Alibaba Cloud E-MapReduce

ベストプラクティス

Document Version20200425

目次

1 クラスター管理

1.1 YARN による cgroups を使用した CPU 使用率の制御

cgroups (control groups) は、プロセスのコレクションのリソース使用量 (CPU、メモリ、ディスク I/O など) を制御、制限、分離する Linux カーネル機能です。

概要

YARN により cgroups 機能が統合されます。 この機能を使用すると、NodeManager が管理する各コンテナーまたはすべてのコンテナーの CPU 使用率を制御できます。

YARN での cgroups 機能の有効化

> 注 :
> デフォルトでは、E-MapReduce クラスターの YARN コンポーネントの cgroups 機能は無効になっています。 必要に応じて機能を有効化できます。

1. Alibaba Cloud E-MapReduce コンソール にログインします。
2. E-MapReduce クラスターの **[YARN]** ページに移動します。
   a. E-MapReduce ホームページで、**[**クラスター管理**]** タブをクリックします。

      [クラスター管理**]** タブが表示されます。
   b. 対象のクラスターを探し、そのクラスター ID をクリックします。
   c. 表示されるページで、左側のナビゲーションペインの **[**クラスターサービス**] > [YARN]** を選択します。
3. cgroups 機能を有効にします。
   a. YARN ページで、右上の **[**操作**] > [EnabledCGroups]** を選択します。

      [クラスターのアクティビティ**]** ダイアログボックスが表示されます。
   b. 必要に応じて関連パラメーターを設定します。
   c. **[OK]** をクリックします。

      [確認**]** ダイアログボックスが表示されます。
   d. **[OK]** をクリックします。

4. クラスターを再起動します。

   a. **[**クラスター管理**]** タブをクリックします。

   b. 対象のクラスターを探し、[操作] 列の **[**詳細**]** > **[**再起動**]** の順に選択します。

CPU 使用率を制御するためのパラメーターの変更

E-MapReduce クラスターの cgroups 機能を有効にした後、YARN コンポーネントの関連パラメーターを設定して、CPU 使用率を制御できます。

1. Alibaba Cloud E-MapReduce コンソール にログインします。
2. E-MapReduce クラスターの **[YARN]** ページに移動します。

   a. E-MapReduce ホームページで、**[**クラスター管理**]** タブをクリックします。

   [クラスター管理**]** タブが表示されます。

   b. 対象のクラスターを探し、そのクラスター ID をクリックします。

   c. 表示されるページで、左側のナビゲーションペインの **[**クラスターサービス**]** > **[YARN]** を選択します。

3. 関連パラメーターを変更します。

   a. YARN ページで、**[**設定**]** タブをクリックします。

   b. 下表に示すようにパラメーターを変更します。

項目	説明
yarn.nodemanager.resource.percentage-physical-cpu-limit	NodeManager が管理するすべてのコンテナーの CPU 使用率の合計を制限する割合。デフォルト値：100。 注： 理論的には、NodeManager が管理するすべてのコンテナーの CPU 使用率の合計は、**yarn.nodemanager.resource.percentage-physical-cpu-limit** パラメーターで指定される上限を越えません。

項目	説明
yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage	各コンテナーの CPU 使用率にハードリミットを設定するかどうかを指定します。 ハードリミットを設定していない場合、コンテナーでは、割り当てられたリソースに加えてアイドル CPU リソースを使用できます。 デフォルト値は false で、コンテナーがアイドル CPU リソースを使用できることを示しています。

すべてのコンテナーの CPU 使用率合計の制御

yarn.nodemanager.resource.percentage-physical-cpu-limit パラメーターを変更することで、NodeManager が管理するすべてのコンテナーの CPU 使用率の合計を制御できます。

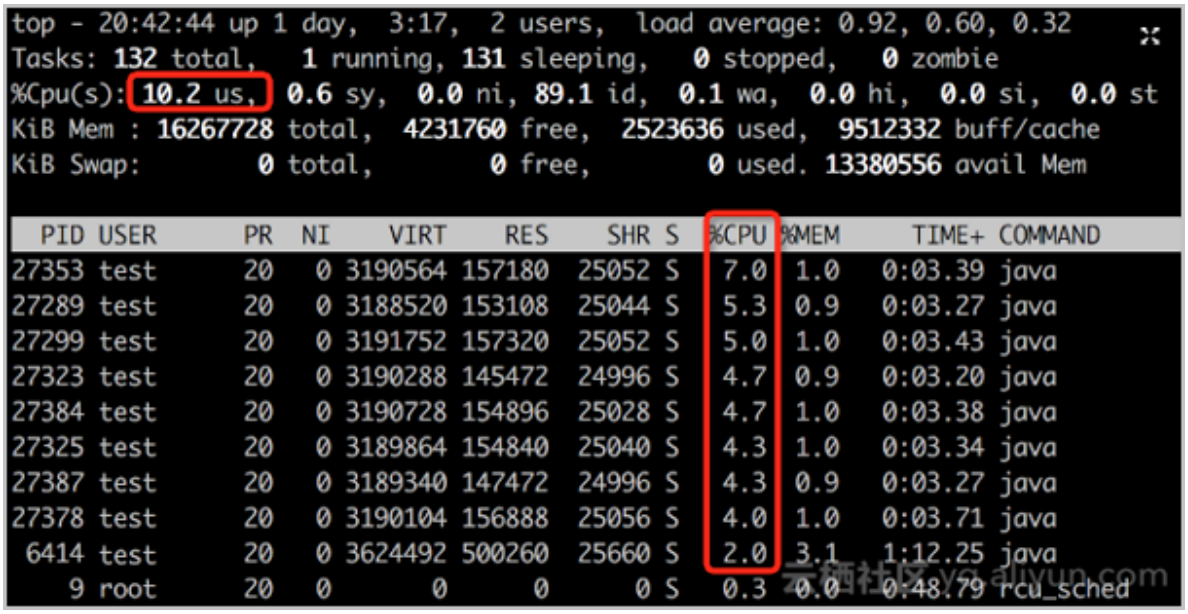
NodeManager をデプロイされた 2 つのノードおよび ResourceManager をデプロイされた 1 つのノードで構成される E-MapReduce クラスターを準備します。 各ノードには 4 コア、64 GB のメモリが搭載されています。 yarn.nodemanager.resource.percentage-physical-cpu-limit パラメーターを設定して、NodeManager でデプロイされたノードの CPU 使用率を制御します。 パラメーターを設定したら、YARN コンポーネントの Hadoop Pi ジョブを実行して CPU 使用率を確認します。 この例では、パラメーターをそれぞれ 10、30、および 50 に設定しています。

 注：
次の図で、%CPU 値は、ユーザー test のプロセスの CPU 使用率がコアで占める割合を示しています。 %Cpu(s) の値は、CPU リソース合計のうち、ユーザー test のすべてのプロセスの CPU 使用率が占める割合を示しています。

- パラメーターを 10 に設定します。

  前の図に示すように、%Cpu(s) は 10.2 で、ユーザー test のすべてのプロセスの %CPU 値の合計です。 CPU 使用率の合計は、次のように計算します：7% + 5.3% + 5% + 4.7% + 4.7% + 4.3% + 4.3% + 4% + 2% = 41.3%。 これは、ユーザー test のすべてのプロセスが 0.413 コアを消費し、ノードの 4 個のコアの約 10% を占めることを表しています。

- パラメーターを 30 に設定します。



前の図に示すように、%Cpu(s) は 32.5 で、ユーザー test のすべてのプロセスの %CPU 値の合計です。 CPU 使用率の合計は、次のように計算します：19% + 18.3% + 18.3% + 17% + 16.7% + 16.3% + 14.7% + 12% = 132.3%。 これは、ユーザー test のすべてのプロセスが 1.323 コアを消費し、ノードの 4 個のコアの約 30% を占めることを表しています。

- パラメーターを 50 に設定します。

前の図に示すように、%Cpu(s) は 48.7 で、ユーザー test のすべてのプロセスの %CPU 値の合計です。 CPU 使用率の合計は、次のように計算します：65.1% + 60.1% + 43.5% + 20.3% + 3.7% + 2% = 194.7%。 これは、ユーザー test のすべてのプロセスが 1.947 コアを消費し、ノードの 4 個のコアの約 50% を占めることを表しています。

各コンテナーの CPU 使用率の制御

理論的には、NodeManager が管理するすべてのコンテナーの CPU 使用率の合計は、**yarn.nodemanager.resource.percentage-physical-cpu-limit** パラメーターで指定される上限を越えません。 NodeManager では、各コンテナーの CPU 使用率を管理、制御するために、共有モードおよび厳密モードを提供しています。 **yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage** パラメーターを設定して、モードを指定できます。

- 共有モードの使用

  共有モードを使用するには、**yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage** パラメーターを false に設定します。 これは、パラメーターのデフォルト

値です。 共有モードでは、コンテナーは、割り当てられたリソースに加えてアイドル CPU リソースを使用できます。

たとえば、**yarn.nodemanager.resource.percentage-physical-cpu-limit** パラメーターを 50 に設定し、**yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage** パラメーターを false に設定します。 クラスターには、NodeManager でデプロイされた 2 つのノードがあり、各ノードには 4 個のコアがあります。 ノードの NodeManager が管理するすべてのコンテナーは、4 コアの最大 50% を消費できます。これは 2 コアと同等です。 各コンテナーに割り当てられる CPU リソースは、次のように計算されます: (1 vcore/8 vcore) × (4 コア × 50%) = 0.25 コア。 共有モードでは、各コンテナーはアイドル CPU リソース (最大 2 コア) を消費できます。

前の図に示すように、ユーザー test のプロセスによって消費される CPU リソースは、0.65 コア、0.61 コア、0.037 コアなど、大きく異なります。 これは、割り当てられた 0.25 コアよりもはるかに多くの CPU リソースが一部のコンテナーで消費されることを示しています。

- 厳格モードの使用

  厳格モードを使用するには、**yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage** パラメーターを true に設定します。 厳密モードでは、アイドル CPU リソースが使用可能であっても、コンテナーでは、割り当てられた CPU リソースのみを消費できます。

  たとえば、**yarn.nodemanager.resource.percentage-physical-cpu-limit** パラメーターを 50 に設定し、**yarn.nodemanager.linux-container-executor.cgroups.strict-resource-usage** パラメーターを true に設定します。

  クラスターには、NodeManager をデプロイされた 2 つのノードがあり、各ノードには 4 個のコアがあります。 各コンテナーに割り当てられる CPU リソースは、次のように計算されます: (1 vcore/8 vcore) × (4 コア × 50%) = 0.25 コア。 前の図に示すように、ユーザー test のプロセスによって消費される CPU リソースはすべて、0.266 コア、0.249 コアなど、割り当てられた 0.25 コアに近くなります。

1.2 さまざまなユーザーのOSSデータの分離

このページでは、リソースアクセス管理 (RAM) を使用して、異なるユーザーの Object Storage Service (OSS) データを分離する方法について説明します。

前提条件

Alibaba Cloud アカウントが作成されます。

背景

E-MapReduce では、RAM を使用してさまざまなユーザーのデータを分離できます。

ステップ 1：RAM コンソールへのログイン

ステップ 2：RAM ユーザーの作成

ステップ 3：認証ポリシーの作成

RAM は、デフォルトの認証ポリシーを提供するだけでなく、認証ポリシーをカスタマイズして認証を柔軟にできます。 ニーズに基づいて、複数の認証ポリシーを作成できます。

1. **[設定モード]** で **[スクリプト]** を選択します。

   [スクリプト] モードでアクセス認証ポリシーを構成する方法の詳細については、「アクセス認証ポリシーの構文と構造」をご参照ください。 次の例では、スクリプトモードで 2 つのアクセス認証ポリシーが作成されます。

テスト環境 (テストバケット)	本番環境 (製品バケット)
{ "Version": "1", "Statement": [{ "Effect": "Allow", "Action": ["oss:ListBuckets"], "Resource": ["acs:oss:*:*:*"] }, { "Effect": "Allow", "Action": ["oss:Listobjects", "oss:GetObject", "oss:PutObject", "oss:DeleteObject"], "Resource": ["acs:oss:*:*:test-bucket", "acs:oss:*:*:test-bucket/*"] }] }	{ "Version": "1", "Statement": [{ "Effect": "Allow", "Action": ["oss:ListBuckets"], "Resource": ["acs:oss:*:*:*"] }, { "Effect": "Allow", "Action": ["oss:Listobjects", "oss:GetObject", "oss:PutObject"], "Resource": ["acs:oss:*:*:prod-bucket", "acs:oss:*:*:prod-bucket/*"] }] }

   上記の認証ポリシーが RAM ユーザーに付与された後、RAM ユーザーは E-MapReduce コンソールで次の制限を受けます。

   - すべてのバケットは、クラスター、ジョブ、および実行計画を作成するための OSS 選択ページに表示されますが、アクセスできるのは認証されたバケットのみです。
   - 認証されたバケットのコンテンツのみにアクセスできます。
   - 認証されたバケットでは、読み書きのみ可能です。 RAM ユーザーが無許可のバケットで読み取りまたは書き込み操作を実行すると、エラーが返されます。

ステップ 4：RAM ユーザーへのアクセス許可の付与

(オプション) ステップ 5：RAM ユーザーに Alibaba Cloud コンソールへのログイン権限の付与

RAM ユーザーの作成時にコンソールログイン権限が RAM ユーザーに付与されていない場合、次のようにして RAM ユーザーに権限を付与できます。

1. **[権限付与]** タブの **[コンソールログイン管理]** セクションで **[ログイン設定の変更]** をクリックします。

ステップ 6：RAM ユーザーでの E-MapReduce コンソールへのログイン

1. RAM ユーザーで Alibaba Cloud コンソール にログインします。
2. コンソールにログインした後、 **[E-MapReduce]** を選択します。

2 データ開発

2.1 E-MapReduce を使用した Kafka クライアントからのメトリクスの収集

このセクションでは、E-MapReduce を使用して Kafka クライアントからメトリクスを収集し、効果的なパフォーマンスモニタリングを行う方法を説明します。

背景

Kafka では、Broker、Consumer、Producer、Stream および Connect のパフォーマンス測定に使用されるメトリクスの集合を提供します。 E-MapReduce では、Ganglia を使用して Kafka Broker の実行ステータスをモニタリングすることにより、この Kafka Brokerのメトリクスが収集されます。 Kafka システムは、2 つのロール、Kafka Broker と 複数の Kafka クライアントで構成されます。 読み書きのパフォーマンスに問題が発生した場合は、Kafka Broker とクライアントの両方で分析する必要があります。 分析の実行には、Kafka クライアントのメトリクスが重要です。

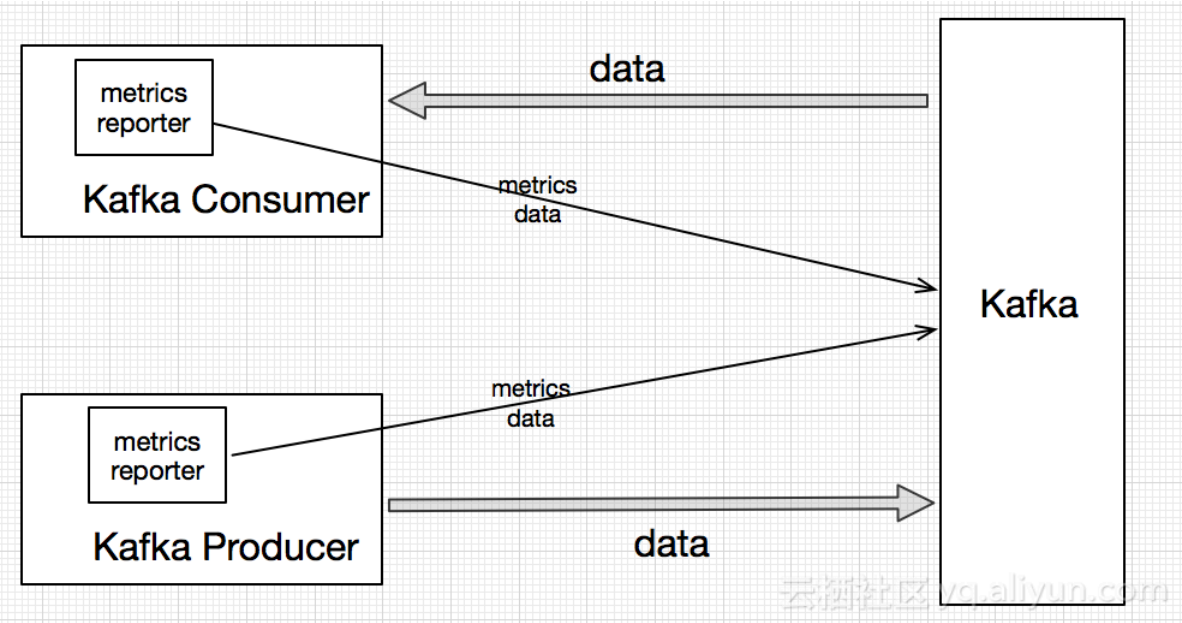
原則

- Kafka のパフォーマンスに関するメトリクスの収集

  Kafka では、複数の外部メトリクスレポーターがサポートされています。 JMX レポーターはデフォルトで Kafka に組み込まれています。 JMX ツールを使用して Kafka のメトリクスを表示します。 カスタマイズメトリクスを収集するため、**org.apache.kafka.common.metrics.MetricsReporter** など独自のメトリクスレポーターを実装します。
- メトリクスの格納

  Kafka メトリクスをカスタマイズします。 さらに、カスタマイズしたメトリクスを後で使用および分析するためには、これらのメトリクスを保持するためのデータストアが必要です。 Kafka 自体がデータストアであるため、サードパーティのデータストアを使用せずに Kafka に

メトリクスを格納できます。 また、Kafka は他のサービスと容易に統合できます。 クライアントからメトリクスを収集します (以下の図を参照)。



E-MapReduce では、emr-kafka-client-metrics のサンプルが提供されます。 source code のリンクからソースコードをダウンロードします。

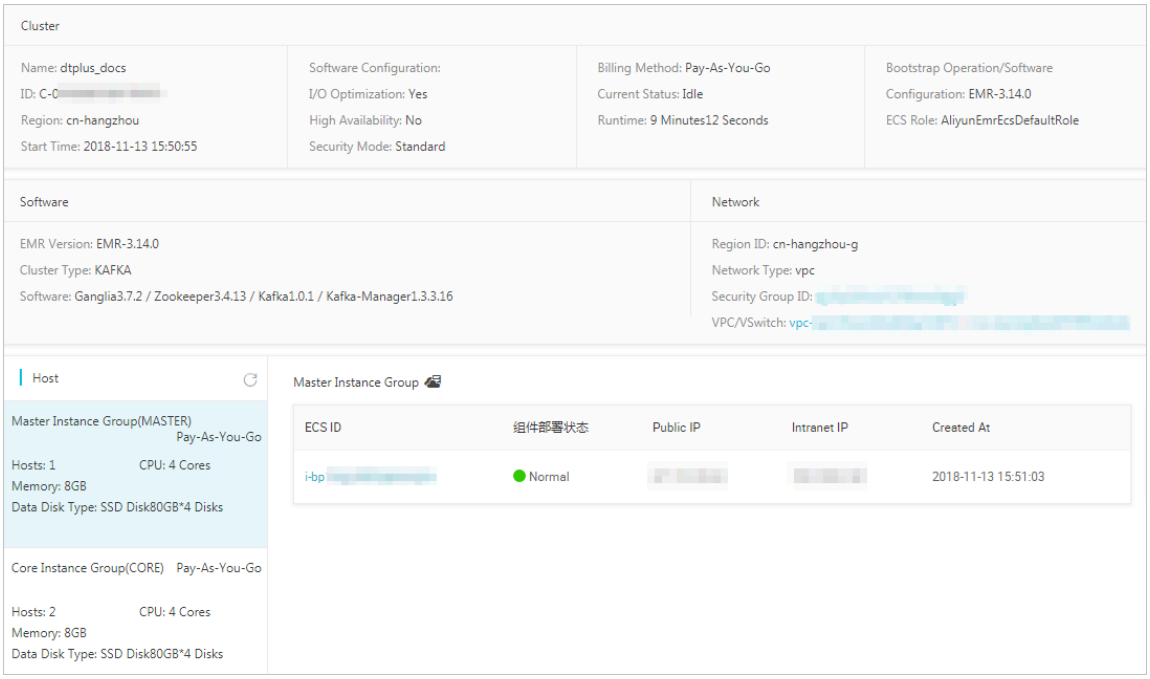
前提条件

- 制限事項
  - サポートされているのは Java だけです。
  - Kafka 0.10 以降のクライアントのみサポートしています。
- E-MapReduce は Maven で jar パッケージを発行しており、コードをご自身でコンパイルする必要はありません。 最新バージョンは download link からダウンロードします。

- このセクションでは、E-MapReduce を使用して自動的に Kafka クラスターを作成します。 詳細は、「クラスターの作成 (Create a cluster)」をご参照ください。

  以下の E-MapReduce および Kafka のバージョンを使用します。

  - EMR バージョン: EMR-3.12.1
  - クラスタータイプ : Kafka
  - ソフトウェア: Kafka-Manager (1.3.3.16)、Kafka (2.11-1.0.1)、ZooKeeper (3.4.12)、および Ganglia (3.7.2)
  - この Kafka クラスターのネットワークタイプは、中国 (杭州) リージョンの VPC です。 マスターインスタンスグループは、パブリック IP と内部ネットワーク IP で設定されています。 詳細は以下の図をご参照ください。



- メトリクスの設定

メトリクス	説明
metric.reporters	メトリクスレポーターのサンプル : org.apache.kafka.clients.reporter.EMRClientMetricsReporter
emr.metrics.reporter.bootstrap.servers	Kafka クラスターのブートストラップサーバーを格納するメトリクスです。
emr.metrics.reporter.zookeeper.connect	Kafka クラスターの Zookeeper アドレスを格納するメトリクスです。

- メトリクスの読み込み
  - jar パッケージ emr-kafka-client-metrics をクライアントに配置します。 jar パッケージのパスをクライアントサイドアプリケーションのクラスパスに追加します。
  - クライアントサイドアプリケーションの jar パッケージに emr-kafka-client-metrics の依存関係をインストールします。

操作手順

1. 最新の emr-kafka-client-metrics パッケージをダウンロードします。

```
wget http://central.maven.org/maven2/com/aliyun/emr/emr-kafka-client-metrics/1.4.3/emr-kafka-client-metrics-1.4.3.jar
```

2. テストトピックを作成します。

```
kafka-topics.sh --zookeeper emr-header-1:2181/kafka-1.0.1 --partitions 10 --replication-factor 2 --topic test-metrics  --create
```

3. emr-kafka-client-metrics パッケージを Kafka クライアントの lib ディレクトリにコピーします。

```
cp emr-kafka-client-metrics-1.4.3.jar /usr/lib/kafka-current/libs/
```

4. テストトピックにデータを書き込みます。 Kafka Producer の設定をローカルの client.conf ファイルに書き込みます。

```
## client.conf:
metric.reporters=org.apache.kafka.clients.reporter.EMRClientMetricsReporter
emr.metrics.reporter.bootstrap.servers=emr-worker-1:9092
emr.metrics.reporter.zookeeper.connect=emr-header-1:2181/kafka-1.0.1
bootstrap.servers=emr-worker-1:9092
## Commnad:
kafka-producer-perf-test.sh --topic test-metrics --throughput 1000 --num-records 100000
--record-size 1024 --producer.config client.conf
```

5. クライアントから現在のメトリクスを閲覧します。 デフォルトのメトリクストピックは _emr-client-metrics です。

```
Kafka-console-consumer.sh -- Topic _ emr-client-metrics -- Bootstrap-server emr-worker-1: 9092
--from-beginning
```

以下のメッセージが返されます。

```
{prefix=kafka.producer, client.ip=192.168.xxx.xxx, client.process=25536@emr-header-1.cluster-xxxx,
attribute=request-rate, value=894.4685104965012, timestamp=1533805225045, group=producer-metrics,
```

```
tag.client-id=producer-1}
```

注：

フィールド名	説明
client.ip	クライアントホストの IP アドレス
client.process	クライアントサイドアプリケーションのプロセス ID
attribute	メトリクスの属性名
value	メトリクスの値
timestamp	メトリクス収集時のタイムスタンプ
tag.xxx	メトリクスのその他のタグ情報

2.2 E-MapReduce を使ったオフラインジョブの処理

このセクションでは、E-MapReduce を使用して OSS からデータを読み取る方法およびデータ収集やデータのクリーンアップなどの一連のオフラインデータ処理操作について説明します。

概要

E-MapReduce クラスターはさまざまなシナリオで使用できます。 E-MapReduce は Hadoop エコシステムと Spark がサポートするすべてのシナリオをサポートします。 E-MapReduce は Hadoop クラスターと Spark クラスターをベースとています。 物理マシンと同様の方法で、E-MapReduce クラスターによってホスティングされている Alibaba Cloud ECS インスタンスを使用できます。

現在使用されている一般的なタイプのビッグデータ処理は、オフラインとオンラインの 2 つです。

- オフラインデータ処理：処理時間の制約なくデータの分析結果を取得する場合に使用します。たとえば、バッチデータ処理シナリオでは、MapReduce、Hive、Pig、および Spark を使用して、OSS からデータを受信し、処理結果を OSS に出力します。
- オンラインデータ処理：リアルタイムストリーミングデータ処理など、処理時間の厳しい要件がある状態でデータの分析結果を取得する場合に使用します。 Spark MLlib、GrapX、および SQL を密接に連携した Spark Streaming は、ストリーミングメッセージの処理に使用できます。

このセクションでは、E-MapReduce でワードカウントというオフラインジョブを実行する方法について説明します。

プロセス

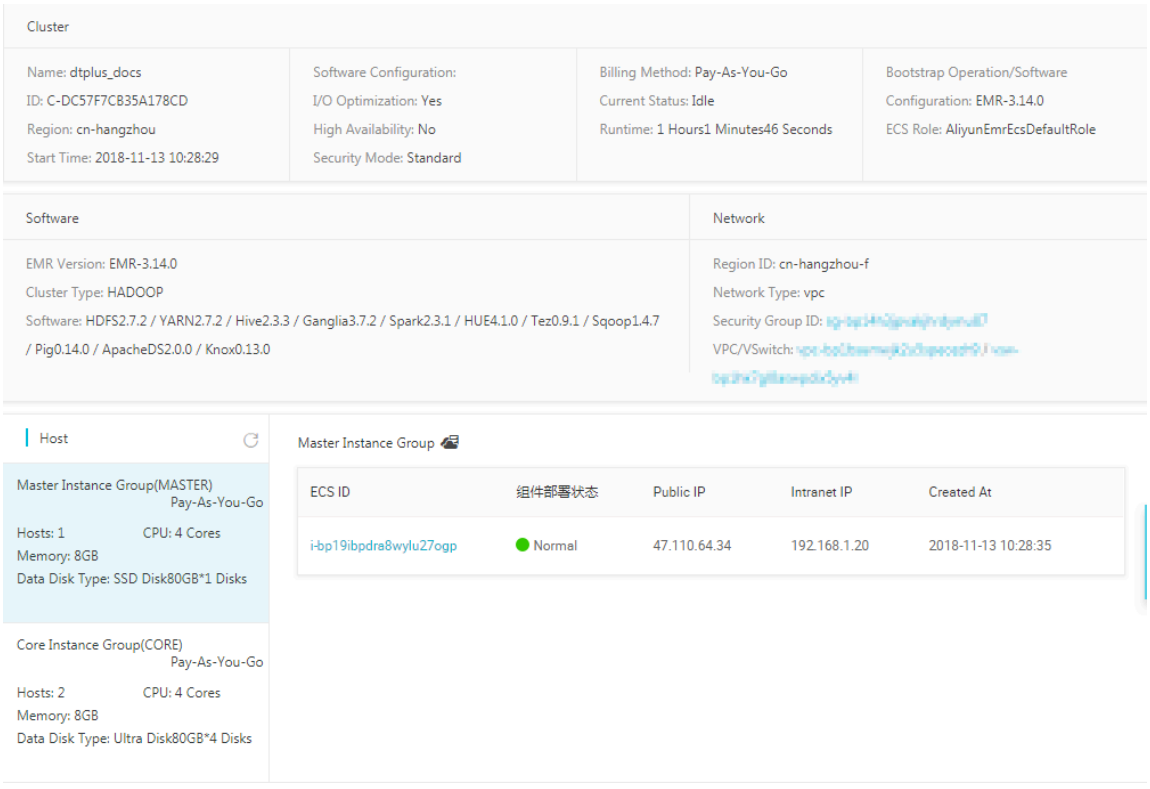
OSS -> EMR -> Hadoop MapReduce

このプロセスには以下の 2 つのステップがあります。

1. データを OSS に保存します。
2. OSS からデータを読み込み、E-MapReduce を使用してデータを分析します。

前提条件

- 以下のステップは、Windows システムで実行されます。 Maven と Java をシステムに確実にインストールし適切に設定します。
- E-MapReduce を使用して Hadoop クラスターを自動的に作成できます。 詳細は、「クラスターの作成 (Create a cluster)」をご参照ください。
  - EMR バージョン：EMR-3.12.1
  - クラスタータイプ：HADOOP
  - ソフトウェア：HDFS2.7.2、YARN2.7.2、Hive2.3.3、Ganglia3.7.2、Spark2.3.1、HUE4.1.0、Zeppelin0.8.0、Tez0.9.1、Sqoop1.4.7、Pig0.14.0、ApacheDS2.0.0、Knox0.13.0
  - この Hadoop クラスターのネットワークタイプは、中国 (杭州) リージョンの VPC です。 マスター インスタンスグループは、パブリック IP と内部ネットワーク IP で設定されています。 高可用性モードは No (非 HA モード) に設定されています。 詳細は次の図をご参照ください。



手順

1. サンプルコードをローカルディスクにダウンロードします。

   システム内の git bash を開き、以下のとおり clone を実行します。

   ```
   git clone https://github.com/aliyun/aliyun-emapreduce-demo.git
   ```

   mvn install コマンドを実行し、コードをコンパイルします。
2. バケットを作成する方法の詳細は、「バケットの作成 (Create a bucket)」をご参照ください。

   > 注：
   > 同じリージョンにバケットと E-MapReduce クラスターを作成する必要があります。
3. jar パッケージとリソースファイルをアップロードします。

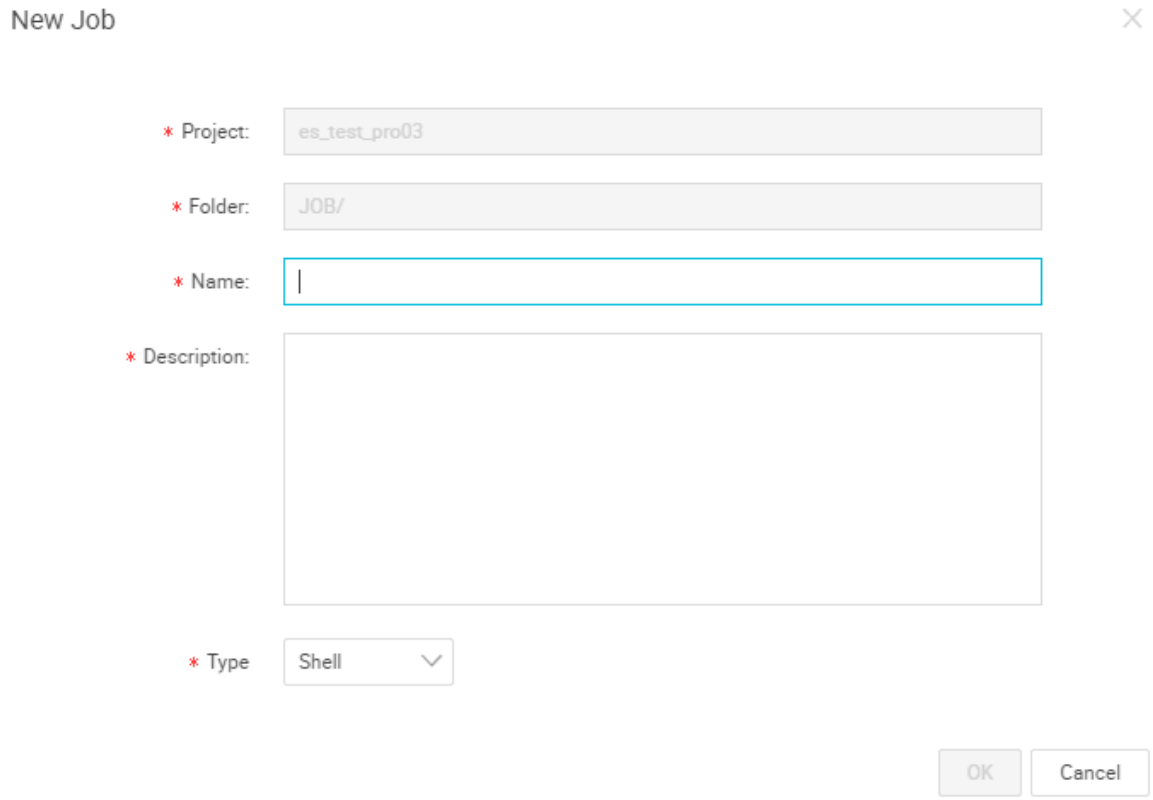
   a. OSS コンソールにログインし、**[ファイル]** タブをクリックします。

   b. **[アップロード]** をクリックし、aliyun-emapreduce-demo/resources ディレクトリにあるリソースファイルとaliyun-emapreduce-demo/target ディレクトリにある jar パッケージをアップロードします。
4. ワークフロープロジェクトを作成します。

   詳細は、「ワークフロープロジェクトマネジメント (Workflow project management)」をご参照ください。

5. ジョブを作成します。

   詳細は、「ジョブの編集 (Edit jobs)」をご参照ください。 MapReduce ジョブを例とします。



6. ジョブを設定したら、**[実行]** をクリックします。 詳細は次の図をご参照ください。

   - OSS 利用方法の詳細は、「OSS 利用説明 (/OSS usage instructions)」をご参照ください。
   - ジョブ設定方法の詳細は、『E-MapReduce ユーザーガイド』の『「ジョブ」』セクションをご参照ください。

   > 注：
   >
   > - OSS 出力 URI がすでに存在する場合は、ジョブを実行するとエラーが発生します。
   > - [OSS UNI の挿入] ボタンをクリックしてファイルプレフィックスとして OSSREF を選択すると、E-MapReduce が OSS ファイルをクラスターにダウンロードし、これらのファイルを指定されたクラスパスに追加します。
   > - 現在すべての操作をサポートしているのは OSS スタンダードストレージだけです。

ログの表示

実行プランのログを閲覧する方法の詳細は、「SSH を利用したクラスター接続 (Connect to a cluster using SSH)」をご参照ください。

2.3 E-MapReduce 上の Kafka でデータを処理する Storm トポロジーの送信

このページでは、Storm クラスターと Kafka クラスターを E-MapReduce にデプロイし、Storm トポロジーを実行して Kafka でデータを消費する方法について説明します。

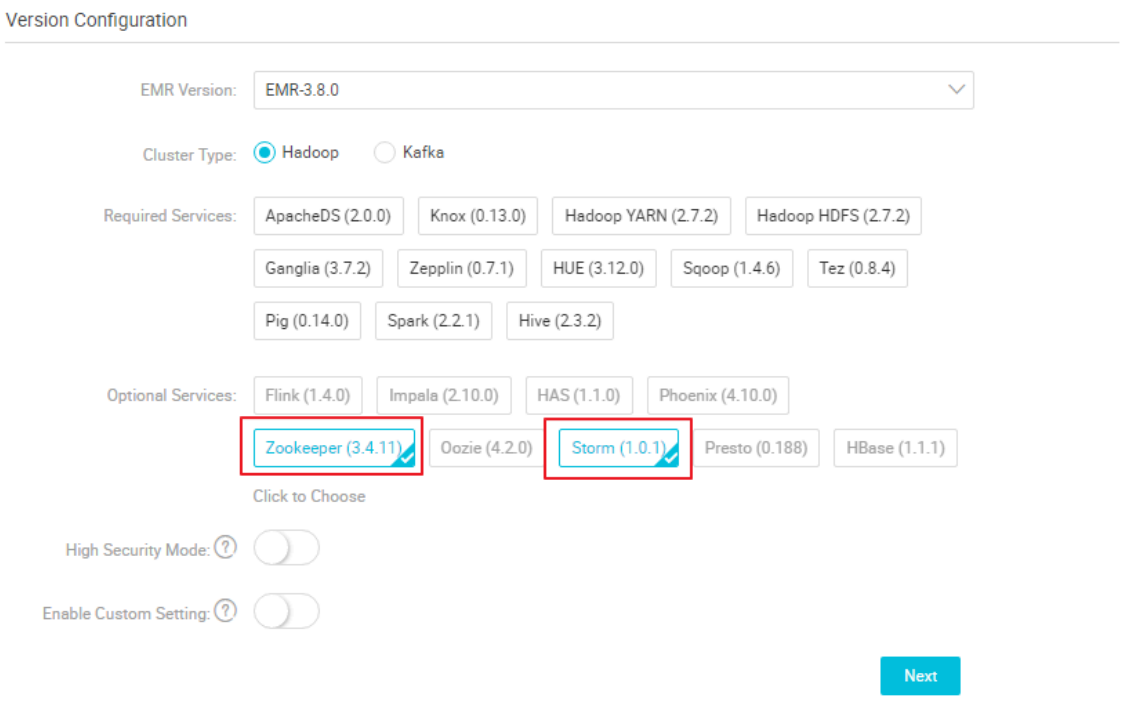
環境の準備

このテストは、中国 (杭州) リージョンにデプロイされている EMR を使用して実行されます。 EMR のバージョンは 3.8.0 です。 このテストに必要なコンポーネントのバージョンは以下のとおりです。

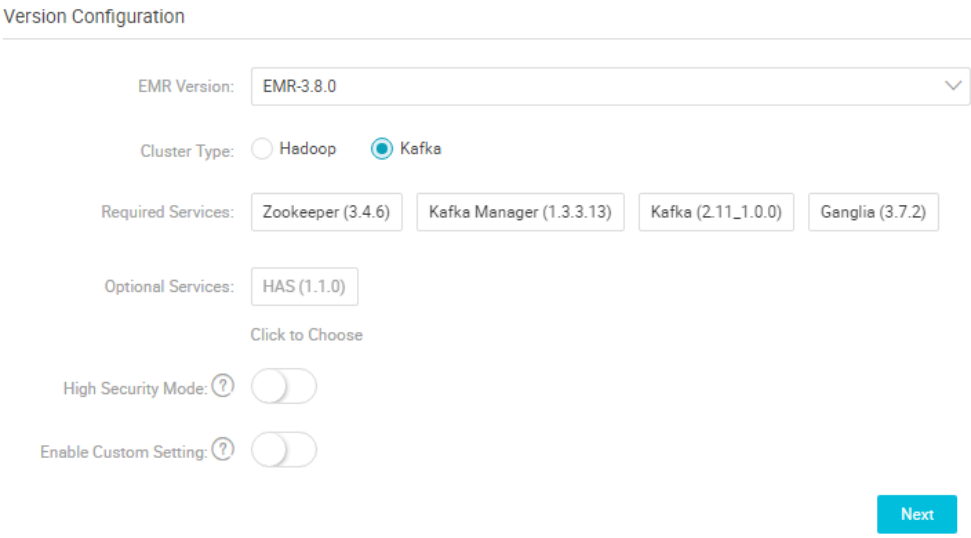
- Kafka: 2.11_1.0.0
- Storm: 1.0.1

このページでは、Alibaba Cloud E-MapReduce を使用して Kafka クラスターを自動的に作成します。 詳細は、「クラスターの作成 (Create a cluster)」をご参照ください。

- Hadoop クラスターの作成



- Kafka クラスターの作成



注：

- ネットワークタイプとしてクラシックネットワークを選択する場合は、Hadoop クラスターと Kafka クラスターを同じセキュリティグループに入れて、インスタンス間の接続を設定する時間を節約します。
- ネットワークタイプとして VPC を選択する場合は、Hadoop クラスターと Kafka クラスターを同じ VPC および同じセキュリティグループに配置して、VPC ピア接続を設定する時間を節約します。
- ECS のネットワークおよびセキュリティグループに精通していれば必要に応じて設定を作成できます。

- Storm の環境変数を設定します。

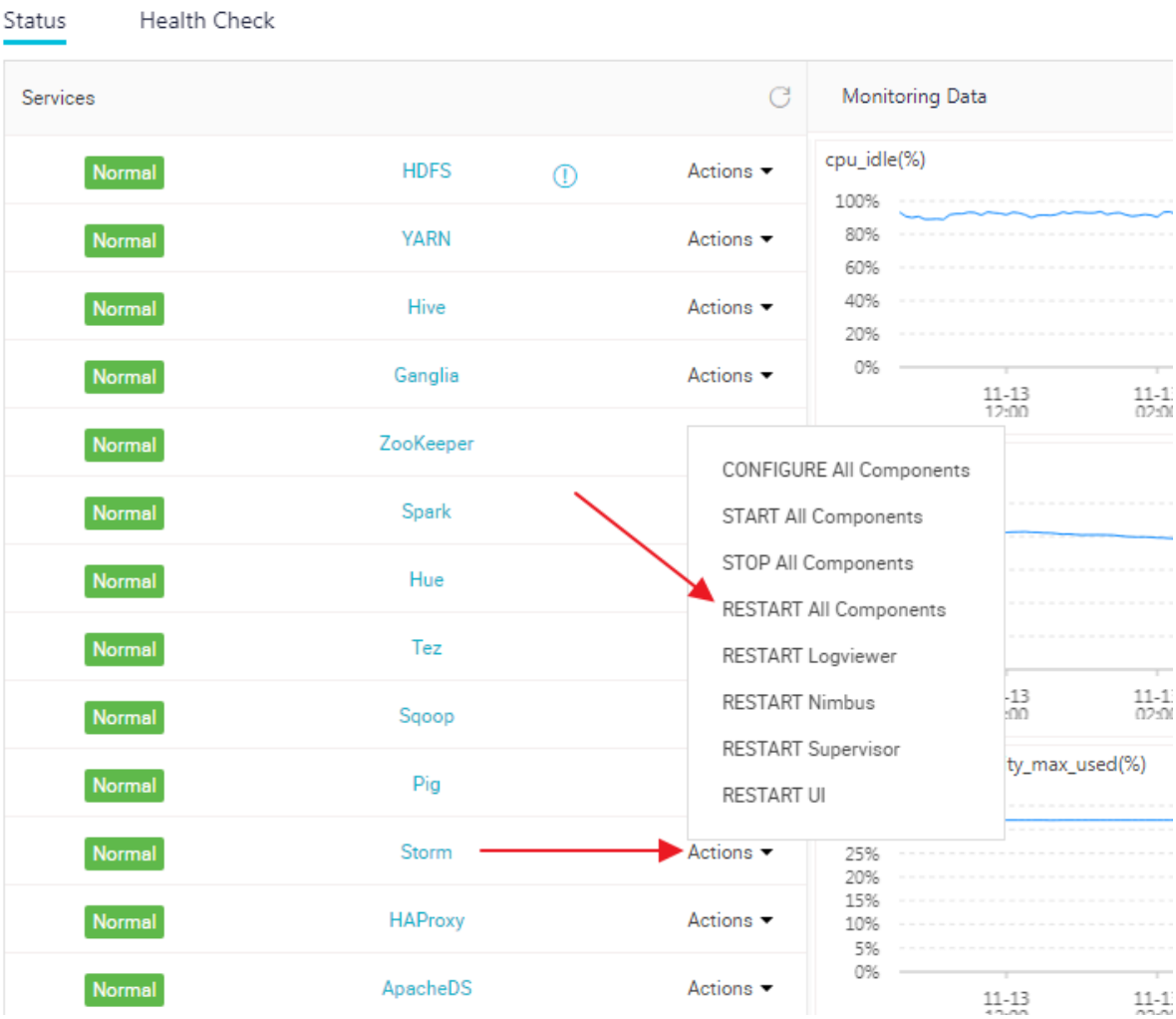
  初期環境で Storm トポロジーを実行すると、Kafka データの消費は失敗します。 このような失敗を回避するには、Storm 環境に以下の依存関係をインストールする必要があります。

  - curator-client
  - curator-framework
  - curator-recipes
  - json-simple
  - metrics-core
  - scala-library
  - zookeeper
  - commons-cli
  - commons-collections
  - commons-configuration
  - htrace-core
  - jcl-over-slf4j
  - protobuf-java
  - guava
  - hadoop-common
  - kafka-clients
  - kafka
  - storm-hdfs
  - storm-kafka

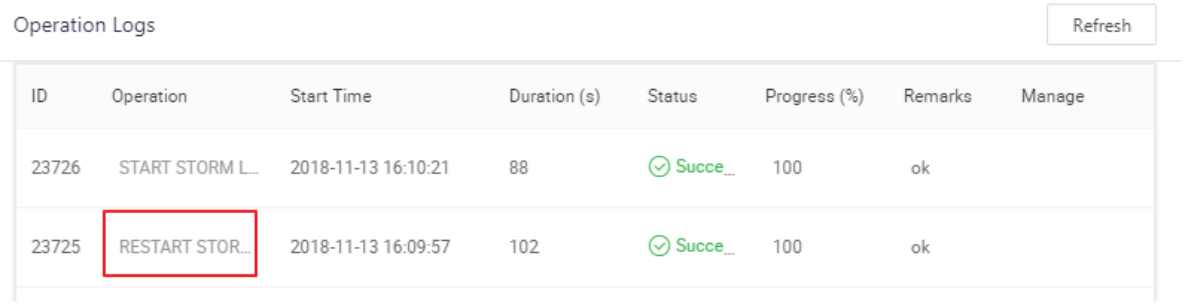
  これらの依存関係はテスト済みです。 追加の依存関係が必要な場合は、以下の操作を実行してそれらを Storm の lib フォルダーに追加します。

```
[hadoop@emr-header-1 ~]$ ll
total 8524
-rw-rw-r-- 1 hadoop hadoop   52988 Jun 14  2015 commons-cli-1.3.1.jar
-rw-rw-r-- 1 hadoop hadoop  588337 Nov 13  2015 commons-collections-3.2.2.jar
-rw-rw-r-- 1 hadoop hadoop  298829 Feb  5  2009 commons-configuration-1.6.jar
-rw-r--r-- 1 root   root     73448 Feb  9 14:01 curator-client-2.10.0.jar
-rw-r--r-- 1 root   root    195437 Feb  9 14:01 curator-framework-2.10.0.jar
-rw-r--r-- 1 root   root    281476 Feb  9 14:01 curator-recipes-2.10.0.jar
-rw-rw-r-- 1 hadoop hadoop   31212 Apr 19  2014 htrace-core-3.0.4.jar
-rw-rw-r-- 1 hadoop hadoop   17289 Jun 11  2012 jcl-over-slf4j-1.6.6.jar
-rw-rw-r-- 1 hadoop hadoop   16046 Aug 13  2009 json-simple-1.1.jar
-rw-rw-r-- 1 hadoop hadoop   82123 Nov 27  2012 metrics-core-2.2.0.jar
-rw-rw-r-- 1 hadoop hadoop  533455 Mar  8  2013 protobuf-java-2.5.0.jar
-rw-r--r-- 1 root   root   5745606 Feb  9 14:01 scala-library-2.11.7.jar
-rw-rw-r-- 1 hadoop hadoop  792964 Feb 24  2014 zookeeper-3.4.6.jar
[hadoop@emr-header-1 ~]$ pwd
/home/hadoop
[hadoop@emr-header-1 ~]$ sudo cp ./* /usr/lib/storm-current/lib/
```

Hadoop クラスター内の各ノードで上記の操作を実行する必要があります。 操作が完了したら、E-MapReduce コンソールで Storm を再起動します (次の図を参照)。



操作ログを閲覧して Stormの 状態をチェックできます。



Storm トポロジーと Kafka トピックの作成

- E-MapReduce は直接使用可能なサンプルコードを提供します。 リンクは以下のとおりです。
  - e-mapreduce-demo
  - e-mapreduce-sdk
- トピックにデータを書き込む

  1. Kafka クラスターにログインします。
  2. 10 個のパーティションと 2 個のレプリカでテストトピックを作成します。

     ```
     /usr/lib/kafka-current/bin/kafka-topics.sh --partitions 10 --replication-factor 2 --zookeeper emr-header-1:/kafka-1.0.0 --topic test --create
     ```

  3. テストトピックに 100 レコードのデータを書き込みます。

     ```
     /usr/lib/kafka-current/bin/kafka-producer-perf-test.sh --num-records 100 --throughput 10000 --record-size 1024 --producer-props bootstrap.servers=emr-worker-1:9092 --topic test
     ```

  > 注 :
  > 上記のコマンドは、Kafka クラスターの emr-header-1 ノードで実行されます。 このコマンドはクライアントノードで実行することもできます。

- Storm トポロジーを実行します。

  Hadoop クラスターにログインし、プロジェクトをコンパイルして /target/shaded ディレクトリの下の examples-1.1-shaded.jar ファイルを emr-header-1 ノードにコピーします。 こ

の例では、ファイルは HDFS ルートディレクトリに格納されています。 以下のコマンドを実行してトポロジーを送信します。

```
/usr/lib/storm-current/bin/storm jar examples-1.1-shaded.jar com.aliyun.emr.example.storm.StormKafkaSample test aaa.bbb.ccc.ddd hdfs://emr-header-1:9000 sample
```

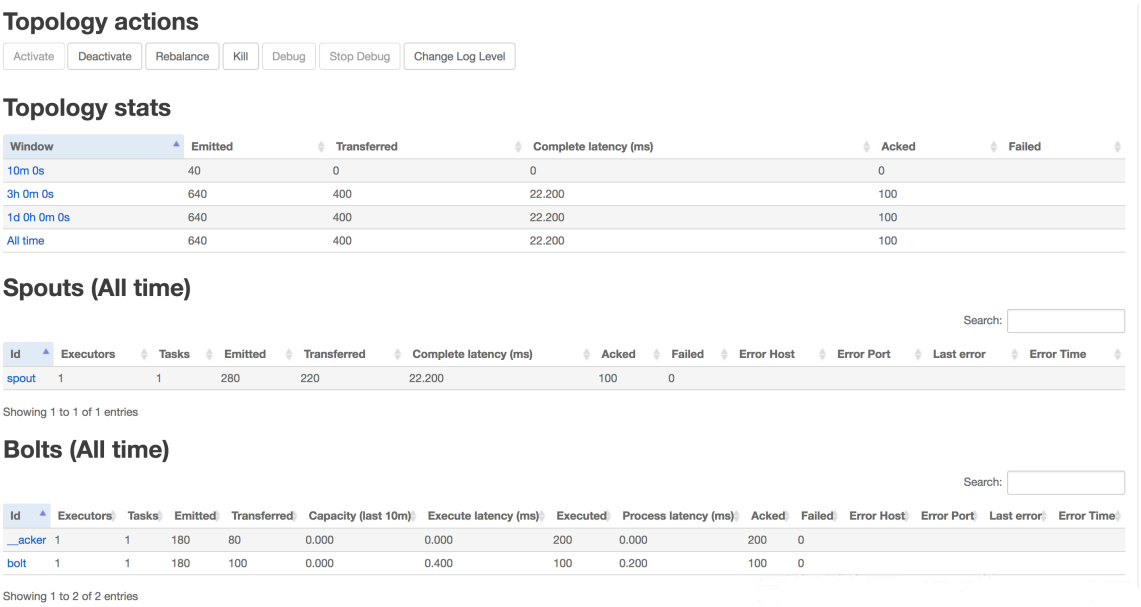
- トポロジーの実行状態を表示
  - Storm の実行状態を表示

    Web UI を使用して、以下のとおりクラスター上のサービスを閲覧できます。

    ■ Knox を使用。 詳細は、「Knox 注意事項 (Knox instructions)」をご参照ください。

    ■ SSH を使用。 詳細は、「SSH を使用してクラスターにログイン (Use SSH to log on to a cluster)」をご参照ください。

    このページでは、SSH を使用して Web UI にアクセスします。 エンドポイントは http://localhost:9999/index.html です。 送信されたトポロジーは表示可能です。 実行ログを閲覧するには、トポロジーをクリックします。



  - HDFS で出力ファイルを表示

    ■ HDFS で出力ファイルを閲覧します。

```
[root@emr-header-1 ~]# hadoop fs -ls /foo/
-rw-r--r--   3 root hadoop     615000 2018-02-11 13:37 /foo/bolt-2-0-1518327393692.txt
-rw-r--r--   3 root hadoop     205000 2018-02-11 13:37 /foo/bolt-2-0-1518327441777.txt
[root@emr-header-1 ~]# hadoop fs -cat /foo/bolt-2-0-1518327441777.txt | wc -l
```

```
200
```

■ Kafka のテストトピックに 120 レコードのデータを書き込みます。

```
[root@emr-header-1 ~]# /usr/lib/kafka-current/bin/kafka-producer-perf-test.
sh --num-records 120 --throughput 10000 --record-size 1024 --producer-props
bootstrap.servers=emr-worker-1:9092 --topic test
120 レコードが送信され、816.326531 レコード/秒 (0.80 MB /秒)、平均レイテンシ
35.37 ミリ秒、最大レイテンシ 134.00 ミリ秒、50 番目は 35 ミリ秒、95 番目は 39
ミリ秒、99 番目は 41 ミリ秒、99.9番目は 134 ミリ秒となります。
```

■ HDFS ファイルの行番号を出力します。

```
[root@emr-header-1 ~]# hadoop fs -cat /foo/bolt-2-0-1518327441777.txt | wc -l
320
```

概要

E-MapReduce に Storm クラスターと Kafka クラスターを正常にデプロイし、Storm トポロジーを実行して Kafka データを消費しました。 E-MapReduce は、Hadoop クラスターで実行して Kafka データを処理できる Spark ストリーミングと Flink コンポーネントもサポートしています。

 注：

E-MapReduce は Storm クラスターオプションは提供していません。 このため、Hadoop クラスターを作成し、Storm コンポーネントをインストールしました。 他のコンポーネントを使用する必要がない場合は、E-MapReduce コンソールでそれらのコンポーネントを簡単に無効にすることができます。 そうすれば、Hadoop クラスターは Storm クラスターと同等となります。

2.4 E-MapReduce で ES-Hadoop を使用

ES-Hadoop は、Elasticsearch (ES) が提供する Hadoop エコシステムの接続に使用されるツールです。 ユーザーはこのツールで MapReduce (MR)、Spark、Hive などのツールを使用して ES でデータを処理できます (このページでは扱いませんが、ES-Hadoop では ES インデックスのスナップショットの作成と HDFS への保存もサポートしています)。

背景

Hadoop エコシステムの利点は、大規模なデータセットを処理することです。 しかし対話式分析の遅延という欠点もあります。 ES は多くの種類のクエリ、特にアドホッククエリを巧みに扱うことができます。 レスポンス時間は短く 1 秒未満です。 ES-Hadoop は両方の利点を兼ね備えています。 ES-Hadoop を使用すれば、ES に格納されているデータを迅速に処理するコードを少し変更するだけで済みます。 ES は高速化も提供します。

ES-Hadoop は MR、Spark、Hive などのデータ処理エンジンのデータソースとして ES を使用します。 ES は、コンピューティングとストレージが分離されているアーキテクチャでストレージの役割を果たします。 これは、MR、Spark、および Hive の他のデータソースについても同じです。 しかし ES のデータフィルタリング機能は他のデータソースと比較して高速です。 この機能は、分析エンジンの最も重要な機能の 1 つです。

EMR はすでに ES-Hadoop と統合されています。 ES-Hadoop は設定なしで直接使用できます。 以下に、EMR 上の ES-Hadoop の例を紹介します。

準備

ES は自動的にインデックスを作成し、入力データでデータタイプを識別します。 この機能はユーザーのいろいろな操作を回避する役に立つ場合があります。 ただし、問題もあります。 最大の問題は、ES が識別するデータ型が正しくない場合があることです。 たとえば、age というフィールドを定義します。 この列のデータ型は INT ですが、ES インデックスでは LONG として識別される場合があります。 ユーザーは、特定のアクションの実行時にデータ型を変換する必要があります。 このような問題を避けるには手動でインデックスを作成するよう推奨します。

以下の例では、 company インデックスと employees' タイプを使用 (ES インデックスをデータベースとして、タイプをデータベース内のテーブルとみなすことができます)。 このタイプは 4 つのフィールドを定義します (フィールドタイプは ES によって定義されます)。

```
{
  "id": long,
  "name": text,
  "age": integer,
  "birth": date
}
```

Kibana でインデックスを作成するには、以下のコマンドを実行します (cURL も使用できます)。

```
PUT company
{
  "mappings": {
    "employees": {
      "properties": {
        "id": {
          &amp;quot;type&amp;quot; :  &amp;quot;long&amp;quot;
        },
        "name": {
          &amp;quot;type&amp;quot; :  &amp;quot;int&amp;quot;、
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "birth": {
          &amp;quot;type&amp;quot; :  &amp;quot;date&amp;quot;、
```

```
      },
      "addr": {
        "type": "text"
      }
    }
  }
 },
 "settings": {
  &amp;quot;index&amp;quot; : 0
    "number_of_shards": "5",
    "number_of_replicas": "1"
  }
 }
}
```

注：
必要に応じて設定でインデックスパラメーターを指定します。 このステップは省略可能です。

以下のように各行が JSON オブジェクトであるファイルを準備します。

```
{"id": 1, "name": "zhangsan", "birth": "1990-01-01", "addr": "No. 969, wenyixi Rd, yuhang
, hangzhou"}
{"id": 2, "name": "lisi", "birth": "1991-01-01", "addr": "No. 556, xixi Rd, xihu, hangzhou"}
{"id": 3, "name": "wangwu", "birth": "1992-01-01", "addr": "No. 699 wangshang Rd,
binjiang, hangzhou"}
```

HDFS 内の指定したディレクトリにファイルを保存します (たとえば、/es-hadoop/employees.txt)。

Mapreduce

以下の例では、HDFS 内の /es-hadoop ディレクトリにある JSON ファイルを読み取り、JSONファイルの各行をドキュメントとして ES に書き込みます。 EsOutputFormat により書き込みはマップステージで終了します。

ES を設定するには、以下のオプションを使用します。

- es.nodes: ES nodes. フォーマットは host:port です。 Alibaba Cloud でホスティングされている ES の場合は、Alibaba Cloud が提供する ES のエンドポイントに値を設定します。
- es.net.http.auth.user: Username.
- es.net.http.auth.pass: Password.
- es.nodes.wan.only: For ES hosted on Alibaba Cloud, set the value to true.
- es.resource: The indices and types of ES.
- es.input.json: 入力ファイルが JSON 形式の場合は値を true に設定します。 それ以外の場合は、map() 関数を使用して入力データを解析し、対応する Writable クラスを出力する必要があります。

:

マップタスクの投機的実行は無効にしてタスク数を削減します。

```
package com.aliyun.emr;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.elasticsearch.hadoop.mr.EsOutputFormat;

public class Test implements Tool {

  private Configuration conf;

  @Override
  public int run(String[] args) throws Exception {

    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

    conf.setBoolean("mapreduce.map.speculative", false);
    conf.setBoolean("mapreduce.reduce.speculative", false);
    conf.set("es.nodes", "<your_es_host>:9200");
    conf.set("es.net.http.auth.user", "<your_username>");
    conf.set("es.net.http.auth.pass", "<your_password>");
    conf.set("es.nodes.wan.only", "true");
    conf.set("es.resource", "company/employees");
    conf.set("es.input.json", "yes");

    Job job = Job.getInstance(conf);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(EsOutputFormat.class);
    job.setMapOutputKeyClass(NullWritable.class);
    job.setMapOutputValueClass(Text.class);
    job.setJarByClass(Test.class);
    job.setMapperClass(EsMapper.class);

    FileInputFormat.setInputPaths(job, new Path(otherArgs[0]));

    return job.waitForCompletion(true) ? 0 : 1;
  }

  @Override
  public void setConf(Configuration conf) {
    this.conf = conf;
  }

  @Override
  public Configuration getConf() {
    return conf;
  }

  public static class EsMapper extends Mapper<Object, Text, NullWritable, Text> {
```

```
    private Text doc = new Text();

    @Override
    protected void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
      if (value.getLength() > 0) {
        doc.set(value);
        context.write(NullWritable.get(), doc);
      }
    }
  }

  public static void main（final String [] args）throws IOException{
    int ret = ToolRunner.run(new Test(), args);
    System.exit(ret);
  }
}
```

mr-test.jar という JAR ファイルにコードをコンパイルしパッケージします。 そのコードを EMR クライアントプログラム (ゲートウェイその他 EMR クラスターの任意のノード) をインストールしたインスタンスにサブミットします。

以下のコマンドを EMR クライアントをインストールした任意のノード上で実行して MapReduce プログラムを実行します。

```
hadoop jar mr-test.jar com.aliyun.emr.Test -Dmapreduce.job.reduces=0 -libjars mr-test.
jar /es-hadoop
```

ES へのデータの書き込みはこの時点で終了しています。 Kibana (または cURL コマンド) を使用してデータの書き込みを照会できます。

```
GET
{
  "query": {
    "match_all": {}
  }
}
```

Spark

この例では MapReduce ではなく Spark を使用して ES 内のインデックスにデータを書き込みます。 Spark では JavaEsSpark クラスを使用して ES に回復力のある分散データセット (RDD) が持続されます。 ES を設定するには MapReduce セクション内上記のオプションを使用する必要もあります。

```
package com.aliyun.emr;

import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.sql.Row;
```

```
import org.apache.spark.sql.SparkSession;
import org.elasticsearch.spark.rdd.api.java.JavaEsSpark;
import org.spark_project.guava.collect.ImmutableMap;

public class Test {

  public static void main(String[] args) {
    SparkConf conf = new SparkConf();
    conf.setAppName("Es-test");
    conf.set("es.nodes", "<your_es_host>:9200");
    conf.set("es.net.http.auth.user", "<your_username>");
    conf.set("es.net.http.auth.pass", "<your_password>");
    conf.set("es.nodes.wan.only", "true");

    SparkSession ss = new SparkSession(new SparkContext(conf));
    final AtomicInteger employeesNo = new AtomicInteger(0);
    JavaRDD<Map<Object, ? >> javaRDD = ss.read().text("hdfs://emr-header-1:9000/es-
hadoop/employees.txt")
        .javaRDD().map((Function<Row, Map<Object, ? >>) row -> ImmutableMap.of("
employees" + employeesNo.getAndAdd(1), row.mkString()));

    JavaEsSpark.saveToEs(javaRDD, "company/employees");
  }
}
```

spark-test.jar という JAR ファイル内にコードをパッケージします。 以下のコマンドを実行してデータを書き込みます。

```
spark-submit --master yarn --class com.aliyun.emr.Test spark-test.jar
```

このタスクの終了後に Kibana または cURL で結果を照会できます。

Spark RDD に追加して実行します。 ES-Hadoop は ES データの読み書き用に Spark SQL コンポーネントも提供しています。 詳細は、ES-Hadoop の『公式 Web サイト』をご参照ください。

Hive

この例では、Hive を介して ES データを読み書きする SQL 文を紹介します。

まず **hive** を実行して CLI にアクセスしテーブルを作成します。

```
CREATE DATABASE IF NOT EXISTS company;
```

次に ES に格納される外部テーブルを作成します。 TBLPROPERTIES を使用してオプションを指定します。

```
CREATE EXTERNAL table IF NOT EXISTS employees(
  id BIGINT,
  name STRING,
  birth TIMESTAMP,
  addr STRING
)
STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler'
TBLPROPERTIES(
    'es.resource' = 'tpcds/ss',
    'es.nodes' = '<your_es_host>',
    'es.net.http.auth.user' = '<your_username>',
```

```
    'es.net.http.auth.pass' = '<your_password>',
    'es.nodes.wan.only' = 'true',
    'es.resource' = 'company/employees'
);
```

注：

birth の列のデータ型を Hive テーブル内の TIMESTAMP に設定します。 ES では DATE に設定します。 Hive および EC ではデータ型の扱いが異なるためこの設定にします。 変換済みの日付データの解析は Hive で ES にデータを書き込むときに失敗する場合があります。 これに対して、Hive で ES データを読み取るときもリターンデータの解析が失敗することがあります。 詳細は、こちらをクリックしてください。

テーブルにデータをいくらか挿入します。

```
INSERT INTO TABLE employees VALUES (1, "zhangsan", "1990-01-01","No. 969, wenyixi Rd
, yuhang, hangzhou");
INSERT INTO TABLE employees VALUES (2, "lisi", "1991-01-01", "No. 556, xixi Rd, xihu,
hangzhou");
INSERT INTO TABLE employees VALUES (3, "wangwu", "1992-01-01", "No. 699 wangshang
 Rd, binjiang, hangzhou");
```

クエリを実行して結果を閲覧します。

```
SELECT * FROM employees LIMIT 100;
OK
1    zhangsan    1990-01-01    No. 969, wenyixi Rd, yuhang, hangzhou
2    lisi    1991-01-01    No. 556, xixi Rd, xihu, hangzhou
3    wangwu    1992-01-01    No. 699 wangshang Rd, binjiang, hangzhou
```

2.5 E-MapReduce での Mongo-Hadoop の利用

Mongo-Hadoop は、Hadoop コンポーネントが MongoDB に接続する場合に MongoDB によって提供されるコンポーネントです。 Mongo-Hadoop の使い方は、前のページで説明した ES-Hadoop の使い方と似ています。 EMR はすでに Mongo-Hadoop と統合されています。 Mongo-Hadoop はデプロイ設定なしで直接使用できます。 このページでは、いくつかの例を使用して Mongo-Hadoop を使用方法について説明します。

準備

以下の例で同じデータモデルを使用します。

```
{
  "id": long,
  "name": text,
  "age": integer,
  "birth": date
```

```
}
```

MongoDB データベースの指定されたコレクション (データベース内のテーブルと同様) にデータを書き込みます。 このため、まずコレクションが MongoDB データベースに確実に存在している必要があります。 まず、MongoDB データベースにアクセスできるクライアントノードで MongoDB クライアントプログラムを実行します。 MongoDB Web サイトからクライアントプログラムをダウンロードしてインストールすることが必要な場合があります。 例として MongoDB 用の ApsaraDB への接続を行います。

```
mongo --host dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717 --authentica
tionDatabase admin -u root -p 123456
```

MongoDB データベースのホスト名は dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com です。 ポート番号は 3717 です。 実際のポート番号は MongoDB クラスターによって異なります。 独自にデプロイした外部 MongoDB クラスターの場合、デフォルトのポート番号は 27017 です。 この例では、-p オプションを使用してパスワードを 123456 に設定しています。 CLI で以下のコマンドを実行して、company データベースに employees という名前のコレクションを作成します。

```
> use company;
> db.createCollection("employees")
```

以下のように各行が JSON オブジェクトであるファイルを用意します。

```
{"id": 1, "name": "zhangsan", "birth": "1990-01-01", "addr": "No. 969, wenyixi Rd, yuhang
, hangzhou"}
{"id": 2, "name": "lisi", "birth": "1991-01-01", "addr": "No. 556, xixi Rd, xihu, hangzhou"}
{"id": 3, "name": "wangwu", "birth": "1992-01-01", "addr": "No. 699 wangshang Rd,
binjiang, hangzhou"}
```

HDFS 内の指定したディレクトリにファイルを保存します (たとえば、/mongo-hadoop/employees.txt)。

Mapreduce

以下の例では、HDFS の /mongo-hadoop ディレクトリにある JSON ファイルを読み取り、JSON ファイルの各行をドキュメントとして MongoDB データベースに書き込みます。

```
package com.aliyun.emr;

import com.mongodb.BasicDBObject;
import com.mongodb.hadoop.MongoOutputFormat;
import com.mongodb.hadoop.io.BSONWritable;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
```

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class Test implements Tool {

  private Configuration conf;

  @Override
  public int run(String[] args) throws Exception {

    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

    conf.set("mongo.output.uri", "mongodb://<your_username>:<your_password>@
dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717/company.employees?
authSource=admin");

    Job job = Job.getInstance(conf);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(MongoOutputFormat.class);
    job.setOutputKeyClass(Text.class);
    job.setMapOutputValueClass(BSONWritable.class);

    job.setJarByClass(Test.class);
    job.setMapperClass(MongoMapper.class);

    FileInputFormat.setInputPaths(job, new Path(otherArgs[0]));

    return job.waitForCompletion(true) ? 0 : 1;
  }

  @Override
  public Configuration getConf() {
    return conf;
  }

  @Override
  public void setConf(Configuration conf) {
    this.conf = conf;
  }

  public static class MongoMapper extends Mapper<Object, Text, Text, BSONWritable> {

    private BSONWritable doc = new BSONWritable();
    private int employeeNo = 1;
    private Text id;

    @Override
    protected void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
      if (value.getLength() > 0) {
        doc.setDoc(BasicDBObject.parse(value.toString()));
        id = new Text("employee" + employeeNo++);
        context.write(id, doc);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    int ret = ToolRunner.run(new Test(), args);
    System.exit(ret);
  }
```

```
}
```

コードをコンパイルし、mr-test.jar という JAR ファイルにパッケージ化します。 以下のコマンドを実行します。

```
hadoop jar mr-test.jar com.aliyun.emr.Test -Dmapreduce.job.reduces=0 -libjars mr-test.jar /mongo-hadoop
```

実行が完了したら、MongoDB クライアントプログラムを使用して結果を閲覧できます。

```
> db.employees.find();
{ "_id" : "employee1", "id" : 1, "name" : "zhangsan", "birth" : "1990-01-01", "addr" : "No. 969, wenyixi Rd, yuhang, hangzhou" }
{ "_id" : "employee2", "id" : 2, "name" : "lisi", "birth" : "1991-01-01", "addr" : "No. 556, xixi Rd, xihu, hangzhou" }
{ "_id" : "employee3", "id" : 3, "name" : "wangwu", "birth" : "1992-01-01", "addr" : "No. 699 wangshang Rd, binjiang, hangzhou" }
```

Spark

この例では、MapReduce ではなく Spark を使用して MongoDB データベースにデータを書き込みます。

```
package com.aliyun.emr;

import com.mongodb.BasicDBObject;
import com.mongodb.hadoop.MongoOutputFormat;
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.hadoop.conf.Configuration;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.bson.BSONObject;
import scala.Tuple2;

public class Test {

  public static void main(String[] args) {

    SparkSession ss = new SparkSession(new SparkContext());

    final AtomicInteger employeeNo = new AtomicInteger(0);
    JavaRDD<Tuple2<Object, BSONObject>> javaRDD =
        ss.read().text("hdfs://emr-header-1:9000/mongo-hadoop/employees.txt")
            .javaRDD().map((Function<Row, Tuple2<Object, BSONObject>>) row -> {
          BSONObject bson = BasicDBObject.parse(row.mkString());
          return new Tuple2<>("employee" + employeeNo.getAndAdd(1), bson);
        });

    JavaPairRDD<Object, BSONObject> documents = JavaPairRDD.fromJavaRDD(javaRDD);

    Configuration outputConfig = new Configuration();
    outputConfig.set("mongo.output.uri", "mongodb://<your_username>:<your_password>@dds-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717/company.employees?authSource=admin");
```

```
    // It is saved as a "Hadoop file." データは実際には MongoOutputFormat クラスを介して
MongoDB データベースに書き込まれます。
    documents.saveAsNewAPIHadoopFile(
        "file:///this-is-completely-unused",
        Object.class,
        BSONObject.class,
        MongoOutputFormat.class,
        outputConfig
    );
  }
}
```

spark-test.jar という名前の JAR ファイルにコードをパッケージ化します。 以下のコマンドを実行してデータを書き込みます。

```
spark-submit --master yarn --class com.aliyun.emr.Test spark-test.jar
```

書き込みが完了したら、MongoDB クライアントを使用して結果を閲覧できます。

Hive

この例では、Hive を使用して SQL 文を介して MongoDB データベースへデータを読み書きする方法について説明します。

まず、**hive** を実行してCLI モードに入り、テーブルを作成します。

```
CREATE DATABASE IF NOT EXISTS company;
```

MongoDB データベースに格納される外部テーブルを作成する必要があります。 その前に、「準備」セクションの説明に従って、 employees という名前の MongoDB コレクションを作成します。

CLI モードに戻り、以下の SQL ステートメントを実行して外部テーブルを作成します。
MongoDB への接続は TBLPROPERTIES 句を介して設定されます。

```
CREATE EXTERNAL TABLE IF NOT EXISTS employees(
  id BIGINT,
  name STRING,
  birth STRING,
  addr STRING
)
STORED BY 'com.mongodb.hadoop.hive.MongoStorageHandler'
WITH SERDEPROPERTIES('mongo.columns.mapping'='{"id":"_id"}')
TBLPROPERTIES('mongo.uri'='mongodb://<your_username>:<your_password>@dds
-xxxxxxxxxxxxxxxxxxxxx.mongodb.rds.aliyuncs.com:3717/company.employees?
authSource=admin');
```

:

Hive の id 列の値は SERDEPROPERTIES を介して MongoDB の _id 列の値にマッピングされます。 列値は必要に応じてマッピングできます。 birth 列のデータ型が STRING に設定されている

ことに注意してください。 Hive と MongoDB は DATE 形式の処理方法が異なるためこの設定にします。 Hive が DATE 形式のデータを MongoDB に送信した後に Hive のデータが照会されると NULL が返される場合があります。

テーブルにデータを挿入します。

```
INSERT INTO TABLE employees VALUES (1, "zhangsan", "1990-01-01","No. 969, wenyixi Rd
, yuhang, hangzhou");
INSERT INTO TABLE employees VALUES (2, "lisi", "1991-01-01", "No. 556, xixi Rd, xihu,
hangzhou");
INSERT INTO TABLE employees VALUES (3, "wangwu", "1992-01-01", "No. 699 wangshang
 Rd, binjiang, hangzhou");
```

以下の文を実行して結果を確認します。

```
SELECT * FROM employees LIMIT 100;
OK
1    zhangsan    1990-01-01    No. 969, wenyixi Rd, yuhang, hangzhou
2    lisi    1991-01-01    No. 556, xixi Rd, xihu, hangzhou
3    wangwu    1992-01-01    No. 699 wangshang Rd, binjiang, hangzhou
```

2.6 E-MapReduce の Analytics Zoo によるディープラーニング

Analytics Zoo は、Apache Spark と Intel BigDL とを統合パイプラインに融合するアナリティクスおよび AI プラットフォームです。 ビッグデータとエンドツーエンドのパイプラインをベースとしたディープラーニングアプリケーションの開発を支援します。 このページでは、Analytics Zoo を使用してAlibaba Cloud E-MapReduce 上でディープラーニングアプリケーションを開発する方法について説明します。

はじめに

Analytics Zoo は、Apache Spark と Intel BigDL とを統合パイプラインに融合するアナリティクスおよび AI プラットフォームです。 ビッグデータとエンドツーエンドのパイプラインをベースとしたディープラーニングアプリケーションの開発を支援します。

システム要件

- JDK 8
- Spark クラスター (EMR でサポートされている Spark 2.x を推奨します)
- Python 2.7 (Python 3.5 または Python 3.6 も可)、pip

Analytics Zoo のインストール

- Analytics Zoo の最新リリースは 0.2.0 です。

- Scala ユーザーの場合のインストール
  - プレビルドバージョンをダウンロードします。

    GitHub の Analytics Zoo ページからプレビルドバージョンをダウンロードできます。
  - make-dist.sh スクリプトを使用して Analytics Zoo をビルドします。

    Apache Maven をインストールし、以下のとおり環境変数 MAVEN_OPTS を設定します。

    ```
    export MAVEN_OPTS="-Xmx2g -XX:ReservedCodeCacheSize=512m"
    ```

    ECS インスタンスを使用してコードをコンパイルする場合は、Maven リポジトリのミラーを変更するよう推奨します。

    ```
    <mirror>
      <id>nexus-aliyun</id>
      <mirrorOf>central</mirrorOf>
      <name>Nexus aliyun</name>
      <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </mirror>
    ```

    Analytics Zoo リリース をダウンロードします。 ファイルを解凍し、対応するディレクトリに移動して、以下のコマンドを実行します。

    ```
    bash make-dist.sh
    ```

    Analytics Zoo をビルドすると dist ディレクトリが見つかります。このディレクトリには、Analytics Zoo プログラムの実行に必要なすべてのファイルが含まれています。 以下のコマンドを使用して、dist ディレクトリ内のファイルを EMR ソフトウェアスタックのディレクトリにコピーします。

    ```
    cp -r dist/ /usr/lib/analytics_zoo
    ```
- Python ユーザーの場合のインストール

  Analytics Zoo は、pip を使用しても使用しなくてもインストール可能です。 Analytics Zoo を pip でインストールすると、PySpark と BigDL がインストールされます。 PySpark はすでに EMR クラスターにインストールされているため、これによりソフトウェアの競合が発生す

る可能性があります。 このような競合を避けるには、Analytics Zoo を pip なしでインストールします。

- pip を使用しないインストール

  まず、以下のコマンドを実行する必要があります。

  ```
  bash make-dist.sh
  ```

  pyzoo ディレクトリに移動して、Analytics Zoo をインストールします。

  ```
  python setup.py install
  ```

- 環境変数の設定

  Analytics Zoo をビルドしたら、dist ディレクトリを EMR ソフトウェアスタックのディレクトリにコピーし、環境変数を設定します。 /etc/profile.d/analytics_zoo.sh ファイルに以下の行を追加します。

  ```
  export ANALYTICS_ZOO_HOME=/usr/lib/analytics_zoo
  export PATH=$ANALYTICS_ZOO_HOME/bin:$PATH
  ```

  SPARK_HOME はすでに EMR に設定されているので、設定する必要はありません。

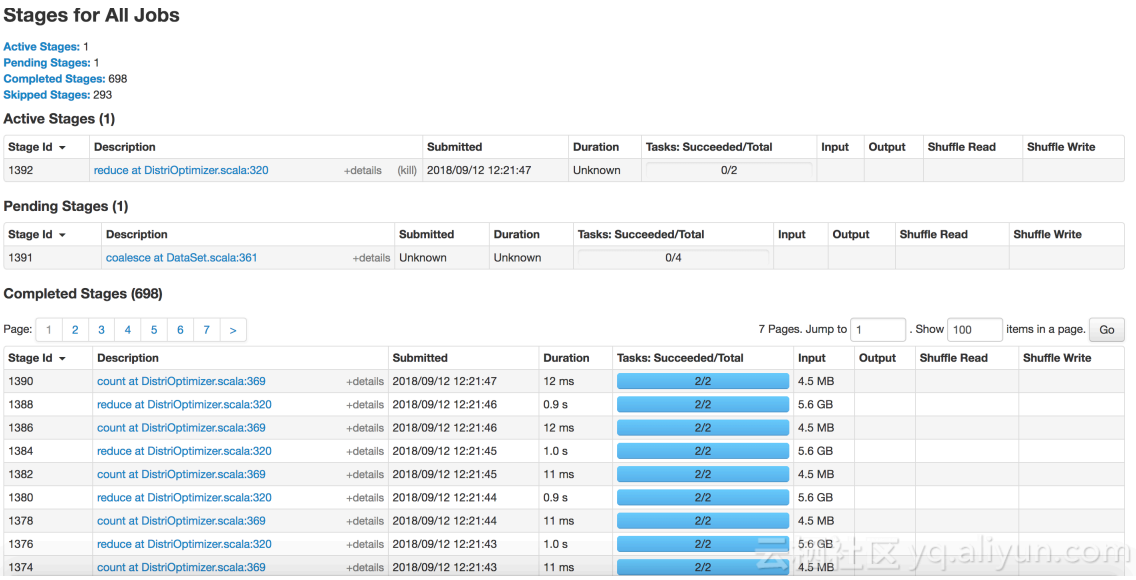
Analytics Zoo の利用

- Spark を使ってディープラーニングモデルを学習させテストします。
  - Analytics Zoo を使用してテキストを分類します。 コードと説明は GitHub にあります。 必要なデータを必要なだけダウンロードします。 以下のコマンドをサブミットします。

    ```
    spark-submit --master yarn \
    --deploy-mode cluster --driver-memory 8g \
    --executor-memory 20g --class com.intel.analytics.zoo.examples.textclassification
    .TextClassification \
    ```

```
/usr/lib/analytics_zoo/lib/analytics-zoo-bigdl_0.6.0-spark_2.1.0-0.2.0-jar-with-dependencies.jar --baseDir /news
```

- SSH プロキシを介して Spark クラスターのインスタンスにログインしジョブのステータスを閲覧します。



ログを介して各エポックの正確さを閲覧することもできます。

```
INFO optim.DistriOptimizer$: [Epoch 2 9600/15107][Iteration 194][Wall Clock 193.266637037s] Trained 128 records in 0.958591653 seconds. スループットは 133.52922 レコード/秒です。 損失は 0.74216986 です。
INFO optim.DistriOptimizer$: [Epoch 2 9728/15107][Iteration 195][Wall Clock 194.224064816s] Trained 128 records in 0.957427779 seconds. スループットは 133.69154 レコード/秒です。 損失は 0.51025534 です。
INFO optim.DistriOptimizer$: [Epoch 2 9856/15107][Iteration 196][Wall Clock 195.189488678s] Trained 128 records in 0.965423862 seconds. スループットは 132.58424 レコード/秒です。 損失は 0.553785 です。
```

```
INFO optim.DistriOptimizer$: [Epoch 2 9984/15107][Iteration 197][Wall Clock 196.164318688s] Trained 128 records in 0.97483001 seconds. スループットは 131.30495 レコード/秒です。 損失は 0.5517549 です。
```

- Analytics Zoo で PySpark と Jupyter を使用してディープラーニングモデルを学習させます。
  - Jupyter をインストールします。

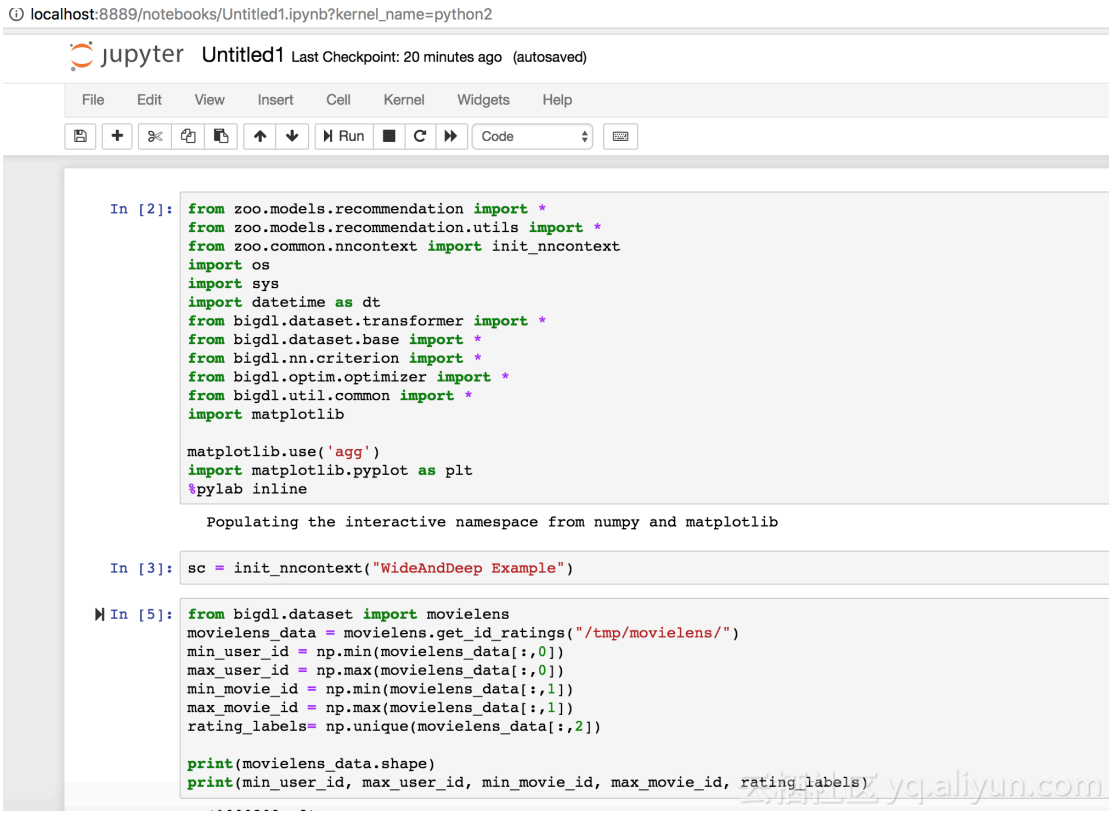
    ```
    pip install jupyter
    ```

  - 以下のコマンドを実行して Jupyter を開始します。

    ```
    jupyter-with-zoo.sh
    ```

  - Analytics Zoo が提供する定義済みの Wide And Deep Learning モデルを使用するよう推奨します。

    1. データをインポートします。



    2. モデルをビルドしてオプティマイザを作成します。

```
In [10]: wide_n_deep = WideAndDeep(5, column_info, "wide_n_deep")
```

```
creating: createZooWideAndDeep
```

```
In [11]: # Create an Optimizer
batch_size = 8000

optimizer = Optimizer(
    model=wide_n_deep,
    training_rdd=train_data,
    criterion=ClassNLLCriterion(),
    optim_method=Adam(learningrate = 0.001, learningrate_decay=0.00005),
    end_trigger=MaxEpoch(10),
    batch_size=batch_size)

# Set the validation logic
optimizer.set_validation(
    batch_size=batch_size,
    val_rdd=test_data,
    trigger=EveryEpoch(),
    val_method=[Top1Accuracy(), Loss(ClassNLLCriterion())]
)
log_dir='/tmp/bigdl_summaries/'
app_name='wide_n_deep-'+dt.datetime.now().strftime("%Y%m%d-%H%M%S")
train_summary = TrainSummary(log_dir=log_dir,
                                        app_name=app_name)
val_summary = ValidationSummary(log_dir=log_dir,
                                      app_name=app_name)
optimizer.set_train_summary(train_summary)
optimizer.set_val_summary(val_summary)
print("saving logs to %s" % (log_dir + app_name))
```


3. 学習プロセスを開始します。

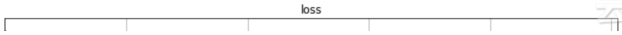
```
In [12]: %%time
# Boot training process
optimizer.optimize()
print("Optimization Done.")
```

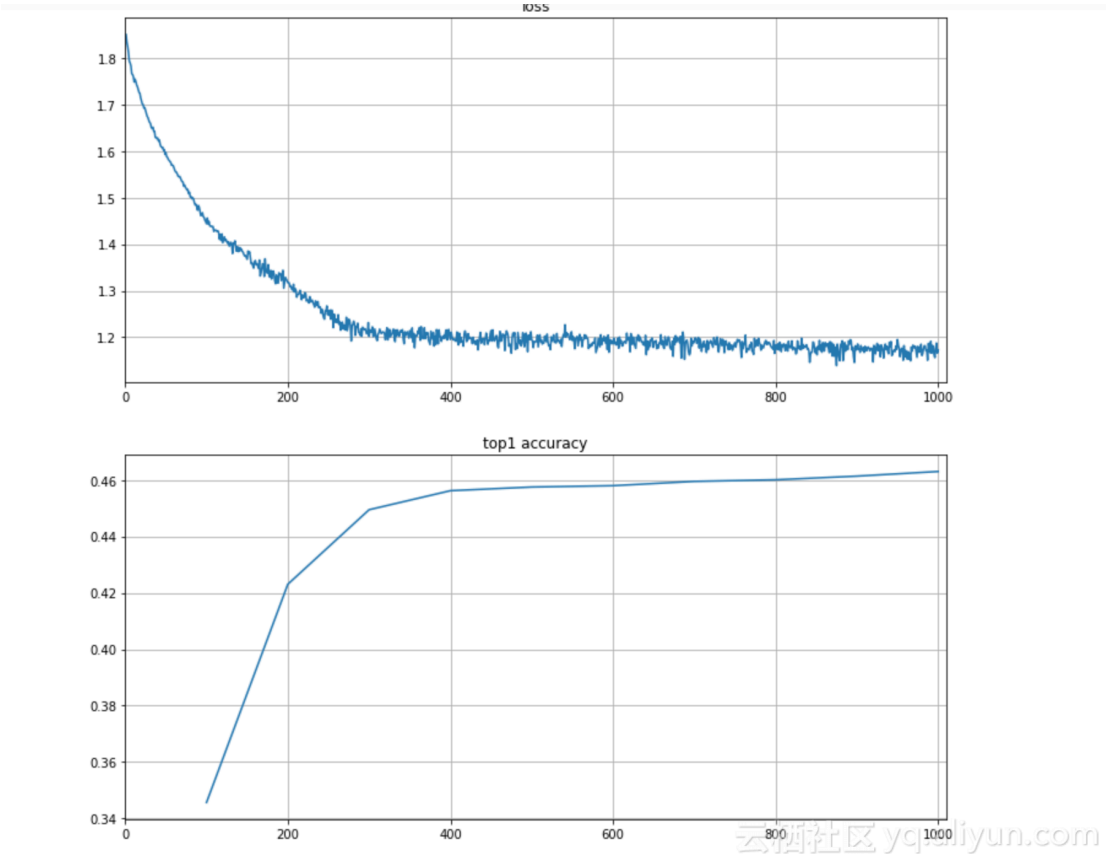
```
Optimization Done.
CPU times: user 85.9 ms, sys: 16.7 ms, total: 103 ms
Wall time: 2min 52s
```


4. 学習結果を閲覧します。

```
In [16]: loss = np.array(train_summary.read_scalar("Loss"))
top1 = np.array(val_summary.read_scalar("Top1Accuracy"))

plt.figure(figsize = (12,12))
plt.subplot(2,1,1)
plt.plot(loss[:,0],loss[:,1],label='loss')
plt.xlim(0,loss.shape[0]+10)
plt.grid(True)
plt.title("loss")
plt.subplot(2,1,2)
plt.plot(top1[:,0],top1[:,1],label='top1')
plt.xlim(0,loss.shape[0]+10)
plt.title("top1 accuracy")
plt.grid(True)
```





2.7 Spark SQL のアダプティブ実行

Alibaba Cloud Elastic MapReduce (E-MapReduce) 3.13.0 の Spark SQL はアダプティブ実行をサポートしています。 自動的に削減タスクの数を設定し、データスキューを解決し、実行プランを動的に最適化する場合に使用します。

解決済みの問題

Spark SQL のアダプティブ実行により以下の問題を解決します。

- シャッフルパーティション数

  現在、Spark SQL の削減ステージのタスク数は **spark.sql.shuffle.partition** パラメーター (デフォルト値は 200) の値によって異なります。 ジョブにこのパラメーターが指定されると、すべてのステージにおける削減タスクの数は、ジョブが実行中のときと同じ値になります。

  ジョブごとに、また 1 つのジョブの削減ステージごとに、実際のデータサイズはかなり異なる場合があります。 たとえば、削減ステージで処理されるデータのサイズは 10 MB の場合もあれば 100 GB の場合もあります。 同じ値を使用してパラメーターを指定すると、実際の処理効率に大きな影響を与えます。 たとえば、10 MB のデータは 1 つのタスクだけで処理可能です。 **spark.sql.shuffle.partition** の値をデフォルト値の 200 に設定すると、10 MB のデータ

は 200 個のタスクによる処理用に分割されます。 これにより、スケジューリングのオーバーヘッドが増加し、処理効率が低下します。

Spark SQL のアダプティブ実行フレームワークは、シャッフルパーティション番号の範囲を設定することで、異なるジョブの異なるステージに合わせて範囲内の削減タスクの数を動的に調整できます。

これにより、最適化にかかるコストが大幅に削減されます (固定値を決める必要がありません)。 さらに、1 つのジョブのさまざまなステージにおける削減タスクの数を動的に調整できます。

パラメーター：

属性	デフォルト値	説明
spark.sql.adaptive.enabled	false	アダプティブ実行を有効または無効にします。
spark.sql.adaptive.minNumPostShufflePartitions	1	削減タスクの最小数
spark.sql.adaptive.maxNumPostShufflePartitions	500	削減タスクの最大数
spark.sql.adaptive.shuffle.targetPostShuffleInputSize	67108864	パーティションサイズをベースに削減タスクの数を動的に調整します。 たとえば、この値が 64 MB に設定されている場合、削減ステージの各タスクは 64 MB を超えるデータを処理します。
spark.sql.adaptive.shuffle.targetPostShuffleRowCount	20000000	パーティション内の行番号に基づいて、削減タスクの数を動的に調整します。 たとえば、値が 20000000 に設定されている場合、削減ステージの各タスクは 20,000,000 行を超えるデータを処理します。

- データスキュー

  データスキューは、SQL 結合操作でよく見られる問題です。 特定のタスクが処理に巻き込むデータが多すぎるというシナリオで、ロングテールにつながります。 現在、Spark SQL は、データの歪みを最適化しません。

  Spark SQL の Adaptive Execution フレームワークは、歪んだデータを自動的に検出し、実行時に最適化を実行します。

  SparkSQL は、パーティション内の歪んだデータを分割し、複数のタスクを介してデータを処理してから、SQL 結合操作を介して結果を結合することにより、データの歪みを最適化します。

  サポートされている結合の種類

データ型	説明
Inner	歪んだデータは両方のテーブルで処理できます。
Cross	歪んだデータは両方のテーブルで処理できます。
LeftSemi	歪んだデータは左のテーブルでのみ処理できます。
LeftAnti	歪んだデータは左のテーブルでのみ処理できます。
LeftOuter	歪んだデータは左のテーブルでのみ処理できます。
RightOuter	歪んだデータは右のテーブルでのみ処理できます。

  パラメーター：

属性	デフォルト値	説明
spark.sql.adaptive.enabled	false	アダプティブ実行フレームワークを有効または無効にします。
spark.sql.adaptive.skewedJoin.enabled	false	歪んだデータの処理を有効または無効にします。

属性	デフォルト値	説明
spark.sql.adaptive.skewedPartitionFactor	10	パーティションが歪んだパーティションとして識別されるのは、以下のシナリオが発生する場合に限られます。 まず、パーティションのサイズがこの値 (すべてのパーティションのサイズの中央値) とspark.sql.adaptive.skewedPartitionSizeThreshold パラメーターの値より大きい場合です。 次に、パーティション内の行数がこの値 (すべてのパーティション内の行数の中央値) およびspark.sql.adaptive.skewedPartitionSizeThreshold パラメーターの値より大きい場合も歪みが検出されます。
spark.sql.adaptive.skewedPartitionSizeThreshold	67108864	歪んだパーティションのサイズしきい値
spark.sql.adaptive.skewedPartitionRowCountThreshold	10000000	歪んだパーティションの行数しきい値
spark.shuffle.statistics.verbose	false	このパラメーターの値が true の場合、MapStatus は歪んだデータを処理するため各パーティション内の行数に関する情報を収集します。

- 実行時の実行プランの最適化

  Spark SQL の Catalyst オプティマイザは、SQL 文から物理実行プランに変換される論理プランを変換し、それらの物理実行プランを実行します。 ただし、Catalyst が作成する物理実行プランは、統計の欠如または不正確さが原因で最適ではない場合があります。 たとえば、

Spark SQL は BroadcastJoin ではなく SortMergeJoinExec を選択することはできますが、BroadcastJoin はシナリオで最適なオプションです。

Spark SQL の Adaptive Execution フレームワークは、シャッフルステージでのシャッフル書き込みのサイズをベースにクエリパフォーマンスを改善するため、SortMergeJoin ではなく BroadcastJoin を使用するかどうか決定します。

パラメーター：

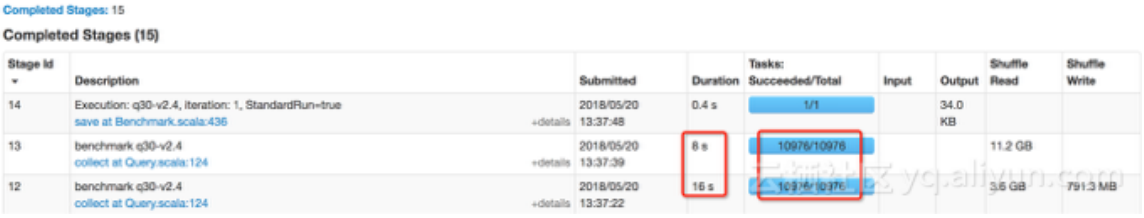
Attribute	デフォルト値	説明
spark.sql.adaptive.enabled	false	アダプティブ実行フレームワークを有効または無効にします。
spark.sql.adaptive.join.enabled	true	より良い結合戦略を実行時に決定するかどうか
spark.sql.adaptiveBroadcastJoinThreshold	spark.sql.autoBroadcastJoinThreshold と同等です。	ブロードキャスト結合を使用して結合クエリを最適化するかどうか決定します。

テスト

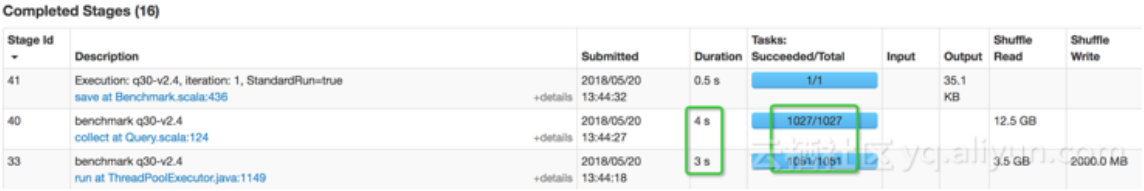
テストサンプルとして TPC-DS クエリをいくつか取ります。

- シャッフルパーティション番号
  - クエリ 30

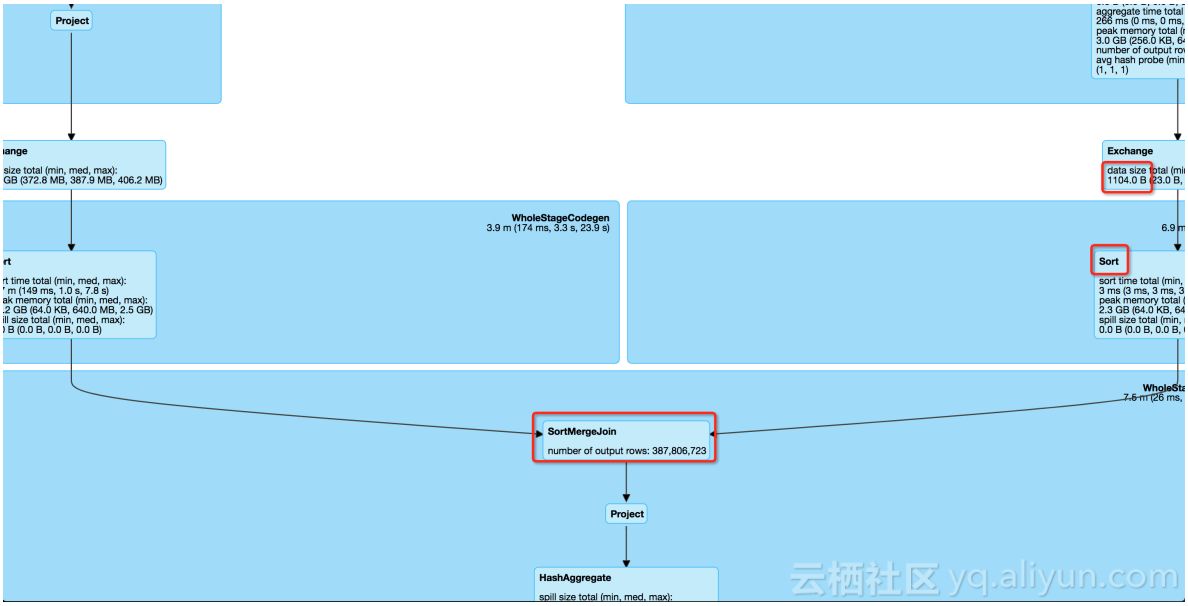
    ネイティブ Spark:



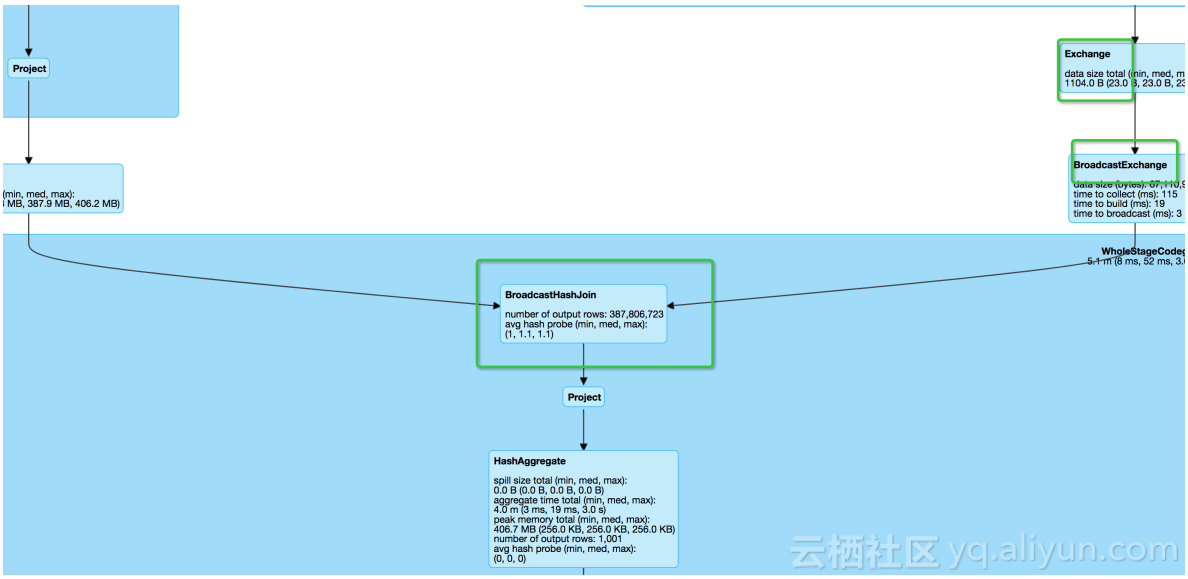
  - 削減タスクの数をアダプティブに調整します。



- 実行時の実行プランの最適化 (SortMergeJoin からBroadcastJoin へ)



アダプティブに BroadcastJoin を使用します。



2.8 Flink ジョブを使用して OSS データを処理

このページでは、E-MapReduce (EMR) を使用して Hadoop クラスターを作成し、Flink ジョブを使用してクラスター内の Object Storage Service (OSS)データを処理する方法について説明します。

- Alibaba Cloud アカウントに登録していること。 詳細については、「Alibaba Cloud アカウントの作成」をご参照ください。
- EMR と OSS をアクティブにしていること。

- Alibaba Cloud アカウントを承認していること。 詳細については、「#unique_16」をご参照ください。

実際の作業においては、OSS に保存されているデータを常に使用する必要があります。 EMRでは、Flink ジョブを実行して、OSS バケットに保存されているデータを使用できます。 EMR で Flink ジョブを作成し、Hadoop クラスターで Flink ジョブを実行し、OSS に保存されているファイルの指定されたコンテンツを取得および出力するには、以下の手順に従います。

ステップ 1：環境の準備

Flink ジョブを作成する前に、ローカルホストで Maven および Java 環境を準備し、EMR で Hadoop クラスターを作成する必要があります。 Maven 3.0 以降を使用している場合は、Java 2.0以前を使用して互換性を確保することを推奨します。

1. Maven と Java をローカルホストにインストールします。
2. EMR コンソール にログインして、Hadoop クラスターを作成します。 **[オプションサービス]** フィールドで [Flink] を選択します。 詳細については、「#unique_8」をご参照ください。

ステップ 2：テストデータの準備

Flink ジョブを作成する前に、テストデータを OSS にアップロードしなければなりません。 以下は、ファイル名 test.txt をアップロードする例です。 ファイルの内容は次のとおりです。Nothing is impossible for a willing heart. While there is a life, there is a hope.

1. OSS コンソール にログインします。
2. バケットを作成し、ファイルをバケットにアップロードします。 詳細については、「#unique_17」および「#unique_18」をご参照ください。

   アップロードされたファイルのサンプルパスは oss://emr-logs2/hengwu/test.txt です。 後で使用するためにパスを保持します。

   > 注：
   > ファイルをアップロードした後、後で使用できるように OSS ログオンウィンドウを開いたままにします。

ステップ 3：JAR ファイルの構築および OSS / Hadoop クラスターへのアップロード

EMR サンプルコード aliyun-emapreduce-demo をダウンロードし、コードをコンパイルして JAR ファイルを作成します。 JAR ファイルを Hadoop クラスターのヘッダーノードまたは OSS バケットにアップロードできます。 以下は、JARファイルを OSS バケットにアップロードする例です。

1. EMR サンプルコード aliyun-emapreduce-demo をご使用のローカルディスクにダウンロードします。
2. **mvn clean package -DskipTests** コマンドで JAR ファイルを作成します。

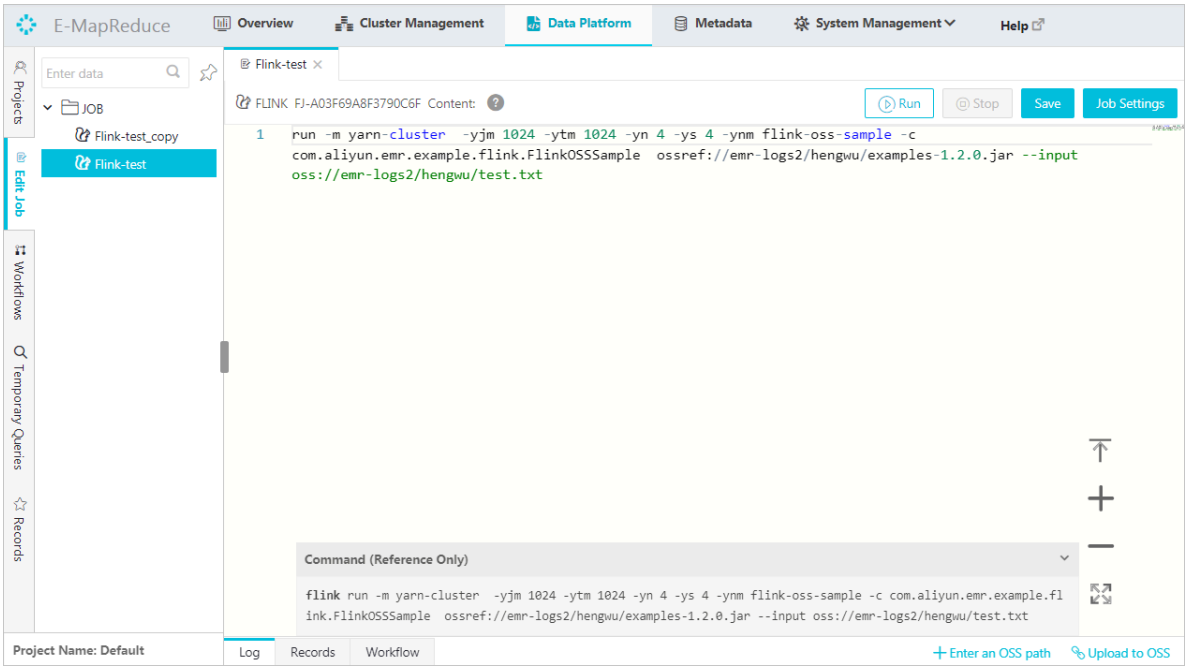
   新しい JAR ファイルは./target/ ディレクトリ (たとえば target / examples-1.2.0.jar) にインストールされます。
3. OSS コンソール を開き、JAR ファイルを OSS ディレクトリにアップロードします。

   JAR ファイルのサンプルパスは、oss#//emr-logs2/hengwu/examples-1.2.0.jar です。 後で使用するためにパスを保持します。

ステップ 4：Flink ジョブの作成および実行

1. EMR コンソール にログインします。
2. **[**データプラットフォーム**]** タブで、プロジェクトを作成します。 詳細については、「#unique_19」をご参照ください。
3. 新しいプロジェクトを開き、 **[**ジョブの編集**]** タブを選択し、 **[Flink]** タイプでジョブを作成します。

4. 新しい Flink ジョブが作成された後、ジョブの**[コンテンツ]** を指定します。



ジョブコンテンツは、コードのスニペットです。 以下に例を示します。

```
run -m yarn-cluster  -yjm 1024 -ytm 1024 -yn 4 -ys 4 -ynm flink-oss-sample -c com.aliyun.emr.example.flink.FlinkOSSSample  ossref://emr-logs2/hengwu/examples-1.2.0.jar --input oss://emr-logs2/hengwu/test.txt
```

上記のコードのパラメーターは、次のように記述されます。

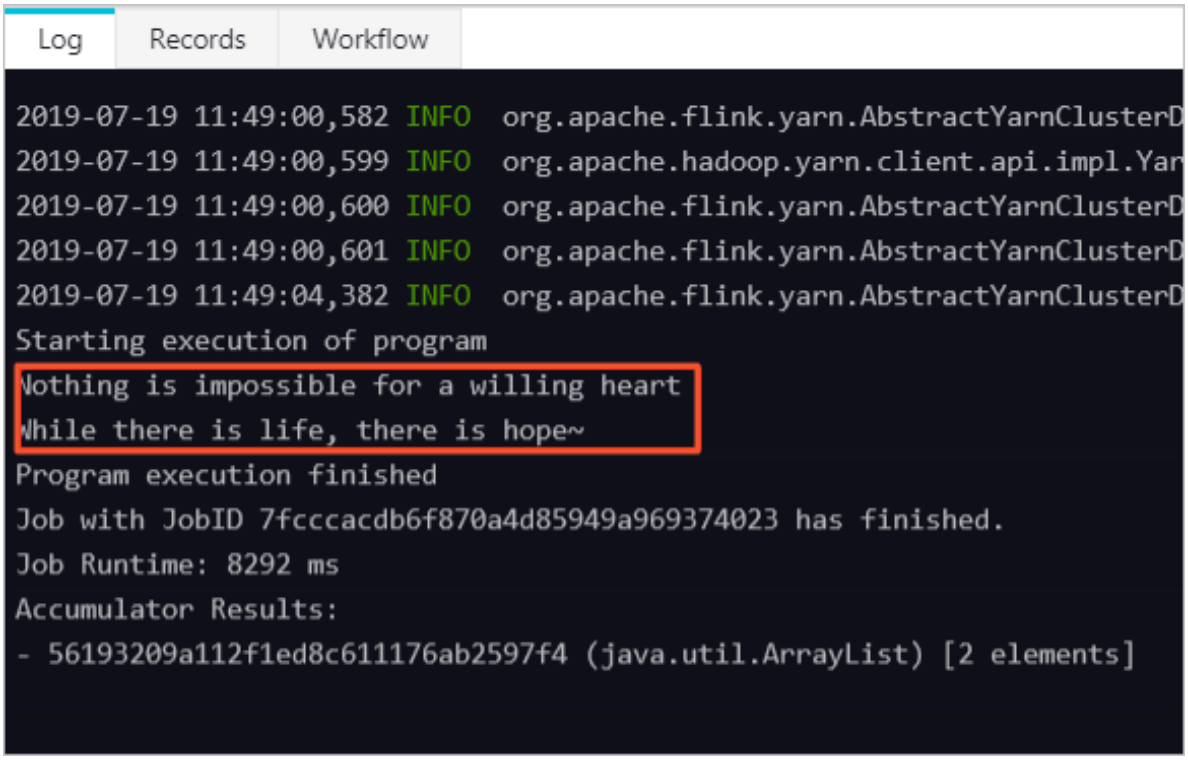
- ossref://emr-logs2/hengwu/examples-1.2.0.jar：アップロードされた JAR ファイルのパスを示します。
- oss://emr-logs2/hengwu/test.txt：テストデータのパスを示します。

> **注：**
> 各パラメータの値を ステップ 1：環境の準備 and ステップ 3：JAR ファイルの構築および OSS / Hadoop クラスターへのアップロード トピックスの設定に基づいた値に置き換えます。

5. ジョブの構成が完了したら、 右上隅の **[実行]** をクリックし、 **[ターゲットクラスター]**フィールドで新しい Hadoop クラスターの名前を選択します。

6. **[OK]** をクリックして Flink ジョブを実行します。

   ジョブの実行中、**[ログ]** ウィンドウが表示されます。 ジョブが完了すると、OSS バケットからファイルのコンテンツが取得され、ログに出力されます。 この時点で、OSS データを消費するために EMR クラスターで実行される Flink ジョブが完了しました。



ステップ 6：ログとジョブの詳細の表示

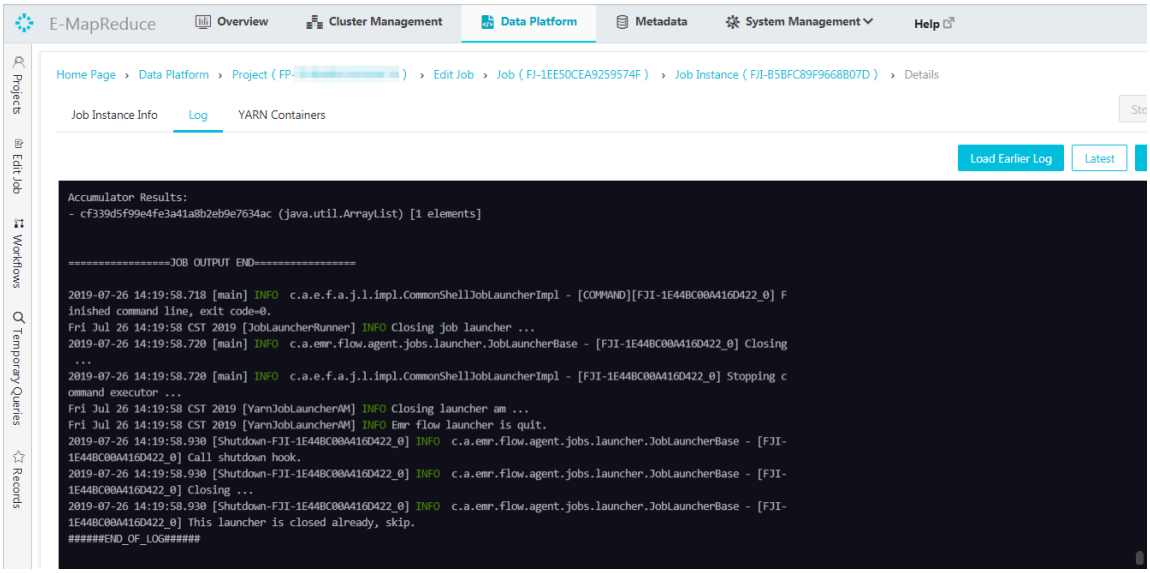
ジョブのログと詳細を表示して、ジョブの失敗の原因とジョブの詳細を特定できます。

1. ジョブのログを表示します。

   EMR コンソールまたは SSH クライアントでログを表示できます。

   - EMR コンソール にログインしてログを表示します。

     コンソールでジョブを送信すると、**[**レコード**]** タブにリストされたジョブの**[**詳細**]** ページを開けるようになります。 **[**詳細**]** ページで、ジョブの結果を表示できます。

   - SSH を使用してログを表示することにより、Hadoop クラスターのヘッダーノードにログインできます。

     デフォルトでは、Flink ジョブのログは、**log4j** ファイルの設定に基づいて /mnt/disk1/log/flink/flink-<user>-client-<hostname>.log ファイルに保存されます。 設定の詳細については、 /etc/ecm/flink-conf/log4j-yarn-session.properties ファイルをご参照ください。

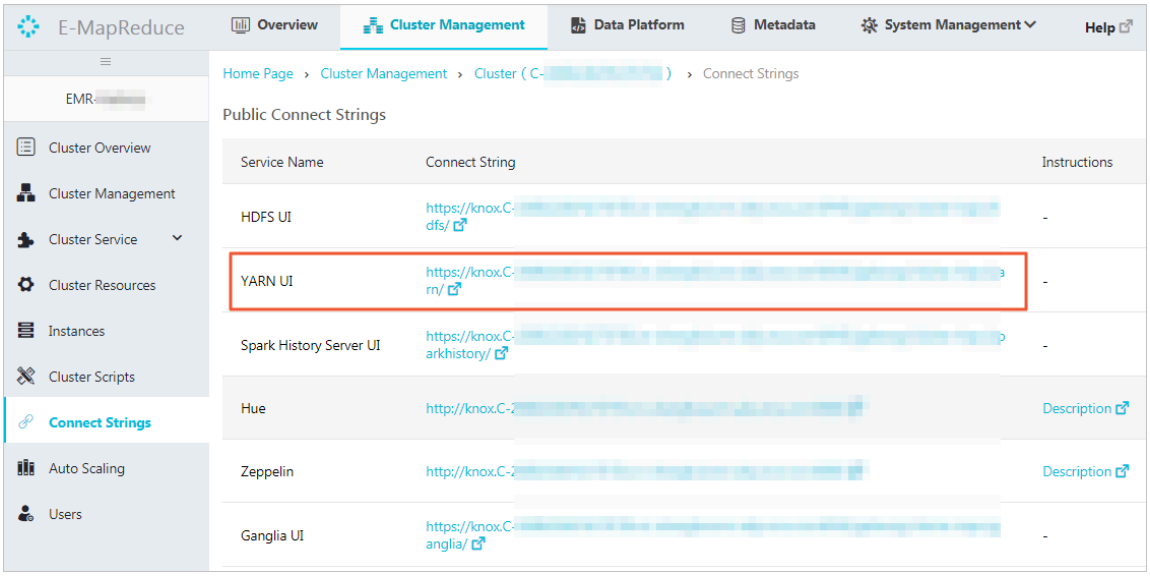
     [ユーザー**]** フィールドは、Flink ジョブを送信するアカウントを示します。 **[**ホスト名**]** フィールドは、ジョブを送信するインスタンスの名前を示します。 root ユーザーとしてログオンし、 **emr-header-1** インスタンスで Flink ジョブを送信すると仮定します。 この場合、ログパスは /mnt/disk1/log/flink/flink-flink-historyserver-0-emr-header-1.cluster-126601.log です。

2. ジョブの詳細を表示します。

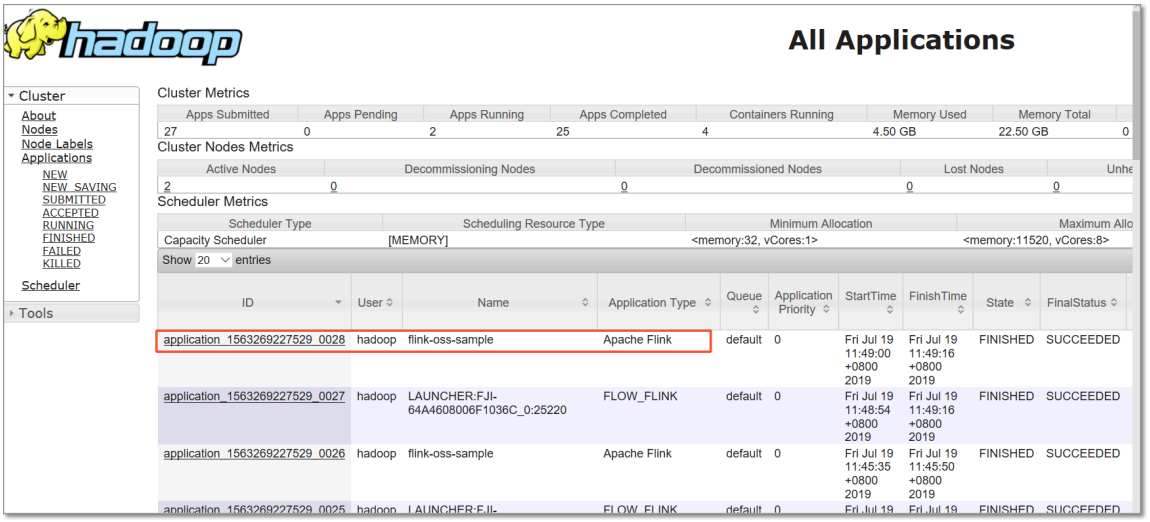
   Yarn UI を使用できます。 SSH と Knox を使って **Yarn UI** にアクセスできます。 SSH の詳細については、「#unique_20」をご参照ください。 Knox については、「#unique_21」と「

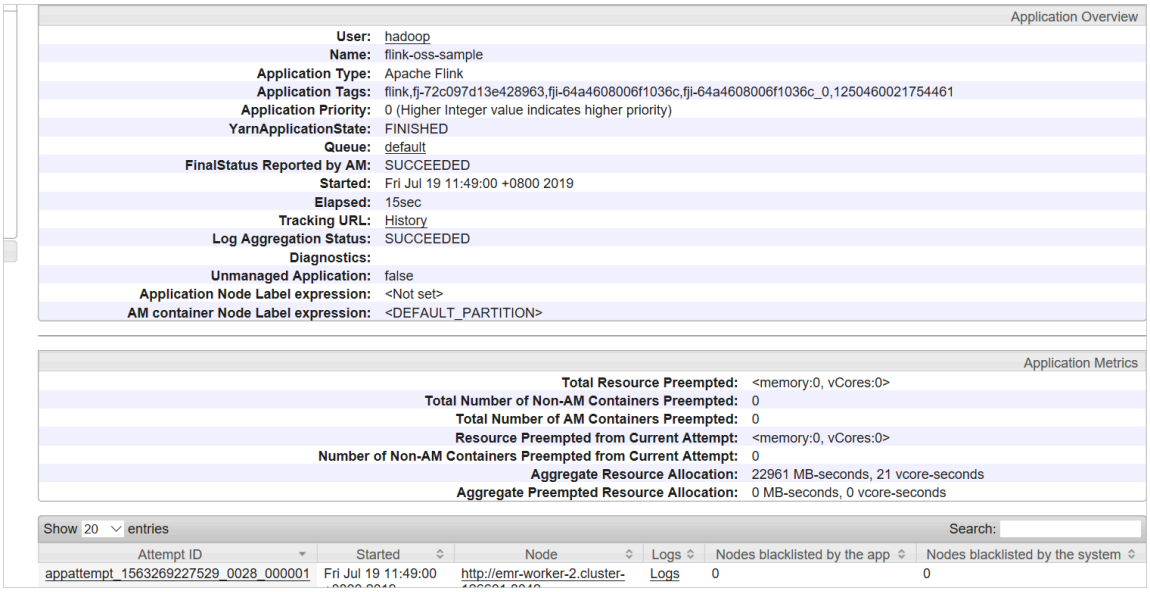
アクセスリンクとアクセスポート」をご参照ください。 以下では、Knox を使ってジョブの詳細を表示する方法を説明します。

a) Hadoop クラスターの **[接続文字列]** タブで、 **Yarn UI** の横のリンクをクリックして、Hadoop コンソールを開きます。



b) Hadoop コンソールでジョブの **ID** をクリックして、ジョブの詳細を表示します。





c) 実行中の Flink ジョブのリストを表示する必要がある場合は、[詳細] ページの **[追跡 URL]** の横にあるリンクをクリックします。 **[Flink Dashboard]** ページに実行中の Flink ジョブのリストが表示されます。

または、 http://emr-header-1:8082 にアクセスして、完了したジョブのリストを表示することもできます。

2.9 E-MapReduce Hive と ApsaraDB for HBase の接続方法

このページでは、E-MapReduce Hiveと ApsaraDB for HBase を接続する方法について説明します。 HBase テーブルの分析は Hive とApsaraDB for HBase 菅の接続に基づいています。

> 注：
> ApsaraDB for HBase は Spark へ統合されます。 その時点で Spark を使用して HBase データを分析することを推奨します。

準備

- 従量課金の EMR クラスターを購入し、実際のシナリオに基づいて設定を作成します。 注：ApsaraDB for HBase と EMR クラスターが同じ VPC 内にあることを確認してください。 クラスターは高可用性を有効にしないことを推奨します。
- EMR クラスター内のすべてのノードの IP アドレスを ApsaraDB for HBase のホワイトリストに追加します。
- ApsaraDB for HBase コンソールの Hive に組み込まれている ZooKeeper のエンドポイントを表示できます。

- ApsaraDB for HBase の HDFS ポートを開くには、チケットを起票し、サポートセンターへお問い合わせください。

手順

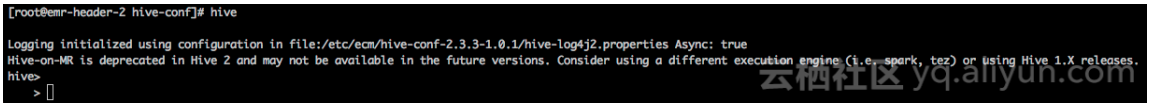
1. Hive 設定の変更
   - Hive 設定ディレクトリ /etc/ecm/hive-conf/ を開きます。
   - **hbase.zookeeper.quorum** プロパティの値をHBase に組み込まれている ZooKeeper のエンドポイントに設定して、hbase-site.xml ファイルを変更します。

```
<property>
        <name>hbase.zookeeper.quorum</name>
        <value>hb-bp1mhyea7754bpigt-001.hbase.rds.aliyuncs.com,hb-
bp1mhyea7754bpigt-002.hbase.rds.aliyuncs.com,hb-bp1mhyea7754bpigt-003.
hbase.rds.aliyuncs.com</value>
    </property>
```

2. Hive テーブルで HBase テーブルへ接続

   HBase ハンドラを使用して Hive でテーブルを作成します。 これにより、 ApsaraDB for HBase でも同じテーブルが作成されます。

   a. ハイブ コマンド ライン インターフェイス (CLI) を開始します。



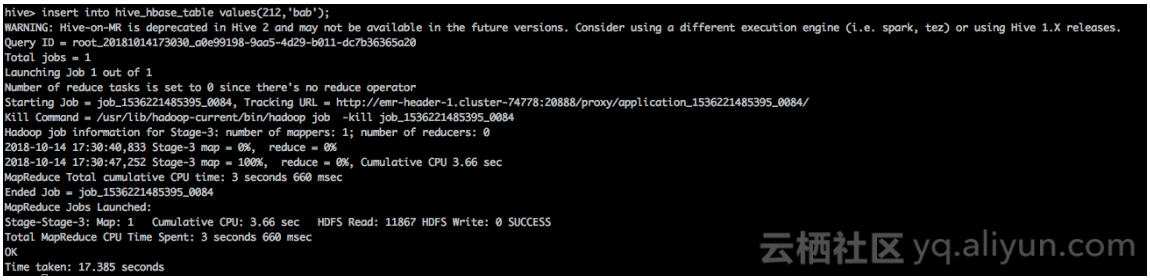
   b. 次の文を使用して、Hive でテーブルを作成します。

```
CREATE TABLE hive_hbase_table(key int, value string)
```

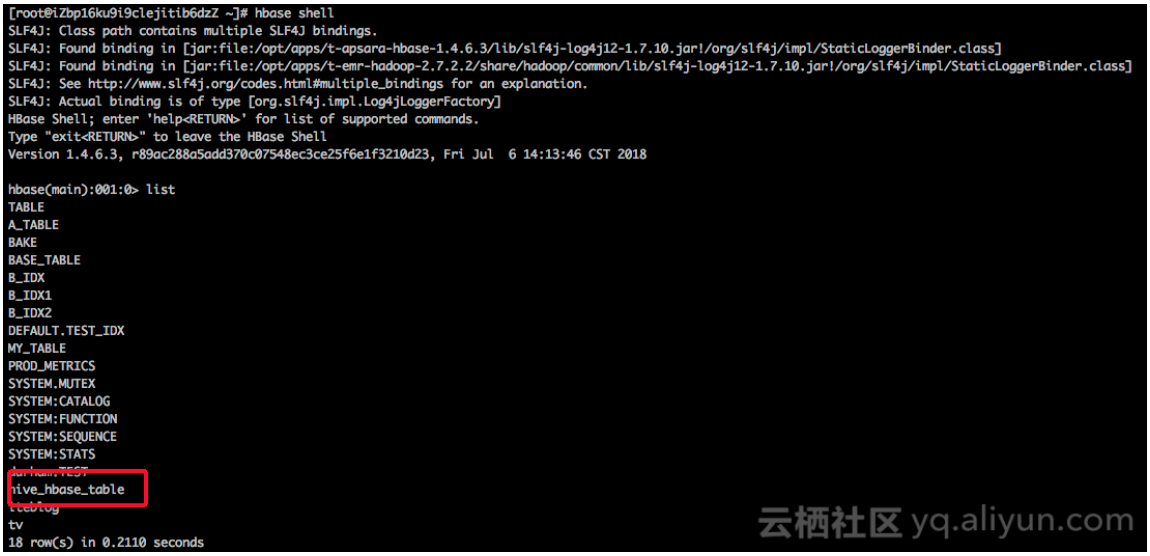
```
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
```

```
TBLPROPERTIES ("hbase.table.name" = "hive_hbase_table", "hbase.mapred.output.
outputtable" = "hive_hbase_table");
```

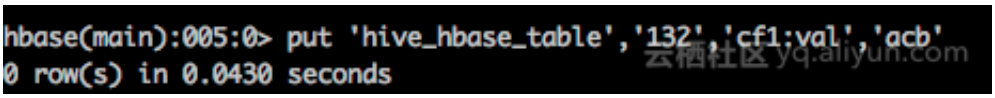
c. Hive の HBase テーブルにデータを挿入します。



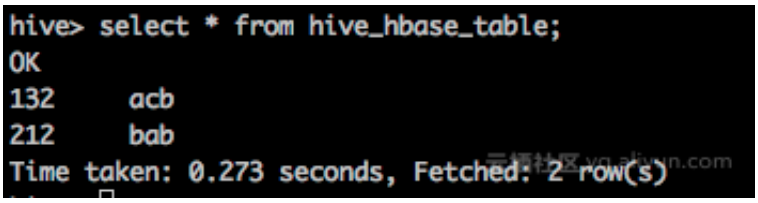
d. HBase テーブルが作成され、データがテーブルに挿入されていることを確認します。



e. put コマンドを使用して HBase テーブルにデータを書き込みます。



Hive テーブルからすべてのデータを選択します。



f. ドロップ コマンドを使用して Hive テーブルを削除します。 HBase テーブルも削除されます。後続の手順で検証されます。

```
hive>
    >
    >
    > drop table hive_hbase_table;
OK
Time taken: 6.307 seconds
```


スキャン コマンドを使用して、HBase の表の内容を表示します。 テーブルが存在しないことを示すエラー メッセージが表示されます。

```
hbase(main):008:0* scan 'hive_hbase_table'
ROW                                                    COLUMN+CELL

ERROR: Unknown table hive_hbase_table!
```


注：

既存の HBase テーブルは、Hive 外部テーブルを使用して接続できます。 Hive 外部テーブルを削除しても、対応する HBase テーブルは削除されません。

g. ApsaraDB for HBase にテーブルを作成し、put コマンドを使用してテスト データをテーブルに書き込みます。

```
hbase(main):020:0* create 'hbase_table','f'
0 row(s) in 1.3010 seconds

=> Hbase::Table - hbase_table
hbase(main):021:0> put 'hbase_table','1122','f:col1','hello'
0 row(s) in 0.0190 seconds

hbase(main):022:0> put 'hbase_table','1122','f:col2','hbase'
0 row(s) in 0.0110 seconds
```


h. HBase テーブルに接続し、HBase テーブルからすべてのデータを選択する Hive 外部テーブルを作成します。

```
hive> create external table hbase_table(key int, col1 string, col2 string)
    > STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
    > WITH SERDEPROPERTIES ("hbase.columns.mapping" = "f:col1,f:col2")
    > TBLPROPERTIES("hbase.table.name" = "hbase_table", "hbase.mapred.output.outputtable" = "hbase_table");
OK
Time taken: 0.129 seconds
hive> select * from hbase_table;
OK
1122    hello   hbase
Time taken: 0.181 seconds, Fetched: 1 row(s)
hive>
```


i. Hive 外部テーブルを削除しても、対応する HBase テーブルが削除されないことを確認します。

```
hive> drop table hbase_table;
OK
Time taken: 0.102 seconds
hive>
```



```
hbase(main):023:0> scan 'hbase_table'
ROW                                   COLUMN+CELL
 1122                                 column=f:col1, timestamp=1539510170256, value=hello
 1122                                 column=f:col2, timestamp=1539510181752, value=hbase
1 row(s) in 0.0160 seconds
```


まとめ

Hive を使用した HBase の操作については「HBase の統合」をご参照ください。 このページで説明した操作は、Alibaba Cloud EMR クラスターにインストールされた Hive に基づいています。 Hive が ECS インスタンスのカスタム MapReduce クラスターにインストールされている場合も、同様の操作になります。 注：Hive の設定ファイル hbase-site.xml は ApsaraDB for HBase の設定ファイルとは異なる場合があります。 Hive を使用して ApsaraDB for HBase へ接続するには、**hbase.zookeeper.quorum** プロパティの設定のみ必要です。

2.10 EMR を使用して MySQL binlog をリアルタイムで伝送

このセクションでは、Alibaba Cloud の SLS プラグイン機能と E-MapReduce クラスターを使用して MySQL binlog の準リアルタイム伝送を実装する方法について説明します。

基本アーキテクチャ

RDS -> SLS -> Spark Streaming -> Spark HDFS

上記のリンクには、3 つのプロセスがあります。

1. SLS に RDS binlog を集める方法
2. Spark Streaming を介して SLS のログを読み取り分析する方法
3. Spark HDFS への 2 番目のリンクで読み取られ処理されるログを保存する方法

環境の準備

1. MySQL データベースをインストールし (RDS および DRDS などの MySQL プロトコルを使用)、log-bin 機能を有効にします。 binlog タイプを ROW モードに設定します。 (RDS はデフォルトで有効になっています)。
2. SLS サービスを有効にします。

操作手順

1. MySQL データベース環境をチェックします。

   a. log-bin 機能が有効になっているかどうか閲覧します。

   ```
   mysql> show variables like "log_bin";
   + ---------------- + ------- + ------ +
   | Variable_name | Value |
   + ---------------- + ------- + ------ +
   | log_bin      | ON    |
   + ---------------- + ------- + ------ +
   1 row in set (0.02 sec)
   ```

   b. binlog の種類を閲覧します。

   ```
   mysql> show variables like "binlog_format";
   + ---------------- + ------- + ------ +
   | Variable_name | Value |
   + ---------------- + ------- + ------ +
   | binlog_format | ROW   |
   + ---------------- + ------- + ------ +
   1 row in set (0.03 sec)
   ```

2. ユーザー権限を追加します。 RDS コンソールから直接ユーザー権限を追加することもできます。

   ```
   CREATE USER canal IDENTIFIED BY 'canal';
   GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON *. * TO 'canal'@'%';
   ```

```
FLUSH PRIVILEGES;
```

3. SLS サービス用の対応する設定ファイルを追加して、データが正しく収集されているかどうかチェックします。

   a. 対応するプロジェクトとログストアを SLS コンソールに追加します。 たとえば、canaltest という名前のプロジェクトと canal という名前のログストアを作成します。

   b. SLS 設定：/etc/ilogtail のディレクトリの下に user_local_config.json というファイルを作成します。

```
{
"metrics": {
  "##1.0##canaltest$plugin-local": {
     "aliuid": "****",
     "enable": true,
     "category": "canal",
     "defaultEndpoint": "********",
     "project_name": "canaltest",
     "region": "cn-hangzhou",
     "version": 2
     "log_type": "plugin",
     "plugin": {
       "inputs": [
         {
           "type": "service_canal",
           "detail": {
             "Host": "******",
             "Password": "****",
             "ServerID": ****,
             "User" : "***",
             "DataBases": [
               "yourdb"
             ],
             "IgnoreTables": [
               "\\S+_inner"
             ],
              "TextToString" : true
           }
         }
       ],
       "flushers": [
         {
           "type": "flusher_sls",
           "detail": {}
         }
       ]
     }
  }
}
}
```

   ホストおよびパスワードなどの詳細な情報は MySQL データベース情報で、ユーザー情報は以前に許可されたユーザー名です。 AliUid、defaultEndpoint、project_name、および

category は、ユーザーと SLS に関連する情報です。 実際の状況に応じて情報を記入します。

c. ログデータが SLS コンソールに正常にアップロードされたかどうかを確認するため約 2 分待ちます。

ログデータの取得が正常に完了していない場合は、トラブルシューティングのプロンプトに従って SLS の取得ログを閲覧します。

4. コードを準備して jar パッケージにコンパイルし、それを OSS にアップロードします。

a. Git を使用して EMR のサンプルコードをコピーし、コードを変更します。 コマンドは以下のとおりです。 `git clone https://github.com/aliyun/aliyun-emapreduce-demo.git` サンプルコードには LoghubSample クラスが含まれており、主に SLS からデータを取得して印刷する場合に使用します。 変更されるコードは以下のとおりです。

```
package com.aliyun.emr.example
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.aliyun.logservice.LoghubUtils
import org.apache.spark.streaming.{ Milliseconds, StreamingContext}
object LoghubSample {
def main(args: Array[String]): Unit = {
if (args.length < 7) {
 System.err.println(
   """Usage: bin/spark-submit --class LoghubSample examples-1.0-SNAPSHOT-
shaded.jar
     |
     |
   """.stripMargin)
 System.exit(1)
}
val loghubProject = args(0)
val logStore = args(1)
val loghubGroupName = args(2)
val endpoint = args(3)
val accessKeyId = args(4)
val accessKeySecret = args(5)
val batchInterval = Milliseconds(args(6).toInt * 1000)
val conf = new SparkConf().setAppName("Mysql Sync")
//    conf.setMaster("local[4]");
val ssc = new StreamingContext(conf, batchInterval)
val loghubStream = LoghubUtils.createStream(
 ssc,
 loghubProject,
 logStore,
 loghubGroupName,
 qendpoint,
 1,
 accessKeyId,
 accessKeySecret,
 StorageLevel.MEMORY_AND_DISK)
loghubStream.foreachRDD(rdd =>
   rdd.saveAsTextFile("/mysqlbinlog")
)
ssc.start()
ssc.awaitTermination()
```

```
}
}
```

主な変更点は以下のとおりです。 loghubStream.foreachRDD(rdd => rdd.saveAsObjectFile("/mysqlbinlog")) サンプルコードが EMR クラスターで実行されると、Spark Streaming から流出するデータは EMR の HDFS に保存されます。

注 :

- サンプルコードをローカルで実行するには、事前にローカル環境に Hadoop クラスターを作成します。
- EMR の Spark SDK が更新されているため、そのコード例は古く、OSS の AccessKey ID と AccessKey Secret をこのパラメーターで直接転送することはできません。 Spark SDK を SparkConf コンストラクタで設定する必要があります (以下の図を参照)。

```
trait RunLocally {
val conf = new SparkConf().setAppName(getAppName).setMaster("local[4]")
conf.set("spark.hadoop.fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssF
ileSystem")
conf.set("spark.hadoop.mapreduce.job.run-local", "true")
conf.set("spark.hadoop.fs.oss.endpoint", "YourEndpoint")
conf.set("spark.hadoop.fs.oss.accessKeyId", "YourId")
conf.set("spark.hadoop.fs.oss.accessKeySecret", "YourSecret")
conf.set("spark.hadoop.job.runlocal", "true")
conf.set("spark.hadoop.fs.oss.impl", "com.aliyun.fs.oss.nat.NativeOssF
ileSystem")
conf.set("spark.hadoop.fs.oss.buffer.dirs", "/mnt/disk1")
val sc = new SparkContext(conf)
def getAppName: String
}
```

- ローカルデバッグ中は、rdd.saveAsObjectFile（ "/ mysqlbinlog"）の /mysqlbinlogloghubStream.foreachRDD(rdd =>）をローカルの HDFS アドレスに変更する必要があります。

b. コードをコンパイルします。

ローカルデバッグが完了したら、以下のコマンドを実行してコードをパッケージ化しコンパイルできます。

```
mvn clean install
```

c. jar パッケージをアップロードします。

バケットが qiaozhou-EMR/jar の OSS インスタンスにディレクトリを作成し、OSS コンソールまたは OSS の SDK を介して /target/shaded ディレクトリの examples-1.1-shaded.jar を OSS ディレクトリにアップロードします。 アップロードされる jar パッケー

ジのアドレスは oss://qiaozhou-EMR/jar/examples-1.1-shaded.jar です。 このアドレスは後ほど使用されます。

5. EMR クラスターとタスクを作成し、実行プランを実行します。

   a. EMR コンソールで EMR クラスターを作成します。約 10 分かかります。

   b. Spark タイプのジョブを作成します。

      SLS_endpointSLS_access_idSLS_secret_key を実際の値に置き換えます。 パラメーターの順序が正しいことを確認します。 順序が正しくないとエラーが報告される可能性があります。

      ```
      --master yarn --deploy-mode client --driver-memory 4g --executor-memory 2g --executor-cores 2 --class com.aliyun.EMR.example.LoghubSample ossref://EMR-test/jar/examples-1.1-shaded.jar canaltest canal sparkstreaming $SLS_endpoint $SLS_access_id $SLS_secret_key 1
      ```

   c. 実行プランが作成されたら、ジョブを EMR クラスターにバインドします。 ジョブの実行を開始します。

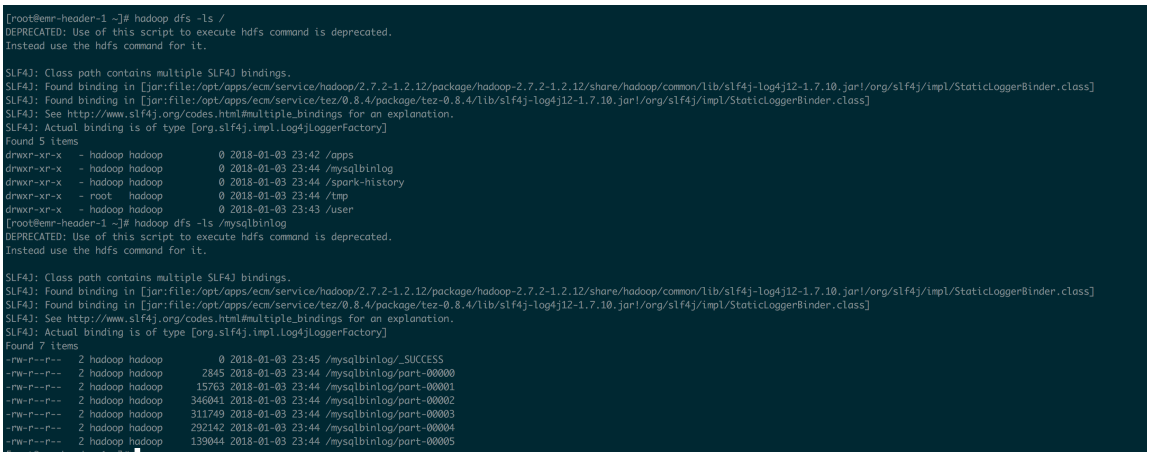
   d. マスターノードの IP アドレスを検索します。

      SSH でログイン後、以下のコマンドを実行します。

      ```
      hadoop fs -ls /
      ```

      mysqlbinlog の先頭にディレクトリが表示され、以下のコマンドで mysqlbinlog ファイルを閲覧できます。

      ```
      hadoop fs -ls /mysqlbinlog
      ```

      hadoop fs -cat /mysqlbinlog/part-00000 コマンドを実行してファイルの内容を閲覧することもできます。

6. トラブルシューティングします。

   正常な結果が表示されない場合は、EMR の実行記録にある問題をトラブルシューティングできます。

2.11 Gateway ノードで Flume を実行してデータを同期する方法

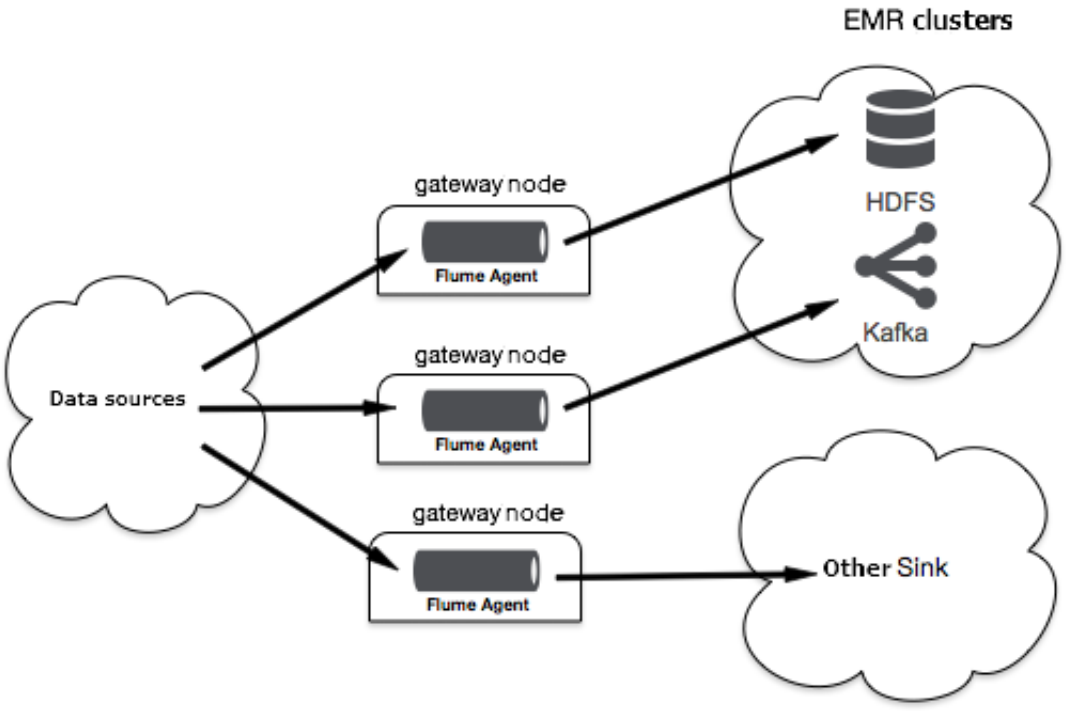
このページでは、Gateway ノードで Flume を実行して、Alibaba Cloud E-MapReduce (EMR) V3.17.0 以降のバージョンに基づいてデータを同期する方法について説明します。

このタスクについて

EMR は V3.16.0 以降の Apache Flume をサポートしており、V3.17.0 以降はデフォルトの監視をサポートしています。

- 基本データフロー

  Gateway ノードで Flume を実行すると、EMR Hadoop クラスターへの影響を回避できます。以下は、Gateway ノードにインストールされている Flume エージェントを介してストリーミングされる基本データフローを表しています。



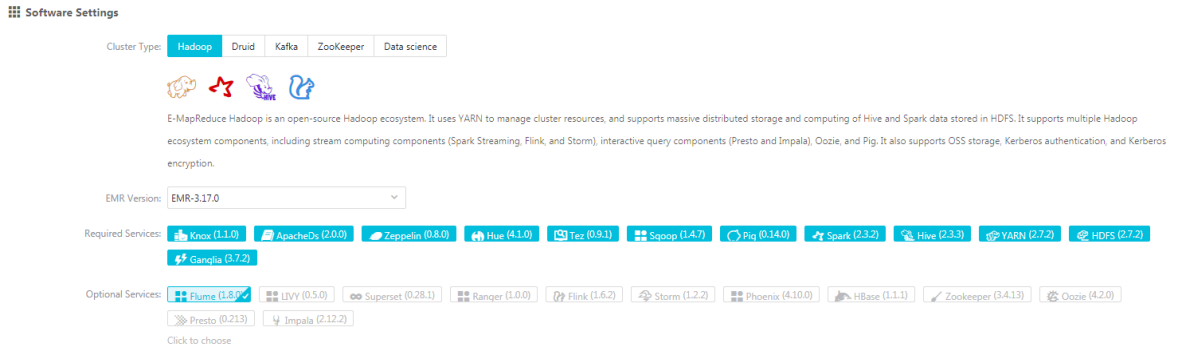
環境の準備

テストは、杭州 (中国東部 1) リージョンに配備されているEMR を使用して実行されます。 EMR のバージョンは V3.17.0 です。 このテストに必要なコンポーネントは次のとおりです。

- Flume: 1.8.0

EMR を使用して、Hadoop クラスターを自動的に作成できます。 詳細については、「クラスターの作成」をご参照ください。

- [クラスターの作成]、次に [クラスタータイプの Flume] をクリックし、**[オプションサービス]** から Flume を選択します。



- Gateway ノードを作成し、さきほど作成した Hadoop クラスターに関連付けます。

設定手順

- Flume を実行
  - The default path of Flume configuration files is /etc/ecm/flume-conf. Flume エージェントの設定ファイル flume.properties の変更については「#unique_25」をご参照ください。 変更後、次のコマンドを使用して Flume エージェントを実行します。

    ```
    nohup flume-ng agent -n a1 -f flume.properties &
    ```

  - -c フラグまたは --conf フラグを使ってデフォルト設定をカスタムファイルに置き換えることができます。 例：

    ```
    nohup flume-ng agent -n a1 -f flume.properties -c path-to-flume-conf &
    ```

  :

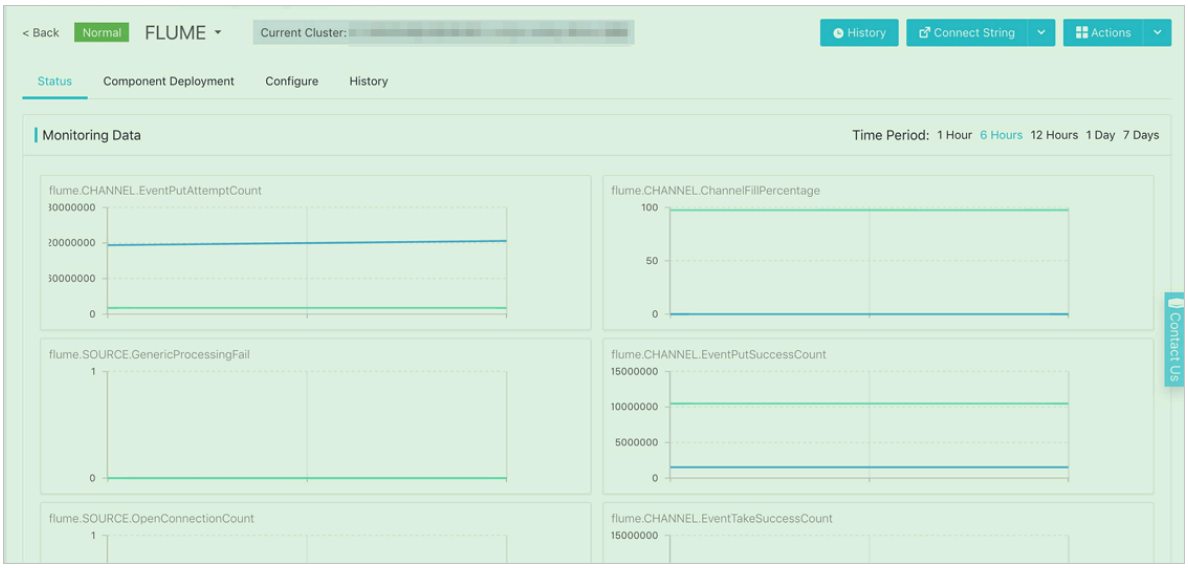
  詳細については「#unique_25」をご参照ください。 Gateways にインストールされた Flume エージェントがデータを HBase に書き込むためにシンクを使用する場合、**zookeeperQuorum** 設定項目を flume.properties 設定ファイルに追加する必要があります。 例：

  ```
  a1.sinks.k1.zookeeperQuorum=emr-header-1.cluster-46349:2181
  ```

  The hostname of the ZooKeeper クラスター emr-header-1.cluster-46349 のホスト名は / etc/ecm/hbase-conf/hbase-site.xml ファイルの **hbase.zookeeper.quorum** 設定項目の値です。

- 監視情報の表示

  Flume エージェントの監視データは、デフォルトでクラスターコンソールに表示されます。[クラスターとサービス] ページで、**[FLUME]** をクリックして以下の図のようにクラスターコンソールにジャンプします。



  :

  監視データは、Flume エージェントのコンポーネント(ソース、チャネル、シンク) で分類されます。 例えば、CHANNEL.channel1 は、channel1 チャネルコンポーネントの監視データを表します。 注：異なるエージェントを構成する場合は、同じコンポーネント名の使用は避けてください。

  適切な構成を作成し、Gangliaを使用してFlume エージェントの監視データを表示するには、公式 Flume ウェブサイトをご参照ください。 この操作を行うと、Flume エージェントの監視データはコンソールに表示されません。

- ログの表示

  デフォルトでは、Flume エージェントのログパスは /mnt/disk1/log/flume/${flume-agent-name}/flume.log です。 /etc/ecm/flume-conf/log4j.properties 設定ファイルを変更してログパスを変更できます。 ただし、デフォルトのログパスは変更しないことを推奨します。

  :

  ログパスには、Flume エージェント名が含まれています。 異なるエージェントのログが同じディレクトリに格納されないように、各エージェントには一意の名前を付けます。

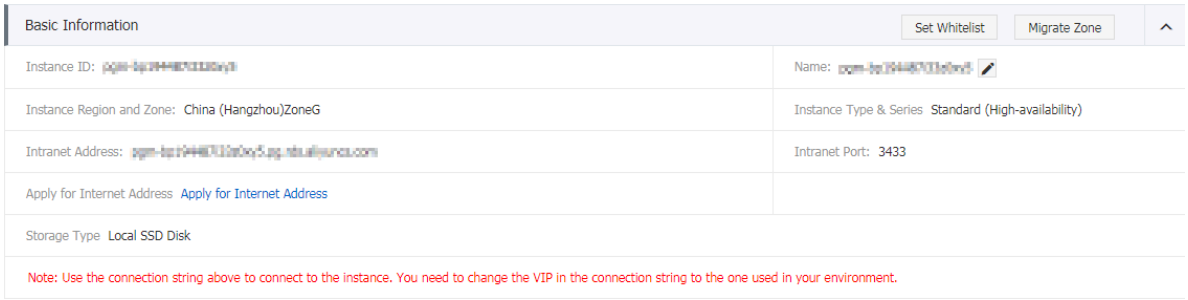
2.12 Sqoop を使用してデータベースから EMR クラスターにデータを転送するためのネットワーク接続を設定する

外部データベースから EMR クラスターにデータを転送する必要がある場合は、ネットワークが接続されていることを確認します。 本ページでは、RDS インスタンスの ApsaraDB、ECS でホストされているユーザー作成データベース、オンプレミスデータベースにアクセスするためのネットワーク接続を設定する方法について説明します。

ApsaraDB for RDS

- クラシックネットワーク

  クラシックネットワークの RDS インスタンスにアクセスする場合、クラシックネットワークにデプロイされた EMR クラスターを使用することを推奨します。 クラシックネットワークの RDS インスタンスには、内部 IP アドレスとパブリック IP アドレスを設定できます。 Sqoop は、マスターノードとワーカーノードでマップタスクを実行してデータを同期します。 ただし、クラシックネットワークの EMR クラスターからパブリックネットワークにアクセスできるのは、マスターノードに限られます。 Sqoop の RDS インスタンスの内部 IP アドレスにアクセスする必要があります。 EMR クラスターの内部 IP アドレスが RDS インスタンスのホワイトリストに含まれていることを確認します。

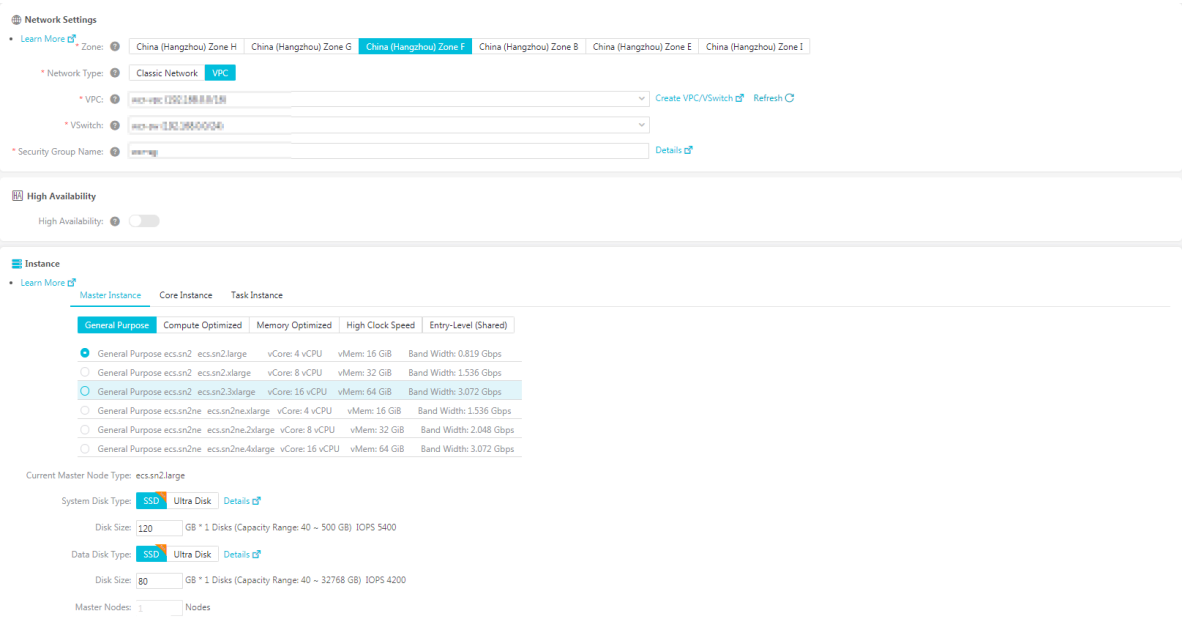
  

  クラシックネットワークの EMR クラスターを作成する方法の詳細については、「クラスターの作成」をご参照ください。

- VPC

  RDS が VPC ネットワーク内にある場合、EMR クラスターの VPC ネットワークを指定する必要があります。 ネットワーク接続をすばやく設定するために、EMR クラスターと RDS インスタ

ンスで同じ VPC を使用することを推奨します。 VPC が異なる場合は、「Express Connect」を使用して、ネットワーク接続を設定します。



ECS ベースのユーザーが作成したデータベース

- クラシックネットワーク

  クラシックネットワークのユーザーが作成したデータベースと RDS インスタンスにアクセスするプロセスは似ています。 EMR クラスターのクラシックネットワークを使用して、ユーザーが作成したデータベースの内部 IP アドレスにアクセスします。 データベースが展開されている ECS インスタンスと EMR クラスターのインスタンスが同じセキュリティグループに入っていることを確認します。 ECS コンソールで**[セキュリティグループ] > [インスタンスの管理] > [インスタンスの追加]**を選択します。
- VPC

  VPC のユーザーが作成したデータベースと VPC RDS インスタンスにアクセスするプロセスは似ています。 EMR クラスターに VPC を使用します。 データベースがデプロイされている ECS インスタンスと EMR クラスターが同じセキュリティグループに入っていることを確認します。

オンプレミスデータベース

EMR クラスターに EIP アドレスを割り当て、データベースのパブリック IP アドレスにアクセスできます。 または、Express Connect を使用して VPC を接続し、データベースにアクセスできます。

- EIP を関連付ける

  パブリックネットワーク経由でオンプレミスデータベースにアクセスできる場合は、VPC EMR クラスターを使用することを推奨します。 VPC に EMR クラスターを作成する 次に、ECS コンソールで choose **[管理] > [設定情報] > [詳細] > [EIP のバインド]** を選択し、 EIP を各 ECS インスタンスに関連付けます。 設定後、ご利用のクラスターはオンプレミスデータベースのパブリック IP アドレスにアクセスできます。

- Express Connect

  オンプレミスデータベースへのパブリックネットワーク経由のアクセスが許可されていない場合は、VPC に EMR クラスターを作成し、Express Connect を使用してオンプレミス IDC と VPC を接続します。 Express Connect の詳細については、「Express Connect」をご参照ください。

2.13 E-MapReduce を使用して Spark Streaming ジョブを送信し Kafka データを消費

このページでは、E-MapReduce (EMR) を使用して Hadoop クラスターと Kafka クラスターを作成し、Spark Streaming ジョブを実行して Kafka データを消費する方法について説明します。

- Alibaba Cloud アカウントに登録していること。 詳細については、「Alibaba Cloud アカウントの準備」をご参照ください。
- EMR をアクティブにしました。
- Alibaba Cloud アカウントを承認しました。 詳細については、「#unique_16」をご参照ください。

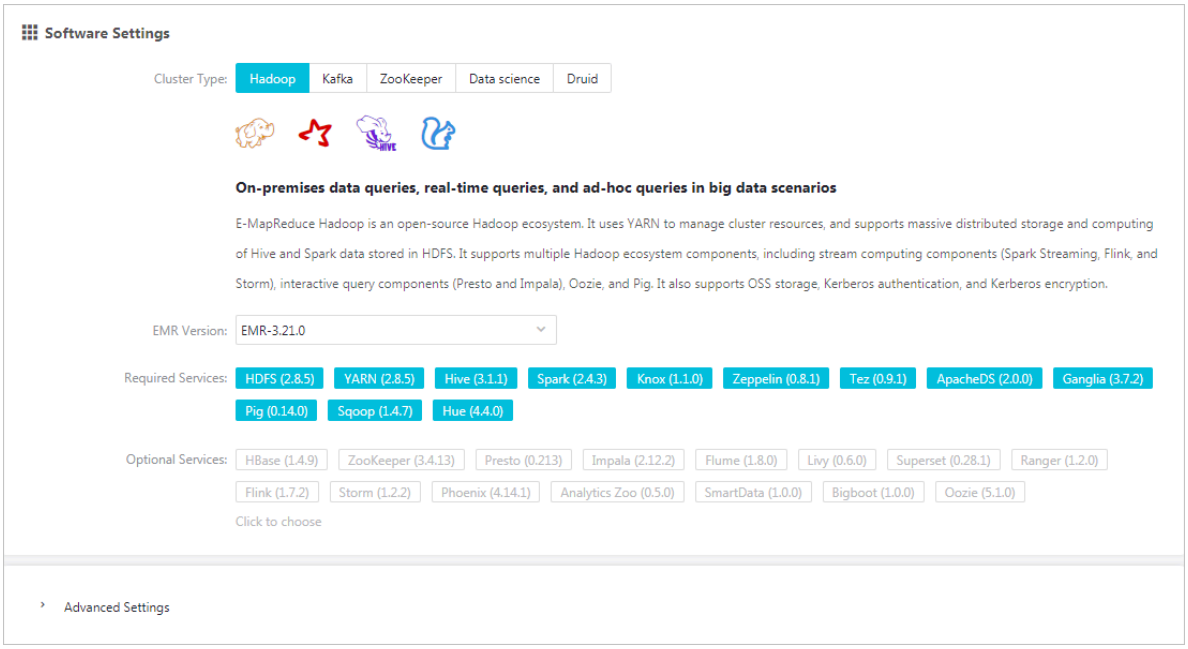
実際のアプリケーションでは、常に Kafka データを使用します。 EMR では、Spark Streaming ジョブを実行して、Kafka データを消費できます。

ステップ 1：Hadoop クラスターと Kafka クラスターの作成

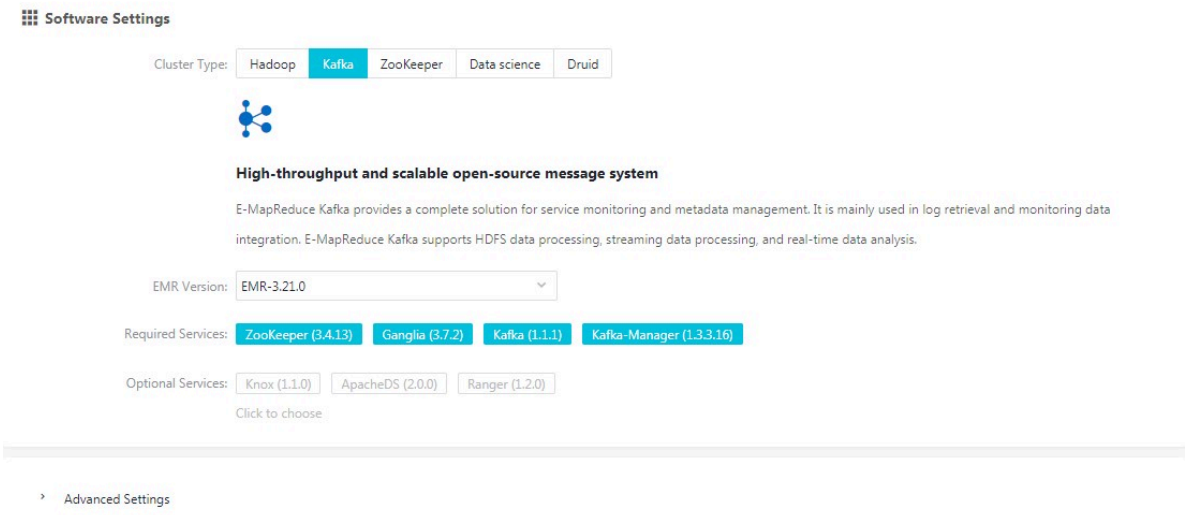
2 つのクラスターを作成する場合、Hadoop クラスターに Kafka クラスターと同じセキュリティグループを指定することを推奨します。 クラスターが異なるセキュリティグループにリンクされている場合、2 つのクラスターは相互にアクセスできません。 相互アクセスを許可するには、セキュリティグループで必要とされる設定を変更しなければなりません。

1. Alibaba Cloud ECS コンソール にログインします。

2. Hadoop クラスターの作成 詳細はについては、「クラスターの作成 」をご参照ください。



3. Kafka クラスターの作成 詳細については、「クラスターの作成」をご参照ください。



ステップ 2：JAR ファイルのダウンロードおよび Hadoop クラスターへのアップロード

この例では、、Demo プロジェクトをカスタマイズおよびコンパイルして新しい JAR ファイルを作成します。 JAR ファイルを Hadoop クラスターの **emr-header-1** インスタンスにアップロードする必要があります。

1. こちら から JAR ファイルをダウンロードします。
2. Alibaba Cloud ECS コンソール にログインします。
3. **[**クラスター管理**]** タブで、ターゲットクラスターの **[**クラスター **ID]** をクリックして Hadoop クラスターを入力します。

4. 左側のナビゲーションウィンドウで **[インスタンス]** を選択し、Hadoop クラスターの **emr-header-1** インスタンスの**IP** アドレス を表示します。
5. SSH を使って **emr-header-1** インスタンスにログインします。
6. JARファイルを **emr-header-1** インスタンスのディレクトリにアップロードします。

> 注：
> この / home / hadoop ディレクトリは、この例ではデータストレージのリポジトリとして指定されます。 JAR ファイルをアップロードした後、後で使用できるようにログインウィンドウを開いたままにすることを推奨します。

ステップ 3：Kafka クラスターでのトピックの作成

ECS コンソールで DDH を作成できます。 詳細については、「#unique_29」をご参照ください。 また、**emr-header-1** インスタンスにログインし、CLI を使ってトピックを作成できます。 この例では、10 個のパーティション、2 個のレプリカを持つ test という名前のトピックを作成できます。

1. Alibaba Cloud EMR コンソール へ移動します。
2. **[クラスター管理]** タブで、目的とする Kafka クラスターの **[クラスター ID]** をクリックして、クラスターの詳細ページを開きます。
3. 左側のナビゲーションペインで、**[インスタンス]** を選択し、Kafka クラスターの**emr-header-1** インスタンスの **IP address** を表示します。
4. SSH クライアントで新しいシェルを開き、 新しいシェルの **emr-header-1** にログインします。
5. 以下のコマンドを実行し、Job を作成します。

```
/usr/lib/kafka-current/bin/kafka-topics.sh --partitions 10 --replication-factor 2 --zookeeper emr-header-1:/kafka-1.0.0 --topic test --create
```

> 注：
> トピックを作成したら、後で使用できるようにこのログイン画面を開いたままにすることを推奨します。

ステップ 4：Spark Streaming ジョブの実行

上記の手順を実行した後、Hadoop クラスターでSpark Streaming ジョブを実行できます。 以下は、データストリームのワード数を計算するジョブを実行する例です。

1. Hadoop クラスターの **emr-header-1** インスタンスのログイン画面に移動します。

   一度ウィンドウを閉じると、再度ログインする必要があります。 その手順については、「ステップ 2：JAR ファイルのダウンロードおよび Hadoop クラスターへのアップロード」をご参照ください。

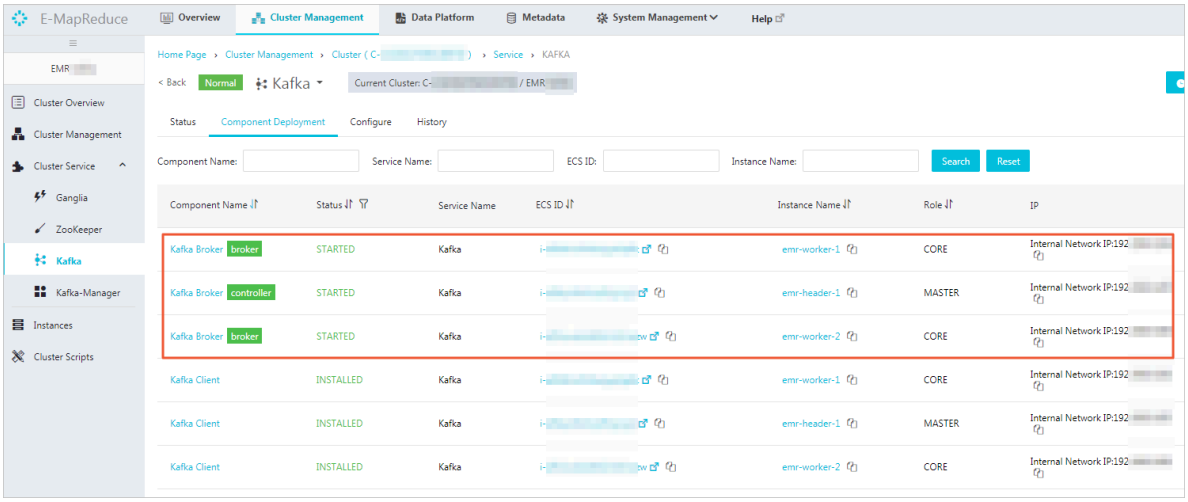
2. カウントのために、次のコマンドを使用して、ジョブを Kafka クラスターに送信します。

   ```
   spark-submit --class com.aliyun.emr.example.spark.streaming.KafkaSample  /home/hadoop/examples-1.2.0-shaded-2.jar 192.168.xxx.xxx:9092 test 5
   ```

   上記のコマンドでは、JAR ファイル名の後のパラメータは次のように説明されています。

   - 192.168.xxx.xxx：Kafka クラスターの ブローカー の内部またはパブリック IP を示します。図 2-1：Kafka クラスターのコンポーネントのリスト が例です。
   - test：トピックの名前を示します。
   - 5：時間間隔を示します。

   図 2-1：Kafka クラスターのコンポーネントのリスト



ステップ 5：Kafkaを使用したメッセージの公開

この手順を実行するときは、Spark Streaming ジョブが実行されていることを確認してください。 Kafka プロデューサーを開始すると、Hadoop クラスターのクライアントインスタンスのシェルに単語数が表示されます。 Kafka クラスターのクライアントインスタンスのシェルに単語を入力すると、値はリアルタイムで更新されます。

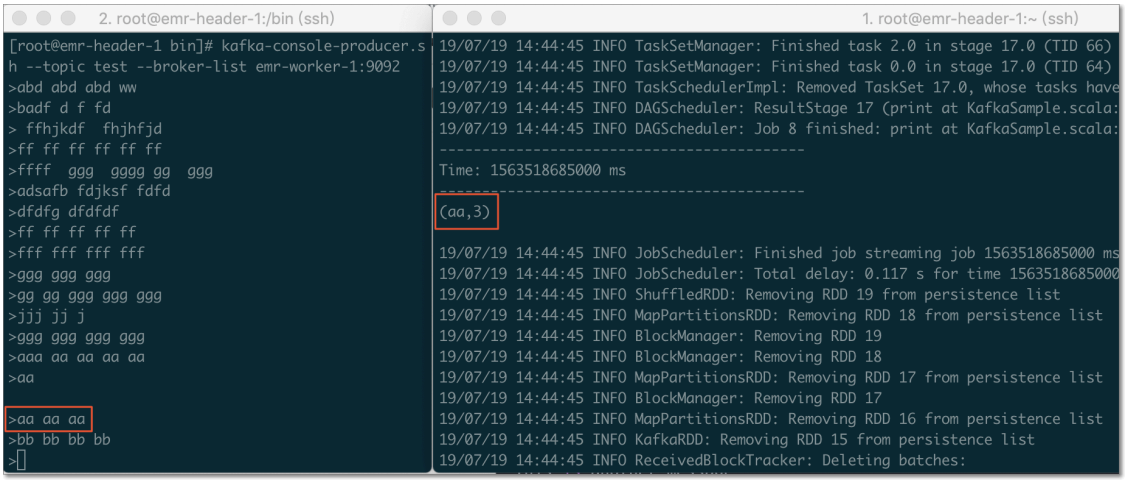
1. **emr-header-1** インスタンスのログイン画面に移動します。

   ウィンドウを閉じる場合は、再度ログオンする必要があります。 ログインの手順については、「ステップ 3：Kafka クラスターでのトピックの作成」をご参照ください。

2. Kafka クラスターのクライアントインスタンスのログイン画面で、次のコマンドを使用してプロデューサーを起動します。

```
/usr/lib/kafka-current/ /bin/kafka-console-producer.sh --topic test --broker-list emr-worker-1:9092
```

3. Kafka のログイン画面に単語を入力すると、単語の数が表示され、Hadoop のログイン画面にリアルタイムで更新されます。

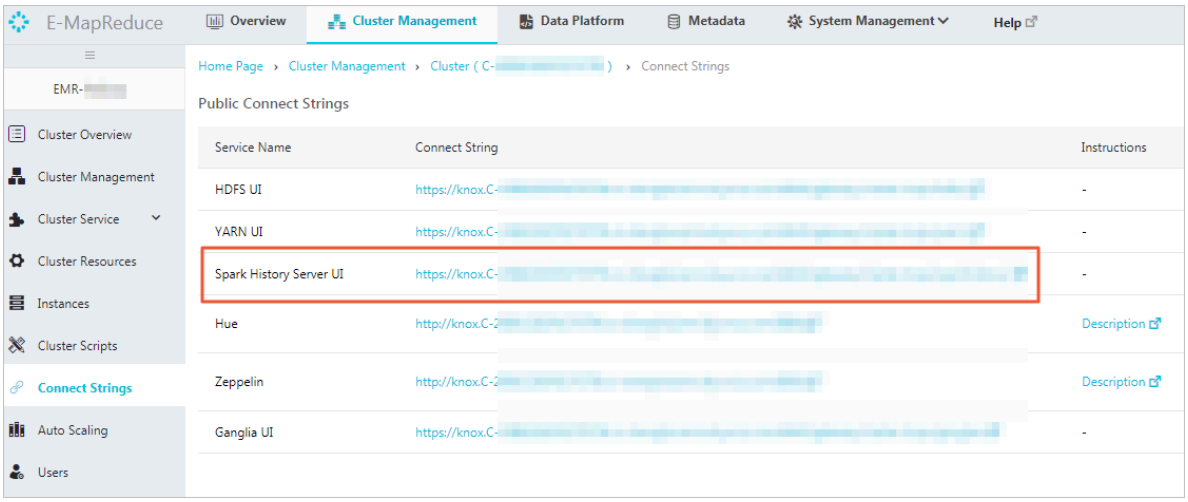


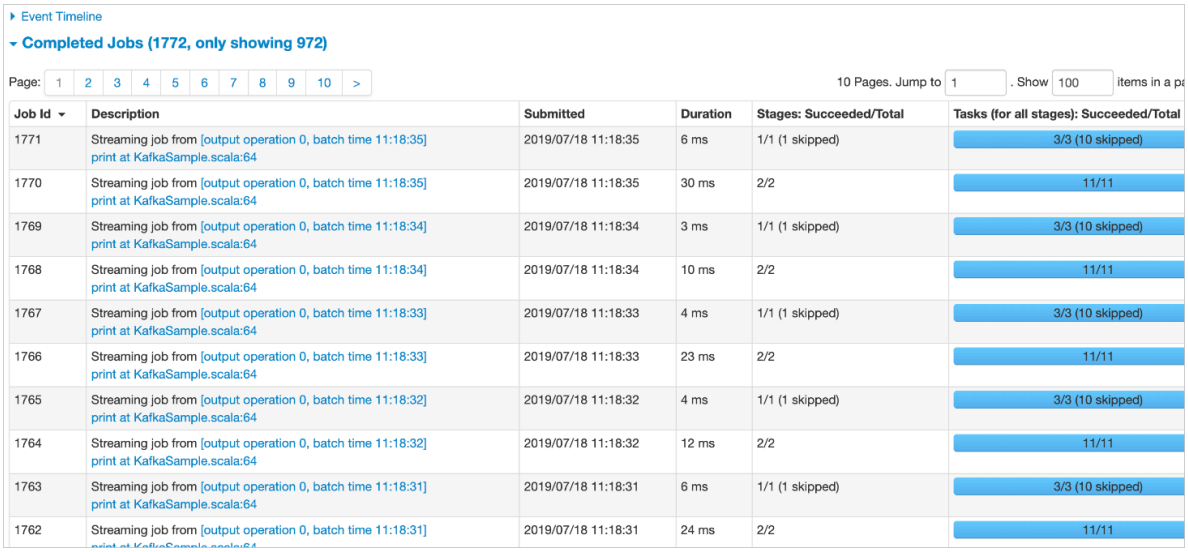
ステップ6：Spark Streaming ジョブの進行状況の表示

Spark Streaming ジョブを実行した後、EMR コンソールでジョブのステータスを表示できます。

1. EMR コンソール を開きます。

2. **[文字列の接続]** ページで、**[Spark History Server UI]** サービス名の横のリンクをクリックし、Spark Streaming のジョブステータスを表示します。 詳細については、[リンクとポートへのアクセス] をご参照ください。





2.14 Kafka Connect を使用したデータの移行

ストリーミングデータの処理中に、Kafka と他のシステム間のデータ同期、または Kafka クラスター間のデータ移行が必要になることがよくあります。 このページでは、Kafka Connect を使用して、Kafka クラスター間でデータを移行する方法について説明します。

- Alibaba Cloud アカウントに登録している必要があります。 詳細については、「Alibaba Cloud アカウントの作成」をご参照ください。
- E-MapReduce をアクティブにする必要があります。
- Alibaba Cloud アカウントの権限が付与されている必要があります。 詳細については、「役割の権限付与」をご参照ください。

Kafka Connect は、Kafka と他のシステム間でストリーミングデータを高速で送信するためのスケーラブルで信頼性の高いツールです。 たとえば、Kafka Connect を使用して、データベースから binlog データを取得し、データベースのデータを Kafka クラスターに移行できます。 この方法で、データベースのデータを移行し、データベースをダウンストリームストリーミングデータ処理システムに間接的に接続できます。 Kafka Connect は、Kafka Connect コネクターの作成と管理に役立つ Representational State Transfer (REST) アプリケーションプログラミングインターフェイス (API) も提供します。

Kafka Connect は、スタンドアロンモードまたは分散モードで実行できます。 スタンドアロンモードでは、すべてのワーカーが同じプロセスで実行されます。 スタンドアロンモードと比較して、分散モードはよりスケーラブルでフォールトトレラントです。 最も一般的に使用されるモードです。実稼働環境用に推奨します。

このページでは、Kafka Connect の REST API を呼び出して、Kafka Connect が分散モードで実行される Kafka クラスター間でデータを移行する方法について説明します。

ステップ 1：クラスターの作成

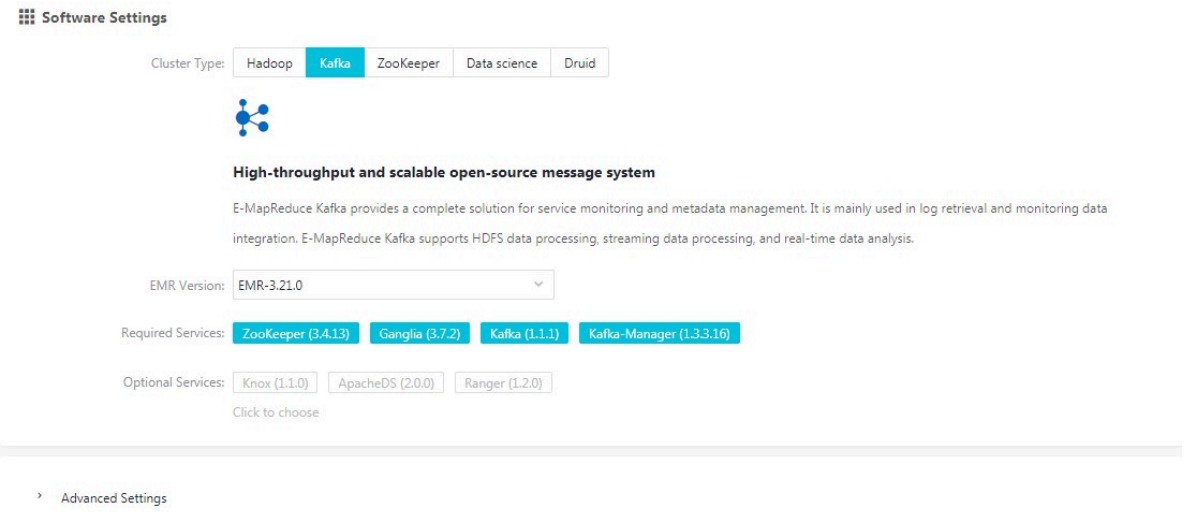
E-MapReduce でソース Kafka クラスターとターゲット Kafkaクラスターを作成します。 Kafka Connect がタスクノードにインストールされます。 そのため、ターゲット Kafka クラスターにタスクノードを作成する必要があります。 クラスターの作成後、デフォルトで Kafka Connect はタスクノードで開始されます。 ポート番号は 8083 です。

ソース Kafka クラスターとターゲット Kafka クラスターを同じセキュリティグループに追加することをお勧めします。 ソース Kafka クラスターとターゲットKafka クラスターが異なるセキュリティグループに属している場合、2 つのクラスターはデフォルトで相互にアクセスできません。相互アクセスを許可するには、セキュリティグループで必要とされる設定を変更しなければなりません。

1. Alibaba Cloud E-MapReduce コンソール にログインします。
2. ソース Kafka クラスターとターゲット Kafka クラスターを作成します。 詳細は、「#unique_27」をご参照ください。

注：

ターゲットKafkaクラスターを作成する場合、 タスクインスタンス 、つまりタスクノードを設定する必要があります。



ステップ2：移行するデータを保存するためのトピックの作成

ソース Kafka クラスターに **connect** というトピックを作成します。

1. Secure Shell (SSH) を使用して、ソース Kafka クラスターのヘッダーノードにログインします。 この例では、ヘッダーノードは **emr-header-1** です。
2. **connect** というトピックを作成するために以下のコマンドをルートとして実行します。

```
kafka-topics.sh --create --zookeeper emr-header-1:2181 --replication-factor 2 --partitions 10 --topic connect
```

```
[root@emr-header-1 ~]# kafka-topics.sh --create --zookeeper emr-header-1:2181 --replication-factor 2 --partitions 10 --topic connect
Created topic "connect".
[root@emr-header-1 ~]#
```

注：

上記の操作を実行した後、後で使用するためにログインウィンドウを保持します。

ステップ3：Kafka Connect コネクターの作成

ターゲットKafka クラスターのタスクノードで、**curl** コマンドを実行し、JavaScript Object Notation (JSON) データを使用して Kafka Connect コネクターを作成します。

1. SSH を使用して、ターゲット Kafka クラスターのタスクノードにログインします。 この例では、タスクノードは **emr-worker-3** です。

2. オプション: Kafka Connect の設定をカスタマイズします。

   ターゲット Kafka クラスターの **Kafka** サービスの **[設定]** ページを開きます。 **connect-distributed.properties** の **offset.storage.topic**、**config.storage.topic**、**status.storage.topic** パラメータをカスタマイズします。 詳細は、「#unique_31」をご参照ください。

   Kafka Connect は、オフセット、設定、タスクステータスを **offset.storage.topic**、**config.storage.topic**、**status.storage.topic** パラメータで指定されるトピックにそれぞれ保存します。 Kafka Connect は、/etc/ecm/kafka-conf/connect-distributed.properties に保存されているデフォルトのパーティションとレプリケーション係数を使用して、これらのトピックを自動的に作成します。

3. root ユーザーとして次のコマンドを実行して、Kafka Connect コネクターを作成します。

```
curl -X POST -H "Content-Type: application/json" --data '{"name": "connect-test",
 "config": { "connector.class": "EMRReplicatorSourceConnector", "key.converter": "
org.apache.kafka.connect.converters.ByteArrayConverter", "value.converter": "org.
apache.kafka.connect.converters.ByteArrayConverter", "src.kafka.bootstrap.servers":
 "${src-kafka-ip}:9092", "src.zookeeper.connect": "${src-kafka-curator-ip}:2181", "dest.
zookeeper.connect": "${dest-kafka-curator-ip}:2181", "topic.whitelist": "${source-topic
}", "topic.rename.format": "${dest-topic}", "src.kafka.max.poll.records": "300" } }' http
://emr-worker-3:8083/connectors
```

   JSON データでは、**[名前]** フィールドは、作成する Kafka Connect コネクター名を示します。 この例では connect-test です。 **config** フィールドは、実際の要件に基づいて設定する必要があります。 次の表は、config フィールドの主要な変数を説明しています。

変数	説明
${source-topic}	移行するデータをソース Kafka クラスターに保存するためのトピック。 たとえば、**connect**。 複数入力するときは、コンマ (,) で区切ります。
${dest-topic}	ターゲット Kafka クラスターでデータが移行されるトピック。 たとえば、**connect.replica**。
${src-kafka-curator-hostname}	ソース Kafka クラスターで ZooKeeper サービスがインストールされているノードの内部 IP アドレス。
${dest-kafka-curator-hostname}	ZooKeeper サービスがターゲット Kafka クラスターにインストールされているノードの内部 IP アドレス。

   注 :
   上記の操作を実行した後、後で使用するためにログインウィンドウを保持します。

ステップ 4：Kafka Connect コネクターとタスクノードのステータスの表示

Kafka Connect コネクターとタスクノードのステータスを表示し、それらが正常なステータスであることを確認します。

1. ターゲット Kafka クラスターのタスクノードのログインウィンドウに戻ります。 この例では、タスクノードは **emr-worker-3** です。
2. root ユーザーとして次のコマンドを実行して、すべての Kafka Connect コネクターを表示します。

   ```
   curl emr-worker-3:8083/connectors
   ```

   ```
   [root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors
   ["connect-test"][root@emr-worker-3 ~]#
   ```

3. root ユーザーとして次のコマンドを実行して、この例で作成された Kafka Connect コネクターのステータス、つまり **connect-test**を表示します。

   ```
   curl emr-worker-3:8083/connectors/connect-test/status
   ```

   ```
   [root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/status
   {"name":"connect-test","connector":{"state":"RUNNING","worker_id":"192.168.  :8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"192.168.  :8083"}],"type":"source"}[r
   oot@emr-worker-3 ~]#
   ```

   Kafka Connect コネクター (この例では **connect-test**) は必ず**RUNNING** ステータスにしてください。

4. rootユーザーとして次のコマンドを実行して、タスクノードの詳細を表示します。

   ```
   curl emr-worker-3:8083/connectors/connect-test/tasks
   ```

   ```
   [root@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/status
   {"name":"connect-test","connector":{"state":"RUNNING","worker_id":"192.168.  :8083"},"tasks":[{"state":"RUNNING","id":0,"worker_id":"192.168.  :8083"}],"type":"source"}[r
   oot@emr-worker-3 ~]# curl emr-worker-3:8083/connectors/connect-test/tasks
   [{"id":{"connector":"connect-test","task":0},"config":{"connector.class":"EMRReplicatorSourceConnector","src.zookeeper.connect":"emr-header-1.cluster-127390:2181","topic.rename.f
   ormat":"connect.replica","dest.zookeeper.connect":"emr-header-1.cluster-127499:2181","task.class":"org.apache.kafka.connect.replicator.EMRReplicatorSourceTask","src.kafka.max.poll.records":"300","name":"connect-test","task.id":"connect-test-0","value.converter":"org.apache.kafka.connect.converters.ByteArrayConverter","key.converter":"org.apache.kafka.connect.converters.ByteArrayConverter","src.kafka.bootstrap.servers":"192.168.  :9092","topic.whitelist":"connect","partition.assignment":"AAAAAAABAAdjb25uZWN0AAAACgAAAAAAAAABAAAAAgAAAAMAAAAEAAAABQAAAAYAAAAHAAAACAAAAAkAAAAA"}}][root@emr-worker-3 ~]#
   ```

   タスクノードに関するエラーメッセージが返されないことを確認します。

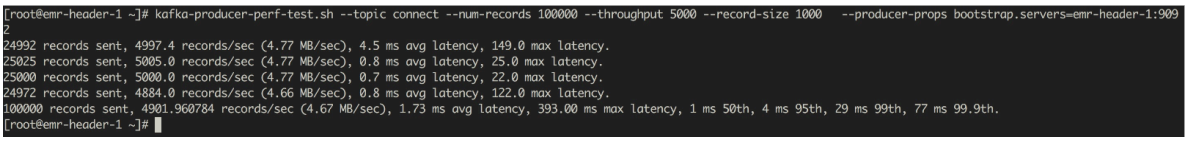
ステップ5：移行データの生成

移行するデータをソース Kafka クラスターの **connect** トピックに送信します。

1. ソースKafkaクラスターのヘッダーノードのログオンウィンドウに戻ります。 この例ではヘッダーノードは **emr-header-1** です。

2. root ユーザーとして以下のコマンドを実行してデータを **connect** トピックに送信します。

```
kafka-producer-perf-test.sh --topic connect --num-records 100000 --throughput 5000
 --record-size 1000 --producer-props bootstrap.servers=emr-header-1:9092
```

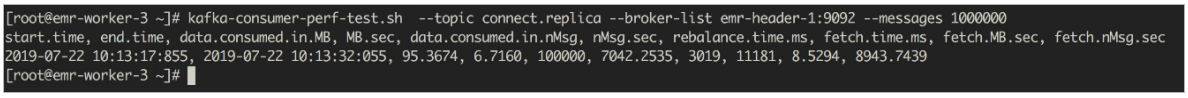


ステップ6：データ移行の結果表示

移行データが生成されると、Kafka Connect はターゲット Kafka クラスター内の対応するトピックにデータを自動的に移行します。 この例では、トピックは **connect.replica** です。

1. ターゲット Kafka クラスターのタスクノードのログインウィンドウに戻ります。 この例では、託すノードは **emr-worker-3** です。
2. root ユーザーとして次のコマンドを実行して、データが移行されているかどうかを確認します。

```
kafka-consumer-perf-test.sh --topic connect.replica --broker-list emr-header-1:9092
 --messages 100000
```



上図に示されるコマンド出力によると、ソース Kafka クラスターに送信された 100,000 のメッセージは、ターゲット Kafka クラスターに移行されます。

概要

このページでは、Kafka Connect を使用して Kafka クラスター間でデータを移行する方法について説明します。 Kafka Connect の詳細については、『Kafka 公式ウェブサイト』および 『REST API』をご覧ください。