如何在CDH 5上运行Spark应用程序

本文转载自：http://blog.cloudera.com/blog/2014/04/how-to-run-a-simple-apache-spark-app-in-cdh-5/

(Editor’s note – this post has been updated to reflect CDH 5.1/Spark 1.0)

Apache Spark is a general-purpose, cluster computing framework that, like MapReduce in Apache Hadoop, offers powerful abstractions for processing large datasets. For various reasons pertaining to performance, functionality, and APIs, Spark is already becoming more popular than MapReduce for certain types of workloads. (For more background about Spark, read this post.)

In this how-to, you’ll learn how to write, compile, and run a simple Spark program written in Scala on CDH 5 (in which Spark ships and is supported by Cloudera). The full code for the example is hosted at https://github.com/sryza/simplesparkapp.

Writing

Our example app will be a souped-up version of WordCount, the classic MapReduce example. In WordCount, the goal is to learn the distribution of letters in the most popular words in our corpus. That is, we want to:

1. Read an input set of text documents
2. Count the number of times each word appears
3. Filter out all words that show up less than a million times
4. For the remaining set, count the number of times each letter occurs

In MapReduce, this would require two MapReduce jobs, as well as persisting the intermediate data to HDFS in between them. In constrast, in Spark, you can write a single job in about 90 percent fewer lines of code.

Our input is a huge text file where each line contains all the words in a document, stripped of punctuation. The full Scala program looks like this:

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

package com.cloudera.sparkwordcount

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

object SparkWordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext(new SparkConf().setAppName("Spark Count"))
    val threshold = args(1).toInt

    // 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 less 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(", "))
  }
}
```

Spark uses “lazy evaluation”, meaning that transformations don’t execute on the cluster until an “action” operation is invoked. Examples of action operations are collect, which pulls data to the client, and saveAsTextFile, which writes data to a filesystem like HDFS.

It’s worth noting that in Spark, the definition of “reduce” is slightly different than that in

MapReduce. In MapReduce, a reduce function call accepts all the records corresponding to a given key. In Spark, the function passed to reduce, or reduceByKey function call, accepts just two arguments – so if it’s not associative, bad things will happen. A positive consequence is that Spark knows it can always apply a combiner. Based on that definition, the Spark equivalent of MapReduce’s reduce is similar to a groupBy followed by a map.

For those more comfortable with Java, here’s the same program using Spark’s Java API:

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

package com.cloudera.sparkwordcount;

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) {
    JavaSparkContext sc =
        new JavaSparkContext(new SparkConf().setAppName("Spark Count"));
    final int threshold = Integer.parseInt(args[1]);

    // split each document into words
    JavaRDD<String> tokenized = sc.textFile(args[0]).flatMap(
      new FlatMapFunction<String, String>() {
```

```
    @Override
    public Iterable<String> call(String s) {
      return Arrays.asList(s.split(" "));
    }
  }
);

// count the occurrence of each word
JavaPairRDD<String, Integer> counts = tokenized.mapToPair(
  new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String s) {
      return new Tuple2<String, Integer>(s, 1);
    }
  }
).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
    }
  }
);

// filter out words with less than threshold occurrences
JavaPairRDD<String, Integer> filtered = counts.filter(
  new Function<Tuple2<String, Integer>, Boolean>() {
    @Override
    public Boolean call(Tuple2<String, Integer> 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());
  }
}
```

Because Java doesn't support anonymous functions, the program is considerably more verbose, but it still requires a fraction of the code needed in an equivalent MapReduce program.

Compiling

We'll use Maven to compile our program. Maven expects a specific directory layout that informs it where to look for source files. Our Scala code goes under src/main/scala, and our Java code goes under src/main/java. That is, we place SparkWordCount.scala in the src/main/scala/com/cloudera/sparkwordcount directory and JavaWordCount.java in the src/main/java/com/cloudera/sparkwordcount directory.

Maven also requires you to place a pom.xml file in the root of the project directory that tells it how to build the project. A few noteworthy excerpts are included below.

To compile Scala code, include:

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

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

which requires adding the scala-tools plugin repository:

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

Then, include Spark and Scala as dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.10.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.0.0-cdh5.1.0</version>
  </dependency>
</dependencies>
```

Finally, to generate our app jar, simply run:

```
mvn package
```

It will show up in the target directory as sparkwordcount-0.0.1-SNAPSHOT.jar.

Running

(Note: the following instructions only work with CDH 5.1/Spark 1.0 and later. To run against CDH 5.0/Spark 0.9, see the instructions here.)

Before running, place the input file into a directory on HDFS. The repository supplies an example input file in its data directory. To run the Spark program, we use the spark-submit script:

```
spark-submit --class com.cloudera.sparkwordcount.SparkWordCount --master local
target/sparkwordcount-0.0.1-SNAPSHOT.jar <input file>  2
```

This will run the application in a single local process. If the cluster is running a Spark standalone cluster manager, you can replace --master local with --master spark://:.

If the cluster is running YARN, you can replace --master local with --master yarn. Spark will determine the YARN ResourceManager's address from the YARN configuration file.

The output of the program should look something like this:

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

Congratulations, you have just run a simple Spark application in CDH 5. Happy Sparking!

Sandy Ryza is data scientist at Cloudera. He is an Apache Hadoop committer and recently led Cloudera's Spark development.