <name>hbase.zookeeper.quorum</name>
       <value>$ZK_IP1,$ZK_IP2,$ZK_IP3</value>
    </property>
```

```
</configuration>
```

その後、 **bin/hbase shell** コマンドを使用してクラスターにアクセスできます。

E-MapReduce を使用して ECS インスタンスを作成すれば、HBase-1.x リソースをダウンロードせず hbase-site.xml ファイルを変更するだけで済みます。

3.2.0 より上のバージョンの設定ファイルは /etc/ecm/hbase-conf/hbase-site.xml にあります。

3.2.0 より前のバージョンの設定ファイルは /etc/emr/hbase-conf/hbase-site.xml にあります。

4. API を使用して HBase クラスターにアクセスするには、以下の Maven 依存関係を追加します。

```
<groupId>org.apache.hbase</groupId>
<artifactId>hbase-client</artifactId>
<version>1.1.1</version>
```

クラスターに接続するための正しい ZooKeeper アドレスを設定します。

```
Configuration config = HBaseConfiguration.create();
config.set(HConstants.ZOOKEEPER_QUORUM,"$ZK_IP1,$ZK_IP2,$ZK_IP3");
Connection connection = ConnectionFactory.createConnection(config);
try {
    Table table = connection.getTable(TableName.valueOf("myLittleHBaseTable"));
    try {
        //Do table operation
    }finally {
        if (table ! = null) table.close();
    }
} finally {
    connection.close();
}
```

Apache HBase 開発の詳細は、Apache HBase Web サイトをご参照ください。

## コーディング例

- 前提条件

  Hbase クラスターにアクセスする ECS インスタンスは、HBase クラスターと同じセキュリティグループに属している必要があります。

- Spark への HBase アクセス許可

  spark-hbase-connector をご参照ください。

- Hadoop への HBase アクセス許可

  「HBase MapReduce の例 (HBase MapReduce Examples)」をご参照ください。

- Hive への HBase アクセス許可

  HBase クラスターにアクセスできるのは E-MapReduce 1.2.0 以降のバージョンのクラスターで実行される Hive だけです。 以下のステップを実行します。

  1. Hive クラスターにログインし、以下の行を追加してホストを変更します。

     ```
     $zk_ip emr-cluster //$zk_ip is the IP address of the ZooKeeper node in the HBase cluster.
     ```

  2. Hive の操作方法の詳細は、「Hive HBase の統合 (Hive HBase Integration)」をご参照ください。

# 7 HBase のバックアップ

E-MapReduce で作成された HBase クラスターでは、HBase に統合されたスナップショット機能を使用して HBase テーブルのバックアップを取り、そのバックアップを OSS にエクスポートできます。

コーディング例：

1. HBase クラスターの作成

   詳細は、「クラスターの作成 (Create clusters)」をご参照ください。

2. テーブルの作成

```
>create 'test','cf'
```

3. Add data

```
> put 'test','a','cf:c1',1
> put 'test','a','cf:c2',2
> put 'test','b','cf:c1',3
> put 'test','b','cf:c2',4
> put 'test','c','cf:c1',5
> put 'test','c','cf:c2',6
```

4. スナップショットの作成

```
hbase snapshot create -n test_snapshot -t test
```

   スナップショットの一覧表示

```
>list_snapshots
SNAPSHOT             TABLE + CREATION TIME
 test_snapshot        test (Sun Sep 04 20:31:00 +0800 2016)
1 row(s) in 0.2080 seconds
```

5. スナップショットを OSS にエクスポート

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot test_snapshot
 -copy-to oss://$accessKeyId:$accessKeySecret@$bucket.oss-cn-hangzhou-internal.
aliyuncs.com/hbase/snapshot/test
```

   注：

   内部エンドポイントを使用して OSS にアクセス

6. 別の HBase クラスターを作成

7. OSS からスナップショットをエクスポート

```
hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot test_snaps
hot -copy-from oss://$accessKeyId:$accessKeySecret@$bucket.oss-cn-hangzhou-
internal.aliyuncs.com/hbase/snapshot/test -copy-to /hbase/
```

8. スナップショットからデータを復元

```
>restore _snapshot 'test_snapshot'
```

```
>scan 'test'
ROW                COLUMN+CELL
 a                column=cf:c1, timestamp=1472992081375, value=1
 a                column=cf:c2, timestamp=1472992090434, value=2
 b                column=cf:c1, timestamp=1472992104339, value=3
 b                column=cf:c2, timestamp=1472992099611, value=4
 c                column=cf:c1, timestamp=1472992112657, value=5
 c                column=cf:c2, timestamp=1472992118964, value=6
3 row(s) in 0.0540 seconds
```

9. スナップショットから新しいテーブルを作成

```
>clone_snapshot 'test_snapshot','test_2'
```

```
>scan 'test_2'
ROW                COLUMN+CELL
 a                column=cf:c1, timestamp=1472992081375, value=1
 a                column=cf:c2, timestamp=1472992090434, value=2
 b                column=cf:c1, timestamp=1472992104339, value=3
 b                column=cf:c2, timestamp=1472992099611, value=4
 c                column=cf:c1, timestamp=1472992112657, value=5
 c                column=cf:c2, timestamp=1472992118964, value=6
3 row(s) in 0.0540 seconds
```

# 8 E-MapReduce クラスター運用ガイド

このページでは、E-MapReduce クラスターの複数の操作方法について説明します。コンポーネントに対する個別の操作の実行がより簡単になります。

:

後段のクラスターバージョン (3.2 以降) では、コンポーネントのすべての操作はコンソールの [クラスター管理] ページから実行できます。 コンポーネントの管理は Web ページを使用して行うよう推奨します。

**一般的な環境変数**

**env** コマンドを入力して、以下の例に類似した環境変数の設定を確認します。 実際の設定は異なる場合があります。 この例はあくまで参考用です。

```
JAVA_HOME=/usr/lib/jvm/java
HADOOP_HOME=/usr/lib/hadoop-current
HADOOP_CLASSPATH=/usr/lib/hbase-current/lib/*:/usr/lib/tez-current/*:/usr/lib/tez-current/lib/*:/etc/emr/tez-conf:/usr/lib/hbase-current/lib/*:/usr/lib/tez-current/*:/usr/lib/tez-current/lib/*:/etc/emr/tez-conf:/opt/apps/extra-jars/*:/opt/apps/extra-jars/*
HADOOP_CONF_DIR=/etc/emr/hadoop-conf
SPARK_HOME=/usr/lib/spark-current
SPARK_CONF_DIR=/etc/emr/spark-conf
HBASE_HOME=/usr/lib/hbase-current
HBASE_CONF_DIR=/etc/emr/hbase-conf
HIVE_HOME=/usr/lib/hive-current
HIVE_CONF_DIR=/etc/emr/hive-conf
PIG_HOME=/usr/lib/pig-current
PIG_CONF_DIR=/etc/emr/pig-conf
TEZ_HOME=/usr/lib/tez-current
TEZ_CONF_DIR=/etc/emr/tez-conf
ZEPPELIN_HOME=/usr/lib/zeppelin-current
ZEPPELIN_CONF_DIR=/etc/emr/zeppelin-conf
HUE_HOME=/usr/lib/hue-current
HUE_CONF_DIR=/etc/emr/hue-conf
PRESTO_HOME=/usr/lib/presto-current
PRESTO_CONF_DIR=/etc/emr/presot-conf
```

**Web UI を使用してコンポーネントを起動および停止**

E-MapReduce サービスの Web UI を使用して、指定された ECS インスタンスで実行されているコンポーネントを起動、停止、および再起動できます。 HDFS サービスなどさまざまなコンポーネントに対する操作は類似しています。 emr-worker-1 インスタンスの DataNode コンポーネントを起動、停止、および再起動するには、以下のステップを実行します。

1. クラスター一覧 のページで、操作を実行するクラスターの "アクション" 列の **[管理]**をクリックします。

2. **[クラスターとサービス]** ページで、 **[HDFS]** リンクをクリックし [HDFS] サービスページに移動します。
3. **[コンポーネントトポロジ]** タブをクリックして、クラスター内のすべてのインスタンスで実行されているコンポーネントの一覧を表示します。
4. emr-worker-1 インスタンス上で実行される DataNode コンポーネントで "アクション" 列の **[開始]** をクリックします。 ダイアログボックスにコミットレコードを入力し、**[OK]** をクリックします。 10 秒後にページを最新の情報に更新します。 DataNode の **"コンポーネントステータス"** 列の値が **"STOPPED"** から **"STARTED"** に切り替わります。

Status　Component Topology　Configuration　Configuration Change History

Hostname :　Host ID :　Search

| Component Name | Components Stat... | Service Name | Host ID | Hostname | Host Role | IP | Need restart | Operation |
|---|---|---|---|---|---|---|---|---|
| HDFS Client | INSTALLED | HDFS | [illegible] | emr-worker-2 | CORE | [illegible] | No | |
| DataNode | STARTED | HDFS | [illegible] | emr-worker-2 | CORE | [illegible] | No | Restart \| Stop |
| NameNode | STARTED | HDFS | [illegible] | emr-header-1 | MASTER | [illegible] | No | Restart \| Stop |
| KMS | STOPPED | HDFS | [illegible] | emr-header-1 | MASTER | [illegible] | No | Start |

5. コンポーネントが起動したら、**"アクション"** 列で **[再起動]** をクリックします。 ダイアログボックスにコミットレコードを入力し、**[OK]** をクリックします。

   40 秒後にページを最新の情報に更新します。 DataNode の "コンポーネントステータス" 列の値が **"STOPPED"** から **"STARTED"** に切り替わります。
6. **"アクション"** 列の **[停止]** をクリックします。 ダイアログボックスにコミットレコードを入力し、**[OK]** をクリックします。

   10 秒後にページを最新の情報に更新します。 DataNode の **"コンポーネントステータス"** 列の値が **"STOPPED"** から **"STARTED"** に切り替わります。

### Web UI を使用してコンポーネントに対する一括操作を実行

指定した単一の ECS インスタンスではなく、複数の ECS インスタンスのコンポーネントに対して一括操作を実行できます。 HDFS サービスを例とします。 すべてのインスタンスの DataNode コンポーネントを再起動するには、以下のステップを実行します。

1. クラスター一覧 ページで、操作を実行するクラスターの **"アクション"** 列の **[管理]** をクリックします。
2. クラスターとサービス ページで、サービス一覧の **[HDFS]** の **[アクション]** をクリックします。

3. [アクション]ドロップダウンリストから **[RESTART DataNode]** を選択します。 ダイアログボックスにコミットレコードを入力し、**[OK]** をクリックします。

   **[HDFS]** をクリックして **[コンポーネントトポロジ]** タブをクリックします。 [コンポーネントトポロジ] タブページで各プロセスのステータスを閲覧します。

   **:**

   クラスターのローリング再起動を実行後に手動でノードを再起動すると、コンソールからエラーが報告されます。



**CLI を使用してコンポーネントを起動および停止**

- YARN

  Account: hadoop

  - ResourceManager (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start resourcemanager
```

```
//Stops a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop resourcemanager
```

- NodeManager (コアコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start nodemanager
//Stops a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop nodemanager
```

- JobHistoryServer (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh start historyserver
//Stops a component.
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh stop historyserver
```

- JobHistoryServer (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh start historyserver
//Stops a component.
/usr/lib/hadoop-current/sbin/mr-jobhistory-daemon.sh stop historyserver
```

- WebProxyServer (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh start proxyserver
//Stops a component.
/usr/lib/hadoop-current/sbin/yarn-daemon.sh stop proxyserver
```

• HDFS

  Account: hdfs

  - NameNode (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh start namenode
//Stops a component.
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh stop namenode
```

  - DataNode (コアコンポーネント)

```
//Starts a component.
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh start datanode
//Stops a component.
/usr/lib/hadoop-current/sbin/hadoop-daemon.sh stop datanode
```

• Hive

  Account: hadoop

  - MetaStore (マスターコンポーネント)

```
//Starts a component. 最大メモリーを大きくするには、HADOOP_HEAPSIZE 環境変数をより大きな値に設定できます。
```

```
HADOOP_HEAPSIZE=512 /usr/lib/hive-current/bin/hive --service metastore >/var/
log/hive/metastore.log 2>&1 &
```

- HiveServer2 (マスターコンポーネント)

```
//Starts a component.
HADOOP_HEAPSIZE=512 /usr/lib/hive-current/bin/hive --service hiveserver2 >/var/
log/hive/hiveserver2.log 2>&1 &
```

- HBase

  運用アカウント : hdfs

  注 : 以下の例は HBase サービスをベースとしています。 対応する設定なしでコンポーネントを起動するとエラーが発生します。

  - HMaster (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh start master
//Restarts a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh restart master
//Stops a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh stop master
```

  - HRegionServer (コアコンポーネント)

```
//Starts a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh start regionserver
//Restarts a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh restart regionserver
//Stops a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh stop regionserver
```

  - ThriftServer (マスターコンポーネント)

```
//Starts a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh start thrift -p 9099 >/var/log/hive/
thriftserver.log 2>&1 &
//Stops a component.
/usr/lib/hbase-current/bin/hbase-daemon.sh stop thrift
```

- Hue

  アカウント : hadoop

```
//Starts a component.
su -l root -c "${HUE_HOME}/build/env/bin/supervisor >/dev/null 2>&1 &"
//Stops a component.
ps aux | grep hue     //Lists information about the currently running Hue processes.
```

```
kill -9 huepid       //Kills the Hue process by its PID.
```

- Zeppelin

  アカウント : hadoop

  ```
  //Starts a component. 最大メモリーを大きくするには、HADOOP_HEAPSIZE 環境変数をより大きな値に設定できます。
  su -l root -c "ZEPPELIN_MEM=\"-Xmx512m -Xms512m\" ${ZEPPELIN_HOME}/bin/zeppelin-daemon.sh start"
  //Stops a component.
  su -l root -c "${ZEPPELIN_HOME}/bin/zeppelin-daemon.sh stop"
  ```

- Presto

  Account: hadoop

  - PrestoServer (マスターコンポーネント)

    ```
    //Starts a component.
    /usr/lib/presto-current/bin/launcher --config=/usr/lib/presto-current/etc/coordinator-config.properties start
    //Stops a component.
    /usr/lib/presto-current/bin/launcher --config=/usr/lib/presto-current/etc/coordinator-config.properties stop
    ```

  - (コアコンポーネント)

    ```
    //Starts a component.
    /usr/lib/presto-current/bin/launcher --config=/usr/lib/presto-current/etc/worker-config.properties start
    //Stops a component.
    /usr/lib/presto-current/bin/launcher --config=/usr/lib/presto-current/etc/worker-config.properties stop
    ```

### CLI を使用してコンポーネントに対する一括操作を実行

すべてのワーカー (コア) ノードに一括操作を実行するスクリプトコマンドを記述します。 EMR クラスターでは、SSH を使用してマスターノードから hadoop アカウントと hdfs アカウントで実行されるすべてのワーカーノードにアクセスできます。

たとえば、以下のコマンドを実行して、すべてのワーカーノードの NodeManager コンポーネントを停止できます。 ワーカーノードの数を 10 と仮定します。

```
for i in `seq 1 10`;do ssh emr-worker-$i /usr/lib/hadoop-current/sbin/yarn-daemon.sh stop nodemanager;done
```