cloudera®

Spark Guide

Important Notice

Cloudera, Inc.
1001 Page Mill Road, Bldg 3
Palo Alto, CA 94304
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: CDH 5.6.x
Date: August 24, 2017

Table of Contents

Apache Spark Overview

Apache Spark is a general framework for distributed computing that offers high performance for both batch and interactive processing. It exposes APIs for Java, Python, and Scala and consists of Spark core and several related projects:

- Spark SQL - Module for working with structured data. Allows you to seamlessly mix SQL queries with Spark programs.
- Spark Streaming - API that allows you to build scalable fault-tolerant streaming applications.
- MLlib - API that implements common machine learning algorithms.
- GraphX - API for graphs and graph-parallel computation.

You can run Spark applications locally or distributed across a cluster, either by using an interactive shell or by submitting an application. Running Spark applications interactively is commonly performed during the data-exploration phase and for ad-hoc analysis.

To run applications distributed across a cluster, Spark requires a cluster manager. Cloudera supports two cluster managers: YARN and Spark Standalone. When run on YARN, Spark application processes are managed by the YARN ResourceManager and NodeManager roles. When run on Spark Standalone, Spark application processes are managed by Spark Master and Worker roles.

Unsupported Features

The following Spark features are not supported:

- Spark SQL:
  - `spark.ml`
  - ML pipeline APIs
- Spark MLib:
  - `spark.ml`
- SparkR
- GraphX
- Spark on Scala 2.11
- Mesos cluster manager

Related Information

- Managing Spark
- Monitoring Spark Applications
- Spark Authentication
- Cloudera Spark forum
- Apache Spark documentation

Running Your First Spark Application

The simplest way to run a Spark application is by using the Scala or Python shells.

1. To start one of the shell applications, run one of the following commands:

   - Scala:

```
$ SPARK_HOME/bin/spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version ...
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information
...
SQL context available as sqlContext.

scala>
```

   - Python:

```
$ SPARK_HOME/bin/pyspark
Python 2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information
...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version ...
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

   In a CDH deployment, *SPARK_HOME* defaults to `/usr/lib/spark` in package installations and `/opt/cloudera/parcels/CDH/lib/spark` in parcel installations. In a Cloudera Manager deployment, the shells are also available from `/usr/bin`.

   For a complete list of shell options, run `spark-shell` or `pyspark` with the `-h` flag.

2. To run the classic Hadoop word count application, copy an input file to HDFS:

```
$ hdfs dfs -put input
```

3. Within a shell, run the word count application using the following code examples, substituting for *namenode_host*, *path/to/input*, and *path/to/output*:

   - Scala

```
scala> val myfile = sc.textFile("hdfs://namenode_host:8020/path/to/input")
scala> val counts = myfile.flatMap(line => line.split(" ")).map(word => (word,
1)).reduceByKey(_ + _)
scala> counts.saveAsTextFile("hdfs://namenode:8020/path/to/output")
```

- Python

```
>>> myfile = sc.textFile("hdfs://namenode_host:8020/path/to/input")
>>> counts = myfile.flatMap(lambda line: line.split(" ")).map(lambda word: (word,
1)).reduceByKey(lambda v1,v2: v1 + v2)
>>> counts.saveAsTextFile("hdfs://namenode:8020/path/to/output")
```

Spark Application Overview

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 job 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.

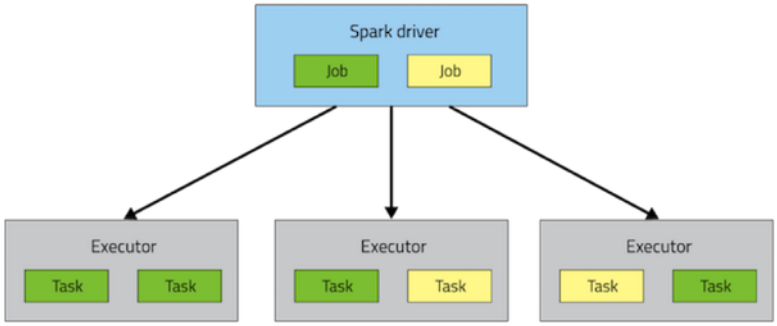
Spark Execution Model

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 executing 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 Spark Applications

When you are ready to move beyond running core Spark applications in an interactive shell, you need best practices for building, packaging, and configuring applications and using the more advanced APIs. This section describes how to develop, package, and run Spark applications, aspects of using Spark APIs beyond core Spark, how to access data stored in Amazon S3, and best practices in building and configuring Spark applications.

Developing and Running a Spark WordCount Application

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

Writing the Application

The 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.

Here are three versions of the program:

- Figure 1: Scala WordCount on page 9
- Figure 2: Python WordCount on page 10
- Figure 3: Java 7 WordCount on page 10


```
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("Spark Count"))

    // 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)).reduceByKey(_ + _)
```

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

Figure 1: Scala WordCount

```
import sys

from pyspark import SparkContext, SparkConf

if __name__ == "__main__":

  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("Spark Count")
  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).map(lambda c: (c,
 1)).reduceByKey(lambda v1,v2:v1 +v2)

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

Figure 2: Python WordCount

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.SparkConf;
import scala.Tuple2;

public class JavaWordCount {
  public static void main(String[] args) {

    // create Spark context with Spark configuration
    JavaSparkContext sc = new JavaSparkContext(new SparkConf().setAppName("Spark Count"));


    // get threshold
    final int threshold = Integer.parseInt(args[1]);

    // read in text file and split each document into words
    JavaRDD<String> tokenized = sc.textFile(args[0]).flatMap(
      new FlatMapFunction() {
        public Iterable call(String s) {
          return Arrays.asList(s.split(" "));
        }
      }
    );

    // count the occurrence of each word
    JavaPairRDD<String, Integer> counts = tokenized.mapToPair(
      new PairFunction() {
        public Tuple2 call(String s) {
          return new Tuple2(s, 1);
        }
```

```
      }
    ).reduceByKey(
      new Function2() {
        public Integer call(Integer i1, Integer i2) {
          return i1 + i2;
        }
      }
    );

    // filter out words with fewer than threshold occurrences
    JavaPairRDD<String, Integer> filtered = counts.filter(
      new Function, Boolean>() {
        public Boolean call(Tuple2 tup) {
          return tup._2 >= threshold;
        }
      }
    );

    // count characters
    JavaPairRDD<Character, Integer> charCounts = filtered.flatMap(
      new FlatMapFunction<Tuple2<String, Integer>, Character>() {
        @Override
        public Iterable<Character> call(Tuple2<String, Integer> s) {
          Collection<Character> chars = new ArrayList<Character>(s._1().length());
          for (char c : s._1().toCharArray()) {
            chars.add(c);
          }
          return chars;
        }
      }
    ).mapToPair(
      new PairFunction<Character, Character, Integer>() {
        @Override
        public Tuple2<Character, Integer> call(Character c) {
          return new Tuple2<Character, Integer>(c, 1);
        }
      }
    ).reduceByKey(
      new Function2<Integer, Integer, Integer>() {
        @Override
        public Integer call(Integer i1, Integer i2) {
          return i1 + i2;
        }
      }
    );

    System.out.println(charCounts.collect());
  }
}
```

Figure 3: Java 7 WordCount

Because Java 7 does not support anonymous functions, this Java program is considerably more verbose than Scala and Python, but still requires a fraction of the code needed in an equivalent MapReduce program. Java 8 supports anonymous functions and their use can further streamline the Java application.

Compiling and Packaging the Scala and Java Applications

The tutorial uses Maven to compile and package the Scala and Java programs. Excerpts of the tutorial `pom.xml` are included below. For best practices using Maven to build Spark applications, see Building Spark Applications on page 24.

To compile Scala, include the Scala tools plug-in:

```
<plugin>
  <groupId>org.scala-tools</groupId>
      <xrefrtifactId>maven-scala-plugin</artifactId>
      <executions>
        <execution>
```

```
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
</plugin>
```

which requires the `scala-tools` plug-in repository:

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

Also, include Scala and Spark as dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <xrefrtifactId>scala-library</artifactId>
    <version>2.10.2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <xrefrtifactId>spark-core_2.10</artifactId>
    <version>1.5.0-cdh5.6.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

To generate the application JAR, run:

```
$ mvn package
```

to create `sparkwordcount-1.0-SNAPSHOT-jar-with-dependencies.jar` in the `target` directory.

Running the Application

1. 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 on HDFS. You can use tutorial example input file:

```
$ wget --no-check-certificate .../inputfile.txt
$ hdfs dfs -put inputfile.txt
```

2. Run one of the applications using `spark-submit`:

   - Scala - Run in a local process with threshold 2:

```
$ spark-submit --class com.cloudera.sparkwordcount.SparkWordCount \
--master local --deploy-mode client --executor-memory 1g \
--name wordcount --conf "spark.app.id=wordcount" \
sparkwordcount-1.0-SNAPSHOT-jar-with-dependencies.jar
hdfs://namenode_host:8020/path/to/inputfile.txt 2
```

If you use the example input file, the output should look something like:

```
(e,6), (p,2), (a,4), (t,2), (i,1), (b,1), (u,1), (h,1), (o,2), (n,4), (f,1), (v,1),
(r,2), (l,1), (c,1)
```

- Java - Run in a local process with threshold 2:

```
$ spark-submit --class com.cloudera.sparkwordcount.JavaWordCount \
--master local --deploy-mode client --executor-memory 1g \
--name wordcount --conf "spark.app.id=wordcount" \
sparkwordcount-1.0-SNAPSHOT-jar-with-dependencies.jar
hdfs://namenode_host:8020/path/to/inputfile.txt 2
```

If you use the example input file, the output should look something like:

```
(e,6), (p,2), (a,4), (t,2), (i,1), (b,1), (u,1), (h,1), (o,2), (n,4), (f,1), (v,1),
(r,2), (l,1), (c,1)
```

- Python - Run on YARN with threshold 2

```
$ spark-submit --master yarn --deploy-mode client --executor-memory 1g \
--name wordcount --conf "spark.app.id=wordcount" wordcount.py
hdfs://namenode_host:8020/path/to/inputfile.txt 2
```

In this case, the output should look something like:

```
[(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 Spark Streaming

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 and divides the data into batches called DStreams. DStreams can be created either from sources such as Kafka, Flume, and Kinesis, or by applying operations on other DStreams. Every input DStream is associated with a `Receiver`, which receives the data from a source and stores it in executor memory.

For detailed information on Spark Streaming, see Spark Streaming Programming Guide.

Spark Streaming and Dynamic Allocation

Starting with CDH 5.5, dynamic allocation is enabled by default, which means that executors are removed when idle. However, dynamic allocation is not effective in Spark Streaming. In Spark Streaming, data comes in every batch, and executors run whenever data is available. If the executor idle timeout is less than the batch duration, executors are constantly being 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. Install Kafka and create a Kafka service.
2. Create a Kafka topic `wordcounttopic` and pass in your ZooKeeper server:

```
$ kafka-topics --create --zookeeper zookeeper_server:2181 --topic wordcounttopic \
--partitions 1 --replication-factor 1
```

3. 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)
        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()
```

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

```
$ spark-submit --master local[2] --conf "spark.dynamicAllocation.enabled=false" \
--jars SPARK_HOME/lib/spark-examples.jar kafka_wordcount.py \
zookeeper_server:2181 wordcounttopic
```

In a CDH deployment, *SPARK_HOME* defaults to `/usr/lib/spark` in package installations and `/opt/cloudera/parcels/CDH/lib/spark` in parcel installations. In a Cloudera Manager deployment, the shells are also available from `/usr/bin`.

Alternatively, you can run on YARN as follows:

```
$ spark-submit --master yarn --deploy-mode client --conf
"spark.dynamicAllocation.enabled=false" \
--jars SPARK_HOME/lib/spark-examples.jar kafka_wordcount.py \
zookeeper_server:2181 wordcounttopic
```

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

```
$ kafka-console-producer --broker-list kafka_broker:9092 --topic wordcounttopic
```

6. 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

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. 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-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.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: functionToCreateContext())
  ssc.start()
  ssc.awaitTermination()
```

For more information, see Checkpointing.

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.

For more information, see Spark Streaming + Kafka Integration Guide and Spark Streaming + Flume Integration Guide.

Configuring Authentication for Long-Running Spark Streaming Jobs

If you are using authenticated Spark communication, you must perform additional configuration steps for long-running Spark Streaming jobs. See Configuring Spark on YARN for Long-running Applications.

Using Spark SQL

Spark SQL lets you query structured data inside Spark programs using either SQL or the `DataFrame` API.

The entry point to all Spark SQL functionality is the `SQLContext` class or one of its descendants. You create a `SQLContext` from a `SparkContext`. With an `SQLContext`, you can create a `DataFrame` from an RDD, a Hive table, or a data source.

To work with data stored in Hive from Spark applications, construct a `HiveContext`, which inherits from `SQLContext`. With a `HiveContext`, you can access tables in the Hive Metastore and write queries using HiveQL. If you use `spark-shell`, a `HiveContext` is already created for you and is available as the `sqlContext` variable. To access Hive tables you must also perform the steps in Accessing Hive from Spark on page 47.

For detailed information on Spark SQL, see the Spark SQL and DataFrame Guide.

Spark SQL Example

This example demonstrates how to use `sqlContext.sql` to create and load a table and select rows from the table into a DataFrame. The next steps use the DataFrame API to filter the rows for salaries greater than 150,000 and show the resulting DataFrame.

1. At the command-line, copy the Hue sample_07 data to HDFS:

```
$ hdfs dfs -put HUE_HOME/apps/beeswax/data/sample_07.csv /user/hdfs
```

   where *HUE_HOME* defaults to `/opt/cloudera/parcels/CDH/lib/hue` (parcel installation) or `/usr/lib/hue` (package installation).

2. Start `spark-shell`:

```
$ spark-shell
```

3. Create a Hive table:

```
scala> sqlContext.sql("CREATE TABLE sample_07 (code string,description string,total_emp
 int,salary int) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TextFile")
```

4. Load data from HDFS into the table:

```
scala> sqlContext.sql("LOAD DATA INPATH '/user/hdfs/sample_07.csv' OVERWRITE INTO TABLE
 sample_07")
```

5. Create a DataFrame containing the contents of the sample_07 table:

```
scala> val df = sqlContext.sql("SELECT * from sample_07")
```

6. Show all rows with salary greater than 150,000:

```
scala> df.filter(df("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|
+-------+--------------------+---------+------+
```

Ensuring HiveContext Enforces Secure Access

To ensure that `HiveContext` enforces ACLs, enable the HDFS-Sentry plug-in as described in Synchronizing HDFS ACLs and Sentry Permissions. Column-level access control for access from Spark SQL is not supported by the HDFS-Sentry plug-in.

Interaction 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 the required privileges on the underlying data files, a SparkSQL query against the view returns an empty result set, rather than an error.

Using the spark-avro Library to Access Avro Data Sources

Spark supports loading and saving DataFrames from a variety of data sources. The spark-avro library allows you to process data encoded in the Avro format using Spark. No Maven `pom.xml` configuration or other changes are required.

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.

Partitioning

The spark-avro library allows you to write and read partitioned data without extra configuration. As you do when writing Parquet, simply pass the columns you want to partition by to the writer. See Figure 7: Writing Partitioned Data on page 19 and Figure 8: Reading Partitioned Data on page 20.

Compression

Specify the compression used on write by setting the Spark configuration `spark.sql.avro.compression.codec`. The supported compression types are `uncompressed`, `snappy`, and `deflate`. Specify the level to use with `deflate` compression in `spark.sql.avro.deflate.level`. For an example, see Figure 6: Writing Deflate Compressed Records on page 19.

Avro Record Name

Specify the record name and namespace to use when writing to disk by passing `recordName` and `recordNamespace` as optional parameters. See Figure 9: Specifying a Record Name on page 20.

Avro to Spark SQL Conversion

The spark-avro library supports conversion for all Avro data types, except complex union 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

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.
- 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 will be 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. See Figure 7: Writing Partitioned Data on page 19 for an example.

API Examples

This section provides examples of using the spark-avro API in all supported languages.

Scala Examples

The easiest way to work with Avro data files in Spark applications is by using the DataFrame API. The `spark-avro` library includes `avro` methods in `SQLContext` for reading and writing Avro files:

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)

// The Avro records are converted to Spark types, filtered, and
// then written back out as Avro records
val df = sqlContext.read.avro("input dir")
df.filter("age > 5").write.avro("output dir")
```

Figure 4: Scala Example with Function

You can also specify the format "`com.databricks.spark.avro`":

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)

val df = sqlContext.read.format("com.databricks.spark.avro").load("input dir")

df.filter("age > 5").write.format("com.databricks.spark.avro").save("output dir")
```

Figure 5: Scala Example with Format

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)

// configuration to use deflate compression
sqlContext.setConf("spark.sql.avro.compression.codec", "deflate")
sqlContext.setConf("spark.sql.avro.deflate.level", "5")

val df = sqlContext.read.avro("input dir")

// writes out compressed Avro records
df.write.avro("output dir")
```

Figure 6: Writing Deflate Compressed Records

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)

import sqlContext.implicits._

val df = Seq(
(2012, 8, "Batman", 9.8),
(2012, 8, "Hero", 8.7),
(2012, 7, "Robot", 5.5),
(2011, 7, "Git", 2.0)).toDF("year", "month", "title", "rating")

df.write.partitionBy("year", "month").avro("output dir")
```

Figure 7: Writing Partitioned Data

This code outputs a directory structure like this:

```
-rw-r--r--   3 hdfs supergroup          0 2015-11-03 14:58 /tmp/output/_SUCCESS
drwxr-xr-x   - hdfs supergroup          0 2015-11-03 14:58 /tmp/output/year=2011
drwxr-xr-x   - hdfs supergroup          0 2015-11-03 14:58 /tmp/output/year=2011/month=7
-rw-r--r--   3 hdfs supergroup        229 2015-11-03 14:58
/tmp/output/year=2011/month=7/part-r-00001-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro
drwxr-xr-x   - hdfs supergroup          0 2015-11-03 14:58 /tmp/output/year=2012
drwxr-xr-x   - hdfs supergroup          0 2015-11-03 14:58 /tmp/output/year=2012/month=7
-rw-r--r--   3 hdfs supergroup        231 2015-11-03 14:58
/tmp/output/year=2012/month=7/part-r-00001-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro
drwxr-xr-x   - hdfs supergroup          0 2015-11-03 14:58 /tmp/output/year=2012/month=8
-rw-r--r--   3 hdfs supergroup        246 2015-11-03 14:58
/tmp/output/year=2012/month=8/part-r-00000-9b89f1bd-7cf8-4ba8-910f-7587c0de5a90.avro
```

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)
val df = sqlContext.read.avro("input dir")

df.printSchema()
df.filter("year = 2011").collect().foreach(println)
```

Figure 8: Reading Partitioned Data

This code automatically detects the partitioned data and joins it all, so it is treated the same as unpartitioned data. This also queries only the directory required, to decrease disk I/O.

```
root
|-- title: string (nullable = true)
|-- rating: double (nullable = true)
|-- year: integer (nullable = true)
|-- month: integer (nullable = true)

[Git,2.0,2011,7]
```

```
import com.databricks.spark.avro._

val sqlContext = new SQLContext(sc)
val df = sqlContext.read.avro("input dir")

val name = "AvroTest"
val namespace = "com.cloudera.spark"
val parameters = Map("recordName" -> name, "recordNamespace" -> namespace)

df.write.options(parameters).avro("output dir")
```

Figure 9: Specifying a Record Name

Java Example

Use the DataFrame API to query Avro files in Java. This example is almost identical to Figure 5: Scala Example with Format on page 19:

```
import org.apache.spark.sql.*;

SQLContext sqlContext = new SQLContext(sc);

// Creates a DataFrame from a file
DataFrame df = sqlContext.read().format("com.databricks.spark.avro").load("input dir");

// Saves the subset of the Avro records read in
df.filter("age > 5").write().format("com.databricks.spark.avro").save("output dir");
```

Python Example

Use the DataFrame API to query Avro files in Python. This example is almost identical to Figure 5: Scala Example with Format on page 19:

```
# Creates a DataFrame from a directory
df = sqlContext.read.format("com.databricks.spark.avro").load("input dir")

#  Saves the subset of the Avro records read in
df.where("age > 5").write.format("com.databricks.spark.avro").save("output dir")
```

Spark SQL Example

You can also write SQL queries against the DataFrames directly:

```
CREATE TEMPORARY TABLE table_name USING com.databricks.spark.avro OPTIONS (path "input
 dir")
```

Using Spark MLlib

CDH 5.5 supports MLlib, 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-1.5/data/mllib/sample_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.MovieLensALS \
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 5 or 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 on page 21.
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.github.fommil.netlib.NativeSystemBLAS
15/07/12 12:33:01 WARN BLAS: Failed to load implementation from:
com.github.fommil.netlib.NativeRefBLAS
15/07/12 12:33:01 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemLAPACK
15/07/12 12:33:01 WARN LAPACK: Failed to load implementation from:
com.github.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.github.fommil.netlib.NativeSystemBLAS
15/07/12 13:32:20 WARN LAPACK: Failed to load implementation from:
com.github.fommil.netlib.NativeSystemLAPACK
Test RMSE = 1.5329939324808561.
```

Accessing Data Stored in Amazon S3 through Spark

To access data stored in Amazon S3 from Spark applications, you use Hadoop file APIs (`SparkContext.hadoopFile`, `JavaHadoopRDD.saveAsHadoopFile`, `SparkContext.newAPIHadoopRDD`, and `JavaHadoopRDD.saveAsNewAPIHadoopFile`) 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.

You can access Amazon S3 by the following methods:

Without credentials:

Run EC2 instances with instance profiles associated with IAM roles that have the permissions you want. Requests from a machine with such a profile authenticate without credentials.

With credentials:

- Specify the credentials in a configuration file, such as `core-site.xml`:

```
<property>
    <name>fs.s3a.access.key</name>
    <value>...</value>
</property>
<property>
    <name>fs.s3a.secret.key</name>
    <value>...</value>
</property>
```

- Specify the credentials at run time. For example:

```
sc.hadoopConfiguration.set("fs.s3a.access.key", "...")
  sc.hadoopConfiguration.set("fs.s3a.secret.key", "...")
```

Reading and Writing Text Files From and To Amazon S3

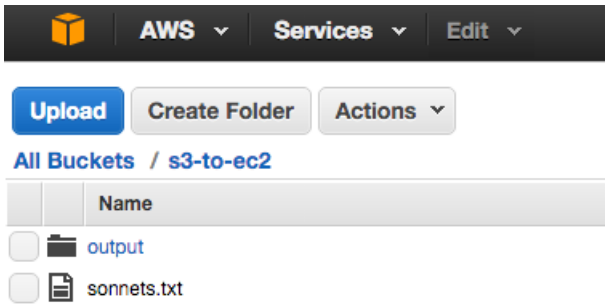
After specifying credentials:

```
scala> sc.hadoopConfiguration.set("fs.s3a.access.key", "...")
scala> sc.hadoopConfiguration.set("fs.s3a.secret.key", "...")
```

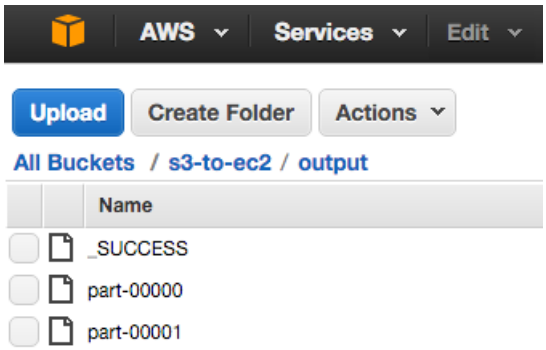
you can perform the word count application:

```
scala> val sonnets = sc.textFile("s3a://s3-to-ec2/sonnets.txt")
scala> val counts = sonnets.flatMap(line => line.split(" ")).map(word => (word,
1)).reduceByKey(_ + _)
scala> counts.saveAsTextFile("s3a://s3-to-ec2/output")
```

on a `sonnets.txt` file stored in Amazon S3:



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. Specify Amazon S3 credentials:

```
scala> sc.hadoopConfiguration.set("fs.s3a.access.key", "...")
scala> sc.hadoopConfiguration.set("fs.s3a.secret.key", "...")
```

2. Read a text file in Amazon S3:

```
scala> val sample_07 = sc.textFile("s3a://s3-to-ec2/sample_07.csv")
```

3. Map lines into columns:

```
scala> import org.apache.spark.sql.Row
scala> val rdd_07 = sample_07.map(_.split('\t')).map(e   Row(e(0), e(1), e(2).trim.toInt,
 e(3).trim.toInt))
```

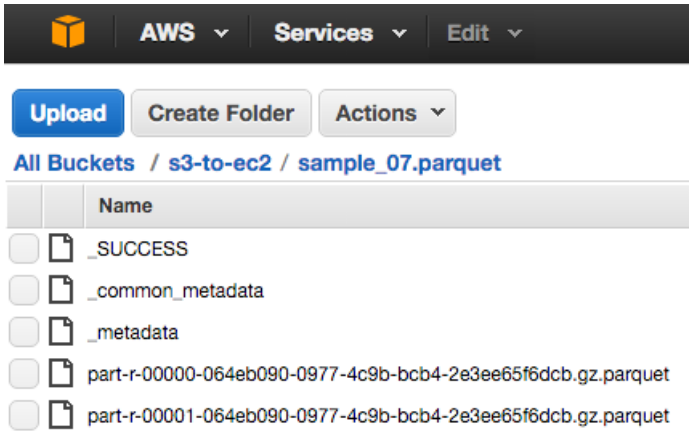
4. Create a schema and apply to the RDD to create a DataFrame:

```
scala> import org.apache.spark.sql.types.{StructType, StructField, StringType,
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_07 = sqlContext.createDataFrame(rdd_07,schema)
```

5. Write DataFrame to a Parquet file:

```
scala> df_07.write.parquet("s3a://s3-to-ec2/sample_07.parquet")
```



The files are compressed with the default gzip compression.

Building Spark Applications

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

For the Maven properties of CDH 5 components, see Using the CDH 5 Maven Repository. For the Maven properties of Kafka, see Maven Artifacts for Kafka.

Building Applications

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

- Compile 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 (CDH 5.5 and higher) 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_2.10</artifactId>
  <version>1.5.0-cdh5.5.0</version>
  <scope>provided</scope>
</dependency>
```

Building Reusable Modules

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=infodynamics-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 -Dpackaging=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;
```

Configuring Spark Applications

You can specify Spark application configuration properties as follows:

- Pass properties using the `--conf` command-line switch; for example:

```
spark-submit \
--class com.cloudera.example.YarnExample \
--master yarn \
--deploy-mode cluster \
--conf "spark.eventLog.dir=hdfs:///user/spark/eventlog" \
lib/yarn-example.jar \
10
```

- Specify properties in `spark-defaults.conf`. See Configuring Spark Application Properties in spark-defaults.conf on page 27.
- Pass properties directly to the SparkConf used to create the SparkContext in your Spark application; for example:

```
val conf = new SparkConf().set("spark.dynamicAllocation.initialExecutors", "5")
val sc = new SparkContext(conf)
```

The order of precedence in configuration properties is:

1. Properties passed to `SparkConf`.
2. Arguments passed to `spark-submit`, `spark-shell`, or `pyspark`.
3. Properties set in `spark-defaults.conf`.

For more information, see Spark Configuration.

Configuring Spark Application Properties in spark-defaults.conf

Specify properties in the `spark-defaults.conf` file in the form `property value`.

You create a comment by putting a hash mark (#) at the beginning of a line. You cannot add comments to the end or middle of a line.

This example shows a `spark-defaults.conf` file:

```
spark.master     spark://mysparkmaster.acme.com:7077
spark.eventLog.enabled    true
spark.eventLog.dir        hdfs:///user/spark/eventlog
# Set spark executor memory
spark.executor.memory     2g
spark.logConf             true
```

Cloudera recommends placing configuration properties that you want to use for every application in `spark-defaults.conf`. See Application Properties for more information.

Configuring Properties in spark-defaults.conf Using Cloudera Manager

You configure properties for all Spark applications in `spark-defaults.conf` as follows:

1. Go to the Spark service.
2. Click the **Configuration** tab.
3. Select **Scope** > **Gateway**.
4. Select **Category** > **Advanced**.
5. Locate the **Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-defaults.conf** property.
6. Specify properties described in Application Properties.

   If more than one role group applies to this configuration, edit the value for the appropriate role group. See .
7. Click **Save Changes** to commit the changes.
8. Deploy the client configuration.

Configuring Properties in spark-defaults.conf Using the Command Line

Important:

- If you use Cloudera Manager, do not use these command-line instructions.
- This information applies specifically to CDH 5.6.x. If you use a lower version of CDH, see the documentation for that version located at Cloudera Documentation.

To configure properties for all Spark applications using the command line, edit the file *SPARK_HOME*`/conf/spark-defaults.conf`.

Configuring Spark Application Logging Properties

You configure Spark application logging properties in a `log4j.properties` file.

Configuring Logging Properties Using Cloudera Manager

1. Go to the Spark service.
2. Click the **Configuration** tab.
3. Select **Scope** > **Gateway**.
4. Select **Category** > **Advanced**.
5. Locate the **Spark Client Advanced Configuration Snippet (Safety Valve) for spark-conf/log4j.properties** property.
6. Specify log4j properties.

   If more than one role group applies to this configuration, edit the value for the appropriate role group. See Modifying Configuration Properties.
7. Click **Save Changes** to commit the changes.
8. Deploy the client configuration.

Configuring Logging Properties Using the Command Line

> **Important:**
>
> - If you use Cloudera Manager, do not use these command-line instructions.
> - This information applies specifically to CDH 5.6.x. If you use a lower version of CDH, see the documentation for that version located at Cloudera Documentation.

To specify logging properties for all users on a machine using the command line, edit the file *SPARK_HOME*`/conf/log4j.properties`. To set it just for yourself or for a specific application, copy *SPARK_HOME*`/conf/log4j.properties.template` to `log4j.properties` in your working directory or any directory in your application's classpath.

Running Spark Applications

You can run Spark applications locally or distributed across a cluster, either by using an interactive shell or by submitting an application. Running Spark applications interactively is commonly performed during the data-exploration phase and for ad-hoc analysis.

Because of a limitation in the way Scala compiles code, some applications with nested definitions running in an interactive shell may encounter a `Task not serializable` exception. Cloudera recommends submitting these applications.

To run applications distributed across a cluster, Spark requires a cluster manager. Cloudera supports two cluster managers: YARN and Spark Standalone. When run on YARN, Spark application processes are managed by the YARN ResourceManager and NodeManager roles. When run on Spark Standalone, Spark application processes are managed by Spark Master and Worker roles.

In CDH 5, Cloudera recommends running Spark applications on a YARN cluster manager instead of on a Spark Standalone cluster manager, for the following benefits:

- You can dynamically share and centrally configure the same pool of cluster resources among all frameworks that run on YARN.
- You can use all the features of YARN schedulers for categorizing, isolating, and prioritizing workloads.
- You choose the number of executors to use; in contrast, Spark Standalone requires each application to run an executor on every host in the cluster.
- Spark can run against Kerberos-enabled Hadoop clusters and use secure authentication between its processes.

For information on monitoring Spark applications, see Monitoring Spark Applications.

Submitting Spark Applications

To submit an application consisting of a Python file or a compiled and packaged Java or Spark JAR, use the `spark-submit` script.

spark-submit Syntax

```
spark-submit --option value \
  application jar | python file [application arguments]
```

Example: Running SparkPi on YARN on page 34 demonstrates how to run one of the sample applications, `SparkPi`, packaged with Spark. It computes an approximation to the value of pi.

Table 1: spark-submit Arguments

Option	Description
application jar	Path to a JAR file containing a Spark application and all dependencies. The path must be globally visible inside your cluster; see Advanced Dependency Management.
python file	Path to a Python file containing a Spark application. The path must be globally visible inside your cluster; see Advanced Dependency Management.
application arguments	Arguments to pass to the main method of your main class.

spark-submit Options

You specify `spark-submit` options using the form `--option value` instead of `--option=value`. (Use a space instead of an equals sign.)

Option	Description
`class`	For Java and Scala applications, the fully-qualified classname of the class containing the main method of the application. For example, `org.apache.spark.examples.SparkPi`.
`conf`	Spark configuration property in *key*=*value* format. For values that contain spaces, surround "*key*=*value*" with quotes (as shown).
`deploy-mode`	Deployment mode: ***cluster*** and ***client***. In cluster mode, the driver runs on worker hosts. In client mode, the driver runs locally as an external client. Use cluster mode with production jobs; client mode is more appropriate for interactive and debugging uses, where you want to see your application output immediately. To see the effect of the deployment mode when running on YARN, see Deployment Modes on page 31. Default: `client`.
`driver-cores`	Number of cores used by the driver, only in cluster mode. Default: 1.
`driver-memory`	Maximum heap size (represented as a JVM string; for example 1024m, 2g, and so on) to allocate to the driver. Alternatively, you can use the `spark.driver.memory` property.
`files`	Comma-separated list of files to be placed in the working directory of each executor. The path must be globally visible inside your cluster; see Advanced Dependency Management.
`jars`	Additional JARs to be loaded in the classpath of drivers and executors in cluster mode or in the executor classpath in client mode. The path must be globally visible inside your cluster; see Advanced Dependency Management.
`master`	The location to run the application.
`packages`	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. The local Maven, Maven central, and remote repositories specified in `repositories` are searched in that order. The format for the coordinates is `groupId:artifactId:version`.
`py-files`	Comma-separated list of .zip, .egg, or .py files to place on `PYTHONPATH`. The path must be globally visible inside your cluster; see Advanced Dependency Management.
`repositories`	Comma-separated list of remote repositories to search for the Maven coordinates specified in `packages`.

Table 2: Master Values

Master	Description
local	Run Spark locally with one worker thread (that is, no parallelism).
local[*K*]	Run Spark locally with *K* worker threads. (Ideally, set this to the number of cores on your host.)

Master	Description
local[*]	Run Spark locally with as many worker threads as logical cores on your host.
spark://*host*:*port*	Run using the Spark Standalone cluster manager with the Spark Master on the specified host and port (7077 by default).
yarn	Run using a YARN cluster manager. The cluster location is determined by *HADOOP_CONF_DIR* or *YARN_CONF_DIR*. See Configuring the Environment on page 33.

Cluster Execution Overview

Spark orchestrates its operations through the driver program. When the driver program is run, the Spark framework initializes executor processes on the cluster hosts that process your data. The following occurs when you submit a Spark application to a cluster:

1. The driver is launched and invokes the `main` method in the Spark application.
2. The driver requests resources from the cluster manager to launch executors.
3. The cluster manager launches executors on behalf of the driver program.
4. The driver runs the application. Based on the transformations and actions in the application, the driver sends tasks to executors.
5. Tasks are run on executors to compute and save results.
6. If dynamic allocation is enabled, after executors are idle for a specified period, they are released.
7. When driver's `main` method exits or calls `SparkContext.stop`, it terminates any outstanding executors and releases resources from the cluster manager.

Running Spark Applications on YARN

When Spark applications run on a YARN cluster manager, resource management, scheduling, and security are controlled by YARN.

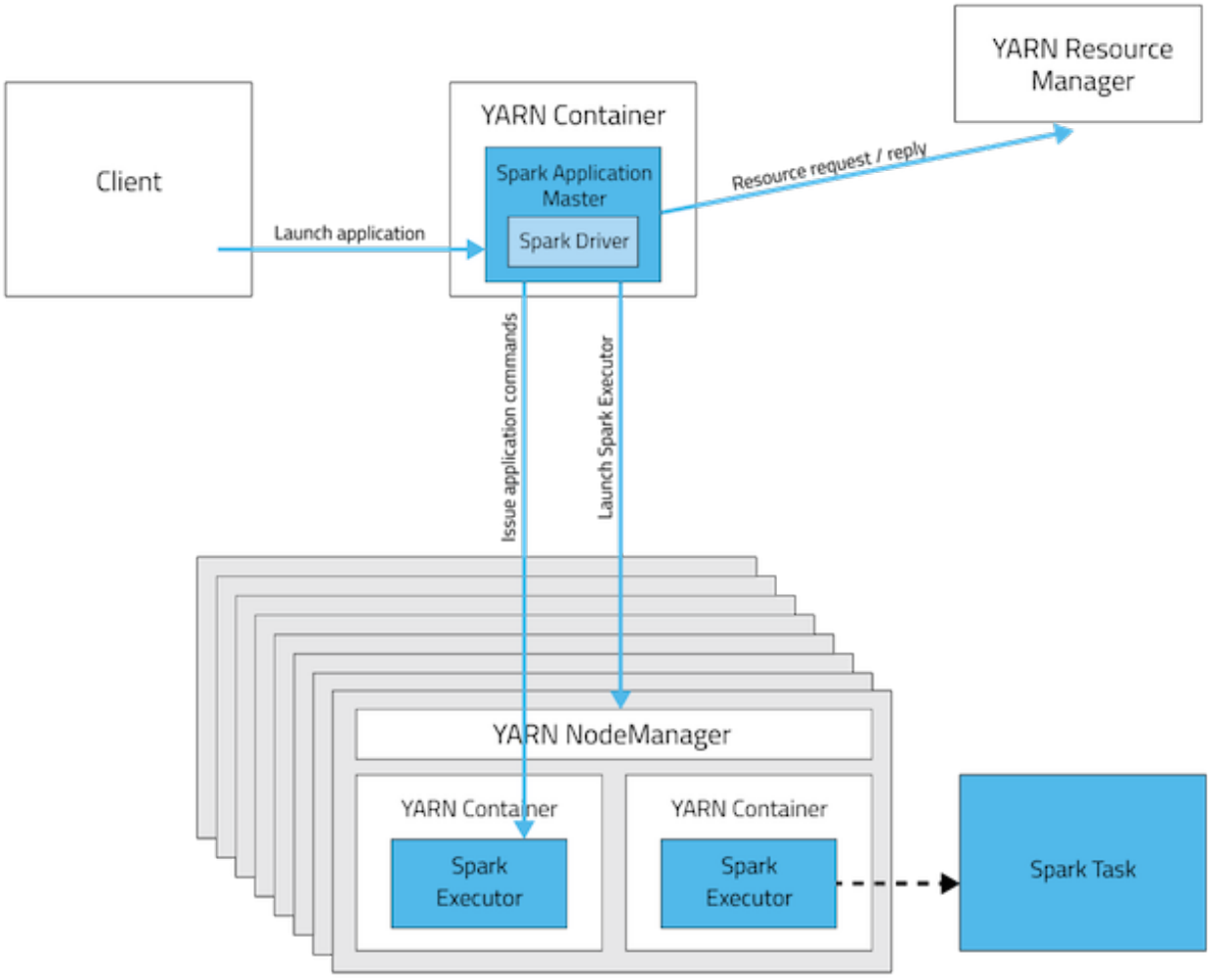
Deployment Modes

In YARN, each application instance has an ApplicationMaster process, which is the first container started for that application. The application is responsible for requesting resources from the ResourceManager. Once the resources are allocated, the application instructs NodeManagers to start containers on its behalf. ApplicationMasters eliminate the need for an active client: the process starting the application can terminate, and coordination continues from a process managed by YARN running on the cluster.

For the option to specify the deployment mode, see spark-submit Options on page 30.

Cluster Deployment Mode

In cluster mode, the Spark driver runs in the ApplicationMaster on a cluster host. A single process in a YARN container is responsible for both driving the application and requesting resources from YARN. The client that launches the application does not need to run for the lifetime of the application.



Cluster mode is not well suited to using Spark interactively. Spark applications that require user input, such as `spark-shell` and `pyspark`, require the Spark driver to run inside the client process that initiates the Spark application.

Client Deployment Mode

In client mode, the Spark driver runs on the host where the job is submitted. The ApplicationMaster is responsible only for requesting executor containers from YARN. After the containers start, the client communicates with the containers to schedule work.

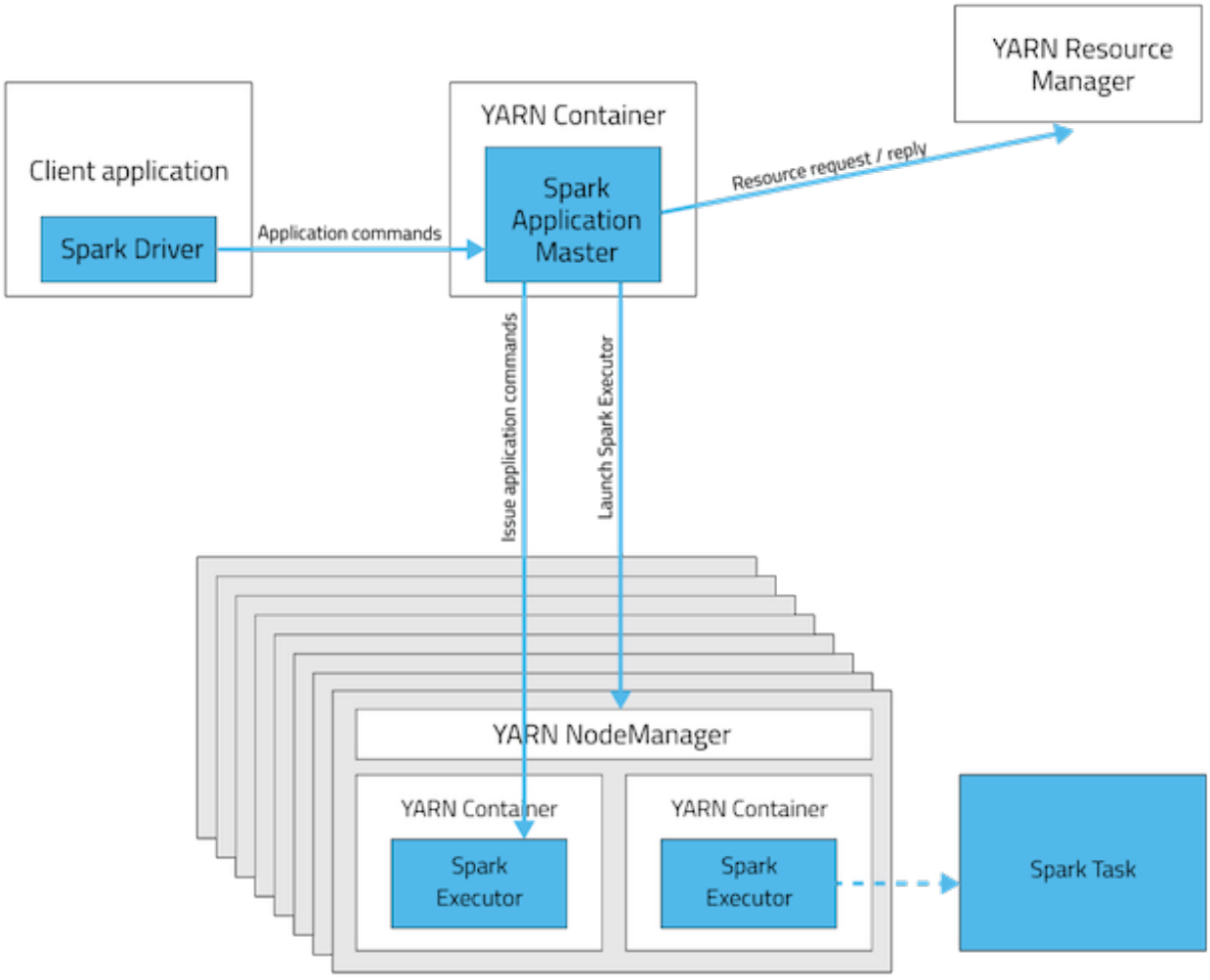


Table 3: Deployment Mode Summary

Mode	YARN Client Mode	YARN Cluster Mode
Driver runs in	Client	ApplicationMaster
Requests resources	ApplicationMaster	ApplicationMaster
Starts executor processes	YARN NodeManager	YARN NodeManager
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers
Supports Spark Shell	Yes	No

Configuring the Environment

Spark requires that the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` environment variable point to the directory containing the client-side configuration files for the cluster. These configurations are used to write to HDFS and connect to the YARN ResourceManager. If you are using a Cloudera Manager deployment, these variables are configured automatically. If you are using an unmanaged deployment, ensure that you set the variables as described in Running Spark on YARN.

Running a Spark Shell Application on YARN

To run the `spark-shell` or `pyspark` client on YARN, use the `--master yarn --deploy-mode client` flags when you start the application.

If you are using a Cloudera Manager deployment, these properties are configured automatically.

Submitting Spark Applications to YARN

To submit an application to YARN, use the `spark-submit` script and specify the `--master yarn` flag. For other `spark-submit` options, see Table 1: spark-submit Arguments on page 29.

Monitoring and Debugging Spark Applications

To obtain information about Spark application behavior you can consult YARN logs and the Spark web application UI. These two methods provide complementary information. For information how to view logs created by Spark applications and the Spark web application UI, see Monitoring Spark Applications.

Example: Running SparkPi on YARN

These examples demonstrate how to use `spark-submit` to submit the SparkPi Spark example application with various options. In the examples, the argument passed after the JAR controls how close to pi the approximation should be.

In a CDH deployment, *SPARK_HOME* defaults to `/usr/lib/spark` in package installations and `/opt/cloudera/parcels/CDH/lib/spark` in parcel installations. In a Cloudera Manager deployment, the shells are also available from `/usr/bin`.

Running SparkPi in YARN Cluster Mode

To run `SparkPi` in cluster mode:

- CDH 5.2 and lower

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode cluster SPARK_HOME/examples/lib/spark-examples.jar 10
```

- CDH 5.3 and higher

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode cluster SPARK_HOME/lib/spark-examples.jar 10
```

The command prints status until the job finishes or you press `control-C`. Terminating the `spark-submit` process in cluster mode does not terminate the Spark application as it does in client mode. To monitor the status of the running application, run `yarn application -list`.

Running SparkPi in YARN Client Mode

To run `SparkPi` in client mode:

- CDH 5.2 and lower

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode client SPARK_HOME/examples/lib/spark-examples.jar 10
```

- CDH 5.3 and higher

```
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
--deploy-mode client SPARK_HOME/lib/spark-examples.jar 10
```

Running Python SparkPi in YARN Cluster Mode

1. Unpack the Python examples archive:

```
sudo su gunzip SPARK_HOME/lib/python.tar.gz
sudo su tar xvf SPARK_HOME/lib/python.tar
```

2. Run the `pi.py` file:

```
spark-submit --master yarn --deploy-mode cluster SPARK_HOME/lib/pi.py 10
```

Configuring Spark on YARN Applications

In addition to spark-submit Options on page 30, options for running Spark applications on YARN are listed in Table 4: spark-submit on YARN Options on page 35.

Table 4: spark-submit on YARN Options

Option	Description
`archives`	Comma-separated list of archives to be extracted into the working directory of each executor. The path must be globally visible inside your cluster; see Advanced Dependency Management.
`executor-cores`	Number of processor cores to allocate on each executor. Alternatively, you can use the `spark.executor.cores` property.
`executor-memory`	Maximum heap size to allocate to each executor. Alternatively, you can use the `spark.executor.memory` property.
`num-executors`	Total number of YARN containers to allocate for this application. Alternatively, you can use the `spark.executor.instances` property.
`queue`	YARN queue to submit to. For more information, see Assigning Applications and Queries to Resource Pools. Default: `default`.

During initial installation, Cloudera Manager tunes properties according to your cluster environment.

In addition to the command-line options, the following properties are available:

Property	Description
`spark.yarn.driver.memoryOverhead`	Amount of extra off-heap memory that can be requested from YARN per driver. Combined with `spark.driver.memory`, this is the total memory that YARN can use to create a JVM for a driver process.
`spark.yarn.executor.memoryOverhead`	Amount of extra off-heap memory that can be requested from YARN, per executor process. Combined with `spark.executor.memory`, this is the total memory YARN can use to create a JVM for an executor process.

Dynamic Allocation

Dynamic allocation allows Spark to dynamically scale the cluster resources allocated to your application based on the workload. When dynamic allocation is enabled and a Spark application has a backlog of pending tasks, it can request executors. When the application becomes idle, its executors are released and can be acquired by other applications.

Starting with CDH 5.5, dynamic allocation is enabled by default. Table 5: Dynamic Allocation Properties on page 36 describes properties to control dynamic allocation.

If you set `spark.dynamicAllocation.enabled` to false or use the `--num-executors` command-line argument or set the `spark.executor.instances` property when running a Spark application, dynamic allocation is disabled. For more information on how dynamic allocation works, see resource allocation policy.

When Spark dynamic resource allocation is enabled, all resources are allocated to the first submitted job available causing subsequent applications to be queued up. To allow applications to acquire resources in parallel, allocate

resources to pools and run the applications in those pools and enable applications running in pools to be preempted. See Dynamic Resource Pools.

If you are using Spark Streaming, see the recommendation in Spark Streaming and Dynamic Allocation on page 13.

Table 5: Dynamic Allocation Properties

Property	Description
spark.dynamicAllocation.executorIdleTimeout	The length of time executor must be idle before it is removed. Default: 60 s.
spark.dynamicAllocation.enabled	Whether dynamic allocation is enabled. Default: true.
spark.dynamicAllocation.initialExecutors	The initial number of executors for a Spark application when dynamic allocation is enabled. Default: 1.
spark.dynamicAllocation.minExecutors	The lower bound for the number of executors. Default: 0.
spark.dynamicAllocation.maxExecutors	The upper bound for the number of executors. Default: Integer.MAX_VALUE.
spark.dynamicAllocation.schedulerBacklogTimeout	The length of time pending tasks must be backlogged before new executors are requested. Default: 1 s.

Optimizing YARN Mode in Unmanaged CDH Deployments

In CDH deployments not managed by Cloudera Manager, Spark copies the Spark assembly JAR file to HDFS each time you run spark-submit. You can avoid this copying by doing one of the following:

- Set spark.yarn.jar to the local path to the assembly JAR: local:/usr/lib/spark/lib/spark-assembly.jar.
- Upload the JAR and configure the JAR location:
  1. Manually upload the Spark assembly JAR file to HDFS:

```
$ hdfs dfs -mkdir -p /user/spark/share/lib
$ hdfs dfs -put SPARK_HOME/assembly/lib/spark-assembly_*.jar
/user/spark/share/lib/spark-assembly.jar
```

     You must manually upload the JAR each time you upgrade Spark to a new minor CDH release.
  2. Set spark.yarn.jar to the HDFS path:

```
spark.yarn.jar=hdfs://namenode:8020/user/spark/share/lib/spark-assembly.jar
```

Using PySpark

Apache Spark provides APIs in non-JVM languages such as Python. Many data scientists use Python because it has a rich variety of numerical libraries with a statistical, machine-learning, or optimization focus.

Running Spark Python Applications

Accessing Spark with Java and Scala offers many advantages: platform independence by running inside the JVM, self-contained packaging of code and its dependencies into JAR files, and higher performance because Spark itself runs in the JVM. You lose these advantages when using the Spark Python API.

Managing dependencies and making them available for Python jobs on a cluster can be difficult. To determine which dependencies are required on the cluster, you must understand that Spark code applications run in Spark executor processes distributed throughout the cluster. If the Python transformations you define use any third-party libraries, such as NumPy or nltk, Spark executors require access to those libraries when they run on remote executors.

Self-Contained Dependencies

In a common situation, a custom Python package contains functionality you want to apply to each element of an RDD. You can use a `map()` function call to make sure that each Spark executor imports the required package, before calling any of the functions inside that package. The following shows a simple example:

```
def import_my_special_package(x):
  import my.special.package
  return x

int_rdd = sc.parallelize([1, 2, 3, 4])
int_rdd.map(lambda x: import_my_special_package(x))
int_rdd.collect()
```

You create a simple RDD of four elements and call it `int_rdd`. Then you apply the function `import_my_special_package` to every element of the `int_rdd`. This function imports `my.special.package` and then returns the original argument passed to it. Calling this function as part of a `map()` operation ensures that each Spark executor imports `my.special.package` when needed.

If you only need a single file inside `my.special.package`, you can direct Spark to make this available to all executors by using the `--py-files` option in your `spark-submit` command and specifying the local path to the file. You can also specify this programmatically by using the `sc.addPyFiles()` function. If you use functionality from a package that spans multiple files, you can make an egg for the package, because the `--py-files` flag also accepts a path to an egg file.

If you have a *self-contained* dependency, you can make the required Python dependency available to your executors in two ways:

- If you depend on only a single file, you can use the `--py-files` command-line option, or programmatically add the file to the `SparkContext` with `sc.addPyFiles(path)` and specify the local path to that Python file.
- If you have a dependency on a self-contained module (a module with no other dependencies), you can create an egg or zip file of that module and use either the `--py-files` command-line option or programmatically add the module to the`SparkContext` with `sc.addPyFiles(path)` and specify the local path to that egg or zip file.

Complex Dependencies

Some operations rely on complex packages that also have many dependencies. For example, the following code snippet imports the Python pandas data analysis library:

```
def import_pandas(x):
 import pandas
 return x

int_rdd = sc.parallelize([1, 2, 3, 4])
int_rdd.map(lambda x: import_pandas(x))
int_rdd.collect()
```

pandas depends on NumPy, SciPy, and many other packages. Although pandas is too complex to distribute as a `*.py` file, you can create an egg for it and its dependencies and send that to executors.

Limitations of Distributing Egg Files

In both self-contained and complex dependency scenarios, sending egg files is problematic because packages that contain native code must be compiled for the specific host on which it will run. When doing distributed computing with industry-standard hardware, you must assume is that the hardware is heterogeneous. However, because of the required C compilation, a Python egg built on a client host is specific to the client CPU architecture. Therefore, distributing an egg for complex, compiled packages like NumPy, SciPy, and pandas often fails. Instead of distributing egg files you should install the required Python packages on each host of the cluster and specify the path to the Python binaries for the worker hosts to use.

Installing and Maintaining Python Environments

Installing and maintaining Python environments is complex but allows you to use the full Python package ecosystem. Ideally, a sysadmin sets up a virtual environment on every host of your cluster with your required dependencies.

If you are using Cloudera Manager, you can deploy the Anaconda distribution as a parcel as follows:

Minimum Required Role: Cluster Administrator (also provided by **Full Administrator**)

1. Add the following URL https://repo.continuum.io/pkgs/misc/parcels/ to the Remote Parcel Repository URLs as described in Parcel Configuration Settings.
2. Download, distribute, and activate the parcel as described in Managing Parcels.

Anaconda is installed in *parcel directory*/Anaconda, where *parcel directory* is /opt/cloudera/parcels by default, but can be changed in parcel configuration settings. The Anaconda parcel is supported by Continuum Analytics.

If you are not using Cloudera Manager, you can set up a virtual environment on your cluster by running commands on each host using Cluster SSH, Parallel SSH, or Fabric. Assuming each host has Python and pip installed, use the following commands to set up the standard data stack (NumPy, SciPy, scikit-learn, and pandas) in a virtual environment on a RHEL 6-compatible system:

```
# Install python-devel:
yum install python-devel

# Install non-Python dependencies required by SciPy that are not installed by default:
yum install atlas atlas-devel lapack-devel blas-devel

# install virtualenv:
pip install virtualenv

# create a new virtualenv:
virtualenv mynewenv

# activate the virtualenv:
source mynewenv/bin/activate

# install packages in mynewenv:
pip install numpy
pip install scipy
pip install scikit-learn
pip install pandas
```

Setting the Python Path

After the Python packages you want to use are in a consistent location on your cluster, set the appropriate environment variables to the path to your Python executables as follows:

- Set the Python executor path with the PYSPARK_PYTHON environment variable, and the Python driver path with the PYSPARK_DRIVER_PYTHON environment variable. These settings apply regardless of whether you are using yarn-client or yarn-cluster mode.
- If you are using yarn-cluster mode, in addition to the above, also set spark.yarn.appMasterEnv.PYSPARK_PYTHON and spark.yarn.appMasterEnv.PYSPARK_DRIVER_PYTHON in spark-defaults.conf (using the safety valve) to the same paths.

To consistently use these variables, add the appropriate export statements:

- Anaconda parcel - `export variable /opt/cloudera/parcels/Anaconda/bin/python`
- Virtual environment - `export variable /path/to/mynewenv/bin/python`

to `spark-env.sh`, checking that other users have not set the variables already with the conditional tests such as:

```
if [ -z "${PYSPARK_PYTHON}" ]; then
export PYSPARK_PYTHON=
fi
```

In Cloudera Manager, set environment variables in `spark-env.sh` as follows:

Minimum Required Role: Configurator (also provided by **Cluster Administrator, Full Administrator**)

1. Go to the Spark service.
2. Click the **Configuration** tab.
3. Search for **Spark Service Advanced Configuration Snippet (Safety Valve) for spark-conf/spark-env.sh**.
4. Add the variables to the property.
5. Click **Save Changes** to commit the changes.
6. Restart the service.
7. Deploy the client configuration.

On the command-line, set environment variables in `/etc/spark/conf/spark-env.sh`.

Running Spark Applications Using IPython and Jupyter Notebooks

IPython Notebook is a system similar to Mathematica that allows you to create "executable documents". IPython Notebooks integrate formatted text (Markdown), executable code (Python), mathematical formulas (LaTeX), and graphics and visualizations (matplotlib) into a single document that captures the flow of an exploration and can be exported as a formatted report or an executable script.

Important:

Cloudera does not support IPython or Jupyter notebooks on CDH. The instructions that were formerly here have been removed to avoid confusion about the support status of these components.

Tuning Spark Applications

This topic describes various aspects in tuning Spark applications. During tuning you should monitor application behavior to determine the effect of tuning actions.

For information on monitoring Spark applications, see Monitoring Spark Applications.

Shuffle Overview

A Spark dataset comprises a fixed number of partitions, each of which comprises a number of records. For the datasets returned by ***narrow*** transformations, such as `map` and `filter`, the records required to compute the records in a single partition reside in a *single partition* in the parent dataset. Each object is only dependent on a single object in the parent. Operations such as `coalesce` can result in a task processing multiple input partitions, but the transformation is still considered narrow because the input records used to compute any single output record can still only reside in a limited subset of the partitions.

Spark also supports transformations with ***wide*** dependencies, such as `groupByKey` and `reduceByKey`. In these dependencies, the data required to compute the records in a single partition can reside in *many partitions* of the parent dataset. To perform these transformations, all of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy this requirement, Spark performs a *shuffle*, which transfers data around the cluster and results in a new stage with a new set of partitions.

For example, consider the following code:

```
sc.textFile("someFile.txt").map(mapFunc).flatMap(flatMapFunc).filter(filterFunc).count()
```

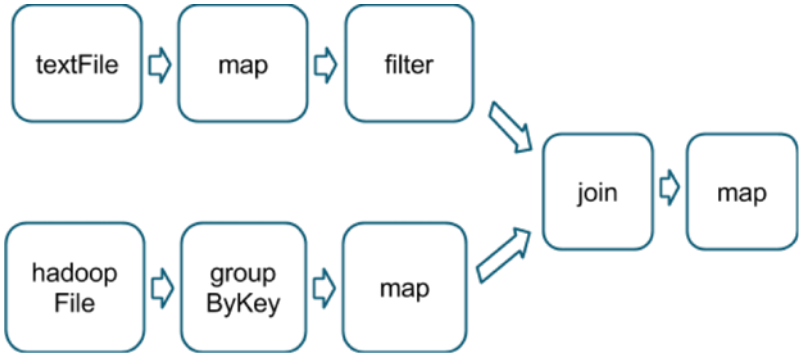
It runs a single action, `count`, which depends on a sequence of three transformations on a dataset derived from a text file. This code runs in a single stage, because none of the outputs of these three transformations depend on data that comes from different partitions than their inputs.

In contrast, this Scala code finds how many times each character appears in all the words that appear more than 1,000 times in a text file:

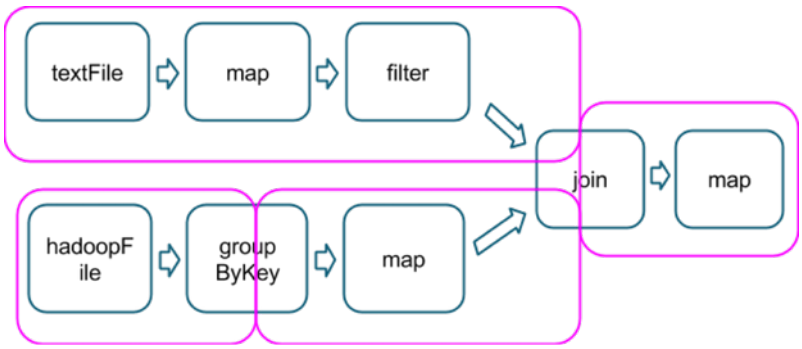
```
val tokenized = sc.textFile(args(0)).flatMap(_.split(' '))
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
val filtered = wordCounts.filter(_._2 >= 1000)
val charCounts = filtered.flatMap(_._1.toCharArray).map((_, 1)).reduceByKey(_ + _)
charCounts.collect()
```

This example has three stages. The two `reduceByKey` transformations each trigger stage boundaries, because computing their outputs requires repartitioning the data by keys.

A final example is this more complicated transformation graph, which includes a `join` transformation with multiple dependencies:



The pink boxes show the resulting stage graph used to run it:



At each stage boundary, data is written to disk by tasks in the parent stages and then fetched over the network by tasks in the child stage. Because they incur high disk and network I/O, stage boundaries can be expensive and should be avoided when possible. The number of data partitions in a parent stage may be different than the number of partitions in a child stage. Transformations that can trigger a stage boundary typically accept a `numPartitions` argument, which specifies into how many partitions to split the data in the child stage. Just as the number of reducers is an important parameter in MapReduce jobs, the number of partitions at stage boundaries can determine an application's performance. Tuning the Number of Partitions on page 44 describes how to tune this number.

Choosing Transformations to Minimize Shuffles

You can usually choose from many arrangements of actions and transformations that produce the same results. However, not all these arrangements result in the same performance. Avoiding common pitfalls and picking the right arrangement can significantly improve an application's performance.

When choosing an arrangement of transformations, minimize the number of shuffles and the amount of data shuffled. Shuffles are expensive operations; all shuffle data must be written to disk and then transferred over the network. `repartition` , `join`, `cogroup`, and any of the `*By` or `*ByKey` transformations can result in shuffles. Not all these transformations are equal, however, and you should avoid the following patterns:

- `groupByKey` when performing an associative reductive operation. For example, `rdd.groupByKey().mapValues(_.sum)` produces the same result as `rdd.reduceByKey(_ + _)`. However, the former transfers the entire dataset across the network, while the latter computes local sums for each key in each partition and combines those local sums into larger sums after shuffling.
- `reduceByKey` when the input and output value types are *different*. For example, consider writing a transformation that finds all the unique strings corresponding to each key. You could use `map` to transform each element into a `Set` and then combine the `Sets` with `reduceByKey`:

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2)).reduceByKey(_ ++ _)
```

  This results in unnecessary object creation because a new set must be allocated for each record.

  Instead, use `aggregateByKey`, which performs the map-side aggregation more efficiently:

```
val zero = new collection.mutable.Set[String]()
rdd.aggregateByKey(zero)((set, v) => set += v,(set1, set2) => set1 ++= set2)
```

- `flatMap-join-groupBy`. When two datasets are already grouped by key and you want to join them and keep them grouped, use `cogroup`. This avoids the overhead associated with unpacking and repacking the groups.

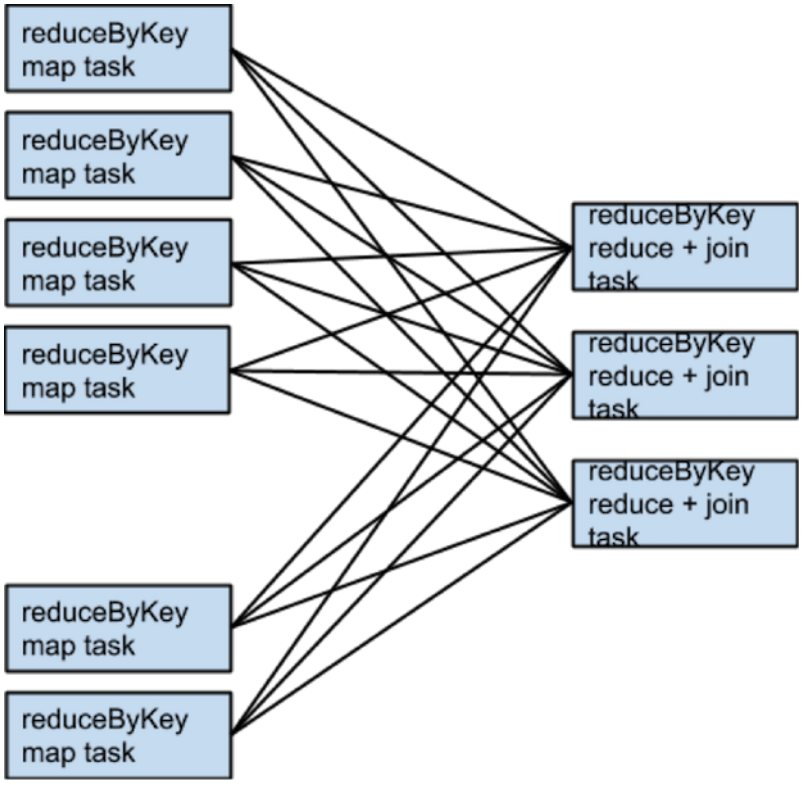
When Shuffles Do Not Occur

In some circumstances, the transformations described previously *do not* result in shuffles. Spark does not shuffle when a previous transformation has already partitioned the data according to the *same partitioner*. Consider the following flow:

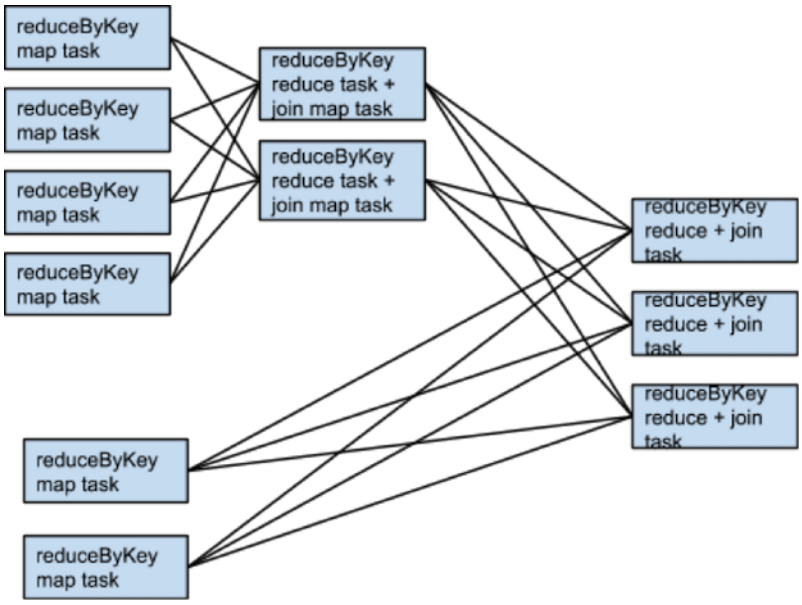
```
rdd1 = someRdd.reduceByKey(...)
rdd2 = someOtherRdd.reduceByKey(...)
rdd3 = rdd1.join(rdd2)
```

Because no partitioner is passed to `reduceByKey`, the default partitioner is used, resulting in `rdd1` and `rdd2` both being hash-partitioned. These two `reduceByKey` transformations result in two shuffles. If the datasets have the same number of partitions, a join requires no additional shuffling. Because the datasets are partitioned identically, the set of keys in any single partition of `rdd1` can only occur in a single partition of `rdd2`. Therefore, the contents of any single output partition of `rdd3` depends only on the contents of a single partition in `rdd1` and single partition in `rdd2`, and a third shuffle is not required.

For example, if `someRdd` has four partitions, `someOtherRdd` has two partitions, and both the `reduceByKeys` use three partitions, the set of tasks that run would look like this:



If `rdd1` and `rdd2` use different partitioners or use the default (hash) partitioner with different numbers of partitions, only one of the datasets (the one with the fewer number of partitions) needs to be reshuffled for the join:



To avoid shuffles when joining two datasets, you can use broadcast variables. When one of the datasets is small enough to fit in memory in a single executor, it can be loaded into a hash table on the driver and then broadcast to every executor. A map transformation can then reference the hash table to do lookups.

When to Add a Shuffle Transformation

The rule of minimizing the number of shuffles has some exceptions.

An extra shuffle can be advantageous when it increases parallelism. For example, if your data arrives in a few large unsplittable files, the partitioning dictated by the `InputFormat` might place large numbers of records in each partition,

while not generating enough partitions to use all available cores. In this case, invoking repartition with a high number of partitions (which triggers a shuffle) after loading the data allows the transformations that follow to use more of the cluster's CPU.

Another example arises when using the `reduce` or `aggregate` action to aggregate data into the driver. When aggregating over a high number of partitions, the computation can quickly become bottlenecked on a single thread in the driver merging all the results together. To lighten the load on the driver, first use `reduceByKey` or `aggregateByKey` to perform a round of distributed aggregation that divides the dataset into a smaller number of partitions. The values in each partition are merged with each other in parallel, before being sent to the driver for a final round of aggregation. See `treeReduce` and `treeAggregate` for examples of how to do that.

This method is especially useful when the aggregation is already grouped by a key. For example, consider an application that counts the occurrences of each word in a corpus and pulls the results into the driver as a map. One approach, which can be accomplished with the `aggregate` action, is to compute a local map at each partition and then merge the maps at the driver. The alternative approach, which can be accomplished with `aggregateByKey`, is to perform the count in a fully distributed way, and then simply `collectAsMap` the results to the driver.

Secondary Sort

The `repartitionAndSortWithinPartitions` transformation repartitions the dataset according to a partitioner and, within each resulting partition, sorts records by their keys. This transformation pushes sorting down into the shuffle machinery, where large amounts of data can be spilled efficiently and sorting can be combined with other operations.

For example, Apache Hive on Spark uses this transformation inside its `join` implementation. It also acts as a vital building block in the *secondary sort* pattern, in which you group records by key and then, when iterating over the values that correspond to a key, have them appear in a particular order. This scenario occurs in algorithms that need to group events by user and then analyze the events for each user, based on the time they occurred.

Tuning Resource Allocation

For background information on how Spark applications use the YARN cluster manager, see Running Spark Applications on YARN on page 31.

The two main resources that Spark and YARN manage are CPU and memory. Disk and network I/O affect Spark performance as well, but neither Spark nor YARN actively manage them.

Every Spark executor in an application has the same fixed number of cores and same fixed heap size. Specify the number of cores with the `--executor-cores` command-line flag, or by setting the `spark.executor.cores` property. Similarly, control the heap size with the `--executor-memory` flag or the `spark.executor.memory` property. The `cores` property controls the number of concurrent tasks an executor can run. For example, set `--executor-cores 5` for each executor to run a maximum of five tasks at the same time. The memory property controls the amount of data Spark can cache, as well as the maximum sizes of the shuffle data structures used for grouping, aggregations, and joins.

Starting with CDH 5.5 dynamic allocation, which adds and removes executors dynamically, is enabled. To explicitly control the number of executors, you can override dynamic allocation by setting the `--num-executors` command-line flag or `spark.executor.instances` configuration property.

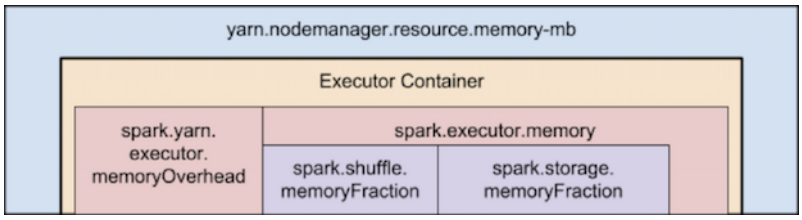
Consider also how the resources requested by Spark fit into resources YARN has available. The relevant YARN properties are:

- `yarn.nodemanager.resource.memory-mb` controls the maximum sum of memory used by the containers on each host.
- `yarn.nodemanager.resource.cpu-vcores` controls the maximum sum of cores used by the containers on each host.

Requesting five executor cores results in a request to YARN for five cores. The memory requested from YARN is more complex for two reasons:

- The `--executor-memory/spark.executor.memory` property controls the executor heap size, but JVMs can also use some memory off heap, for example for interned Strings and direct byte buffers. The value of the `spark.yarn.executor.memoryOverhead` property is added to the executor memory to determine the full memory request to YARN for each executor. It defaults to `max`(384, .07 * `spark.executor.memory`).
- YARN may round the requested memory up slightly. The `yarn.scheduler.minimum-allocation-mb` and `yarn.scheduler.increment-allocation-mb` properties control the minimum and increment request values, respectively.

The following diagram (not to scale with defaults) shows the hierarchy of memory properties in Spark and YARN:



Keep the following in mind when sizing Spark executors:

- The ApplicationMaster, which is a non-executor container that can request containers from YARN, requires memory and CPU that must be accounted for. In ***client*** deployment mode, they default to 1024 MB and one core. In ***cluster*** deployment mode, the ApplicationMaster runs the driver, so consider bolstering its resources with the `--driver-memory` and `--driver-cores` flags.
- Running executors with too much memory often results in excessive garbage-collection delays. For a single executor, use 64 GB as an upper limit.
- The HDFS client has difficulty processing many concurrent threads. At most, five tasks per executor can achieve full write throughput, so keep the number of cores per executor below that number.
- Running tiny executors (with a single core and just enough memory needed to run a single task, for example) offsets the benefits of running multiple tasks in a single JVM. For example, broadcast variables must be replicated once on each executor, so many small executors results in many more copies of the data.

Resource Tuning Example

Consider a cluster with six hosts running NodeManagers, each equipped with 16 cores and 64 GB of memory.

The NodeManager capacities, `yarn.nodemanager.resource.memory-mb` and `yarn.nodemanager.resource.cpu-vcores`, should be set to 63 * 1024 = 64512 (megabytes) and 15, respectively. Avoid allocating 100% of the resources to YARN containers because the host needs some resources to run the OS and Hadoop daemons. In this case, leave one GB and one core for these system processes. Cloudera Manager accounts for these and configures these YARN properties automatically.

You might consider using `--num-executors 6 --executor-cores 15 --executor-memory 63G`. However, this approach does not work:

- 63 GB plus the executor memory overhead does not fit within the 63 GB capacity of the NodeManagers.
- The ApplicationMaster uses a core on one of the hosts, so there is no room for a 15-core executor on that host.
- 15 cores per executor can lead to bad HDFS I/O throughput.

Instead, use `--num-executors 17 --executor-cores 5 --executor-memory 19G`:

- This results in three executors on all hosts except for the one with the ApplicationMaster, which has two executors.
- `--executor-memory` is computed as (63/3 executors per host) = 21. 21 * 0.07 = 1.47. 21 - 1.47 ~ 19.

Tuning the Number of Partitions

Spark has limited capacity to determine optimal parallelism. Every Spark stage has a number of tasks, each of which processes data sequentially. The number of tasks per stage is the most important parameter in determining performance.

As described in Spark Execution Model on page 8, Spark groups datasets into stages. The number of tasks in a stage is the same as the number of partitions in the last dataset in the stage. The number of partitions in a dataset is the same as the number of partitions in the datasets on which it depends, with the following exceptions:

- The `coalesce` transformation creates a dataset with *fewer* partitions than its parent dataset.
- The `union` transformation creates a dataset with the *sum* of its parents' number of partitions.
- The `cartesian` transformation creates a dataset with the *product* of its parents' number of partitions.

Datasets with no parents, such as those produced by `textFile` or `hadoopFile`, have their partitions determined by the underlying MapReduce `InputFormat` used. Typically, there is a partition for each HDFS block being read. The number of partitions for datasets produced by `parallelize` are specified in the method, or `spark.default.parallelism` if not specified. To determine the number of partitions in an dataset, call `rdd.partitions().size()`.

If the number of tasks is smaller than number of slots available to run them, CPU usage is suboptimal. In addition, more memory is used by any aggregation operations that occur in each task. In `join`, `cogroup`, or `*ByKey` operations, objects are held in in hashmaps or in-memory buffers to group or sort. `join`, `cogroup`, and `groupByKey` use these data structures in the tasks for the stages that are on the fetching side of the shuffles they trigger. `reduceByKey` and `aggregateByKey` use data structures in the tasks for the stages on both sides of the shuffles they trigger. If the records in these aggregation operations exceed memory, the following issues can occur:

- Holding a high number records in these data structures increases garbage collection, which can lead to pauses in computation.
- Spark spills them to disk, causing disk I/O and sorting that leads to job stalls.

To increase the number of partitions if the stage is reading from Hadoop:

- Use the `repartition` transformation, which triggers a shuffle.
- Configure your `InputFormat` to create more splits.
- Write the input data to HDFS with a smaller block size.

If the stage is receiving input from another stage, the transformation that triggered the stage boundary accepts a `numPartitions` argument:

```
val rdd2 = rdd1.reduceByKey(_ + _, numPartitions = X)
```

Determining the optimal value for *X* requires experimentation. Find the number of partitions in the parent dataset, and then multiply that by 1.5 until performance stops improving.

You can also calculate *X* in a more formulaic way, but some quantities in the formula are difficult to calculate. The main goal is to run enough tasks so that the data destined for each task fits in the memory available to that task. The memory available to each task is:

```
(spark.executor.memory * spark.shuffle.memoryFraction * spark.shuffle.safetyFraction)/
spark.executor.cores
```

`memoryFraction` and `safetyFraction` default to 0.2 and 0.8 respectively.

The in-memory size of the total shuffle data is more difficult to determine. The closest heuristic is to find the ratio between shuffle spill memory and the shuffle spill disk for a stage that ran. Then, multiply the total shuffle write by this number. However, this can be compounded if the stage is performing a reduction:

```
(observed shuffle write) * (observed shuffle spill memory) * (spark.executor.cores)/
(observed shuffle spill disk) * (spark.executor.memory) * (spark.shuffle.memoryFraction)
 * (spark.shuffle.safetyFraction)
```

Then, round up slightly, because too many partitions is usually better than too few.

When in doubt, err on the side of a larger number of tasks (and thus partitions). This contrasts with recommendations for MapReduce, which unlike Spark, has a high startup overhead for tasks.

Reducing the Size of Data Structures

Data flows through Spark in the form of records. A record has two representations: a deserialized Java object representation and a serialized binary representation. In general, Spark uses the deserialized representation for records in memory and the serialized representation for records stored on disk or transferred over the network. For sort-based shuffles, in-memory shuffle data is stored in serialized form.

The `spark.serializer` property controls the serializer used to convert between these two representations. Cloudera recommends using the Kryo serializer, `org.apache.spark.serializer.KryoSerializer`.

The footprint of your records in these two representations has a significant impact on Spark performance. Review the data types that are passed and look for places to reduce their size. Large deserialized objects result in Spark spilling data to disk more often and reduces the number of deserialized records Spark can cache (for example, at the `MEMORY` storage level). The Apache Spark tuning guide describes how to reduce the size of such objects. Large serialized objects result in greater disk and network I/O, as well as reduce the number of serialized records Spark can cache (for example, at the `MEMORY_SER` storage level.) Make sure to register any custom classes you use with the `SparkConf#registerKryoClasses` API.

Choosing Data Formats

When storing data on disk, use an extensible binary format like Avro, Parquet, Thrift, or Protobuf and store in a sequence file.

Spark and Hadoop Integration

This section describes how to write to various Hadoop ecosystem components from Spark.

Writing to HBase from Spark

You can use Spark to process data that is destined for HBase. See Importing Data Into HBase Using Spark.

You can also use Spark in conjunction with Apache Kafka to stream data from Spark to HBase. See Importing Data Into HBase Using Spark and Kafka.

Because Spark does not have a dependency on HBase, in order to access HBase from Spark, you must do the following:

- Manually provide the location of HBase configurations and classes to the driver and executors. You do so by passing the locations to both classpaths when you run `spark-submit`, `spark-shell`, or `pyspark`:
  - **parcel installation**

```
--driver-class-path /etc/hbase/conf:/opt/cloudera/parcels/CDH/lib/hbase/lib/*
--conf
"spark.executor.extraClassPath=/etc/hbase/conf/:/opt/cloudera/parcels/CDH/lib/hbase/lib/*"
```

  - **package installation**

```
--driver-class-path /etc/hbase/conf:/usr/lib/hbase/lib/hbase/lib/*
--conf "spark.executor.extraClassPath=/etc/hbase/conf/:/usr/lib/hbase/lib/*"
```

- Add an HBase gateway role to all YARN worker hosts and the edge host where you run `spark-submit`, `spark-shell`, or `pyspark` and deploy HBase client configurations.

Limitations in Kerberized Environments

The following limitations apply to Spark applications that access HBase in a Kerberized cluster:

- The application must be restarted every seven days.
- If the cluster also has HA enabled, you must specify the `keytab` and `principal` parameters in your command line (as opposed to using `kinit`). For example:

```
spark-shell --jars MySparkHbaseApp.jar --principal ME@DOMAIN.COM --keytab
/path/to/local/keytab ...
```

```
spark-submit --class com.example.SparkHbaseApp --principal ME@DOMAIN.COM --keytab
/path/to/local/keytab
SparkHBaseApp.jar [ application parameters....]"
```

For further information, see Spark Authentication.

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 the required privileges on the underlying data files, a SparkSQL query against the view returns an empty result set, rather than an error.

Running Spark Jobs from Oozie

For CDH 5.4 and higher you can invoke Spark jobs from Oozie using the Spark action. For information on the Spark action, see Oozie Spark Action Extension.

In CDH 5.4, to enable dynamic allocation when running the action, specify the following in the Oozie workflow:

```
<spark-opts>--conf spark.dynamicAllocation.enabled=true
--conf spark.shuffle.service.enabled=true
--conf spark.dynamicAllocation.minExecutors=1
</spark-opts>
```

If you have enabled the shuffle service in Cloudera Manager, you do not need to specify `spark.shuffle.service.enabled`.

Building and Running a Crunch Application with Spark

Developing and Running a Spark WordCount Application on page 9 provides a tutorial on writing, compiling, and running a Spark application. Using the tutorial as a starting point, do the following to build and run a Crunch application with Spark:

1. Along with the other dependencies shown in the tutorial, add the appropriate version of the `crunch-core` and `crunch-spark` dependencies to the Maven project.

```
<dependency>
  <groupId>org.apache.crunch</groupId>
  <artifactId>crunch-core</artifactId>
  <version>${crunch.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.crunch</groupId>
  <artifactId>crunch-spark</artifactId>
  <version>${crunch.version}</version>
  <scope>provided</scope>
</dependency>
```

2. Use `SparkPipeline` where you would have used `MRPipeline` in the declaration of your Crunch pipeline. `SparkPipeline` takes either a String that contains the connection string for the Spark master (`local` for local mode, `yarn` for YARN) or a `JavaSparkContext` instance.
3. As you would for a Spark application, use `spark-submit` start the pipeline with your Crunch application *app-jar-with-dependencies*.jar file.

For an example, see Crunch demo. After building the example, run with the following command:

```
spark-submit --class com.example.WordCount
crunch-demo-1.0-SNAPSHOT-jar-with-dependencies.jar \
hdfs://namenode_host:8020/user/hdfs/input hdfs://namenode_host:8020/user/hdfs/output
```