Cloudera Runtime 7.1.1

Developing Apache Spark Applications

Date published: 2019-09-23
Date modified: 2020-12-15

https://docs.cloudera.com/

Legal Notice

Contents

Introduction 5

Spark application model 5

Spark execution model 5

Developing and running an Apache Spark WordCount application 6

Using the Spark DataFrame API 9

Building Spark Applications 11
- Best practices for building Apache Spark applications 11
- Building reusable modules in Apache Spark applications 11
- Packaging different versions of libraries with an Apache Spark application 13

Using Spark SQL 13
- SQLContext and HiveContext 14
- Querying files into a DataFrame 15
- Spark SQL example 15
- Interacting with Hive views 17
- Performance and storage considerations for Spark SQL DROP TABLE PURGE 17
- TIMESTAMP compatibility for Parquet files 18
- Accessing Spark SQL through the Spark shell 20

Calling Hive user-defined functions (UDFs) 20

Using Spark Streaming 21
- Spark Streaming and Dynamic Allocation 22
- Spark Streaming Example 22
- Enabling fault-tolerant processing in Spark Streaming 24
- Configuring authentication for long-running Spark Streaming jobs 25
- Building and running a Spark Streaming application 25
- Sample pom.xml file for Spark Streaming with Kafka 27

Accessing external storage from Spark 29
- Accessing data stored in Amazon S3 through Spark 30
  - Examples of accessing Amazon S3 data from Spark 31

Accessing Hive from Spark 33

Accessing HDFS Files from Spark..... 34

Accessing ORC Data in Hive Tables..... 34
Accessing ORC files from Spark..... 35
Predicate push-down optimization..... 36
Loading ORC data into DataFrames using predicate push-down..... 36
Optimizing queries using partition pruning..... 37
Enabling vectorized query execution..... 37
Reading Hive ORC tables..... 38

Accessing Avro data files from Spark SQL applications..... 38

Accessing Parquet files from Spark SQL applications..... 40

Using Spark MLlib..... 40
Running a Spark MLlib example..... 40
Enabling Native Acceleration For MLlib..... 40

Using custom libraries with Spark..... 41

Introduction

Apache Spark enables you to quickly develop applications and process jobs.

Apache Spark is designed for fast application development and processing. Spark Core is the underlying execution engine; other services, such as Spark SQL, MLlib, and Spark Streaming, are built on top of the Spark Core.

Depending on your use case, you can extend your use of Spark into several domains, including the following:

- Spark DataFrames
- Spark SQL
- Calling Hive user-defined functions from Spark SQL
- Spark Streaming
- Accessing HBase tables, HDFS files, and ORC data (Hive)
- Using custom libraries

Related Information

Apache Spark Quick Start

Apache Spark Overview

Apache Spark Programming Guide

Spark application model

Apache Spark is widely considered to be the successor to MapReduce for general purpose data processing on Apache Hadoop clusters. Like MapReduce applications, each Spark application is a self-contained computation that runs user-supplied code to compute a result. As with MapReduce jobs, Spark applications can use the resources of multiple hosts. However, Spark has many advantages over MapReduce.

In MapReduce, the highest-level unit of computation is a *job*. A job loads data, applies a map function, shuffles it, applies a reduce function, and writes data back out to persistent storage. In Spark, the highest-level unit of computation is an *application*. A Spark application can be used for a single batch job, an interactive session with multiple jobs, or a long-lived server continually satisfying requests. A Spark application can consist of more than just a single map and reduce.

MapReduce starts a process for each task. In contrast, a Spark application can have processes running on its behalf even when it's not running a job. Furthermore, multiple tasks can run within the same executor. Both combine to enable extremely fast task startup time as well as in-memory data storage, resulting in orders of magnitude faster performance over MapReduce.

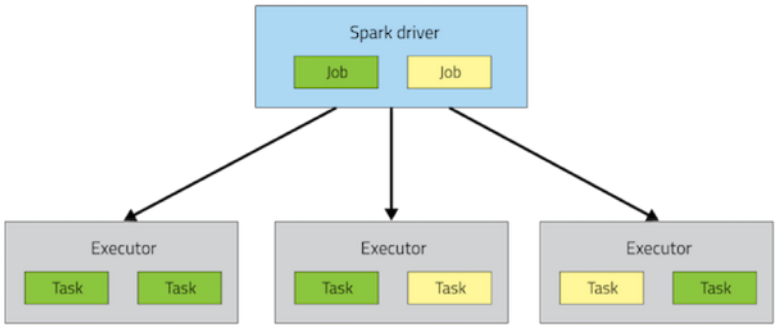
Spark execution model

Spark application execution involves runtime concepts such as *driver*, *executor*, *task*, *job*, and *stage*. Understanding these concepts is vital for writing fast and resource efficient Spark programs.

At runtime, a Spark application maps to a single *driver* process and a set of *executor* processes distributed across the hosts in a cluster.

The driver process manages the job flow and schedules tasks and is available the entire time the application is running. Typically, this driver process is the same as the client process used to initiate the job, although when run on YARN, the driver can run in the cluster. In interactive mode, the shell itself is the driver process.

The executors are responsible for performing work, in the form of tasks, as well as for storing any data that you cache. Executor lifetime depends on whether dynamic allocation is enabled. An executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime.



Invoking an action inside a Spark application triggers the launch of a *job* to fulfill it. Spark examines the dataset on which that action depends and formulates an execution plan. The execution plan assembles the dataset transformations into stages. A *stage* is a collection of tasks that run the same code, each on a different subset of the data.

Developing and running an Apache Spark WordCount application

This tutorial describes how to write, compile, and run a simple Spark word count application in two of the languages supported by Spark: Scala and Python. The Scala code was originally developed for a Cloudera tutorial written by Sandy Ryza.

About this task

This example application is an enhanced version of WordCount, the canonical MapReduce example. In this version of WordCount, the goal is to learn the distribution of letters in the most popular words in a corpus. The application:

1. Creates a SparkConf and SparkContext. A Spark application corresponds to an instance of the SparkContext class. When running a shell, the SparkContext is created for you.
2. Gets a word frequency threshold.
3. Reads an input set of text documents.
4. Counts the number of times each word appears.
5. Filters out all words that appear fewer times than the threshold.
6. For the remaining words, counts the number of times each letter occurs.

In MapReduce, this requires two MapReduce applications, as well as persisting the intermediate data to HDFS between them. In Spark, this application requires about 90 percent fewer lines of code than one developed using the MapReduce API.

Procedure

1. Create an empty directory named sparkwordcount in your home directory, and enter it:

```
mkdir $HOME/sparkwordcount
cd $HOME/sparkwordcount
```

2. For the Scala version, create the ./com/cloudera/sparkwordcount subdirectories. For Python, skip this step.

```
mkdir -p com/cloudera/sparkwordcount
```

3. Create the WordCount program in either Scala or Python, using the specified file names and paths:
   - Scala (./com/cloudera/sparkwordcount/SparkWordCount.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SparkWordCount {
  def main(args: Array[String]) {
    // create Spark context with Spark configuration
    val sc = new SparkContext(new SparkConf().setAppName("SparkWordCou
nt"))

    // get threshold
    val threshold = args(1).toInt

    // read in text file and split each document into words
    val tokenized = sc.textFile(args(0)).flatMap(_.split(" "))

    // count the occurrence of each word
    val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)

    // filter out words with fewer than threshold occurrences
    val filtered = wordCounts.filter(_._2 >= threshold)

    // count characters
    val charCounts = filtered.flatMap(_._1.toCharArray).map((_, 1)).re
duceByKey(_ + _)

    System.out.println(charCounts.collect().mkString(", "))
  }
}
```

   - Python (./SparkWordCount.py):

```
import sys

from pyspark import SparkContext, SparkConf
if __name__ == "__main__":
  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("SparkWordCount")
  sc = SparkContext(conf=conf)
  # get threshold
  threshold = int(sys.argv[2])

  # read in text file and split each document into words
  tokenized = sc.textFile(sys.argv[1]).flatMap(lambda line: line.split("
 "))

  # count the occurrence of each word
  wordCounts = tokenized.map(lambda word: (word, 1)).reduceByKey(lambda
v1,v2:v1 +v2)

  # filter out words with fewer than threshold occurrences
  filtered = wordCounts.filter(lambda pair:pair[1] >= threshold)

  # count characters
  charCounts = filtered.flatMap(lambda pair:pair[0]).map(lambda c: c).m
ap(lambda c: (c, 1)).reduceByKey(lambda v1,v2:v1 +v2)

  list = charCounts.collect()
  print repr(list)[1:-1]
```

This tutorial uses Maven to compile and package the Scala program. Download the tutorial pom.xml file to the parent $HOME/sparkwordcount directory and modify the sections listed below. For best practices using Maven to build Spark applications, see Building Spark Applications on page 11.

4. In the application pom.xml file, include the Scala tools repo and plugin, and add Scala and Spark as dependencies:

```
<pluginRepositories>
<pluginRepository>
    <id>scala-tools.org</id>
    <name>Scala-tools Maven2 Repository</name>
    <url>http://scala-tools.org/repo-releases</url>
  </pluginRepository>
</pluginRepositories>
```

```
  <build>
    <sourceDirectory>${project.basedir}</sourceDirectory>
    <plugins>
      <plugin>
        <groupId>org.scala-tools</groupId>
        <artifactId>maven-scala-plugin</artifactId>
        <version>2.15.2</version>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.11.12</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.4.0.7.0.0.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

5. Make sure that you are in the parent $HOME/sparkwordcount directory, and then generate the application JAR as follows:

```
mvn package
```

This creates sparkwordcount-0.0.1-SNAPSHOT.jar in the target directory.

The input to the application is a large text file in which each line contains all the words in a document, stripped of punctuation. Put an input file in a directory in an S3 bucket that is accessible by your Spark cluster. You can use the tutorial example input file.

6. Run the applications using spark-submit:

- Scala on YARN with threshold 2:

```
spark-submit --class SparkWordCount \
  --master yarn --deploy-mode client --executor-memory 1g \
```

```
  --conf "spark.yarn.access.hadoopFileSystems=s3a://<bucket_name>" \
  target/sparkwordcount-0.0.1-SNAPSHOT.jar \
  s3a://<bucket_name>/<input_filename> 2
```

- Python on YARN with threshold 2:

```
spark-submit --master yarn --deploy-mode client --executor-memory 1g \
  --conf "spark.yarn.access.hadoopFileSystems=s3a://<bucket_name>" \
  SparkWordCount.py s3a://<bucket_name>/<input_filename> 2
```

If you used the example input file, the output is similar to the following:

Scala:

```
19/07/24 09:18:57 INFO scheduler.DAGScheduler: Job 0 finished: collect at
 SparkWordCount.scala:25, took 10.188651 s
(p,2), (t,2), (b,1), (h,1), (n,4), (f,1), (v,1), (r,2), (l,1), (e,6), (a
,4), (i,1), (u,1), (o,2), (c,1)
```

Python:

```
19/07/24 09:23:55 INFO scheduler.DAGScheduler: Job 0 finished: collect a
t /home/user/sparkwordcount/SparkWordCount.py:26, took 11.762150 s
(u'a', 4), (u'c', 1), (u'e', 6), (u'i', 1), (u'o', 2), (u'u', 1), (u'b',
1), (u'f', 1), (u'h', 1), (u'l', 1), (u'n', 4), (u'p', 2), (u'r', 2), (u
't', 2), (u'v', 1)
```

Using the Spark DataFrame API

About this task

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R or in the Python pandas library. You can construct DataFrames from a wide array of sources, including structured data files, Apache Hive tables, and existing Spark resilient distributed datasets (RDD). The Spark DataFrame API is available in Scala, Java, Python, and R.

This section provides examples of DataFrame API use.

To list JSON file contents as a DataFrame:

1. Upload the people.txt and people.json example files to your object store:

```
hdfs dfs -put people.txt people.json s3a://<bucket_name>/
```

2. Launch the Spark shell:

```
spark-shell --conf "spark.yarn.access.hadoopFileSystems=s3a:
//<bucket_name>/"
```

3. At the Spark shell, type the following:

```
scala> val df = spark.read.format("json").load("s3a://<bucket_name>/peo
ple.json")
```

4. Using df.show, display the contents of the DataFrame:

```
scala> df.show
+----+-------+
| age|   name|
```

```
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

The following examples use Scala to access DataFrame df defined in the previous subsection:

```
// Select all rows, but increment age by 1
scala> df.select(df("name"), df("age") + 1).show()
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     null|
|   Andy|       31|
| Justin|       20|
+-------+---------+

// Select people older than 21
scala> df.filter(df("age") > 21).show()
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+

// Count people by age
scala> df.groupBy("age").count().show()
+----+-----+

| age|count|
+----+-----+
|  19|    1|
|null|    1|
|  30|    1|
+----+-----+
```

The following example uses the DataFrame API to specify a schema for people.txt, and then retrieves names from a temporary table associated with the schema:

```
val people = sc.textFile("s3a://<bucket_name>/people.txt")
val schemaString = "name age"
import org.apache.spark.sql.types.{StructType,StructField,StringType}
import org.apache.spark.sql.Row

val schema = StructType(schemaString.split(" ").map(fieldName => StructField
(fieldName, StringType, true)))
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
val peopleDataFrame = spark.createDataFrame(rowRDD, schema)

peopleDataFrame.createOrReplaceTempView("people")
val results = spark.sql("SELECT name FROM people")
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

This produces output similar to the following:

```
Name: Michael
Name: Andy
Name: Justin
```

Building Spark Applications

You can use Apache Maven to build Spark applications developed using Java and Scala.

For the Maven properties of CDH components, see Using the Cloudera Runtime Maven Repository.

Best practices for building Apache Spark applications

Follow these best practices when building Apache Spark Scala and Java applications:

- Compile your applications against the same version of Spark that you are running.
- Build a single assembly JAR ("Uber" JAR) that includes all dependencies. In Maven, add the Maven assembly plug-in to build a JAR containing all dependencies:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

This plug-in manages the merge procedure for all available JAR files during the build. Exclude Spark, Hadoop, and Kafka classes from the assembly JAR, because they are already available on the cluster and contained in the runtime classpath. In Maven, specify Spark, Hadoop, and Kafka dependencies with scope provided. For example:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>2.4.0.7.0.0.0</version>
  <scope>provided</scope>
</dependency>
```

Building reusable modules in Apache Spark applications

Using existing Scala and Java classes inside the Spark shell requires an effective deployment procedure and dependency management. For simple and reliable reuse of Scala and Java classes and complete third-party libraries, you can use a *module*, which is a self-contained artifact created by Maven. This module can be shared by multiple users. This topic shows how to use Maven to create a module containing all dependencies.

Create a Maven Project

1. Use Maven to generate the project directory:

```
$ mvn archetype:generate -DgroupId=com.mycompany -DartifactId=mylibrary \
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Download and Deploy Third-Party Libraries

1. Prepare a location for all third-party libraries that are not available through Maven Central but are required for the project:

```
mkdir libs
cd libs
```

2. Download the required artifacts.
3. Use Maven to deploy the library JAR.
4. Add the library to the dependencies section of the POM file.
5. Repeat steps 2-4 for each library. For example, to add the JIDT library:
   a. Download and decompress the zip file:

```
curl http://lizier.me/joseph/software/jidt/download.php?file=infodynamic
s-dist-1.3.zip > infodynamics-dist-1.3.zip
unzip infodynamics-dist-1.3.zip
```

   b. Deploy the library JAR:

```
$ mvn deploy:deploy-file \
-Durl=file:///$HOME/.m2/repository -Dfile=libs/infodynamics.jar \
-DgroupId=org.jlizier.infodynamics -DartifactId=infodynamics -Dpackag
ing=jar -Dversion=1.3
```

   c. Add the library to the dependencies section of the POM file:

```
<dependency>
  <groupId>org.jlizier.infodynamics</groupId>
  <artifactId>infodynamics</artifactId>
  <version>1.3</version>
</dependency>
```

6. Add the Maven assembly plug-in to the plugins section in the pom.xml file.
7. Package the library JARs in a module:

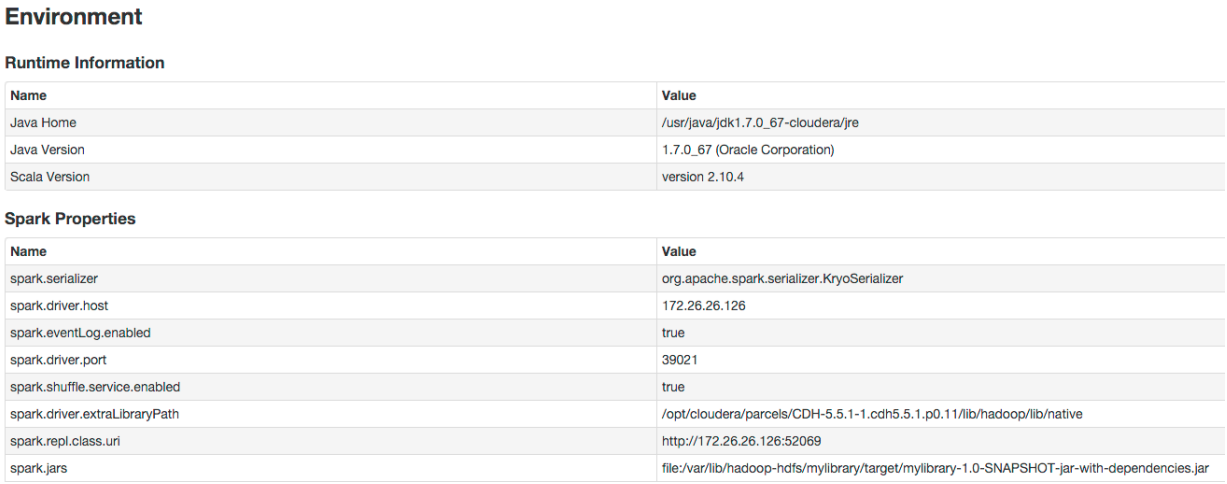
```
mvn clean package
```

Run and Test the Spark Module

1. Run the Spark shell, providing the module JAR in the --jars option:

```
spark-shell --jars target/mylibrary-1.0-SNAPSHOT-jar-with-dependencies.jar
```

2. In the Environment tab of the Spark Web UI application (http://*driver_host*:4040/environment/), validate that the spark.jars property contains the library. For example:



Environment

Runtime Information

Name	Value
Java Home	/usr/java/jdk1.7.0_67-cloudera/jre
Java Version	1.7.0_67 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.serializer	org.apache.spark.serializer.KryoSerializer
spark.driver.host	172.26.26.126
spark.eventLog.enabled	true
spark.driver.port	39021
spark.shuffle.service.enabled	true
spark.driver.extraLibraryPath	/opt/cloudera/parcels/CDH-5.5.1-1.cdh5.5.1.p0.11/lib/hadoop/lib/native
spark.repl.class.uri	http://172.26.26.126:52069
spark.jars	file:/var/lib/hadoop-hdfs/mylibrary/target/mylibrary-1.0-SNAPSHOT-jar-with-dependencies.jar

3. In the Spark shell, test that you can import some of the required Java classes from the third-party library. For example, if you use the JIDT library, import MatrixUtils:

```
$ spark-shell
...
scala> import infodynamics.utils.MatrixUtils;
```

Packaging different versions of libraries with an Apache Spark application

To use a version of a library in your application that is different than the version of that library that is shipped with Spark, use the Apache Maven Shade Plugin. This process is technically known as relocation, and often referred to as shading.

See Relocating Classes for an example.

Using Spark SQL

This section provides information about using Spark SQL.

Using SQLContext, Apache Spark SQL can read data directly from the file system. This is useful when the data you are trying to analyze does not reside in Apache Hive (for example, JSON files stored in HDFS).

Using HiveContext, Spark SQL can also read data by interacting with the Hive MetaStore. If you already use Hive, you should use HiveContext; it supports all Hive data formats and user-defined functions (UDFs), and it enables you to have full access to the Hive parser. HiveContext extends SQLContext, so HiveContext supports all SQLContext functionality.

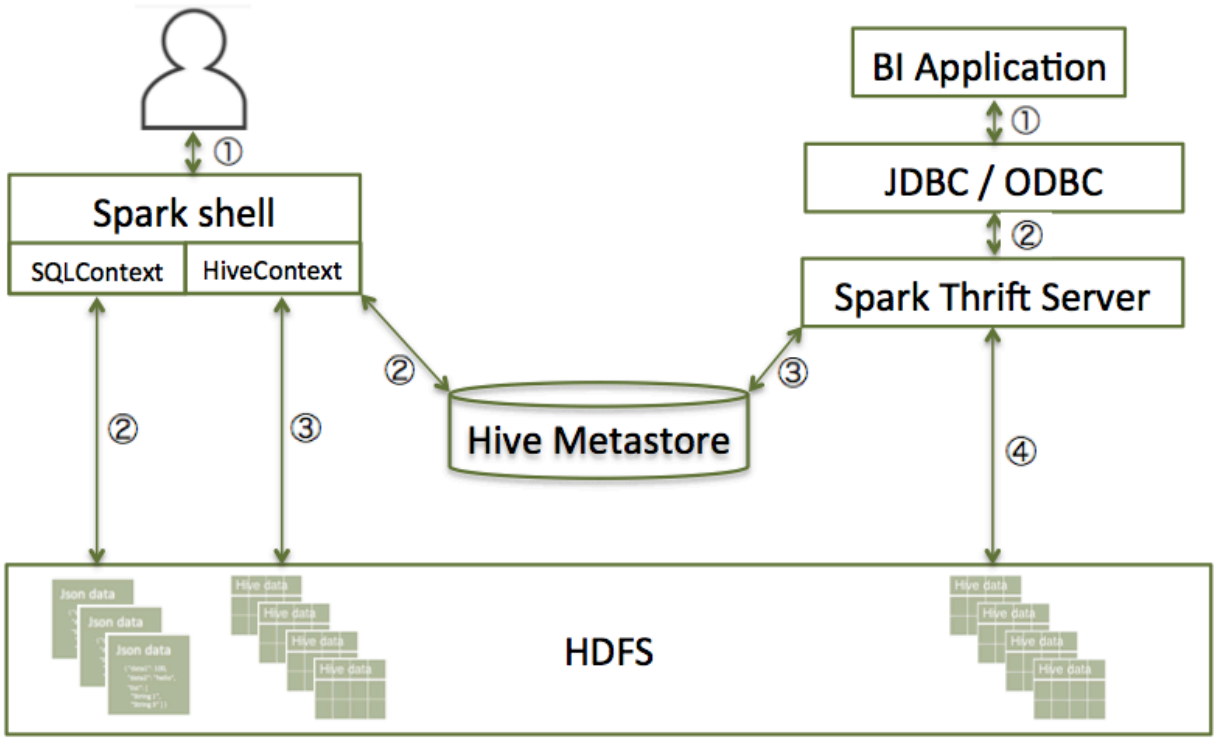
There are two ways to interact with Spark SQL:

- Interactive access using the Spark shell (see "Accessing Spark SQL through the Spark Shell" in this guide).
- From an application, operating through one of the following two APIs and the Spark Thrift server:
  - JDBC, using your own Java code or the Beeline JDBC client
  - ODBC, through the Simba ODBC driver

  For more information about JDBC and ODBC access, see "Accessing Spark SQL through JDBC: Prerequisites" and "Accessing Spark SQL through JDBC and ODBC" in this guide.

The following diagram illustrates the access process, depending on whether you are using the Spark shell or business intelligence (BI) application:

The following diagram illustrates the access process, depending on whether you are using the Spark shell or business intelligence (BI) application:



The following subsections describe how to access Spark SQL through the Spark shell, and through JDBC and ODBC.

Related Information

Beeline Command Line Shell

SQLContext and HiveContext

Beginning in Spark 2.0, all Spark functionality, including Spark SQL, can be accessed through the SparkSessions class, available as spark when you launch spark-shell. You can create a DataFrame from an RDD, a Hive table, or a data source.

Note: Hive and Impala tables and related SQL syntax are interchangeable in most respects. Because Spark uses the underlying Hive infrastructure, with Spark SQL you write DDL statements, DML statements, and queries using the Hive syntax. For interactive query performance, you can access the same tables through Impala using impala-shell or the Impala JDBC and ODBC interfaces.

If you use spark-submit, use code like the following at the start of the program (this example uses Python):

```
from pyspark import SparkContext, HiveContext
sc = SparkContext(appName = "test")
sqlContext = HiveContext(sc)
```

The host from which the Spark application is submitted or on which spark-shell or pyspark runs must have a Hive gateway role defined in Cloudera Manager and client configurations deployed.

When a Spark job accesses a Hive view, Spark must have privileges to read the data files in the underlying Hive tables. Currently, Spark cannot use fine-grained privileges based on the columns or the WHERE clause in the view

definition. If Spark does not have the required privileges on the underlying data files, a SparkSQL query against the view returns an empty result set, rather than an error.

Querying files into a DataFrame

If you have data files that are outside of a Hive or Impala table, you can use SQL to directly read JSON or Parquet files into a DataFrame.

JSON

```
df = sqlContext.sql("SELECT * FROM json.`input dir`")
```

Parquet

```
df = sqlContext.sql("SELECT * FROM parquet.`input dir`")
```

Spark SQL example

This example demonstrates how to use spark.sql to create and load two tables and select rows from the tables into two DataFrames. The next steps use the DataFrame API to filter the rows for salaries greater than 150,000 from one of the tables and shows the resulting DataFrame. Then the two DataFrames are joined to create a third DataFrame. Finally the new DataFrame is saved to a Hive table.

1. Copy the Hue sample_07.csv and sample_08.csv files to your object store in a location accessible by the Spark cluster:

   ```
   hdfs dfs -put /opt/cloudera/parcels/CDH/lib/hue/apps/beeswax/data/sample
   _0* s3a://<bucket_name>/
   ```

2. Launch spark-shell:

   ```
   spark-shell --conf "spark.yarn.access.hadoopFileSystems=s3a:
   //<bucket_name>"
   ```

3. Create Hive tables sample_07 and sample_08:

   ```
   scala> spark.sql("CREATE EXTERNAL TABLE sample_07 (code string,description
    string,total_emp int,salary int) ROW FORMAT DELIMITED FIELDS TERMINATED B
   Y '\t' STORED AS TextFile LOCATION 's3a://<bucket_name>/s07/'")
   scala> spark.sql("CREATE EXTERNAL TABLE sample_08 (code string,descript
   ion string,total_emp int,salary int) ROW FORMAT DELIMITED FIELDS TERMINA
   TED BY '\t' STORED AS TextFile LOCATION 's3a://<bucket_name>/s08/'")
   ```

4. In another session, launch Beeline:

   ```
   beeline -u "jdbc:hive2://<HiveServer2_host>:10001/default;principal=hive/
   _HOST@CLOUDERA.SITE;transportMode=http;httpPath=cliservice"
   ```

5. In Beeline, show the Hive tables:

   ```
   0: jdbc:hive2://hs2.cloudera.site:> show tables;
   +------------+--+
   |  tab_name  |
   +------------+--+
   | sample_07  |
   | sample_08  |
   ```

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

6. In the Spark shell, load the data from the CSV files into the tables:

```
scala> spark.sql("LOAD DATA INPATH 's3a://<bucket_name>/sample_07.csv'
OVERWRITE INTO TABLE sample_07")
scala> spark.sql("LOAD DATA INPATH 's3a://<bucket_name>/sample_08.csv'
OVERWRITE INTO TABLE sample_08")
```

7. Create DataFrames containing the contents of the sample_07 and sample_08 tables:

```
scala> val df_07 = spark.sql("SELECT * from sample_07")
scala> val df_08 = spark.sql("SELECT * from sample_08")
```

8. Show all rows in df_07 with salary greater than 150,000:

```
scala> df_07.filter(df_07("salary") > 150000).show()
```

The output should be:

```
+-------+--------------------+---------+------+
|   code|         description|total_emp|salary|
+-------+--------------------+---------+------+
|11-1011|    Chief executives|   299160|151370|
|29-1022|Oral and maxillof...|     5040|178440|
|29-1023|       Orthodontists|     5350|185340|
|29-1024|     Prosthodontists|      380|169360|
|29-1061|   Anesthesiologists|    31030|192780|
|29-1062|Family and genera...|   113250|153640|
|29-1063| Internists, general|    46260|167270|
|29-1064|Obstetricians and...|    21340|183600|
|29-1067|            Surgeons|    50260|191410|
|29-1069|Physicians and su...|   237400|155150|
+-------+--------------------+---------+------+
```

9. Create the DataFrame df_09 by joining df_07 and df_08, retaining only the code and description columns.

```
scala> val df_09 = df_07.join(df_08, df_07("code") === df_08("code")).se
lect(df_07.col("code"),df_07.col("description"))
scala> df_09.orderBy($"code".asc).show()
```

The new DataFrame looks like:

```
+-------+--------------------+
|   code|         description|
+-------+--------------------+
|00-0000|     All Occupations|
|11-0000|Management occupa...|
|11-1011|    Chief executives|
|11-1021|General and opera...|
|11-1031|         Legislators|
|11-2011|Advertising and p...|
|11-2021|  Marketing managers|
|11-2022|      Sales managers|
|11-2031|Public relations ...|
|11-3011|Administrative se...|
|11-3021|Computer and info...|
|11-3031|  Financial managers|
|11-3041|Compensation and ...|
|11-3042|Training and deve...|
|11-3049|Human resources m...|
|11-3051|Industrial produc...|
|11-3061| Purchasing managers|
```

```
|11-3071|Transportation, s...|
|11-9011|Farm, ranch, and ...|
|11-9012|Farmers and ranchers|
+-------+--------------------+
```

10. Save DataFrame df_09 as the Hive table sample_09:

```
scala> df_09.write.option("path","s3a://<bucket_name>/s09/").saveAsTable(
"sample_09")
```

11. In Beeline, show the Hive tables:

```
0: jdbc:hive2://hs2.cloudera.site:> show tables;
+------------+--+
|  tab_name  |
+------------+--+
| sample_07  |
| sample_08  |
| sample_09  |
+------------+--+
```

Here is an equivalent program in Python, that you could submit using spark-submit:

```
from pyspark import SparkContext, SparkConf, HiveContext

if __name__ == "__main__":

  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("Data Frame Join")
  sc = SparkContext(conf=conf)
  sqlContext = HiveContext(sc)
  df_07 = sqlContext.sql("SELECT * from sample_07")
  df_07.filter(df_07.salary > 150000).show()
  df_08 = sqlContext.sql("SELECT * from sample_08")
  tbls = sqlContext.sql("show tables")
  tbls.show()
  df_09 = df_07.join(df_08, df_07.code == df_08.code).select(df_07.code,d
f_07.description)
  df_09.show()
  df_09.write.saveAsTable("sample_09")
  tbls = sqlContext.sql("show tables")
  tbls.show()
```

Note: Instead of displaying the tables using Beeline, the show tables query is run using the Spark SQL API.

Interacting with Hive views

When a Spark job accesses a Hive view, Spark must have privileges to read the data files in the underlying Hive tables. Currently, Spark cannot use fine-grained privileges based on the columns or the WHERE clause in the view definition. If Spark does not have the required privileges on the underlying data files, a SparkSQL query against the view returns an empty result set, rather than an error.

Performance and storage considerations for Spark SQL DROP TABLE PURGE

The PURGE clause in the Hive DROP TABLE statement causes the underlying data files to be removed immediately, without being transferred into a temporary holding area (such as the HDFS trashcan).

Although the PURGE clause is recognized by the Spark SQL DROP TABLE statement, this clause is currently not passed along to the Hive statement that performs the “drop table” operation behind the scenes. Therefore, if you know the PURGE behavior is important in your application for performance, storage, or security reasons, do the DROP TABLE directly in Hive, for example through the `beeline` shell, rather than through Spark SQL.

The immediate deletion aspect of the PURGE clause could be significant in cases such as:

- If the cluster is running low on storage space and it is important to free space immediately, rather than waiting for the HDFS trashcan to be periodically emptied.
- If the underlying data files reside on the Amazon S3 filesystem. Moving files to the HDFS trashcan from S3 involves physically copying the files, meaning that the default DROP TABLE behavior on S3 involves significant performance overhead.
- If the underlying data files contain sensitive information and it is important to remove them entirely, rather than leaving them to be cleaned up by the periodic emptying of the trashcan.

TIMESTAMP compatibility for Parquet files

Impala stores and retrieves the TIMESTAMP values verbatim, with no adjustment for the time zone. When writing Parquet files, Hive and Spark SQL both normalize all TIMESTAMP values to the UTC time zone. During a query, Spark SQL assumes that all TIMESTAMP values have been normalized this way and reflect dates and times in the UTC time zone. Therefore, Spark SQL adjusts the retrieved date/time values to reflect the local time zone of the server. SPARK-12297 introduces a configuration setting, spark.sql.parquet.int96TimestampConversion=true, that you can set to change the interpretation of TIMESTAMP values read from Parquet files that were written by Impala, to match the Impala behavior.

Note: This compatibility workaround only applies to Parquet files created by Impala and has no effect on Parquet files created by Hive, Spark or other Java components.

The following sequence of examples show how, by default, TIMESTAMP values written to a Parquet table by an Apache Impala SQL statement are interpreted differently when queried by Spark SQL, and vice versa.

The initial Parquet table is created by Impala, and some TIMESTAMP values are written to it by Impala, representing midnight of one day, noon of another day, and an early afternoon time from the Pacific Daylight Savings time zone. (The second and third tables are created with the same structure and file format, for use in subsequent examples.)

```
[localhost:21000] > create table parquet_table(t timestamp) stored as parqu
et;
[localhost:21000] > create table parquet_table2 like parquet_table stored as
 parquet;
[localhost:21000] > create table parquet_table3 like parquet_table stored as
 parquet;
[localhost:21000] > select now();
+-------------------------------+
| now()                         |
+-------------------------------+
| 2018-03-23 14:07:01.057912000 |
+-------------------------------+
[localhost:21000] > insert into parquet_table
                  > values ('2018-03-23'), (now()), ('2000-01-01 12:00:00');
[localhost:21000] > select t from parquet_table order by t;
+-------------------------------+
| t                             |
+-------------------------------+
| 2000-01-01 12:00:00           |
| 2018-03-23 00:00:00           |
| 2018-03-23 14:08:54.617197000 |
```

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

By default, when this table is queried through the Spark SQL using `spark-shell`, the values are interpreted and displayed differently. The time values differ from the Impala result set by either 4 or 5 hours, depending on whether the dates are during the Daylight Savings period or not.

```
scala> spark.sql("select t from jdr.parquet_table order by t").show(truncat
e=false);
+--------------------------+
|t                         |
+--------------------------+
|2000-01-01 04:00:00.0     |
|2018-03-22 17:00:00.0     |
|2018-03-23 07:08:54.617197|
+--------------------------+
```

Running the same Spark SQL query with the configuration setting spark.sql.parquet.int96TimestampConversion=true applied makes the results the same as from Impala:

```
$ spark-shell --conf spark.sql.parquet.int96TimestampConversion=true
...
scala> spark.sql("select t from jdr.parquet_table order by t").show(trunca
te=false);
+--------------------------+
|t                         |
+--------------------------+
|2000-01-01 12:00:00.0     |
|2018-03-23 00:00:00.0     |
|2018-03-23 14:08:54.617197|
+--------------------------+
```

The compatibility considerations also apply in the reverse direction. The following examples show the same Parquet values as before, this time being written to tables through Spark SQL.

```
$ spark-shell
scala> spark.sql("insert into jdr.parquet_table2 select t from jdr.parque
t_table");
scala> spark.sql("select t from jdr.parquet_table2 order by t").show(trun
cate=false);
+--------------------------+
|t                         |
+--------------------------+
|2000-01-01 04:00:00.0     |
|2018-03-22 17:00:00.0     |
|2018-03-23 07:08:54.617197|
+--------------------------+
```

Again, the configuration setting spark.sql.parquet.int96TimestampConversion=true means that the values are both read and written in a way that is interoperable with Impala:

```
$ spark-shell --conf spark.sql.parquet.int96TimestampConversion=true
...
scala> spark.sql("insert into jdr.parquet_table3 select t from jdr.parquet
_table");
```

```
scala> spark.sql("select t from jdr.parquet_table3 order by t").show(trunc
ate=false);
+--------------------------+
|t                         |
+--------------------------+
|2000-01-01 12:00:00.0     |
|2018-03-23 00:00:00.0     |
|2018-03-23 14:08:54.617197|
+--------------------------+
```

Accessing Spark SQL through the Spark shell

Use the following steps to access Spark SQL using the Spark shell.

The following sample command launches the Spark shell on a YARN cluster:

spark-shell --num-executors 1 --executor-memory 512m --master yarn-client

To read data directly from the file system, construct a SQLContext. For an example that uses SQLContext and the Spark DataFrame API to access a JSON file, see Using the Spark DataFrame API on page 9.

To read data by interacting with the Hive Metastore, construct a HiveContext instance (HiveContext extends SQLContext). For an example of the use of HiveContext (instantiated as val sqlContext), see "Accessing ORC Files from Spark" in this guide.

Calling Hive user-defined functions (UDFs)

Use the following steps to call Hive user-defined functions.

About this task

You can call built-in Hive UDFs, UDAFs, and UDTFs and custom UDFs from Spark SQL applications if the functions are available in the standard Hive JAR file. When using Hive UDFs, use HiveContext (not SQLContext).

Using Built-in UDFs

The following interactive example reads and writes to HDFS under Hive directories, using hiveContext and the built-in collect_list(col) UDF. The collect_list(col) UDF returns a list of objects with duplicates. In a production environment, this type of operation runs under an account with appropriate HDFS permissions; the following example uses hdfs user.

1. Launch the Spark Shell on a YARN cluster:

```
spark-shell --num-executors 2 --executor-memory 512m --master yarn-client
```

2. Invoke the Hive collect_list UDF:

```
scala> spark.sql("from TestTable SELECT key, collect_list(value) group by
 key order by key").collect.foreach(println)
```

Using Custom UDFs

You can register custom functions in Python, Java, or Scala, and then use them within SQL statements.

When using a custom UDF, ensure that the .jar file for your UDF is included with your application, or use the --jars command-line option to specify the file.

The following example uses a custom Hive UDF. This example uses the more limited SQLContext, instead of HiveContext.

1. Launch spark-shell with hive-udf.jar as its parameter:

```
spark-shell --jars <path-to-your-hive-udf>.jar
```

2. From spark-shell, define a function:

```
scala> spark.sql("""create temporary function balance as 'org.package.hi
veudf.BalanceFromRechargesAndOrders'""");
```

3. From spark-shell, invoke your UDF:

```
scala> spark.sql("""
create table recharges_with_balance_array as
select
  reseller_id,
  phone_number,
  phone_credit_id,
  date_recharge,
  phone_credit_value,
  balance(orders,'date_order', 'order_value', reseller_id, date_recharge,
phone_credit_value) as balance
from orders
""");
```

Using Spark Streaming

This section provides information on using Spark streaming.

Before you begin

Before running a Spark Streaming application, Spark and Kafka must be deployed on the cluster.

Unless you are running a job that is part of the Spark examples package installed by Cloudera Data Platform (CDP), you must add or retrieve the CDP spark-streaming-kafka .jar file and associated .jar files before running your Spark job.

About this task

Spark Streaming is an extension of core Spark that enables scalable, high-throughput, fault-tolerant processing of data streams. Spark Streaming receives input data streams called Discretized Streams (DStreams), which are essentially a continuous series of RDDs. DStreams can be created either from sources such as Kafka, Flume, and Kinesis, or by applying operations on other DStreams.

You can process data using complex algorithms expressed with high-level functions like map, reduce, join, and wind ow, and send results to file systems, databases, and live dashboards.

For detailed information on Spark Streaming, see Spark Streaming Programming Guide in the Apache Spark documentation.

Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches:

Apache Spark has built-in support for the Apache Kafka 08 API. If you want to access a Kafka 0.10 cluster using new Kafka 0.10 APIs (such as wire encryption support) from Spark streaming jobs, the spark-kafka-0-10-connector package supports a Kafka 0.10 connector for Spark streaming. See the package readme file for additional documentation.

The remainder of this subsection describes general steps for developers using Spark Streaming with Kafka on a Kerberos-enabled cluster; it includes a sample pom.xml file for Spark Streaming applications with Kafka. For additional examples, see the Apache GitHub example repositories for Scala, Java, and Python.

Important: Dynamic Resource Allocation does not work with Spark Streaming.

Related Information

Apache Streaming Programming Guide
Apache GitHub Scala Streaming Examples
Apache GitHub Java Streaming Examples
Apache GitHub Python Streaming Examples

Spark Streaming and Dynamic Allocation

Dynamic allocation is enabled by default, which means that executors are removed when idle. Dynamic allocation conflicts with Spark Streaming operations.

In Spark Streaming, data comes in batches, and executors run whenever data is available. If the executor idle timeout is less than the batch duration, executors are constantly added and removed. However, if the executor idle timeout is greater than the batch duration, executors are never removed. Therefore, Cloudera recommends that you disable dynamic allocation by setting spark.dynamicAllocation.enabled to false when running streaming applications.

Spark Streaming Example

This example uses Kafka to deliver a stream of words to a Python word count program.

1. Create a Kafka topic wordcounttopic:

```
kafka-topics --create --zookeeper zookeeper_server:2181 --topic wordcountt
opic --partitions 1 --replication-factor 1
```

2. Create a Kafka word count Python program adapted from the Spark Streaming example kafka_wordcount.py. This version divides the input stream into batches of 10 seconds and counts the words in each batch:

```
from __future__ import print_function

import sys

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
```

```
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: kafka_wordcount.py <zk> <topic>", file=sys.stderr)
        sys.exit(-1)

    sc = SparkContext(appName="PythonStreamingKafkaWordCount")
    ssc = StreamingContext(sc, 10)

    zkQuorum, topic = sys.argv[1:]
    kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer
", {topic: 1})
    lines = kvs.map(lambda x: x[1])
    counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word:
 (word, 1)).reduceByKey(lambda a, b: a+b)
    counts.pprint()

    ssc.start()
    ssc.awaitTermination()
```

3. Submit the application using spark-submit with dynamic allocation disabled and specifying your ZooKeeper server and topic. To run locally, you must specify at least two worker threads: one to receive and one to process data:

```
spark-submit --master yarn --deploy-mode client --conf "spark.dynamicAll
ocation.enabled=false" --jars SPARK_HOME/lib/spark-examples.jar kafka_wo
rdcount.py zookeeper_server:2181 wordcounttopic
```

   In a Cloudera Data Platform deployment, *SPARK_HOME* defaults to /opt/cloudera/parcels/CDH/lib/spark. The shells are also available from /usr/bin.

4. In another window, start a Kafka producer that publishes to wordcounttopic:

```
kafka-console-producer --broker-list kafka_broker:9092 --topic wordcountt
opic
```

5. In the producer window, type the following:

```
hello
hello
hello
hello
hello
hello
gb
gb
gb
gb
gb
gb
```

   Depending on how fast you type, in the Spark Streaming application window you will see output like:

```
-------------------------------------------
Time: 2016-01-06 14:18:00
-------------------------------------------
(u'hello', 6)
(u'gb', 2)

-------------------------------------------
Time: 2016-01-06 14:18:10
-------------------------------------------
(u'gb', 4)
```

Enabling fault-tolerant processing in Spark Streaming

Important: Spark Streaming checkpoints do not work across Spark upgrades or application upgrades. If you are upgrading Spark or your streaming application, you must clear the checkpoint directory.

For long-running Spark Streaming jobs, make sure to configure the maximum allowed failures in a given time period. For example, to allow 3 failures per hour, set the following parameters (in spark-defaults.conf or when submitting the job):

```
spark.yarn.maxAppAttempts=3
spark.yarn.am.attemptFailuresValidityInterval=1h
```

If the driver host for a Spark Streaming application fails, it can lose data that has been received but not yet processed. To ensure that no data is lost, you can use Spark Streaming recovery. Recovery uses a combination of a write-ahead log and checkpoints. Spark writes incoming data to HDFS as it is received and uses this data to recover state if a failure occurs.

To enable Spark Streaming recovery:

1. Set the spark.streaming.receiver.writeAheadLog.enable parameter to true in the SparkConf object.
2. Create a StreamingContext instance using this SparkConf, and specify a checkpoint directory.
3. Use the getOrCreate method in StreamingContext to either create a new context or recover from an old context from the checkpoint directory:

```
from __future__ import print_function

import sys

from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

checkpoint = "hdfs://ns1/user/systest/checkpoint"
# Function to create and setup a new StreamingContext
def functionToCreateContext():

  sparkConf = SparkConf()
  sparkConf.set("spark.streaming.receiver.writeAheadLog.enable", "true")
  sc = SparkContext(appName="PythonStreamingKafkaWordCount",conf=sparkConf
)
  ssc = StreamingContext(sc, 10)

  zkQuorum, topic = sys.argv[1:]
  kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consume
r", {topic: 1})
  lines = kvs.map(lambda x: x[1])
  counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word: (w
ord, 1)).reduceByKey(lambda a, b: a+b)
  counts.pprint()

  ssc.checkpoint(checkpoint)   # set checkpoint directory
  return ssc

if __name__ == "__main__":
  if len(sys.argv) != 3:
    print("Usage: kafka_wordcount.py <zk> <topic>", file=sys.stderr)
    exit(-1)
  ssc = StreamingContext.getOrCreate(checkpoint, lambda: functionToCrea
teContext())
  ssc.start()
```

```
ssc.awaitTermination()
```

For more information, see Checkpointing in the Apache Spark documentation.

To prevent data loss if a receiver fails, receivers must be able to replay data from the original data sources if required.

- The Kafka receiver automatically replays if the spark.streaming.receiver.writeAheadLog.enable parameter is set to true.
- The receiverless Direct Kafka DStream does not require the spark.streaming.receiver.writeAheadLog.enable parameter and can function without data loss, even without Streaming recovery.
- Both Flume receivers packaged with Spark replay the data automatically on receiver failure.

Configuring authentication for long-running Spark Streaming jobs

Long-running applications such as Spark Streaming jobs must be able to write data continuously, which means that the user may need to delegate tokens possibly beyond the default lifetime. This workload type requires passing Kerberos principal and keytab to the spark-submit script using the --principal and --keytab parameters. The keytab is copied to the host running the ApplicationMaster, and the Kerberos login is renewed periodically by using the principal and keytab to generate the required delegation tokens needed for HDFS.

Note: For secure distribution of the keytab to the ApplicationMaster host, the cluster should be configured for TLS/SSL communication for YARN and HDFS encryption.

Building and running a Spark Streaming application

Use the following steps to build and run a Spark streaming job for Cloudera Data Platform (CDP).

Depending on your compilation and build processes, one or more of the following tasks might be required before running a Spark Streaming job:

- If you are using maven as a compile tool:
  1. Add the Cloudera repository to your pom.xml file:

```
<repository>
    <id>cloudera</id>
    <name>Cloudera Repository</name>
    <url>https://archive.cloudera.com/cdh7/7.1.1.0/maven-repository/</ur
l>
</repository>
```

  2. Specify the Cloudera version number for Spark streaming Kafka and streaming dependencies to your pom.xml file:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka_2.10</artifactId>
    <version>2.0.0.2.4.2.0-90</version>
</dependency>

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
    <version>2.4.0.7.0.0.0</version>
    <scope>provided</scope>
</dependency>
```

     Note that the correct version number includes the Spark version and the Cloudera Runtime version.

3. (Optional) If you prefer to pack an uber .jar rather than use the default ("provided"), add the maven-shade-plug in to your pom.xml file:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <filters>
            <filter>
                <artifact>*:*</artifact>
                <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                </excludes>
            </filter>
        </filters>
        <finalName>uber-${project.artifactId}-${project.version}</finalN
ame>
    </configuration>
</plugin>
```

- Instructions for submitting your job depend on whether you used an uber .jar file or not:
  - If you kept the default .jar scope and you can access an external network, use --packages to download dependencies in the runtime library:

```
spark-submit --master yarn-client \
    --num-executors 1 \
    --packages org.apache.spark:spark-streaming-kafka_2.10:2.0.0.2.4.2.0
-90 \
    --repositories https://archive.cloudera.com/cdh7/7.1.1.0/maven-rep
ository/ \
    --class <user-main-class> \
    <user-application.jar> \
    <user arg lists>
```

    The artifact and repository locations should be the same as specified in your pom.xml file.
  - If you packed the .jar file into an uber .jar, submit the .jar file in the same way as you would a regular Spark application:

```
spark-submit --master yarn-client \
    --num-executors 1 \
    --class <user-main-class> \
    <user-uber-application.jar> \
    <user arg lists>
```

1. Select or create a user account to be used as principal.

   This should not be the kafka or spark service account.
2. Generate a keytab for the user.
3. Create a Java Authentication and Authorization Service (JAAS) login configuration file: for example, key.conf.

4. Add configuration settings that specify the user keytab.

   The keytab and configuration files are distributed using YARN local resources. Because they reside in the current directory of the Spark YARN container, you should specify the location as ./v.keytab.

   The following example specifies keytab location ./v.keytab for principal vagrant@example.com:

   ```
   KafkaClient {
      com.sun.security.auth.module.Krb5LoginModule required
      useKeyTab=true
      keyTab="./v.keytab"
      storeKey=true
      useTicketCache=false
      serviceName="kafka"
      principal="vagrant@EXAMPLE.COM";
   };
   ```

5. In your spark-submit command, pass the JAAS configuration file and keytab as local resource files, using the --files option, and specify the JAAS configuration file options to the JVM options specified for the driver and executor:

   ```
   spark-submit \
       --files key.conf#key.conf,v.keytab#v.keytab \
       --driver-java-options "-Djava.security.auth.login.config=./key.conf" \
       --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.c
   onfig=./key.conf" \
   ...
   ```

6. Pass any relevant Kafka security options to your streaming application.

   For example, the KafkaWordCount example accepts PLAINTEXTSASL as the last option in the command line:

   ```
   KafkaWordCount /vagrant/spark-examples.jar c6402:2181 abc ts 1 PLAINTEXT
   SASL
   ```

Sample pom.xml file for Spark Streaming with Kafka

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>test</groupId>
    <artifactId>spark-kafka</artifactId>
    <version>1.0-SNAPSHOT</version>

    <repositories>
        <repository>
            <id>hortonworks</id>
            <name>hortonworks repo</name>
            <url>http://repo.hortonworks.com/content/repositories/releases/
</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka_2.10</artifactId>
            <version>2.0.0.2.4.2.0-90</version>
```

```
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>2.0.0.2.4.2.0-90</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <defaultGoal>package</defaultGoal>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
            <resource>
                <directory>src/test/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <configuration>
                    <encoding>UTF-8</encoding>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>copy-resources</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <recompileMode>incremental</recompileMode>
                    <args>
                        <arg>-target:jvm-1.7</arg>
                    </args>
                    <javacArgs>
                        <javacArg>-source</javacArg>
                        <javacArg>1.7</javacArg>
                        <javacArg>-target</javacArg>
                        <javacArg>1.7</javacArg>
                    </javacArgs>
                </configuration>
                <executions>
                    <execution>
                        <id>scala-compile</id>
                        <phase>process-resources</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>scala-test-compile</id>
                        <phase>process-test-resources</phase>
                        <goals>
                            <goal>testCompile</goal>
```

```
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>

                <executions>
                    <execution>
                        <phase>compile</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>


            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                            </excludes>
                        </filter>
                    </filters>
                    <finalName>uber-${project.artifactId}-${project.versi
on}</finalName>
                </configuration>
            </plugin>

        </plugins>

    </build>
</project>
```

Accessing external storage from Spark

Spark can access all storage sources supported by Hadoop, including a local file system, HDFS, HBase, Amazon S3, and Microsoft ADLS.

Spark supports many file types, including text files, RCFile, SequenceFile, Hadoop InputFormat, Avro, Parquet, and compression of all supported files.

For developer information about working with external storage, see External Datasets in the upstream Apache Spark *RDD Programming Guide*.

Accessing data stored in Amazon S3 through Spark

Important: Cloudera components writing data to S3 are constrained by the inherent limitation of Amazon S3 known as “eventual consistency”. For more information, see Data Storage Considerations.

To access data stored in Amazon S3 from Spark applications: use Hadoop file APIs (SparkContext.hadoopFile, Java HadoopRDD.saveAsHadoopFile, SparkContext.newAPIHadoopRDD, and JavaHadoopRDD.saveAsNewAPIHadoop File) for reading and writing RDDs, providing URLs of the form s3a://*[***BUCKET_NAME***]*/path/to/file. You can read and write Spark SQL DataFrames using the Data Source API.

Make sure that your environment is configured to allow access to the buckets you need. You must also configure

- for Spark 2, the spark.yarn.access.hadoopFileSystems parameter
- for Spark 3, the spark.kerberos.access.hadoopFileSystems parameter

to include the buckets you need to access. You can do this using the Spark client configuration, or at runtime as a command line parameter.

For example:

Spark 2

Client configuration (/etc/spark/conf/spark-defaults.conf)

```
spark.yarn.access.hadoopFileSystems=s3a://bucket1,s3a://bucket2
```

spark-shell

```
spark-shell --conf "spark.yarn.access.hadoopFileSystems=s3a://bu
cket1,s3a://bucket2" ...
```

spark-submit

```
spark-submit --conf "spark.yarn.access.hadoopFileSystems=s3a://b
ucket1,s3a://bucket2" ...
```

Spark 3

Client configuration (/etc/spark3/conf/spark-defaults.conf)

```
spark.kerberos.access.hadoopFileSystems=s3a://bucket1,s3a://buck
et2
```

spark3-shell

```
spark3-shell --conf "spark.kerberos.access.hadoopFileSystems=s3a
://bucket1,s3a://bucket2" ...
```

spark3-submit

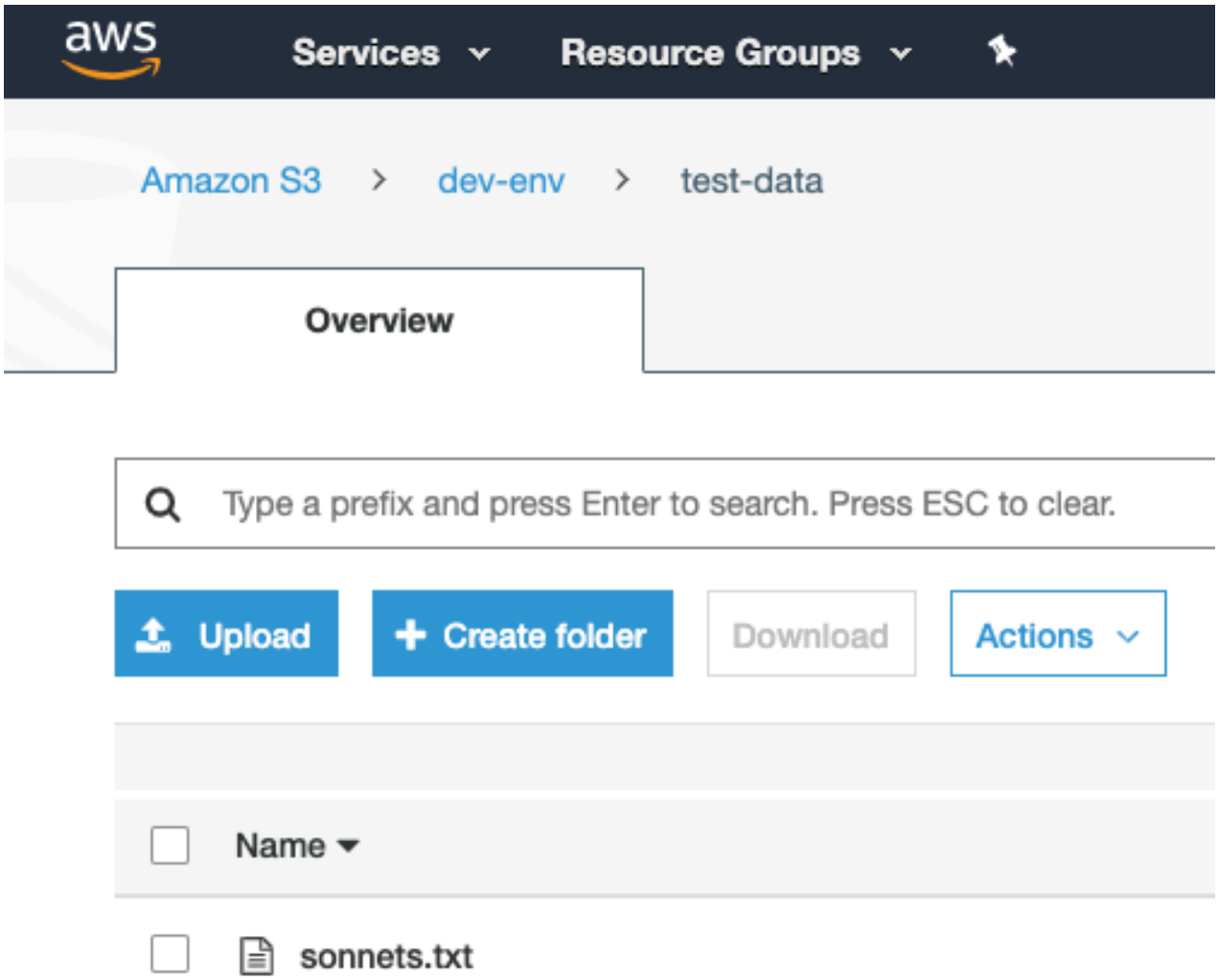
```
spark3-submit --conf "spark.kerberos.access.hadoopFileSystems=s3
a://bucket1,s3a://bucket2" ...
```

Examples of accessing Amazon S3 data from Spark

The following examples demonstrate basic patterns of accessing data in S3 using Spark. The examples show the setup steps, application code, and input and output files located in S3.

Reading and Writing Text Files From and To Amazon S3

Run a word count application on a file stored in Amazon S3 (sonnets.txt in this example):



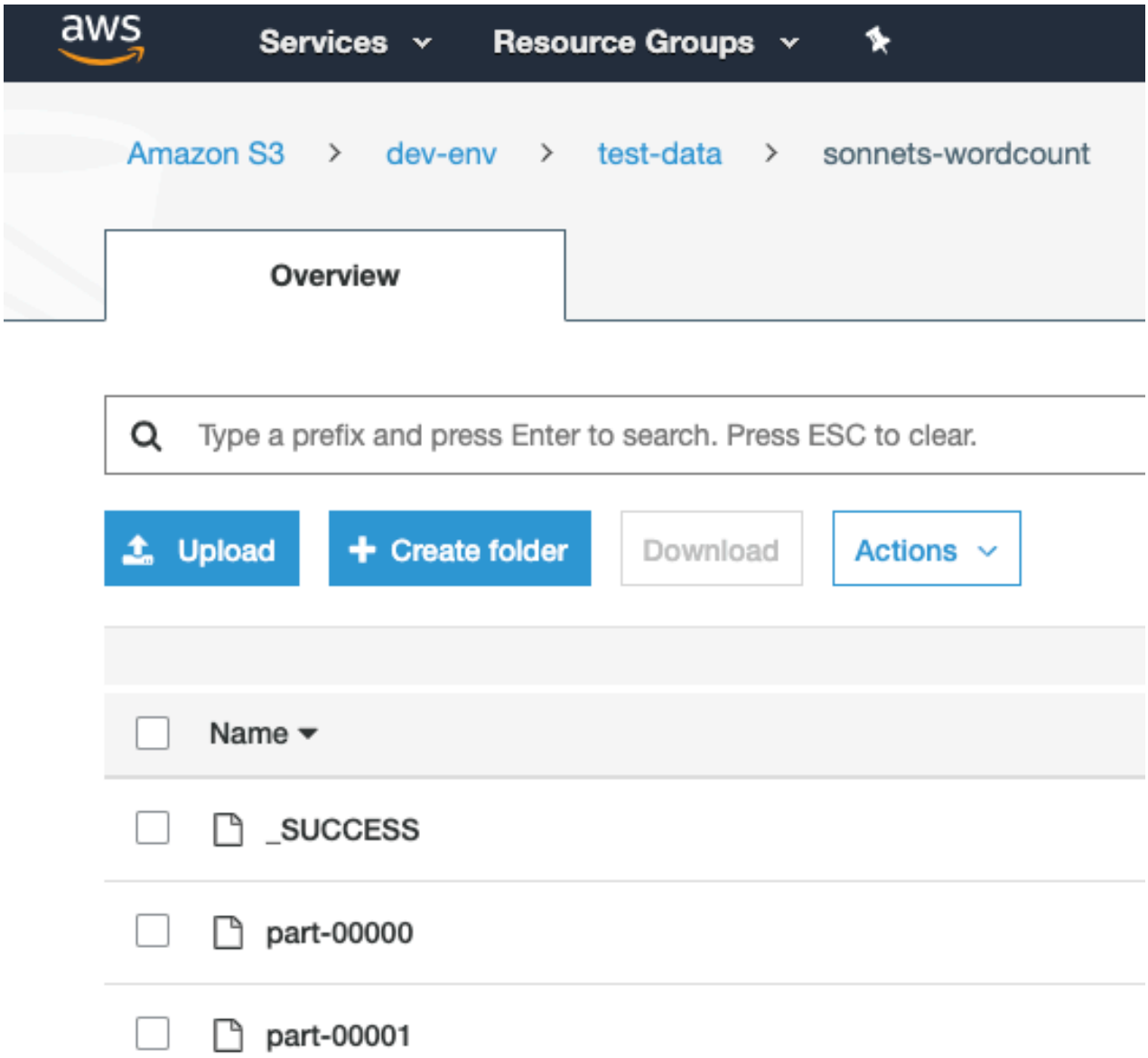
Scala

```
val sonnets = sc.textFile("s3a://dev-env/test-data/sonnets.txt")
val counts = sonnets.flatMap(line => line.split(" ")).map(word =>
 (word, 1)).reduceByKey(_ + _)
counts.saveAsTextFile("s3a://dev-env/test-data/sonnets-wordcoun
t")
```

Python

```
sonnets = sc.textFile("s3a://dev-env/test-data/sonnets.txt")
counts = sonnets.flatMap(lambda line: line.split(" ")).map(lambda
 word: (word, 1)).reduceByKey(lambda v1,v2: v1 + v2)
counts.saveAsTextFile("s3a://dev-env/test-data/sonnets-wordco
unt")
```

Yielding the output:



Reading and Writing Data Sources From and To Amazon S3

The following example illustrates how to read a text file from Amazon S3 into an RDD, convert the RDD to a DataFrame, and then use the Data Source API to write the DataFrame into a Parquet file on Amazon S3:

1. Read a text file in Amazon S3:

```
val sample_data = sc.textFile("s3a://dev-env-data/test-data/sample_data.
csv")
```

2. Map lines into columns:

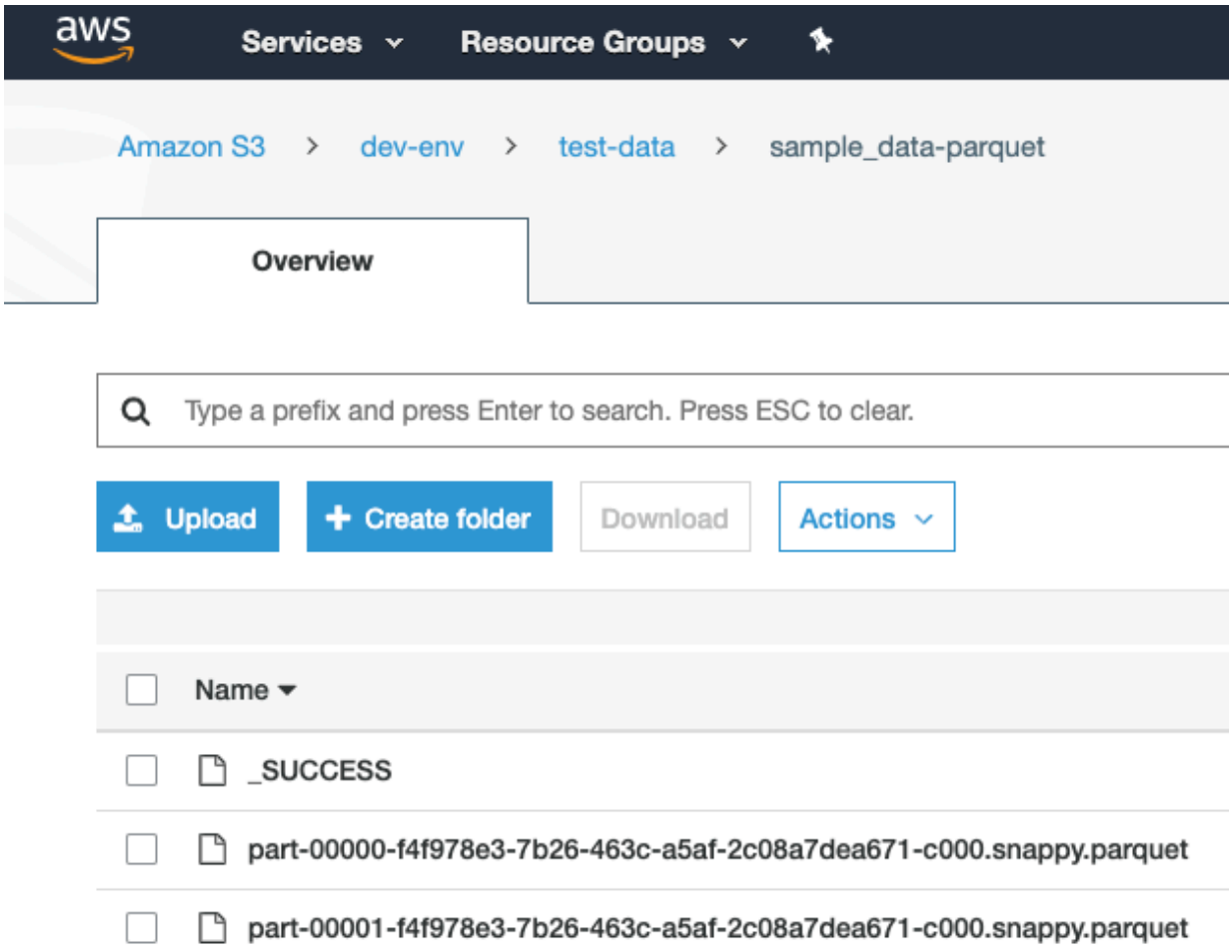
```
import org.apache.spark.sql.Row
val rdd_sample = sample_data.map(_.split('\t')).map(e # Row(e(0), e(1),
e(2).trim.toInt, e(3).trim.toInt))
```

3. Create a schema and apply to the RDD to create a DataFrame:

```
scala> import org.apache.spark.sql.types.{StructType, StructField, Strin
gType, IntegerType};
scala> val schema = StructType(Array(
  StructField("code",StringType,false),
  StructField("description",StringType,false),
  StructField("total_emp",IntegerType,false),
  StructField("salary",IntegerType,false)))
scala> val df_sample = spark.createDataFrame(rdd_sample,schema)
```

4. Write DataFrame to a Parquet file:

```
df_sample.write.parquet("s3a://dev-env-data/test-data/sample_data-parque
t")
```



The files are compressed with the default snappy compression.

Accessing Hive from Spark

The host from which the Spark application is submitted or on which spark-shell or pyspark runs must have a Hive gateway role defined in Cloudera Manager and client configurations deployed.

When a Spark job accesses a Hive view, Spark must have privileges to read the data files in the underlying Hive tables. Currently, Spark cannot use fine-grained privileges based on the columns or the WHERE clause in the view

definition. If Spark does not have the required privileges on the underlying data files, a SparkSQL query against the view returns an empty result set, rather than an error.

Accessing HDFS Files from Spark

This section contains information on running Spark jobs over HDFS data.

Specifying Compression

To add a compression library to Spark, you can use the --jars option. For an example, see "Adding Libraries to Spark" in this guide.

To save a Spark RDD to HDFS in compressed format, use code similar to the following (the example uses the GZip algorithm):

```
rdd.saveAsHadoopFile("/tmp/spark_compressed",
                     "org.apache.hadoop.mapred.TextOutputFormat",
                     compressionCodecClass="org.apache.hadoop.io.compress.
GzipCodec")
```

For more information about supported compression algorithms, see "Configuring HDFS Compression" in the HDP Data Storage guide.

Accessing HDFS from PySpark

When accessing an HDFS file from PySpark, you must set HADOOP_CONF_DIR in an environment variable, as in the following example:

```
$ export HADOOP_CONF_DIR=/etc/hadoop/conf
$ pyspark
$ >>>lines = sc.textFile("hdfs://namenode.example.com:8020/tmp/PySparkTest/f
ile-01")
.......
```

If HADOOP_CONF_DIR is not set properly, you might see an error similar to the following:

```
2016-08-22 00:27:06,046|t1.machine|INFO|1580|140672245782272|MainThread|Py4J
JavaError: An error occurred while calling z:org.apache.spark.api.python.Pyt
honRDD.collectAndServe.
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|: or
g.apache.hadoop.security.AccessControlException: SIMPLE authentication is no
t enabled.  Available:[TOKEN, KERBEROS]
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at s
un.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
2016-08-22 00:27:06,047|t1.machine|INFO|1580|140672245782272|MainThread|at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAcces
sorImpl.java:57)
2016-08-22 00:27:06,048|t1.machine|INFO|1580|140672245782272|MainThread|at
{code}
```

Accessing ORC Data in Hive Tables

Apache Spark in CDP supports the Optimized Row Columnar (ORC) file format, a self-describing, type-aware, column-based file format that is one of the primary file formats supported in Apache Hive.

ORC reduces I/O overhead by accessing only the columns that are required for the current query. It requires significantly fewer seek operations because all columns within a single group of row data (known as a "stripe") are stored together on disk.

Spark ORC data source supports ACID transactions, snapshot isolation, built-in indexes, and complex data types (such as array, map, and struct), and provides read and write access to ORC files. It leverages the Spark SQL Catalyst engine for common optimizations such as column pruning, predicate push-down, and partition pruning.

This subsection has several examples of Spark ORC integration, showing how ORC optimizations are applied to user programs.

Related Information

ORC File Format

Apache Hive ACID Transactions

Accessing ORC files from Spark

Use the following steps to access ORC files from Apache Spark.

About this task

To start using ORC, you can define a SparkSession instance:

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()
import spark.implicits._
```

The following example uses data structures to demonstrate working with complex types. The Person struct data type has a name, an age, and a sequence of contacts, which are themselves defined by names and phone numbers.

Procedure

1. Define Contact and Person data structures:

   ```
   case class Contact(name: String, phone: String)
   case class Person(name: String, age: Int, contacts: Seq[Contact])
   ```

2. Create 100 Person records:

   ```
   val records = (1 to 100).map { i =>;
     Person(s"name_$i", i, (0 to 1).map { m => Contact(s"contact_$m", s"ph
   one_$m") })
   }
   ```

   In the physical file, these records are saved in columnar format. When accessing ORC files through the DataFrame API, you see rows.

3. To write person records as ORC files to a directory named “people”, you can use the following command:

   ```
   records.toDF().write.format("orc").save("people")
   ```

4. Read the objects back:

   ```
   val people = sqlContext.read.format("orc").load("people.json")
   ```

5. For reuse in future operations, register the new "people" directory as temporary table “people”:

   ```
   people.createOrReplaceTempView("people")
   ```

6. After you register the temporary table “people”, you can query columns from the underlying table:

```
spark.sql("SELECT name FROM people WHERE age < 15").count()
```

Results

In this example the physical table scan loads only columns name and age at runtime, without reading the contacts column from the file system. This improves read performance.

You can also use Spark DataFrameReader and DataFrameWriter methods to access ORC files.

Related Information

Apache Spark DataFrameReader Methods

Apache Spark DataFrameWriter Methods

Predicate push-down optimization

The columnar nature of the ORC format helps avoid reading unnecessary columns, but it is still possible to read unnecessary rows. The example in this subsection reads all rows in which the age value is between 0 and 100, even though the query requested rows in which the age value is less than 15 ("...WHERE age < 15"). Such full table scanning is an expensive operation.

ORC avoids this type of overhead by using predicate push-down, with three levels of built-in indexes within each file: file level, stripe level, and row level:

- File-level and stripe-level statistics are in the file footer, making it easy to determine if the rest of the file must be read.
- Row-level indexes include column statistics for each row group and position, for finding the start of the row group.

ORC uses these indexes to move the filter operation to the data loading phase by reading only data that potentially includes required rows.

This combination of predicate push-down with columnar storage reduces disk I/O significantly, especially for larger datasets in which I/O bandwidth becomes the main bottleneck to performance.

ORC predicate push-down is enabled by default in Spark SQL.

Loading ORC data into DataFrames using predicate push-down

DataFrames are similar to Spark RDDs but have higher-level semantics built into their operators. This allows optimization to be pushed down to the underlying query engine.

Here is the Scala API version of the SELECT query used in the previous section, using the DataFrame API:

```
val spark = SparkSession.builder().getOrCreate()
val people = spark.read.format("orc").load("peoplePartitioned")
people.filter(people("age") < 15).select("name").show()
```

DataFrames are not limited to Scala. There is a Java API and a Python API binding:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
people = spark.read.format("orc").load("peoplePartitioned")
people.filter(people.age < 15).select("name").show()
```

Optimizing queries using partition pruning

When predicate push-down optimization is not applicable—for example, if all stripes contain records that match the predicate condition—a query with a WHERE clause might need to read the entire data set. This becomes a bottleneck over a large table. Partition pruning is another optimization method; it exploits query semantics to avoid reading large amounts of data unnecessarily.

Partition pruning is possible when data within a table is split across multiple logical partitions. Each partition corresponds to a particular value of a partition column and is stored as a subdirectory within the table root directory on HDFS. Where applicable, only the required partitions (subdirectories) of a table are queried, thereby avoiding unnecessary I/O.

Spark supports saving data in a partitioned layout seamlessly, through the partitionBy method available during data source write operations. To partition the "people" table by the “age” column, you can use the following command:

```
people.write.format("orc").partitionBy("age").save("peoplePartitioned")
```

As a result, records are automatically partitioned by the age field and then saved into different directories: for example, peoplePartitioned/age=1/, peoplePartitioned/age=2/, and so on.

After partitioning the data, subsequent queries can omit large amounts of I/O when the partition column is referenced in predicates. For example, the following query automatically locates and loads the file under peoplePartitioned/ag e=20/and omits all others:

```
val peoplePartitioned = spark.read.format("orc").load("peoplePartitioned")
peoplePartitioned.createOrReplaceTempView("peoplePartitioned")
spark.sql("SELECT * FROM peoplePartitioned WHERE age = 20")
```

Enabling vectorized query execution

About this task

Vectorized query execution is a feature that greatly reduces the CPU usage for typical query operations such as scans, filters, aggregates, and joins. Vectorization is also implemented for the ORC format. Spark also uses Whole Stage Codegen and this vectorization (for Parquet) since Spark 2.0.

Use the following steps to implement the new ORC format and enable vectorization for ORC files with SparkSQL.

Procedure

1. In the Cloudera Data Platform (CDP) Management Console, go to Data Hub Clusters.
2. Find and select the cluster you want to configure.
3. Click the link for the Cloudera Manager URL.
4. Go to Clusters <Cluster Name>Spark serviceConfiguration .
5. Select the ScopeGateway and CategoryAdvanced filters.
6. Add the following properties to Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/ spark-defaults.conf:
   - spark.sql.orc.enabled=true – Enables the new ORC format to read/write Spark data source tables and files.
   - spark.sql.hive.convertMetastoreOrc=true – Enables the new ORC format to read/write Hive tables.
   - spark.sql.orc.char.enabled=true – Enables the new ORC format to use CHAR types to read Hive tables. By default, STRING types are used for performance reasons. This is an optional configuration for Hive compatibility.
7. Click Save Changes, and then restart Spark and any other components that require a restart.

Reading Hive ORC tables

For existing Hive tables, Spark can read them without createOrReplaceTempView. If the table is stored as ORC format (the default), predicate push-down, partition pruning, and vectorized query execution are also applied according to the configuration.

```
spark.sql("SELECT * FROM hiveTable WHERE age = 20")
```

Accessing Avro data files from Spark SQL applications

Spark SQL supports loading and saving DataFrames from and to a variety of data sources. With the spark-avro library, you can process data encoded in the Avro format using Spark.

The spark-avro library supports most conversions between Spark SQL and Avro records, making Avro a first-class citizen in Spark. The library automatically performs the schema conversion. Spark SQL reads the data and converts it to Spark's internal representation; the Avro conversion is performed only during reading and writing data.

By default, when pointed at a directory, read methods silently skip any files that do not have the .avro extension. To include all files, set the avro.mapred.ignore.inputs.without.extension property to false. See Configuring Spark Applications.

Writing Compressed Data Files

To set the compression type used on write, configure the spark.sql.avro.compression.codec property:

```
sqlContext.setConf("spark.sql.avro.compression.codec","codec")
```

The supported *codec* values are uncompressed, snappy, and deflate. Specify the level to use with deflate compression in spark.sql.avro.deflate.level.

Accessing Partitioned Data Files

The spark-avro library supports writing and reading partitioned data. You pass the partition columns to the writer.

Specifying Record Name and Namespace

Specify the record name and namespace to use when writing to disk by passing recordName and recordNamespace as optional parameters.

Spark SQL

You can write SQL queries to query a set of Avro files. First, create a temporary table pointing to the directory containing the Avro files. Then query the temporary table:

```
sqlContext.sql("CREATE TEMPORARY TABLE table_name
  USING com.databricks.spark.avro OPTIONS (path "input_dir"))
df = sqlContext.sql("SELECT * FROM table_name")
```

Avro to Spark SQL Conversion

The spark-avro library supports conversion for all Avro data types:

- boolean -> BooleanType
- int -> IntegerType
- long -> LongType

- float -> FloatType
- double -> DoubleType
- bytes -> BinaryType
- string -> StringType
- record -> StructType
- enum -> StringType
- array -> ArrayType
- map -> MapType
- fixed -> BinaryType

The spark-avro library supports the following union types:

- union(int, long) -> LongType
- union(float, double) -> DoubleType
- union(any, null) -> any

The library does not support complex union types.

All doc, aliases, and other fields are stripped when they are loaded into Spark.

Spark SQL to Avro Conversion

Every Spark SQL type is supported:

- BooleanType -> boolean
- IntegerType -> int
- LongType -> long
- FloatType -> float
- DoubleType -> double
- BinaryType -> bytes
- StringType -> string
- StructType -> record
- ArrayType -> array
- MapType -> map
- ByteType -> int
- ShortType -> int
- DecimalType -> string
- BinaryType -> bytes
- TimestampType -> long

Limitations

Because Spark is converting data types, keep the following in mind:

- Enumerated types are erased - Avro enumerated types become strings when they are read into Spark, because Spark does not support enumerated types.
- Unions on output - Spark writes everything as unions of the given type along with a null option.
- Avro schema changes - Spark reads everything into an internal representation. Even if you just read and then write the data, the schema for the output is different.
- Spark schema reordering - Spark reorders the elements in its schema when writing them to disk so that the elements being partitioned on are the last elements.

Accessing Parquet files from Spark SQL applications

Spark SQL supports loading and saving DataFrames from and to a variety of data sources and has native support for Parquet.

To read Parquet files in Spark SQL, use the spark.read.parquet("*path*") method.

To write Parquet files in Spark SQL, use the DataFrame.write.parquet("*path*") method.

To set the compression type, configure the spark.sql.parquet.compression.codec property:

```
spark.conf.set("spark.sql.parquet.compression.codec","codec")
```

The supported *codec* values are: uncompressed, gzip, lzo, and snappy. The default is gzip.

Currently, Spark looks up column data from Parquet files by using the names stored within the data files. This is different than the default Parquet lookup behavior of Impala and Hive. If data files are produced with a different physical layout due to added or reordered columns, Spark still decodes the column data correctly. If the logical layout of the table is changed in the metastore database, for example through an ALTER TABLE CHANGE statement that renames a column, Spark still looks for the data using the now-nonexistent column name and returns NULLs when it cannot locate the column values. To avoid behavior differences between Spark and Impala or Hive when modifying Parquet tables, avoid renaming columns, or use Impala, Hive, or a CREATE TABLE AS SELECT statement to produce a new table and new set of Parquet files containing embedded column names that match the new layout.

Using Spark MLlib

MLlib is Spark's machine learning library. For information on MLlib, see the Machine Learning Library (MLlib) Guide.

Running a Spark MLlib example

To try Spark MLlib using one of the Spark example applications, do the following:

1. Download MovieLens sample data and copy it to HDFS:

```
$ wget --no-check-certificate \
https://raw.githubusercontent.com/apache/spark/branch-2.4/data/mllib/sam
ple_movielens_data.txt
$ hdfs dfs -copyFromLocal sample_movielens_data.txt /user/hdfs
```

2. Run the Spark MLlib MovieLens example application, which calculates recommendations based on movie reviews:

```
$ spark-submit --master local --class org.apache.spark.examples.mllib.Mo
vieLensALS \
SPARK_HOME/lib/spark-examples.jar \
--rank 5 --numIterations 5 --lambda 1.0 --kryo sample_movielens_data.txt
```

Enabling Native Acceleration For MLlib

MLlib algorithms are compute intensive and benefit from hardware acceleration. To enable native acceleration for MLlib, perform the following tasks.

Install Required Software

- Install the appropriate libgfortran 4.6+ package for your operating system. No compatible version is available for RHEL 6.

OS	Package Name	Package Version
RHEL 7.1	libgfortran	4.8.x
SLES 11 SP3	libgfortran3	4.7.2
Ubuntu 12.04	libgfortran3	4.6.3
Ubuntu 14.04	libgfortran3	4.8.4
Debian 7.1	libgfortran3	4.7.2

- Install the GPL Extras parcel or package.

Verify Native Acceleration

You can verify that native acceleration is working by examining logs after running an application. To verify native acceleration with an MLlib example application:

1. Do the steps in "Running a Spark MLlib Example."
2. Check the logs. If native libraries are not loaded successfully, you see the following four warnings before the final line, where the RMSE is printed:

```
15/07/12 12:33:01 WARN BLAS: Failed to load implementation from: com.git
hub.fommil.netlib.NativeSystemBLAS
15/07/12 12:33:01 WARN BLAS: Failed to load implementation from: com.gith
ub.fommil.netlib.NativeRefBLAS
15/07/12 12:33:01 WARN LAPACK: Failed to load implementation from: com.gi
thub.fommil.netlib.NativeSystemLAPACK
15/07/12 12:33:01 WARN LAPACK: Failed to load implementation from: com.git
hub.fommil.netlib.NativeRefLAPACK
Test RMSE = 1.5378651281107205.
```

You see this on a system with no libgfortran. The same error occurs after installing libgfortran on RHEL 6 because it installs version 4.4, not 4.6+.

After installing libgfortran 4.8 on RHEL 7, you should see something like this:

```
15/07/12 13:32:20 WARN BLAS: Failed to load implementation from: com.git
hub.fommil.netlib.NativeSystemBLAS
15/07/12 13:32:20 WARN LAPACK: Failed to load implementation from: com.gi
thub.fommil.netlib.NativeSystemLAPACK
Test RMSE = 1.5329939324808561.
```

Using custom libraries with Spark

Spark comes equipped with a selection of libraries, including Spark SQL, Spark Streaming, and MLlib.

If you want to use a custom library, such as a compression library or Magellan, you can use one of the following two spark-submit script options:

- The --jars option, which transfers associated .jar files to the cluster. Specify a list of comma-separated .jar files.
- The --packages option, which pulls files directly from Spark packages. This approach requires an internet connection.

For example, you can use the --jars option to add codec files. The following example adds the LZO compression library:

```
spark-submit --driver-memory 1G \
    --executor-memory 1G \
    --master yarn-client \
    --jars /usr/hdp/2.6.0.3-8/hadoop/lib/hadoop-lzo-0.6.0.2.6.0.3-8.jar \
    test_read_write.py
```

For more information about the two options, see Advanced Dependency Management on the Apache Spark "Submitting Applications" web page.

Note:

If you launch a Spark job that references a codec library without specifying where the codec resides, Spark returns an error similar to the following:

```
Caused by: java.lang.IllegalArgumentException: Compression codec com
.hadoop.compression.lzo.LzoCodec not found.
```

To address this issue, specify the codec file with the --jars option in your job submit command.

Related Information

Submitting Applications: Advanced Dependency Management