

Big Data Analytics

Isara Anantavasilp

Lecture 9-10: MapReduce Continued

WordCount Revisited

- In the previous version of WordCount, we use main function as the driver

```
public static void main(String[] args)
throws Exception { ... }
```

- Drawbacks:
 - You have to recompile the code, rebuild the JAR file if you want to **reconfigure** your task
 - Example: Number of reducers, required libraries

Tool and ToolRunner

- **ToolRunner** is a utility class that runs classes which implements Tool
 - Our new WordCount will implement Tool too

```
public class WordCount extends Configured  
implements Tool {
```

- ToolRunner delegates to **GenericOptionParser**
 - Parses command line arguments (options)
 - Sets parsed arguments on Configuration object

Revamped WordCount

Configured is an implementation class of the interface **Configurable**



```
public class WordCount extends Configured implements Tool
{
    public static class TokenCounterMapper
        extends Mapper<Object, Text, Text, IntWritable>{
    }
    public static class IntSumReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {
    }

    public int run(String[] args) throws Exception {
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCount(), args);
        System.exit(exitCode);
    }
}
```

WordCount run()

```
public int run(String[] args) throws Exception {  
    Configuration conf = getConf();  
  
    args = new GenericOptionsParser(conf, args).getRemainingArgs();  
  
    Job job = Job.getInstance(conf);  
  
    job.setJarByClass(WordCountConf.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setReducerClass(IntSumReducer.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
    return (job.waitForCompletion(true) ? 0 : 1);  
}
```

n -Gram

- **n -Gram** is a contiguous sequence of n items from a given sample of text or speech
 - Elements that always come together
 - This could be syllables, letters, words, etc.
- **2-gram** or **bigram** is a pair of items that always come together
 - Example: to + be, t+o,
- Let's implement a bigram analysis

Bigram Mapper

```
public static class BiGramMapper
    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 {
        String[] words = value.toString().split(" ");

        Text bigram = new Text();
        String prev = null;

        for (String s : words) {
            if (prev != null) {
                bigram.set(prev + "\t+\t" + s);
                context.write(bigram, one);
            }

            prev = s;
        }
    }
}
```

Split string with space

Create a pair of words:
previous + current

Add that to the context

We also use Hadoop library

- If you take a look at the driver, you will notice that we changed the reducer

```
job.setReducerClass(IntSumReducer.class);
```

- It is provided by Hadoop and is imported here:

```
import  
org.apache.hadoop.mapreduce.lib.reduce.  
IntSumReducer;
```

- It uses the same logic as our reducer

To run the Bigram code

- **Compile:**

```
hadoop com.sun.tools.javac.Main  
BiGramCount.java
```

- **Make Jar file:**

```
jar cf bigram.jar BiGramCount*.class
```

- **Run the program (We also set reduce task to 1):**

```
hadoop jar bigram.jar BiGramCount -D  
mapred.reduce.tasks=1 [input] [output]
```

Bigram Analysis

- Let's have a look at the Bigram output

```
cat part-r-00000
```

- We can also sort and filter (all in one line):

```
cat part-r-00000 | sort -t$'\t' -k4  
-nr | head -n 20
```

Exercise (5 points)

- Sort the result of WordCount
- BiGram suffers from the same “dirty” characters as WordCount (commas, full stops, etc.) as well as tabs and spaces.
Clean BiGram result.

Finding Top 10 Words

- We could use Unix tool to sort and filter word count results, but it will not scale
- Let's use MapReduce instead
- Download and run TopTenWords.java using WordCount result as input

```
hadoop jar toptenwords.jar  
TopTenWords out/part-r-00000 out-  
topten
```

TreeMap

- TopTenWords utilizes Java **TreeMap**
 - Efficient means to store key-value pairs in sorted order
 - Guaranteed to sort in ascending order
 - Use method put (key, value) to add element to tree
 - Key is always sorted

TopTenWords - Mapper

- Map function adds words to the tree map
 - If tree is larger than 10, remove the lowest one
- Then in **cleanup** method, it **emits** values to context
 - The results of the map are written at the end of the all loops, not during the loops
 - cleanup is called once after all key/value pairs have been presented to the map method
- The output key is `NullWritable` which writes null value (zero-length serialization)
 - `NullWritable` can be used in both key or value if you do not need to write anything (empty value or any value)

TopTenWords - Mapper

```
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    // (word, count) tuple
    String[] words = value.toString().split("\t") ;
    if (words.length < 2) {
        return;
    }

    topN.put(Integer.parseInt(words[1]), new Text(value));

    if (topN.size() > 10) {
        topN.remove(topN.firstKey());
    }
}

protected void cleanup(Context context) throws IOException,
    InterruptedException {
    for (Text t : topN.values()) {
        context.write(NullWritable.get(), t);
    }
}
```

Use count as key

Use (word, count) as value

Key could be anything

TopTenWords - Reducer

- Reducer employs the same concept as mapper
- It takes the top ten list from all mappers and filter out only the highest 10
- Note that it takes `NullWritable` as input (and output)

```
public void reduce(NullWritable key, Iterable<Text> values,
    Context context) throws IOException, InterruptedException {
    for (Text value : values) {
        String[] words = value.toString().split("\t") ;
        topN.put(Integer.parseInt(words[1]), new Text(value));
        if (topN.size() > 10) {
            topN.remove(topN.firstKey());
        }
    }
    for (Text word : topN.descendingMap().values()) {
        context.write(NullWritable.get(), word);
    }
}
```


Multi-Step Processes

- Previously, we have only one mapper
- Sometimes, we need several mappers to work together e.g. pre-processing data, trimming text or set text cases
- **ChainMapper** class allows several mappers to work as a pipeline
- We can specify which mappers to be used and they will be executed one after one
- Output of the first mapper, will be the input of the second mapper

ChainMapper

- Mappers are added to the configured job using the following method:

```
ChainMapper.addMapper(  
    JobConf job,  
    Class<? extends Mapper<K1,V1,K2,V2>> class,  
    Class<? extends K1> inputKeyClass,  
    Class<? extends V1> inputValueClass,  
    Class<? extends K2> outputKeyClass,  
    Class<? extends V2> outputValueClass,  
    JobConf mapperConf)
```

addMapper Arguments

- **job**: JobConf to add the Mapper class
- **class**: Mapper class to add
- **inputKeyClass**: mapper input key class
- **inputValueClass**: mapper input value class
- **outputKeyClass**: mapper output key class
- **outputValueClass**: mapper output value class
- **mapperConf**: a **JobConf** with the configuration for the Mapper class
- Input and output classes must match those in classes declaration

Adding Mapper in the Driver

```
public int run(String[] args) throws Exception {  
    ...
```

```
    Configuration lowerCaseMapperConf= new Configuration(false);  
    ChainMapper.addMapper(job,  
        LowerCaseMapper.class,  
        Object.class, Text.class,  
        IntWritable.class, Text.class,  
        lowerCaseMapperConf);
```

**IntWritable is not needed.
This could be just NullWritable**



```
    Configuration tokenizerConf= new Configuration(false);  
    ChainMapper.addMapper(job,  
        TokenizerMapper.class,  
        IntWritable.class, Text.class,  
        Text.class, IntWritable.class,  
        tokenizerConf);  
}
```

Mapper Declarations

- Inputs and outputs of all mappers must be corresponding to each other in both class declarations and in the configuration

```
public static class LowerCaseMapper  
    extends Mapper <Object, Text,  
        IntWritable, Text>
```

```
public static class TokenizerMapper  
    extends Mapper <IntWritable, Text,  
        Text, IntWritable>
```

LowerCaseMapper

- Let us define a simple pre-processing mapper
- LowerCaseMapper uses String's `toLowerCase()` method to change all input text to lower case

```
public static class LowerCaseMapper
    extends Mapper<Object, Text, IntWritable, Text> {

    private Text lowercased = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        lowercased.set(value.toString().toLowerCase());
        context.write(new IntWritable(1), lowercased);
    }
}
```

Configuration Steps

- We have to modify Tokenizer in WordCount
- Add the mappers into the chain
 - Make sure that the types of all mappers in the chain are corresponding to each other
- The rest is the same
- The source code is in Moodle

Exercise (5 points)

- Clean text files
 - Change text to lower case
 - Add another mapper to trim words and remove special characters (e.g. “”, :, ;, ...)
- Count Top-10 using TreeMap
- Work with group of 1-2 people