## Owen Galvin

## HU Extension School E-63 Big Data Analytics

## Assignment 03

### Handed out: 02/12/2016 Due by 11:30PM on Friday, 02/19/2016

Download and Install VMWare Workstation or Fusion product from HU VMWare store.

**Problem 1)** Modify attached class WordCount.java so that its result excludes the following stop words:

|  |  |  |
| --- | --- | --- |
| I  a  about  an  are  as  at  be  by  com  for  from how | in  is  it  of  on  or  that the  this to  was  what  when where | who  will  with the www |

as well as special characters (dashes, parentases, etc). Stop word lists could be much lomger than this. You do not have to be extremely thoroughful. You would like to get a more or less clean list of ordinary words with the numbers of their occurances. Do not fret. Be reasonable. Perform analysis on te text of James Joyce’s Ulysis.

**Solution: Running as joe, check hadoop dir from previous Assign02, to make sure everything looks ok, see the 4300.txt is in ulysses directory:**

|  |
| --- |
|  |

**Then in terms of environment setup for all the problems, I wound up going primarily with Eclipse installed onto my VM. I had spent many hours trying to get Eclipse able to run anything useful on Windows 7 host & then getting jar etc. on command line, which did work but was way too onerous in terms of being able to debug also. My eclipse project folder = proj in home directory of joe. In there create a jvLib folder and then an other subfolder named hadoop. Into that last directory, copy all the .jar files specified in the lecture.**

|  |
| --- |
|  |

**Then using Build Path option in Eclipse, add all of those .jar files as a User Library named hadoop, on the left below. As a result my lec03 Eclipse project has references to everything.**

|  |  |
| --- | --- |
|  |  |

**I used the original WordCounter.java as the template code. Fromthere I will go over the changes I made, top to bottom, highlighting my actual code changes among the existing java as appropriate. First was set the package to my very simple pkg value, striving for simplicity just to get things to work. Additionally import java.util.Arrays, used later in the stop word cleansing.**

|  |
| --- |
| package pkg;  import java.io.IOException;  import java.util.Arrays;  import java.util.StringTokenizer; |

**After renaming the class to WordCountP1, construct a static string array of the stop words, though leaving out “I” for now since the string check later on will force lower-case on the found values and don’t want to confuse “i” with “I”.**

|  |
| --- |
| public class WordCountP1 {  // NOTE: "I" moved to separate check  public static final String[] STOP\_WORDS = new String[] {"a","about","an","are","as","at","be","by","com","for","from","how","in","is","it","of","on","or","that","the","this","to","was","what","when","where","who","will","with","the","www"  }; |

**Next change involves pulling the word into its own variable, for easier text checking. Use Java String.ReplaceAll to remove any character that is not alpha-numeric, via an appropriate regular expression. Next, boolean variable only sets to true if the encountered word exactly equals upper-case I character. Finally the two remaining parts are put together, so that a given word is emitted only if it is not a word in the STOP\_WORDS array and is not the letter “I”.**

**\*\*\* My original assumption was that we were cleaning special characters by removing words that exactly = a special character. I’ve updated the code to instead cleanse existing words but I wanted to mention as I might have missed a screenshot update along the way \*\*\***

|  |
| --- |
| 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()) {  String rawText = itr.nextToken();  //perhaps overly-harsh regex, e.g will remove hyphens from middle of words  String cleanWord = rawText.replaceAll("[^A-Za-z0-9 ]", "");    boolean isFirstPersonPronoun = cleanWord.equals(new String("I"));  if (!Arrays.asList(STOP\_WORDS).contains(cleanWord.toLowerCase())  && !isFirstPersonPronoun) {  word.set(cleanWord);    context.write(word, one);  }  }  }  } |

**Final change is very simple, as part of the job configuration, update the job name and class to my renamed WordCountP1.**

|  |
| --- |
| Job job = Job.getInstance(conf, "wordcountP1");  job.setJarByClass(WordCountP1.class); |

**Now I’ll discuss my general Run environment, which involves a \_hadoop directory as well as a pkg subdirectory (which is the name of the Java package). I felt it cleaner to manually copy items into this structure as necessary, in part simply to make my .doc screenshots clearer. Also, probably won’t have time for similar screenshots for following problems, though the steps will basically be the same.Below you see I copied the three WordCountP1 class files into pkg.**

|  |
| --- |
|  |

**To generate the matching .jar file, within Eclipse right-click on the project and select Export, choosing “Jar file” from the following dialog.**

|  |
| --- |
|  |

**On next dialog, browse to, and select only, the WordCountP1.java resource. For the export destination, select the proj/\_hadoop directory mentioned above.**

|  |
| --- |
|  |

**Check my terminal is in the right place, ls to confirm contents. Very first step had already been to verify ulysses directory in HDFS, so should be good to go. Submit the job to hadoop, passing in the name of the .jar, followed by the full class path, then input directory & the output directory that gets created.**

|  |
| --- |
| **...** |

**Check results by confirming output file was generated and then examining the first 10/last 10 lines. Compared to orig file I can see the output is much cleaner, none of the words beginning with special characters at the beginning. Examining the full ouput in gpedit after get’ng out of HDFS, as well as testing with smaller files confirms the stop words were removed also.**

|  |
| --- |
|  |

**Problem 2)** Write another MapReduce program which would read the “word count” output of the previous job and order the results by the declaining number of occurances.

**Solution: I’ve actually solved this problem last as I got hung up on it initially and got nowhere, circling back once everything was complete. The first item to note is that the input I’m using is the direct output of Problem 1, the final version with special characters stripped from words (as well as stop word removal). I’ve renamed that output file as P1\_out\_clean.txt and placed in home directory of Joe. From there I will push into hdfs, first creating a directory named P2\_in and then using put to copy the file in. Confirm result via list on the new hdfs directory.**

|  |
| --- |
|  |

**I’ll run through the code quickly, since there are more details in later problems. In the end I had taken the Inverter.java that I updated to the newer API in Problem 5 and from there made a few quick switches to get it to sort in descending order. Since there is an existing SortComparatorClass for LongWritables, the first thing I added was an import for that type. From there I changed the output key to be LongWritable, since sorting of keys is part of the map reduce process. Then I took the value, which would be the right hand side of the line parsing and represent the wordcount and cast it as a Long, which could then be instanciated into a LongWritable variable named count, written via context as the key.**

|  |
| --- |
| public class P2Sort extends Configured implements Tool {  public static class MapClass  extends Mapper<Text, Text, LongWritable, Text>{  public void map(Text key, Text value, Context context) throws IOException, InterruptedException {  Long rawCount = Long.parseLong(value.toString());  LongWritable count = new LongWritable(rawCount);    context.write(count, key);  }  } |

**Update the Reduce to take the LongWritable key from map as input and then write that out as output, because I think it will look cleaner on the right hand side of the output. Also the value coming from map is received as a simple Text, not an Iterable of Text items. The contents of reduce as defined in the Inverter template are cleaned out since we don’t need to perform any calculations, simply swap the key and value positions.**

|  |
| --- |
| public static class Reduce extends Reducer<LongWritable, Text, Text, LongWritable> {    public void reduce(LongWritable key, Text value,  Context context ) throws IOException, InterruptedException {  context.write(value, key);  }  } |

**A few notable changes in run, will only paste relevant portions below. First edit is to replace comma delimter from Inverter.java with a tab character, since that matches format of input file. Add MapOutputKey/Value setting, since these differ from those in the reduce function. Then add the magic setSortComparatorClass() method on the Job class, passing in the value that will sort keys in descending order. (At one point I was going to multiply the count values by -1, assemble a value consisting of “word count”, and then...) Finally, update the value class on the final output to LongWritable, for the count value.**

|  |
| --- |
| public int run(String[] args) throws Exception {  Configuration conf = getConf();  conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", "\t");  . . .  job.setMapOutputKeyClass(LongWritable.class);  job.setMapOutputValueClass(Text.class);  job.setSortComparatorClass(LongWritable.DecreasingComparator.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(LongWritable.class); |

**In terms of code/jar prep for execution, repeat the same steps described in Problem1. Starting in my \_hadoop directory, copy the three .class files from Eclipse proj bin/pkg into /pkg subfolder, Export a P2Sort.jar from Eclipse into root of \_hadoop. Confirm they are all there with an ls and then execute the hadoop job, passing in correct jar, and path to P2Sort class + in/out directories. Begin + end of cmd output below.**

|  |
| --- |
| **...** |

**Final step is to confirm results of the generated file, displaying first 10 & last 10 lines in the file. As desired they are sorted in descending word count order. (Double check that stop word list from Problem 1 and confirm that “and” was not in there, little worried for a moment.)**

|  |
| --- |
|  |

**Problem 3)** Create a program that will find out how many words appear only once, how many twice, three times, four times and so on in James Joyce’s Ulysis

**Solution: I’m doing some things out of order, so safest to use the original raw output from Assignment 2 as the input to this problem. I’ve renamed that to full\_count.txt and copy that into HDFS from joe’s home directory, confirm file has made it in.**

|  |
| --- |
|  |

**In terms of creating the java code, I started out with Invert.java and updated, keeping it in the older API format. The first change was to add an import, since IntWritable will be needed.**

|  |
| --- |
| import org.apache.hadoop.io.IntWritable; |

**Update the generic interface to output an IntWritable as key in the output, instead of Text. Similar update to the OutputCollector. Then add some simple code to take the value (word count) and cast as an Integer. Not sure if I tried to read it in as an IntWritable in the first place... either way this works fine, multi-step allows for better debugging. Now instanciate an IntWritable using that number value. Output from map in format of <wordCount, word>.**

|  |
| --- |
| //NOTE: still in old API format  public class P3Counter extends Configured implements Tool {    public static class MapClass extends MapReduceBase  implements Mapper<Text, Text, IntWritable, Text> {    public void map(Text key, Text value,  OutputCollector<IntWritable, Text> output,  Reporter reporter) throws IOException {    Integer rawCount = Integer.parseInt(value.toString());  IntWritable count = new IntWritable(rawCount);    output.collect(count, key);  }  } |

**Update the reduce function and Reducer interface reference to expect IntWritable as the type of key. Update latter interface + OutputCollector to support output in the format of <IntWritable, IntWritable> , since the file we are writing will be in format of “numberOfAppearances Count”. Create an empy IntWritable, to hold the final number we want in a format acceptable to Hadoop. The next block replaces the original ‘csv’ code, using a loop through the values Iterator to count the number of items (Text words). Push that value into the result variable and emit everything as key = numberOfAppearances & value = the total of words with that numberOfAppearances.**

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

**I’m going to skip screenshots of the minor renaming of Inverter references to my new P3Counter class, below is the remaining set of changes. Need to individually set the types on the Map outputs since they differ from those in Reduce... perhaps could have skipped the setMapOutputKeyClass setting since that is the same. Then update the Output Key/Value types to match the Intwritables going into the output file. Finally, change the key.value separator from the comma in the Inverter example to a tab, which I understand to be the default separator anyway.**

|  |
| --- |
| //map emits diff key/value types vs. reduce  job.setMapOutputKeyClass(IntWritable.class);  job.setMapOutputValueClass(Text.class);    job.setInputFormat(KeyValueTextInputFormat.class);  job.setOutputFormat(TextOutputFormat.class);    job.setOutputKeyClass(IntWritable.class);  job.setOutputValueClass(IntWritable.class);    job.set("key.value.separator.in.input.line", "\t"); |

**Next steps were essentially described in Problem1. Starting in my \_hadoop directory, copy the three .class files from Eclipse proj bin into /pkg subfolder, Export a P3Counter.jar from Eclipse into root of \_hadoop. Confirm they are all there.**

|  |
| --- |
|  |

**The full\_count.txt had been put into hdfs as first step, ready to run. Do so with P3Counter class/jar references, truncated output.**

|  |
| --- |
| **...** |

**Now to check the output, which doesn’t have an “-r” unlike before. First 20 lines are below, tail isn’t as interesting, with a bunch of numberOfAppearances with count = 1.**

|  |
| --- |
|  |

**Problem 4)** Combine operations of two MapReduce programs in Problems 1 and 3 above into a single program with chained MapReduce jobs.

**Solution: Create a new P4Chained class, with the top section of the class pretty much a copy and paste of the associated Map & Reduce classes from Problems 1 and 3. The big “difference” here is that first I needed to update Problem 3 to the new API. I won’t have time to go over those changes here and they will presumably get 95% covered in Problem 5 since my orig P3 was based on Inverter w/old API. No need to rename the inner classes since both map/reduce names are diff to begin with. Additionally I had trouble with import references, start with the WordCount (new API) imports and then pretty much trial and error until everything agreed.**

**As to the latter section, more directly tied to the chaining, create two createP\*Job functions, the first one related to the actions orig. going on in Problem 1. Set the Mapper/Combiner/Reducer classes same as in P1, which are same values as in orig WordCount.java. Below generally matches same settings from P1. None of the below examples use highlighting since everything is more or less “new”.**

|  |
| --- |
| private Job createP1Job(Configuration conf, Path in, Path out)  throws IOException {  Job job = Job.getInstance(conf, "P4Chained");  job.setJobName("job1");  job.setJarByClass(P4Chained.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class);  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.setInputPaths(job, in);  FileOutputFormat.setOutputPath(job, out);    return job;  } |

**The createP3Job similar but the settings of course match those from Problem 3.**

|  |
| --- |
| private Job createP3Job(Configuration conf, Path in, Path out)  throws IOException {  Job job = Job.getInstance(conf, "P4Chained");  job.setJobName("job2");  job.setJarByClass(P4Chained.class);  job.setMapperClass(MapClass.class);  job.setReducerClass(Reduce.class);    job.setInputFormatClass(KeyValueTextInputFormat.class);  //map emits diff key/value types vs. reduce  job.setMapOutputKeyClass(IntWritable.class);  job.setMapOutputValueClass(Text.class);  job.setOutputKeyClass(IntWritable.class);  job.setOutputValueClass(IntWritable.class);    FileInputFormat.setInputPaths(job, in);  FileOutputFormat.setOutputPath(job, out);  conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", "\t");    return job;  } |

**Below is the run function, which follows general format of the ChainedHistogram example from the lecture, updated to new API. In theory I should have at least checked the return value on the first job before moving on to the 2nd but that is rather minor, all things considered.**

|  |
| --- |
| public int run(String[] args) throws Exception, IOException {  Configuration conf = new Configuration();  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();    Path in = new Path(otherArgs[0]);  Path out = new Path(otherArgs[1]);  Path temp = new Path("chain-temp");    Job job1 = createP1Job(conf, in, temp);  Integer job1Result = job1.waitForCompletion(true) ? 0 : 1;  Job job2 = createP3Job(conf, temp, out);  System.exit(job2.waitForCompletion(true) ? 0 : 1);    //comment out during DEBUG  cleanup(temp, conf);    return 0;  } |

**Finally, I’ll mention the cleanup function, pulled from the lecture pdf. It doesn’t seem to actually do what it is supposed to but I didn’t try to troubleshoot. I’m happy to have output in the final directory.**

|  |
| --- |
| private void cleanup(Path temp, Configuration conf)  throws IOException {  FileSystem fs = temp.getFileSystem(conf);  fs.delete(temp, true);  } |

**Now on to execution, copy the P4Chained .class, export .jar for same, all going to my \_hadoop (/pkg) structure. Confirm they are all in place and run the job, display beg & end command output.**

|  |
| --- |
| ..... |

**Look at the first 20 lines of output, everything looks fine. Numbers don’t come really close to those from Problem 3 since that used the raw word count values from Assignment 2, not the stop-word-cleansed version. Also, the chain-temp hdfs directory was not deleted, guessing the cleanup() function in P4Chained might need to be updated with correct references to deal with HDFS, not going to worry about it now.**

|  |
| --- |
|  |

**Problem 5)** Move attached Inverter.java class from old MapReduce API to the new API. Demonstrate that new and old clas produce the same result. Use patent data set to demonstrate your work.

**Solution: First set of code changes involve the import statements. I basically added in the imports present in the WordCount file and then removed the old API ones as Java told me they were unneeded. Most likely there were one or two new APIs I also had to add in or remove. Then to update the MapClass, remove the interface mention of Mapper and replace MapReduceBase superclass with Mapper. Replace OutputCollector/Reporter with Context arg, add in coverage of InterruptedException. Finally replace output.collect with context.write.**

|  |
| --- |
| public static class MapClass  extends Mapper<Text, Text, Text, Text>{  public void map(Text key, Text value, Context context) throws IOException, InterruptedException {  context.write(value, key);  } |

**Similar updates to Reduce, additionally the Iterator becomes an Iterable. That latter change necessitates replace the while values.hasNext loop with a for each loop. And then we can pull the currently iterated value (val variable) directly into csv once it is cast to a String.**

|  |
| --- |
| public static class Reduce extends Reducer<Text, Text, Text, Text> {    public void reduce(Text key, Iterable<Text> values,  Context context ) throws IOException, InterruptedException {    String csv = "";  for(Text val : values) {  if (csv.length() > 0) csv += ",";  csv += val.toString();  }  context.write(key, new Text(csv));  }  } |

**In the run method, the syntax of setting the text delimiter changes and is set on Configuration object, not Job. Syntax on creating a new Job changes, and name is set in that getInstance() method instead of via SetJobName. Use setInputFormatClass instead of setInputFormat & FileInputFormat.addInputPath in place of FileInputFormat.setInputPaths, same for output (maybe that one isn’t an API change?). And finally replace the JobClient.runJob(job) with waitForCompletion, called directly on the job object.**

|  |
| --- |
| public int run(String[] args) throws Exception {  Configuration conf = getConf();  conf.set("mapreduce.input.keyvaluelinerecordreader.key.value.separator", ",");    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  if (otherArgs.length < 2) {  System.err.println("Usage: P5Inverter\_newAPI <in> [<in>...] <out>");  System.exit(2);  }    Job job = Job.getInstance(conf, "P5Inverter\_newAPI");  job.setJarByClass(P5Inverter\_newAPI.class);  job.setMapperClass(MapClass.class);  job.setReducerClass(Reduce.class);    job.setInputFormatClass(KeyValueTextInputFormat.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    for (int i = 0; i < otherArgs.length - 1; ++i) {  FileInputFormat.addInputPath(job, new Path(otherArgs[i]));  }  FileOutputFormat.setOutputPath(job,  new Path(otherArgs[otherArgs.length - 1]));  System.exit(job.waitForCompletion(true) ? 0 : 1);    return 0;  } |

**Preparing for execution, copy the patent data file (cite75\_99.txt) from joe’s home directory into HDFS. Make an hdfs dir named patents and then pass in put command to copy the patent file into it, confirm put was successful.**

|  |
| --- |
|  |

**From Eclipse bin/pkg dir, copy the old and new Inverter classes, both prefixed with P5, to \_hadoop/pkg. Export .jar for both, placed in root /\_hadoop. Confirm they are all in place.**

|  |
| --- |
|  |

**Run the job using the old API, ie orig Inverter file/class, display beg & end command output.**

|  |
| --- |
| **...** |

**Followed by running the job with the new API, display beg & end.**

|  |
| --- |
| **....** |

**Now to compare the output of both, I’m taking the equality in Bytes Written value at the end there to be a good sign. Run a head on both files to look at the first 20 lines and everything looks to be the same. I updated the fs –cat I used for the last problem, leaving the file name in place, which turned out to be correct for the orig API output. I tried same for P5\_newAPI\_out dir and it failed though – apparently there the file got generated with a –r in the middle.**

|  |
| --- |
|  |
|  |

**Now for a tail check, but only 10 records so I can get in same screenshot.**

|  |
| --- |
|  |

If you have your own working VM with installed CDH5.5.1, do this assignment on that VM. If, for what ever reason, you do not posses a working VM with CDH5.5.1, please be free to download Clouder’s Getting started VM and do you assignment on that VM.

Capture all steps of your implementation with comments indicating what is it you are accomplishing with every step in an MS Word document. Upload to the class site. Please upload your working Java files as well. Please post comments and questions to the class Discussion Board on the Canvas site.