CSCI-B 649 CLOUD COMPUTING
Project-I Report
Word Count

Vraj Parikh (parikhv@indiana.edu),
Marshal Patel (marshalp@indiana.edu),
Shanmukh Sista (ssista@indiana.edu)

February 1, 2015

Transformation of Data

The transformation of data for hadoop happens in the following stages:

1. Load the input file from HDFS.
2. Read the file line by line and extract tokens from the file.
3. Configure a new Hadoop job be setting the inputs, outputs, file, map and reduce job classes.
4. Run the job and wait until the job finishes.

Out of these steps, two most important classes are the Map and Reduce classes.

Map class

```
1  public static class Map extends Mapper<LongWritable, Text, Text,
       IntWritable>{
2       private final static IntWritable one = new IntWritable(1);
3       private Text word = new Text();    // type of output key
4       public void map(LongWritable key, Text value, Context context) throws
            IOException, InterruptedException {
5         /* Another variant of string tokenizer which we have implemented. By
               default the output string treats "word", "word," , "word:" as
              three seperate words. We think this is not an efficient way of
              counting. We have added common sentence tokens so that these
              tokens are not included in the word. */
6         StringTokenizer itr = new StringTokenizer(value.toString(), ",.;-'\"
              \t\n\r");// line to string token
7         while (itr.hasMoreTokens()) {
8           word.set(itr.nextToken());      // set word as each input keyword
9           context.write(word, one);       // create a pair <keyword, 1>
10        }
11      }
12    }
```

The first and the foremost prerequisite to assign a map class is to exted the Mapper class in Hadoop. In line 4 we define a map method that would be called when the map-reduce job starts. This map method reads the input file line by line and then extracts tokens by using the Java

class StringTokenizer. ***We have tweaked this implementation for the tokenizer to accept more tokens thant the default tokens i.e. spaces and tabs. While running and examining the output of the original program, we found that the words "you", "you!", "you?", "you," are treated as four different words. Logically, this should not be the output of the program. So we have added more tokens to filter the input text and generate the count more accurately.***

On line 7, inside the while loop, the 'word' variable is set to the next token to continue processing and the current word's key value pair is added to the Mapper Context and the count for this word is set to 1, which is of type IntWritable.

Reduce class

The reduce class in the WordCount Program is responsible for aggregating all the counts for the map tasks count and write it to the output directory.Given below is the implementation and overview of the Reduce class.

```
1  public static class Reduce extends Reducer<Text,IntWritable,Text,
       IntWritable> {
2      private IntWritable result = new IntWritable();
3      public void reduce(Text key, Iterable<IntWritable> values,
4                         Context context
5                         ) throws IOException, InterruptedException {
6        int sum = 0; // initialize the sum for each keyword
7        for (IntWritable val : values) {
8          sum += val.get();
9        }
10       result.set(sum);
11       context.write(key, result); // create a pair <keyword, number of
             occurences>
12     }
13 }
```

Similar to the Mapper class, the Reduce class extends the Reducer class. We then define a reduce method which accepts the intermediate key values and the Hadoop Mapper Context as inputs. It then calculates the sum of the counts for a given key and writes the result for the given key.

Combiner class

A combiner class is responsible for intermediate processing of the key value pairs before sending these pairs as inputs to the Reducer. In the WordCount Program, the map method assigns a value for each key. So if a key is repeated multiple times in the mapper, each of these keys will have a count of 1. When processing large datasets, this can be an overhead as there is a better way of aggreating the keys before reducing. A combiner class is exactly same as the reducer class in implementation logic but it calculates the sum for a map operation and this key value pair is sent to the reducer for further processing. The code for implementing combiner consists of assigning a combiner class, similar to the Mapper and Reducer class. We can use the reducer class for the combiner as well.

```
1      //Set the combiner class to perform the combining of key-values for a
           given map
2      job.setCombinerClass(Reduce.class);
```

The output for our program is stored in HDFS and can be accessed by executing cat command on hdfs. This file can be later copied to the local file system. A part of the output is shown below.

```
1 writer  1
2 writhe  1
3 writhed 2
```

```
4 writing 6
5 written 4
6 wrong 6
7 wrongs  1
```

Initialization

The main method for the Java Program initializes a new Job and specified configuration parameters for the Mapper, Reducer and Combiner classes. Below is the code for the main method.

```
1  // Driver program
2    public static void main(String[] args) throws Exception {
3      Configuration conf = new Configuration();
4      String[] otherArgs = new GenericOptionsParser(conf, args).
           getRemainingArgs(); // get all args
5      if (otherArgs.length != 2) {
6        System.err.println("Usage: WordCount <in> <out>");
7        System.exit(2);
8      }
9      // create a job with name "wordcount"
10     Job job = new Job(conf, "wordcount");
11     job.setJarByClass(WordCount.class);
12     job.setMapperClass(Map.class);
13     //Set the combiner class to perform the combining of key-values for a
           given map
14     job.setCombinerClass(Reduce.class);
15     job.setReducerClass(Reduce.class);
16     // set output key type
17     job.setOutputKeyClass(Text.class);
18     // set output value type
19     job.setOutputValueClass(IntWritable.class);
20     //set the HDFS path of the input data
21     FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
22     // set the HDFS path for the output
23     FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
24     //Wait till job completion
25     System.exit(job.waitForCompletion(true) ? 0 : 1);
26   }
```

Data Flow for Hadoop

The data flow for Hadoop can be described below:

(a) Input file for the program execution is copied the hdfs partition. This gets read when the program executes.

(b) In the next stage, i.e. the map task, the mapper loads the input data in Main memory and processes this to produce intermediate map files which are written to the disk for reduction. The combiner processes this file before it is written to the disk.

(c) During reduction stage, the processed map files are loaded in the main memory and further reduced to give us the output.

(d) At the end, Output file is written to the HDFS.

(e) Once the reduce tasks are completed, all the temporary map files are deleted from the filesystem.

Memory Management

All hadoop disk operations are performed inside Hadoop's Distributed File system. A HDFS cluster consists of two nodes namely

1. Name Node

   Manages all the filesystem metadata

2. Datanode

   Datanode stores the actual data.

Hadoop manages the available memory for all its tasks by by allocating available memory to various tasks. Admins can specify the configuration parameter mapreduce.map|reduce.memory.mb in megabytes for the map/reduce tasks. This way an memory can be managed efficiently depending on the input size and the number of map/reduce jobs running simultaneously.