Big Data Processing 1st Assignment

Shi Li

A) Pre-processing

- Creating new package in terminal

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

- Import files

```
hadoop fs -mkdir input
hadoop fs -put workspace/Lishi/input/pg100.txt input
hadoop fs -put workspace/Lishi/input/pg3200.txt input
hadoop fs -put workspace/Lishi/output/pg31100.txt input
hadoop fs -ls input
```

- Reference

We used the skeleton of code of Standford University From:
http://snap.stanford.edu/class/cs246-data-2014/WordCount.java
Mining Massive Datasets Hadoop tutorial.

- Some changes are shown below

a) A CSV output is designed so we added the following code in the Hadoop Driver:

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

b) In the map function, we added the toLowerCase() method in the words iteration in order to remove duplicates:

```
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);
        }
    }
}
```

c) In the reduce function, before writing the outputs, a if statement that adds a condition on the number of occurrences of words

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

- Running the Hadoop job

Using the follwing command in the terminal:

`hadoop jar Lishi.jar.Lishi.mdp.StopWords input output`

- Result Saving

After run the code, we wrote commend `input/output` in Run-Configuration window in order to extracting output.

The result was saved in `stopwords.csv`.

B) Questions

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()`. In our case, we use `job.setNumReduceTasks(10)`.

Execution time :

2mins, 20sec.

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?

Added following code:

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

Execution time:

1min, 57sec

Why:

We can see the execution time is shorter than previous one, the reason is because Without Combiner, the job performs aggregation of some sorts, and the reduce input group counter is smaller than the reduce input record counter. Also, the job performs a large shuffler. In opposite, the Combiner can reduce the amount of data that has to be written to disk in order to reduce execution time.

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?

Added following code:

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

Execution Time:

1min, 55sec

Why:

The execution time is a little bit quicker than previous execution. This is because intermediate compressing of map can reduce the amount of data to be stored to disk and written over the network.

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?

Added following code:

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

Execution Time:

6mins, 18sec

Why:

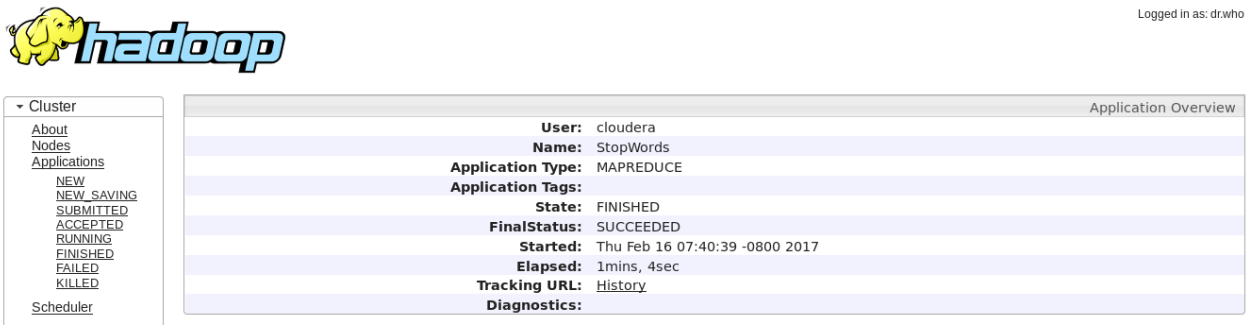
The execution time is much larger than previous one. This is because 50 reducers cause the disk head to move too much. More CPUs and more disks.

Execution Time Summary Table

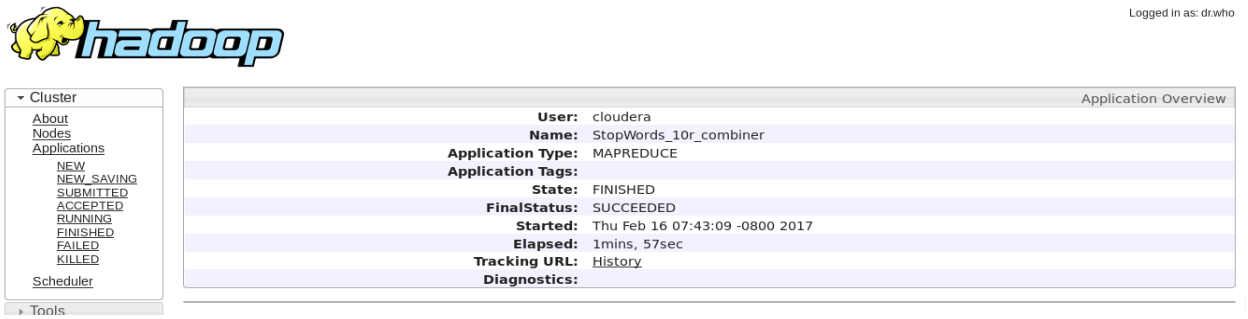
Hadoop Job	Execution Time
withoutcombiner_10reducers	2mins, 20sec.
withcombiner_10reducers	1min, 57sec.
withcombiner_10reducers_compression	1min, 55sec.
withcombiner_50reducers	6mins, 18sec.

Hadoop Screenshots

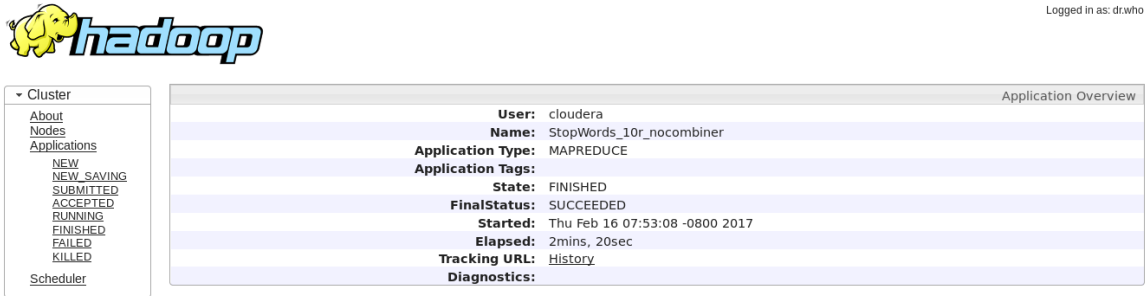
a) Stop-words count



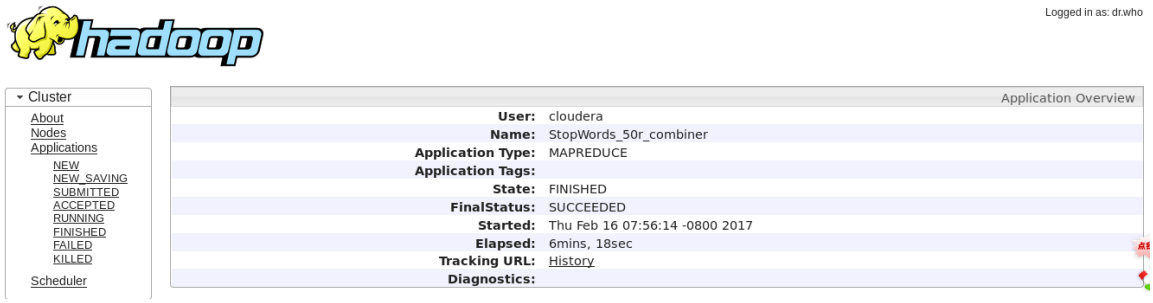
b)10 reducer with combiner



c)10 reducers without combiner



d) 50 reducers with combiner



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

we set in the Hadoop driver the output format of both keys and values as Text:

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

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

Our map function will get, for each word, the filename in which it appears. Also, it will have to remove the stopwords.

Note : We changed the format of the `stopwords.csv` output file of the previous MapReduce program to a more convenient format, `stopwords.txt`.

```
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/Lishi/output/StopWords.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);

    }
}
```

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
word3, doc1.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()));

    }
}
```

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.

We add a custom counter:

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

In this case, the map function is the same as before. But the reduce function will be different. We add a if statement in order to conditionally select only the unique words, that is to say the words for which the collection of filenames is of length 1. We also add within the reduce function, our `UNIQUE_WORDS` 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).incremen
t(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, we can see our counter value in the output after running, representing the number of unique words:

```
centralemdp.Lishi.lishi_unique$CUSTOM_COUNTER
    UNIQUE_WORDS=68476
```

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:

The reducer stored the filenames in a collection but with duplicates filenames, this is because we can count the occurences of each filename in the value part of the mapper output.in order to get some output like this:

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

We implement with the `Collections.frequency(array, object)` method . Reduce function is:

```
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()));

    }
}
```

The 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
...
```

C) Conclusion

MapReduce is a good way to splits massive data into independent chunks which are processed by the map tasks in a completely parallel manner. Hadoop sends the map and reduce task to the appropriate serves in the cluster. We can see compressed and combined data run shorter time than uncompressed and uncombined. Also too many reducers can cause heavy traffic and increase running time.