16-10-2017

SIMEON BINIATIDIS

Hesperange, Luxembourg, 5816

The Hadoop **WordCount MapReduce** workflow works as follows:

1. Hadoop reads Input, breaks it using new line characters as separator, then runs the map function passing each line as an argument with the line number as **Key** and line contents as **Value**.
2. The **map** function tokenizes the line, and for each token (word), emits a Key-Value pair (word,1).
3. Hadoop collects all the (word,1) pairs, sorts them by the word, groups all Values emitted against each unique Key, and invokes the **reduce** function once for each unique Key, passing Key and Values for that key as an argument.
4. The reduce function counts the number of occurrences of each word using the values and emits it as a key-value pair.
5. Hadoop writes the final output to the output directory.

In Hadoop, MapReduce is a computation decomposing large manipulation jobs into individual tasks that can be executed in parallel cross a cluster of servers. The results of tasks can be joined together to compute final results.

MapReduce consists of 2 big steps:

**Map Function –** It takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (Key-Value pair).

e.g. (Synthesis, 1), (SYNTHESIS, 1), (ANALYSIS, 1), (Analysis, 1), (Results, 1), (RESULTS, 1), (REsults, 1),

(SyNthesis, 1)

**Reduce Function –** Takes the output from Map as an input and combines those data tuples into a smaller set of tuples.

e.g. (SYNTHESIS, 3), (ANALYSIS, 2), (RESULTS, 3).

To further analyze a typical program workflow of an archetypal MapReduce job we would identify and qualify roughly 5 discrete steps.

1. **Splitting** – Splitting parameter can be anything, e.g. splitting by space, comma, semicolon, or even by a new line (‘\n’).
2. **Mapping** – as explained above
3. **Intermediate splitting** – the entire process in parallel on different clusters. In order to group them in “Reduce Phase” the similar KEY data should be on same cluster.
4. **Reduce** – It is mostly a “GroupBy” phase.
5. **Combining** – The last phase where all the data (individual result set from each cluster) is combined together to form a Result

A characteristic “printout” of an implementation in Java follows

{Hadoop itself is implemented in Java}

**import** org.apache.hadoop.conf.Configuration;  
**import** org.apache.hadoop.fs.Path;  
**import** org.apache.hadoop.io.IntWritable;  
**import** org.apache.hadoop.io.Text;  
**import** org.apache.hadoop.mapreduce.Job;  
**import** org.apache.hadoop.mapreduce.Mapper;  
**import** org.apache.hadoop.mapreduce.Reducer;  
**import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
**import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
**import** org.apache.hadoop.util.GenericOptionsParser;  
  
**import** java.io.IOException;  
**import** java.util.StringTokenizer;  
  
  
*/\*\*  
 \* <p>The WordCount program counts the number of word occurrences within a set of input documents  
 \* using MapReduce. The code has three parts: mapper, reducer, and the main program.</p>  
 \*/***public class** WordCount {  
 */\*\*  
 \* <p>  
 \* The mapper extends from the org.apache.hadoop.mapreduce.Mapper interface. When Hadoop runs,  
 \* it receives each new line in the input files as an input to the mapper. The �map� function  
 \* tokenizes the line, and for each token (word) emits (word,1) as the output. </p>  
 \*/* **public static class** TokenizerMapper  
 **extends** Mapper<Object, Text, Text, IntWritable>{  
  
 **private final static** IntWritable ***one*** = **new** IntWritable(1);  
 **private** Text **word** = **new** Text();  
  
 **public void** map(Object key, Text value, Context context) **throws** IOException, InterruptedException {  
 StringTokenizer itr = **new** StringTokenizer(value.toString());  
  
 **while** (itr.hasMoreTokens()) {  
 **word**.set(itr.nextToken());  
 context.write(**word**, ***one***);  
  
 }  
 }  
 }  
  
 */\*\*  
 \* <p>Reduce function receives all the values that has the same key as the input, and it outputs the key  
 \* and the number of occurrences of the key as the output.</p>  
 \*/* **public static class** IntSumReducer **extends** Reducer<Text,IntWritable,Text,IntWritable> {  
 **private** IntWritable **result** = **new** IntWritable();  
  
 **public void** reduce(Text key, Iterable<IntWritable> values,Context context) **throws** IOException, InterruptedException {  
 **int** sum = 0;  
  
 **for** (IntWritable val : values) {  
 sum += val.get();  
 }  
 **result**.set(sum);  
 context.write(key, **result**);  
 }  
 }  
  
 */\*\*  
 \* <p> As input this program takes any text file. Create a folder called input in HDFS (or in local directory if you are running this locally)  
 \* <li>Option1: You can compile the sample by ant from sample directory. To do this, you need to have Apache Ant installed in your system.  
 \* Otherwise, you can use the compiled jar included with the source code.  
 \* The jar file exists in ../out/artifacts/WordCount\_jar/  
 \* Then run the command > java jar WordCount.jar input output </li>  
 \* <li>Optionally you can run the WordCount class directly from your IDE passing input output as arguments. \* This will run the sample same as before.  
 \* Running MapReduce Jobs from IDE in this manner is very useful for debugging your MapReduce Jobs. </li>  
 \* </ol>  
 \** ***@param args*** *\** ***@throws*** *Exception  
 \*/* **public static void** main(String[] args) **throws** Exception {  
  
 Configuration conf = **new** Configuration();  
 String[] otherArgs = **new** GenericOptionsParser(conf, args).getRemainingArgs();  
  
 **if** (otherArgs.**length** != 2) {  
 System.***err***.println(**"Usage: wordcount <in> <out>"**);  
 System.*exit*(2);  
 }  
 Job job = Job.*getInstance*(conf, **"word count"**);  
 job.setJarByClass(WordCount.**class**);  
 job.setMapperClass(TokenizerMapper.**class**);  
  
 job.setCombinerClass(IntSumReducer.**class**);  
  
 job.setReducerClass(IntSumReducer.**class**);  
 job.setOutputKeyClass(Text.**class**);  
 job.setOutputValueClass(IntWritable.**class**);  
  
 FileInputFormat.*addInputPath*(job, **new** Path(otherArgs[0]));  
 FileOutputFormat.*setOutputPath*(job, **new** Path(otherArgs[1]));  
  
 System.*exit*(job.waitForCompletion(**true**) ? 0 : 1);  
  
 }  
}

Obviously, this simplified solution uses a regular Text File (in this case of randomized words) in an input directory, then the algorithm MapReduce’s it to output directory. There is a MongoDB connector to Hadoop, in the form of jar files. Using this connector we could change Input, Output Text formats of MapReduce programs to **MongoInput** and **MongoOutput** classes of a MongoDB Collection.

MongoDB stores data as BSON (Binary JSON) objects, so we would have the Mapper input value as a **BasicDBObject.** The same would be passed into the map method. Map method would work on each object. get() method would be used to parse a BasicDBObject and extract required fields from it. In the Reducer class we would send Mapper output to the context.