MapReduce Programming Examples

Wordcount

In this example, we will create a random text input file using the **`RandomTextWriter`** application in Hadoop and compile and run a **`WordCount`** program to process these files and report the number of times each word appears in the input file.

For help on the various **`hadoop`** command line syntaxes, we shall refer you to the MapReduce Tutorial and HDFS Command Guide.

The java code for Wordcount is presented below:

```java
package edu.cmu;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
     word.set(tokenizer.nextToken());

     output.collect(word, one);
    }
   }
  }

  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
     sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
   }
  }

  public static void main(String[] args) throws Exception {
   JobConf conf = new JobConf(WordCount.class);
   conf.setJobName("wordcount");

   conf.setOutputKeyClass(Text.class);
   conf.setOutputValueClass(IntWritable.class);

45         conf.setMapperClass(Map.class);
46         conf.setCombinerClass(Reduce.class);
47         conf.setReducerClass(Reduce.class);
48 
49         conf.setInputFormat(TextInputFormat.class);
50         conf.setOutputFormat(TextOutputFormat.class);
51 
52         FileInputFormat.setInputPaths(conf, new Path(args[0]));
53         FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54 
55         JobClient.runJob(conf);
56     }
57 }
```

Code Walkthrough: The code consists of the `Wordcount` class, which also has two subclasses; `Map` and `Reduce`. The `main` method of the class (lines 38 through 58) sets up the Map Reduce job by defining the input and output types, map and reduce classes to be used, as well as the input and output paths (in this example, they are set to be the first and second command line parameters respectively.

The input format used for this example is the `TextInputFormat`. HDFS automatically breaks files into chunks (default 64 MB) and stores it on the file system. When a MapReduce job is launched, each file block gets a Map task to process it. The `TextInputFormat` class automatically splits each block into records for each line. The map task iterates over each record and performs a map operation over each record. A map operation receives an entire line from the text file as input; The byte offset of that line is the key and the value is the text content of that line. In this example, the Map function simply tokenizes the line, and emits each token as a key, with the value 1. These are forwarded to the reduce function.

The key-value pairs that are emitted from the Map operation are sorted and sent to the reducers in what is known as the shuffle operation. The key space is partitioned across the number of reducers, but a reducer is guaranteed to receive all the values for a particular key that is assigned to it. Since the extracted token on each line acts as the key, each reducer is guaranteed to receive all the values (1s) for a token.

Follow the steps below to compile and run the program:

1. Login to the master instance of your hadoop cluster to author the `WordCount.java` file.
2. Compile your code and package it as a jar using the following commands:

   ```
   mkdir wordcount_classes
   javac -classpath ${HADOOP_HOME}/hadoop-core.jar -d wordcount_classes WordCount.java
   jar -cvf wordcount.jar -C wordcount_classes/ .
   ```

   Please note the trailing `.` at the end of the `jar` command.
3. Run `RandomTextWriter` with parameter `/wordocunt/input` to generate the input data. Use the default configuration to generate about 40 GB of data, (10 GB of data per node).
4. Once the program has finished, use the HDFS commands to check the contents of `/wordcount/input`
5. Run your MapReduce jar file (`wordcount.jar`) over `/wordcount/input`. Use `/wordcount/output` as the destination for your output files.

Hadoop Sort

In this example, we will run a simple Sort application that is located in the Hadoop examples jar. For this purpose, we will use the `RandomWriter` example program to randomly generate binary data into input files which will be sorted by the sort application.

The sort application itself is quite rudimentary, it uses the MapReduce framework to sort data, the Map operation simply transmits the data as key,value pairs and these pairs are globally sorted when they arrive at the reduce phase through the shuffle operation. This is an example of a binary application in Hadoop.

1. Run `RandomWriter` using the parameter `/sort/input` to generate the input data. Use the default configuration to generate 10 GB of data per node.
2. Run the Hadoop Sort example over `/sort/input` and store the results in `/sort/output`.

Click through to the next page when you are ready.