

Massive Data Processing

A Variation of an Inverted Index

Assignment_1 Wanqing WANG B00693850

You are asked to implement an inverted index in MapReduce for the document corpus of pg100.txt (from <http://www.gutenberg.org/cache/epub/100/pg100.txt>), pg31100.txt (from <http://www.gutenberg.org/cache/epub/31100/pg31100.txt>) and pg3200.txt (from <http://www.gutenberg.org/cache/epub/3200/pg3200.txt>). This corpus includes the complete works of William Shakespear, Mark Twain and Jane Austen, respectively.

An inverted index provides for each distinct word in a document corpus, the filenames that contain this word, along with some other information (e.g., count/position within each document).

For example, assume you are given the following corpus, consisting of doc1.txt, doc2.txt and doc3.txt:

- doc1.txt: "This is a very useful program. It is also quite easy."
- doc2.txt: "This is my first MapReduce program."
- doc3.txt: "Its result is an inverted index."

An inverted index would contain the following data (in random order):

| word | filename |
|---------|------------------------------|
| this | doc1.txt, doc2.txt |
| is | doc1.txt, doc2.txt, doc3.txt |
| a | doc1.txt |
| program | doc1.txt, doc2.txt |

a) Run a MapReduce program to identify stop words (words with frequency > 4000) for the given document corpus. Store them in a single csv file on HDFS

(stopwords.csv). You can edit the several parts of the reducers' output after the job finishes (with hdfs commands or with a text editor), in order to merge them as a single csv file.

In this assignment, the environment in which the program was implemented is Eclipse in cloudera-quickstart-vm. In order to execute the code, we need to obey the rules of Hadoop.

By applying this command to get the version of hadoop.

```
[cloudera@quickstart Assignment_InvertedIndex]$ hadoop version
Hadoop 2.6.0-cdh5.8.0
Subversion
http://github.com/cloudera/hadoop -r
57e7b8556919574d517e874abfb7ebe31a366c2b
Compiled by jenkins on 2016-06-16T19:38Z
Compiled with protoc 2.5.0
From source with checksum 9e99ecd28376acfd5f78c325dd939fed
This command was run using /usr/lib/hadoop/hadoop-common-2.6.0-cdh5.8.0.jar
```

Here is the main process of the assignment.

First in the Eclipse, creat a new Java Project called Assignment_InvertedIndex. A folder named Assignment_InvertedIndex would appear in the workspace which would be the main environment where to launch the whole program.

Then by creating a package and new classes, we can implement the code in the workspace.

It is crucial is to add External JARS to the library so that we can import the packages to make sure the execution.

Preparing Data

Create input and output folders into the Assignment_InvertedIndex workspace through the Terminal.

```
[cloudera@quickstart ~]$ cd workspace/Assignment_InvertedIndex
[cloudera@quickstart Assignment_InvertedIndex]$ mkdir input
[cloudera@quickstart Assignment_InvertedIndex]$ mkdir output
```

Download the text files into the input directory. Once finishing this step, put the files into HDFS for running the code.

```
[cloudera@quickstart Assignment_InvertedIndex]$ hadoop fs -put input/pg100.txt input
[cloudera@quickstart Assignment_InvertedIndex]$ hadoop fs -put input/pg31100.txt input
[cloudera@quickstart Assignment_InvertedIndex]$ hadoop fs -put input/pg3200.txt input
```

Now check if all those files are in HDFS:

```
[cloudera@quickstart Assignment_InvertedIndex]$ hadoop fs -ls input
Found 3 items
-rw-r--r--  1 cloudera cloudera    5589889 2017-02-15 14:44 input/pg100.txt
-rw-r--r--  1 cloudera cloudera    4454050 2017-02-15 14:45 input/pg31100.txt
-rw-r--r--  1 cloudera cloudera    16013935 2017-02-15 14:44 input/pg3200.txt
```

Implement MapReduce functions

The intuition is to find the stopwords in our corpus which performs the same as doing a wordcount and adding some condition on the number of occurrence of the words in the reduce task.

The first part of the MapReduce Program is to identify stop words(with frequency >4000) for the given document corpus.

Here is the main part of the main function:

```
Job job = new Job(getConf(), "StopWords_Identifier");

    job.setJarByClass(StopWords.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    job.getConfiguration().set(
```

```
"mapreduce.output.textoutputformat.separator", ",");  
job.setNumReduceTasks(1);
```

The first line is to construct a job of mapreduce with 2 parameters “getConf” and the name of the job. The second line is the computational program with class name “StopWords_Identifier”. Actually, we need to implement 3 classes for the program instead of 2 (map and reduce). The StopWords_Identifier class is to set mapreduce how to run map and reduce functions which in a word is to construct a job for mapreduce.

The next following codes are setting map and reduce classes, defining types of key/value which is also to be treated as the key/value of files stored on HDFS.

In the map function, there are some key points of this part: to count the words, it is important to take text-transform into consideration. So the line `toLowerCase()` is to solve that part in order to remove the duplicates.

Here is the code of map function:

```
public static class Map extends  
    Mapper<LongWritable, Text, Text, IntWritable> {  
        private final static IntWritable ONE = new IntWritable(1);  
        private Text word = new Text();  
  
        @Override  
        public void map(LongWritable key, Text value, Context context)  
            throws IOException, InterruptedException {  
            for (String token : value.toString().split("\\s+")) {  
                word.set(token.toLowerCase());  
                context.write(word, ONE);  
            }  
        }  
    }  
}
```

In the reduce function for this program, one of the key points is to add a condition (or constraint) on the number of occurrence of the words before getting the (key/value) outputs:

```
public static class Reduce extends  
    Reducer<Text, IntWritable, Text, IntWritable> {  
        @Override  
        public void reduce(Text key, Iterable<IntWritable> values,
```

```

        Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            if (sum > 4000) {
                context.write(key, new IntWritable(sum));
            }
        }
    }
}

```

Getting Results

Once finishing implementing the mapreduce program, use “Run Configurations” to test the correction and in this way, it is faster than using Terminal to get output results.

The other way is to use Terminal by exporting Jar package first (Here in this assignment is InvertedIndex.jar). Then run hadoop job by applying the following command for StopWords.java

```

[cloudera@quickstart ~]$ hadoop jar InvertedIndex.jar
mdp.invertedindex.StopWords_Identifier input output3

```

Once the job is done, the results can be found located in the HDFS output3 directory by using the following command in the terminal:

```

[cloudera@quickstart ~]$ hadoop fs -ls output3
Found 2 items
-rw-r--r--  1 cloudera cloudera      0 2017-02-16 05:01 output3/_SUCCESS
-rw-r--r--  1 cloudera cloudera 1177 2017-02-16 05:01
output3/part-r-0000

```

In the StopWords.java, the result is generated by 1 reducer, so there is only 1 output file.csv. By applying the command below, the result will be found in the output folder.

```

[cloudera@quickstart ~]$ hadoop fs -getmerge output3
workspace/Assignment_InvertedIndex/output/StopWords_Identifier.csv

```


StopWords.csv X

```

|,202317
a,99209
about,7350
after,4446
all,18828
am,6435
an,12270
and,167100
any,8201
are,13112
as,29531
at,21165
be,27239
been,10036
before,4219
but,30329

```

The log execution time in the Hadoop YARN ResourceManager: 1mins, 34sec


MapReduce Job job_1487264000041_0001
Logged in as: drwho

Application

Job

Overview
 Counters
 Configuration
 Map tasks
 Reduce tasks

 Tools

Job Overview

| | |
|----------------------|------------------------------|
| Job Name: | StopWords_Identifier |
| User Name: | cloudera |
| Queue: | root.cloudera |
| State: | SUCCEEDED |
| Uberized: | false |
| Submitted: | Thu Feb 16 11:08:23 PST 2017 |
| Started: | Thu Feb 16 11:08:51 PST 2017 |
| Finished: | Thu Feb 16 11:10:26 PST 2017 |
| Elapsed: | 1mins, 34sec |
| Diagnostics: | |
| Average Map Time | 57sec |
| Average Shuffle Time | 19sec |
| Average Merge Time | 9sec |
| Average Reduce Time | 9sec |

i. Use 10 reducers and do not use a combiner. Report the execution time.

The number of reducers is defined in the Hadoop driver configuration by the method `job.setNumReduceTasks()`. By modifying the parameter of `NumReduceTasks` to 10, the number of reducers will update to 10.

The execution time without combiner: **3mins, 7sec**



MapReduce Job job_1487264000041_0002

Logged in as: dr:who

- Application
- Job
 - Overview
 - Counters
 - Configuration
 - Map tasks
 - Reduce tasks
- Tools

| Job Overview | |
|----------------------|------------------------------|
| Job Name: | StopWords_10reducers |
| User Name: | cloudera |
| Queue: | root.cloudera |
| State: | SUCCEEDED |
| Uberized: | false |
| Submitted: | Thu Feb 16 11:13:08 PST 2017 |
| Started: | Thu Feb 16 11:13:26 PST 2017 |
| Finished: | Thu Feb 16 11:16:33 PST 2017 |
| Elapsed: | 3mins, 7sec |
| Diagnostics: | |
| Average Map Time | 1mins, 5sec |
| Average Shuffle Time | 49sec |
| Average Merge Time | 5sec |
| Average Reduce Time | 9sec |

ii - Run the same program again, this time using a Combiner. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

By adding 1 line of code: `job.setCombinerClass(Reduce.class);` the program will be updated with a combiner. Combiner is optional in the mapreduce program. In another word, It is also a kind of reduce operation which is designed to combine duplicated keys after map operation.

The execution time with combiner: **2mins, 25sec**



MapReduce Job job_1487264000041_0003

Logged in as: dr:who

- Application
- Job
 - Overview
 - Counters
 - Configuration
 - Map tasks
 - Reduce tasks
- Tools

| Job Overview | |
|----------------------|-------------------------------|
| Job Name: | StopWords_10reducers_combiner |
| User Name: | cloudera |
| Queue: | root.cloudera |
| State: | SUCCEEDED |
| Uberized: | false |
| Submitted: | Thu Feb 16 11:20:21 PST 2017 |
| Started: | Thu Feb 16 11:20:38 PST 2017 |
| Finished: | Thu Feb 16 11:23:04 PST 2017 |
| Elapsed: | 2mins, 25sec |
| Diagnostics: | |
| Average Map Time | 56sec |
| Average Shuffle Time | 42sec |
| Average Merge Time | 0sec |
| Average Reduce Time | 4sec |

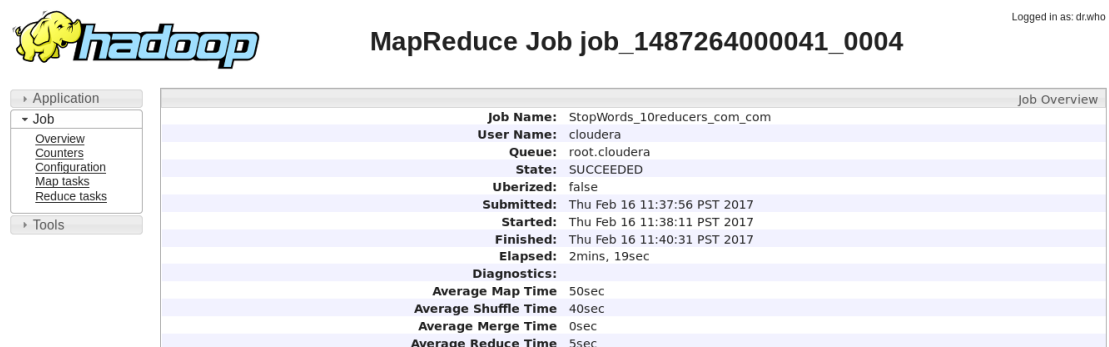
By implementing combiner in mapreduce, the execution time was shorter than that without combiner. The reason is that each map function might produce a lot of output, combiner is the role of mapper to do merge duplicated keys generated from mapper to reduce the amount of data transferred to reducer. The most basic operation is to realize the merge of local keys, which is somehow like the function of reducer.

iii - Run the same program again, this time compressing the intermediate results of map (using any codec you wish). Report the execution time. Is there any difference in the execution, time compared to the previous execution? Why?

By running the same program with implementing the following code into the Hadoop driver configuration :

```
FileOutputStream.setCompressOutput(job, true);
FileOutputStream.setOutputCompressorClass(job,
org.apache.hadoop.io.compress.SnappyCodec.class);
```

The execution time with combiner and compressing: **2mins, 19sec**



The execution time with intermediate results of map is shorter. As intermediate results between map and reduce phrase are not huge, Hadoop can efficiently handle jobs composed by map functions by chaining several mappers which in this way to save the disk I/O cost between map phrases.

iv - Run the same program again, this time using 50 reducers. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

By modifying the parameter in `job.setNumReduceTasks()` to set the number of reducers as 50.

The execution time with 50 reducers combining intermediate results of map:
7mins, 41sec



MapReduce Job job_1487264000041_0005

Logged in as: dr:who

| | |
|---------------|---|
| Application | Job Overview |
| Job | |
| Overview | Job Name: StopWords_50reducers_com_com |
| Counters | User Name: cloudera |
| Configuration | Queue: root.cloudera |
| Map tasks | State: SUCCEEDED |
| Reduce tasks | Uberized: false |
| | Submitted: Thu Feb 16 11:50:48 PST 2017 |
| | Started: Thu Feb 16 11:51:05 PST 2017 |
| | Finished: Thu Feb 16 11:58:46 PST 2017 |
| | Elapsed: 7mins, 41sec |
| Tools | Diagnostics: |
| | Average Map Time: 53sec |
| | Average Shuffle Time: 41sec |
| | Average Merge Time: 0sec |
| | Average Reduce Time: 4sec |

The execution time is much larger with 50 reducers. The reason is that with more reducers, the I/O needs to use more memory and CPU to do the calculation.

Summary of Execution Time

| Hadoop Job | Execution Time |
|-------------------------------|----------------|
| StopWords_Identifier | 1mins, 34sec |
| StopWords_10reducers | 3mins, 7sec |
| StopWords_10reducers_combiner | 2mins, 25sec |
| StopWords_10reducers_com_com | 2mins, 19sec |
| StopWords_50reducers_com_com | 7mins, 41sec |

Question b)

Implement a simple inverted index for the given document corpus, as shown in the previous Table, skipping the words of stopwords.csv.

The intuition in implementing a simple inverted index is that, the program generates outputs in the form of “word, filenames” which the same meaning as “key, value”. The code to form output keys and values as text is implemented as follow.

```
job.setOutputKeyClass(Text.class);  
job.setOutputValueClass(Text.class);
```

The map function shows below:

Important: In the map function, the file generated with StopWords_Identifier was reused. It was important to set the right directory of the text file.

```

public static class Map extends Mapper<LongWritable, Text, Text, Text> {
    private Text word = new Text();
    private Text filename = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        HashSet<String> stopwords = new HashSet<String>();
        BufferedReader Reader = new BufferedReader(
            new FileReader(
                new File(

"/home/cloudera/workspace/Assignment_InvertedIndex/output/StopWords_Identifier.txt"
            )));
        String pattern;
        while ((pattern = Reader.readLine()) != null) {
            stopwords.add(pattern.toLowerCase());
        }

        String filenameStr = ((FileSplit) context.getInputSplit())
            .getPath().getName();
        filename = new Text(filenameStr);

        for (String token : value.toString().split("\\s+")) {
            if (!stopwords.contains(token.toLowerCase())) {
                word.set(token.toLowerCase());
            }
        }

        context.write(word, filename);
    }
}

```

The format of the output:

```

word1, doc1.txt
word1, doc1.txt
word1, doc1.txt
word1, doc2.txt
word2, doc1.txt
word2, doc2.txt
...

```

Reducer stores all the filenames for each word in a HashSet.

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        HashSet<String> set = new HashSet<String>();

        for (Text value : values) {
            set.add(value.toString());
        }

        StringBuilder builder = new StringBuilder();

        String prefix = "";
        for (String value : set) {
            builder.append(prefix);
            prefix = ", ";
            builder.append(value);
        }

        context.write(key, new Text(builder.toString()));
    }
}
```

The format of outputs:

```
word1 -> doc1.txt, doc2.txt
word2 -> doc1.txt, doc2.txt, doc3.txt
word3 -> doc1.txt
...
```

Question c)

How many unique words exist in the document corpus (excluding stop words)? Which counter(s) reveal(s) this information? Define your own counter for the number of words appearing in a single document only. What is the value of this counter? Store the final value of this counter on a new file on HDFS.

Implement counter:

```
public static enum CUSTOM_COUNTER {  
    UNIQUE_WORDS,  
};
```

There is no need to change the map function, however, in reduce function, to elect unique words in the documents and count the number of those, a conditional statement and a counter are necessary for this part of the program.

Implement if statement and counter:

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {  
  
    @Override  
    public void reduce(Text key, Iterable<Text> values, Context context)  
        throws IOException, InterruptedException {  
  
        HashSet<String> set = new HashSet<String>();  
  
        for (Text value : values) {  
            set.add(value.toString());  
        }  
  
        if (set.size() == 1) {  
  
            context.getCounter(CUSTOM_COUNTER.UNIQUE_WORDS).increment(1);  
  
            StringBuilder builder = new StringBuilder();  
  
            String prefix = "";  
            for (String value : set) {  
                builder.append(prefix);  
                prefix = ", ";  
                builder.append(value);  
            }  
  
            context.write(key, new Text(builder.toString()));  
  
        }  
    }  
}
```

Question d)

Extend the inverted index of (b), in order to keep the frequency of each word for each document. The new output should be of the form:

| this | filename and frequency |
|---------|------------------------------------|
| this | doc1.txt#1, doc2.txt#1 |
| is | doc1.txt#2, doc2.txt#1, doc3.txt#1 |
| a | doc1.txt#1 |
| program | doc1.txt#1, doc2.txt#1 |

which means that the word frequency should follow a single '#' character, which should follow the filename, for each file that contains this word. You are required to use a Combiner.

Similar with Question C, there is no need to change the map function. The method is to implement lines of codes to make reducer get the function to store filenames in a collection without removing duplicates filenames:

```
word1 -> doc1.txt, doc1.txt, doc1.txt, doc2.txt
word2 -> doc1.txt, doc2.txt, doc2.txt, doc3.txt
word3 -> doc1.txt
...
```

The key point to keeping the duplicates is that in reduce function the frequency of each word can be calculated by counting the number of occurrence of filenames:

Reduce function:

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        ArrayList<String> list = new ArrayList<String>();

        for (Text value : values) {
```

```

        list.add(value.toString());
    }

    HashSet<String> set = new HashSet<String>(list);
    StringBuilder builder = new StringBuilder();

    String prefix = "";
    for (String value : set) {
        builder.append(prefix);
        prefix = ", ";
        builder.append(value + "#" + Collections.frequency(list, value));
    }

    context.write(key, new Text(builder.toString()));
}
}

```

It gives a (key, value) pair output of the form (word, collection of filenames with frequency):

```

word1 -> doc1.txt#3, doc2.txt#1
word2 -> doc1.txt#1, doc2.txt#2, doc3.txt#1
word3 -> doc1.txt#1
...

```