

Big Data Processing & Analytics - 1st Assignment

A variation of an inverted index

Yijing RONG

A.

In this exercise, the objective is to identify stopwords (words with frequency >4000) for the given corpus. This is similar to the wordcount program and the additional part is to add a condition on the frequency of the words in the reduce task, so that the stopwords are screened out.

I used a code from Stanford given by CS246: Mining Massive Datasets Hadoop tutorial: <http://snap.stanford.edu/class/cs246-data-2014/WordCount.java>.

Here are some changes on the Stanford code:

(1) In order to export a csv file to store the results, the following code is added to the program ->

```
job.getConfiguration().set(
    "mapreduce.output.textoutputformat.separator", ",");
```

(2) In order to avoid repetition, we apply the toLowerCase() method in the corpus in the 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);
        }
    }
}
```

(3) Before writing the (key, value) outputs, a "if" statement is used to add a condition on the frequency of the words (stopwords: frequency > 4000) in the reduce function ->


```

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));
        }
    }
}

```

Once the Map Reduce program is completed, run the code -> export the jar -> operate the following command line in the terminal to run the Hadoop job ->

Hadoop jar Assignment1.jar assignment_1_yijing.StopWords_main input output



MapReduce Job job_1487238424658_0001

Logged in as: drwho

Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
Tools

Job Overview

Job Name: StopWords

User Name: cloudera

Queue: root.cloudera

State: SUCCEEDED

Uberized: false

Submitted: Thu Feb 16 04:59:43 PST 2017

Started: Thu Feb 16 04:59:56 PST 2017

Finished: Thu Feb 16 05:01:25 PST 2017

Elapsed: 1mins, 28sec

Diagnostics:

Average Map Time: 1mins, 6sec

Average Shuffle Time: 9sec

Average Merge Time: 3sec

Average Reduce Time: 4sec

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

The method `job.setNumReduceTasks()` is used to define the number of reducers in the Hadoop driver configuration. Here, we use 10 reducers ->

```
job.setNumReduceTasks(10);
```

Execution time: 2mins, 25sec.



MapReduce Job job_1487238424658_0002

Logged in as: drwho

Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
Tools

Job Overview

Job Name: StopWords_10r_nocombiner

User Name: cloudera

Queue: root.cloudera

State: SUCCEEDED

Uberized: false

Submitted: Thu Feb 16 05:08:12 PST 2017

Started: Thu Feb 16 05:08:21 PST 2017

Finished: Thu Feb 16 05:10:47 PST 2017

Elapsed: 2mins, 25sec

Diagnostics:

Average Map Time: 42sec

Average Shuffle Time: 36sec

Average Merge Time: 5sec


Average Reduce Time: 8sec

(ii) Run the same program again, this time using a Combiner. Report the execution time.

The number of reducers is the same as the former program. To apply a Combiner, we need to add the following code in the Hadoop driver configuration ->

```
job.setCombinerClass(Reduce.class);
```

Execution time: 1mins, 55sec.



MapReduce Job job_1487238424658_0003

Logged in as: drwho

Application

Job

- Overview
- Counters
- Configuration
- Map tasks
- Reduce tasks

Tools

Job Overview	
Job Name:	StopWords_10r_combiner
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 05:28:30 PST 2017
Started:	Thu Feb 16 05:28:40 PST 2017
Finished:	Thu Feb 16 05:30:35 PST 2017
Elapsed:	1mins, 55sec
Diagnostics:	
Average Map Time	33sec
Average Shuffle Time	33sec
Average Merge Time	0sec
Average Reduce Time	4sec

Is there any difference in the execution time, compared to the previous execution? Why?

Comparing the 2 above executives, the program using a combiner is quicker than the previous one. By running the Combiner here, it reduces the amount of data being sent to the reducers, which means it reduces the amount of data that has to be written to disk and transferred over the network in between the Map and Reduce stages of computation, thus improves the performance and efficiency.

(iii) Run the same program again, this time compressing the intermediate results of map (using any codec you wish). Report the execution time.

This time we still use 10 reducers and a combiner, but also we added the following code to compress the intermediate results of map ->

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

Execution time: 2mins, 12sec.



Logged in as: dr.who

Cluster		Application Overview	
About		User:	cloudera
Nodes		Name:	StopWords_withcombiner_10reducers_withcompression
Applications		Application Type:	MAPREDUCE
NEW		Application Tags:	
NEW SAVING		State:	FINISHED
SUBMITTED		FinalStatus:	SUCCEEDED
ACCEPTED		Started:	Thu Feb 16 14:12:32 -0800 2017
RUNNING		Elapsed:	2mins, 12sec
FINISHED		Tracking URL:	History
FAILED		Diagnostics:	
KILLED			
Scheduler			

Is there any difference in the execution, time compared to the previous execution? Why?

The executive time is quicker than the first one but a little bit slower than the second one. Data compression definitely saves a great deal of storage space and speed up the movement of data throughout the cluster. Compressing the intermediate output of the map phase in the Map Reduce processing flow is also useful, as map function output is written to disk and transferred across the network to the reduce tasks. Thus compressing data in Hadoop is more efficient and necessary when the data volumes is huge.

(iv) Run the same program again, this time using 50 reducers. Report the execution time.

The program is all the same as above, except that the number of reducers is changed to 50 in the Hadoop driver configuration ->

```
job.setNumReduceTasks(50);
```

Execution time: 6mins, 50sec.



MapReduce Job job_1487238424658_0005

Logged in as: dr.who

Application		Job Overview	
Job		Job Name:	StopWords_50r_combiner
Overview		User Name:	cloudera
Counters		Queue:	root.cloudera
Configuration		State:	SUCCEEDED
Map tasks		Uberized:	false
Reduce tasks		Submitted:	Thu Feb 16 05:46:07 PST 2017
		Started:	Thu Feb 16 05:46:17 PST 2017
		Finished:	Thu Feb 16 05:53:07 PST 2017
		Elapsed:	6mins, 50sec
		Diagnostics:	
		Average Map Time	32sec
		Average Shuffle Time	37sec
		Average Merge Time	0sec
		Average Reduce Time	3sec

Is there any difference in the execution time, compared to the previous execution? Why?

From the execution result, we can infer that increasing the number of reducers increases the time for running reduce phase. If the output data from the mapper are not that big, increase the number of reducers could hit the performance, because results need to be transferred to different reducers, the operations are increased as

it creates more files as each reducer create its own file. Besides, data need to be split across the entire number of reducers which require more network transfer time and parsing time. The advice is to choose an appropriate number of reducers.

Execution Time Summary

Hadoop Job	Execution Time
StopWords_nocombiner_10reducers	2mins, 25sec
StopWords_withcombiner_10reducers	1mins, 55sec
StopWords_withcombiner_10reducers_withcompression	2mins, 12sec
StopWords_withcombiner_50reducers_withcompression	6mins, 50sec

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

For this part, we start the Map Reduce from scratch. For each word found in corpus, the program needs to output a (key, value) pair of the form (word, collection of filenames). Consequently, we set the output format of both keys and values as Text using the following code ->

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

The map function will get the filename in which it appears for each word.

And we remove the stop words as following ->

```
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/InvertedIndex/output/StopWords_main.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);
}
}

```

It gives a (key, value) pair output of the form (word, filename):

```

word1, doc1.txt
word1, doc1.txt
word1, doc1.txt
word1, doc2.txt
word2, doc1.txt
word2, doc2.txt
word2, doc2.txt
word2, doc3.txt
word3, doc1.txt
...

```

Then, the reducer stores all the filenames for each word in a HashSet (a collection that cannot accept duplicates).

```

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()));
    }
}

```

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

word1 -> doc1.txt, doc2.txt

word2 -> doc1.txt, doc2.txt, doc3.txt

word3 -> doc1.txt

...

(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.

In the driver, we add a custom counter ->

```

public class InvertedIndex_unique extends Configured implements Tool {

    public static enum CUSTOM_COUNTER {
        UNIQUE_WORDS,
    };
}

```

In this case, the map function is the same as the previous one. But the reduce function was modified. A "if" statement is applied in order to add a condition to select only the

unique words, that is to say the words for which the collection of filenames is of length 1. Within the reduce function, the UNIQUE_WORDS counter is also added ->

```
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()));  
  
        }  
    }  
}
```

Then, after running the Hadoop job, the counter value is shown in the output, representing the number of unique words: UNIQUE_WORDS=68476

(D) Extend the inverted index of (b), in order to keep the frequency of each word for each document.

We extend the previous code of question b) that we applied before. In this case, the map function is the same as the two maps functions used before. The intuition here is that the reducer should be able to store the filenames in a collection but this time without removing the duplicate filenames, in order to get something like this:

word1 -> doc1.txt, doc1.txt, doc1.txt, doc2.txt

word2 -> doc1.txt, doc2.txt, doc2.txt, doc3.txt

word3 -> doc1.txt

...

In order to keep the duplicates because then we are able with the reducer to count the occurrences of each filename in the value part of the mapper output, with the

Collections.frequency(array, object) method in Java.

By implementing this new method, our new reduce function would be now ->

```
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

...

(D) Conclusion

- The program using a combiner is quicker than the one without a combiner. By running the Combiner, it reduces the amount of data being sent to the reducers, which means it reduces the amount of data that has to be written to disk and transferred over the network in between the Map and Reduce stages of computation, thus improves the performance and efficiency.
- Data compression saves a great deal of storage space and speed up the movement of data throughout the cluster. Compressing the intermediate output of the map phase in the Map Reduce processing flow is also useful, as map function output is

written to disk and transferred across the network to the reduce tasks. Thus compressing data in Hadoop is more efficient and necessary when the data volumes is huge.

- Increasing the number of reducers increases the time for running reduce phase. If the output data from the mapper are not that big, increase the number of reducers could hit the performance, because results need to be transferred to different reducers, the operations are increased as it creates more files as each reducer create its own file. Besides, data need to be split across the entire number of reducers which require more network transfer time and parsing time. The advice is to choose an appropriate number of reducers.