HY-562 Advanced Topics in Databases

1st Assignment

Andreas Michelakis – Computer Science Department - University of Crete

Andreas Michelakis
20/10/2017

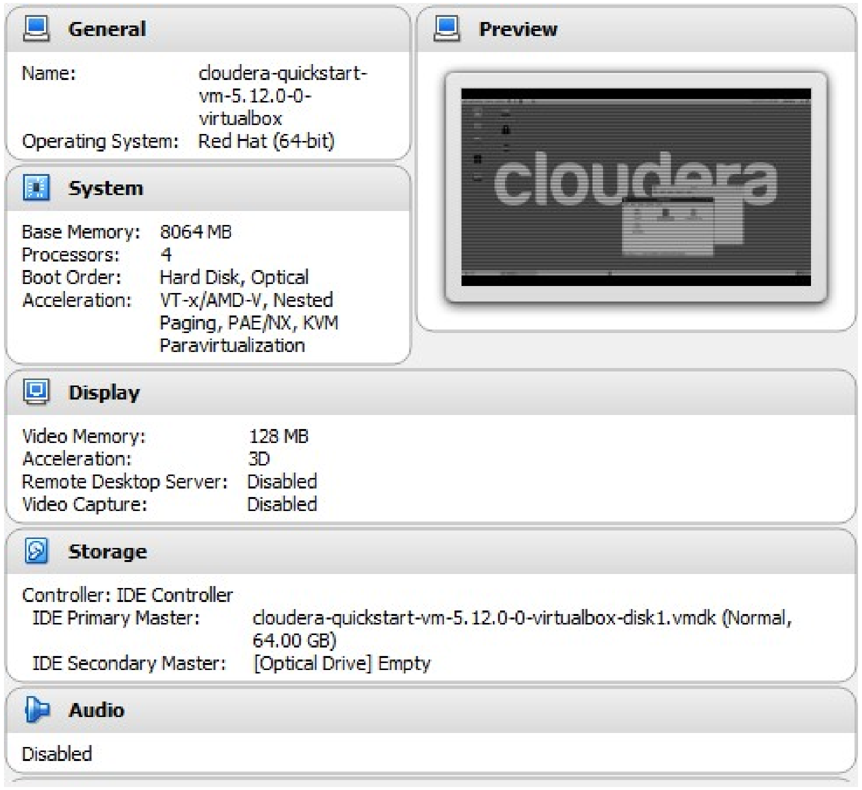
HY-562 Advanced Topics in Databases
1st Assignment
Andreas Michelakis (ID 1043)

Table of Contents

Setup

Oracle Vm VirtualBox

As instructed by the assignment description I chose to run an appliance of Cloudera virtual machine on Oracle Vm Virtual box. Specifically, I set up the virtual machine with the following settings:



General

Name:	cloudera-quickstart-vm-5.12.0-0-virtualbox
Operating System:	Red Hat (64-bit)

System

Base Memory:	8064 MB
Processors:	4
Boot Order:	Hard Disk, Optical
Acceleration:	VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

Display

Video Memory:	128 MB
Acceleration:	3D
Remote Desktop Server:	Disabled
Video Capture:	Disabled

Storage

Controller: IDE Controller

IDE Primary Master:	cloudera-quickstart-vm-5.12.0-0-virtualbox-disk1.vmdk (Normal, 64.00 GB)
IDE Secondary Master:	[Optical Drive] Empty

Audio

Disabled

Figure 1 (as shown in Oracle Vm Virtual Box)

Exercise 1

For exercise 1 I implemented two jobs, with a different mapper and reducer class for each job.

The first job (job) uses the wordMap.class as a mapper class and the ov4kReduce.class as a combiner and reducer class.

```
job.setMapperClass(wordMap.class);

job.setCombinerClass(ov4kReduce.class);

job.setReducerClass(ov4kReduce.class);
```

The second job (job1) uses the WordMapper2.class as a mapper class and the SumeReducer2.class as a combiner and reducer class. A sort comparator class (IntComparator.class) is also used to sort the result by frequency.

```
job1.setMapperClass(WordMapper2.class);

job1.setReducerClass(SumReducer2.class);

job1.setCombinerClass(SumReducer2.class);

job1.setSortComparatorClass(IntComparator.class);
```

In order to control job order I used the library org.apache.hadoop.mapreduce.lib.jobcontrol . This way I can set different configuration for the two jobs and also easily determine the order of execution.

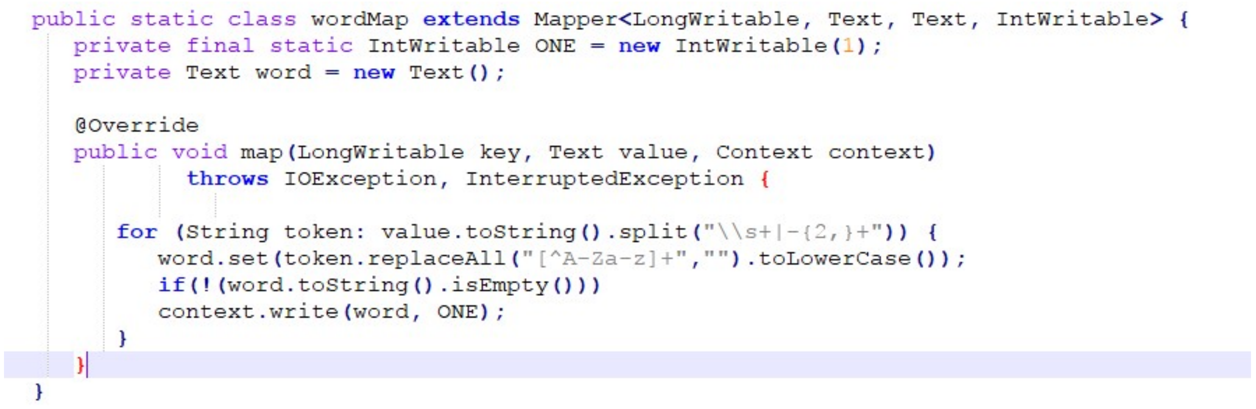
```
ControlledJob controlledJob = new ControlledJob(conf1);

controlledJob.setJob(job);

jobControl.addJob(controlledJob);

// make job1 dependent on job && add the job to the job control

controlledJob1.addDependingJob(controlledJob);

jobControl.addJob(controlledJob1);
```

The output of the second job is one part-r* file which contains all stopwords found in the given document corpus in Comma Separated Value format. I can use the following set of commands to create a stopwords.csv file

```
hadoop fs  –cat [path to output]/part* | hadoop fs -put – [path to desired hdfs directory]stopwords.csv
```

First Job “Find StopWords”

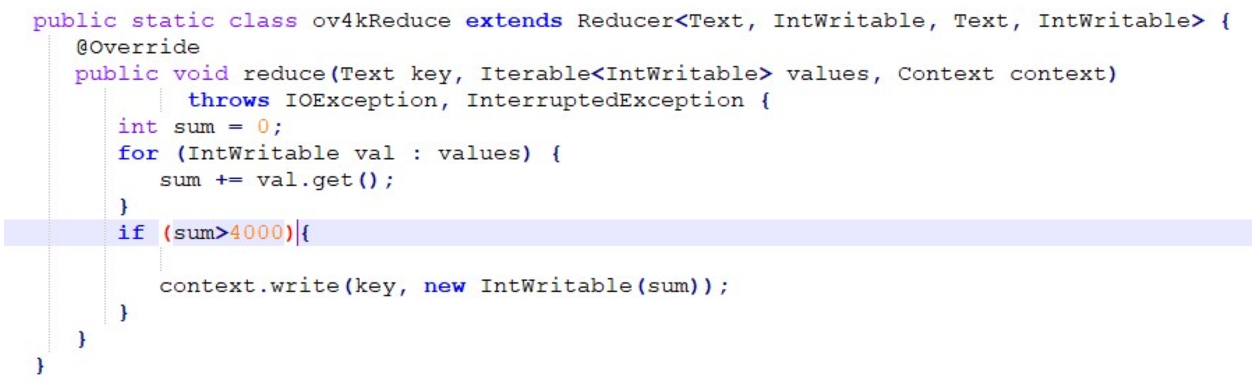
wordMap.class



```
public static class wordMap 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+|-{2,}+")) {
         word.set(token.replaceAll("[^A-Za-z]+","").toLowerCase());
         if(!(word.toString().isEmpty()))
         context.write(word, ONE);
      }
   }
}
```

wordMap.class extends the default Mapper class, iterates the document by splitting at white space, replaces all numerical values with blank “ ” and writes key,value pairs that consist of lowercase words that are not blanks and an IntWritable with a 1 value. contex.write(word,ONE);

ov4kReducer.class

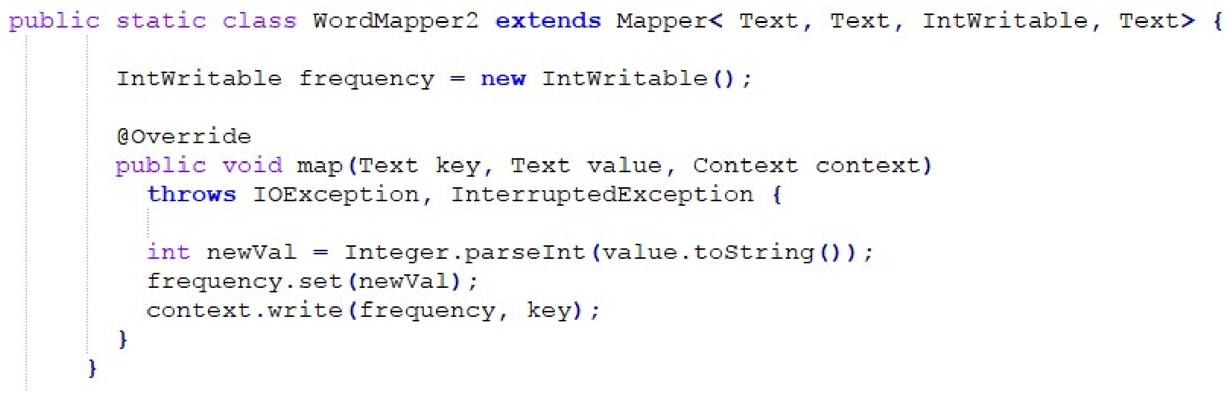


```
public static class ov4kReduce extends Reducer<Text, IntWritable, Text, IntWritable> {
   @Override
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
           throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
         sum += val.get();
      }
      if (sum>4000){

         context.write(key, new IntWritable(sum));
      }
   }
}
```

Over4kReducer.class extends the default Reducer class. Gathers the sum of values for each key using a for loop and if the final summary is over four thousand writes to context as ouput the key(word) and the sum value.

Second Job “Invert key,value pairs/Sort by frequency”

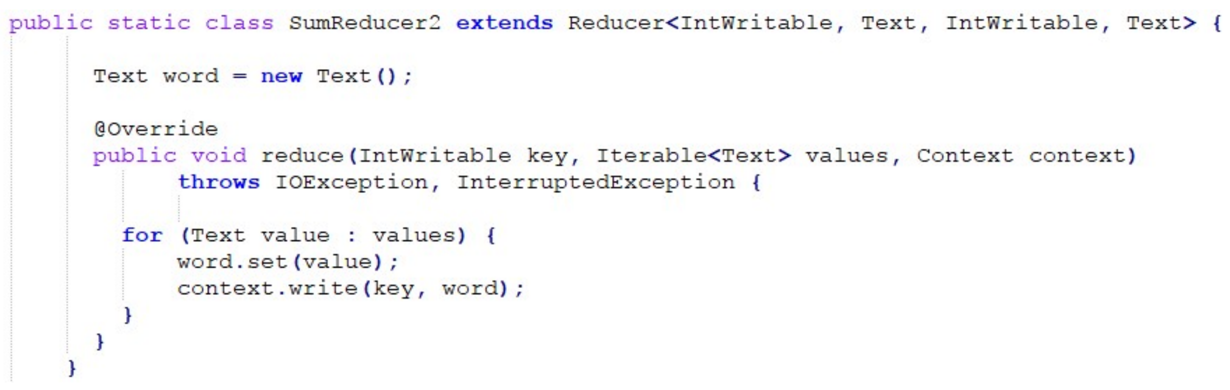
WordMapper2.class



```java
public static class WordMapper2 extends Mapper< Text, Text, IntWritable, Text> {

        IntWritable frequency = new IntWritable();

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

          int newVal = Integer.parseInt(value.toString());
          frequency.set(newVal);
          context.write(frequency, key);
        }
    }
```

Takes as input the outcome key, value pairs of the first job and transforms them to key,value pairs where key is the first job value as string and the value is the first job key.

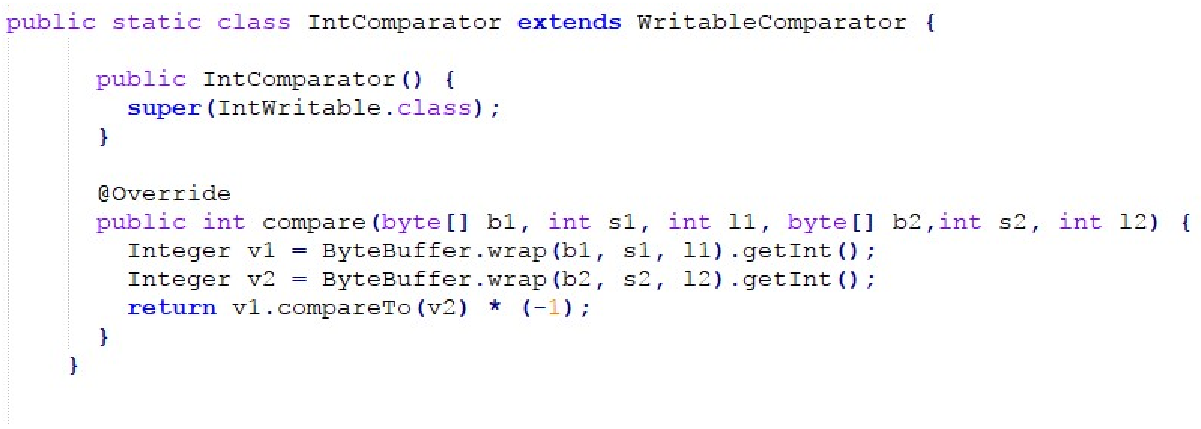
SumReducer2.class



```java
public static class SumReducer2 extends Reducer<IntWritable, Text, IntWritable, Text> {

        Text word = new Text();

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

          for (Text value : values) {
                word.set(value);
                context.write(key, word);
          }
        }
    }
```

Reduces the output of the mapper to one file.

IntComparator.class



```
public static class IntComparator extends WritableComparator {

    public IntComparator() {
      super(IntWritable.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2,int s2, int l2) {
      Integer v1 = ByteBuffer.wrap(b1, s1, l1).getInt();
      Integer v2 = ByteBuffer.wrap(b2, s2, l2).getInt();
      return v1.compareTo(v2) * (-1);
    }
  }
```

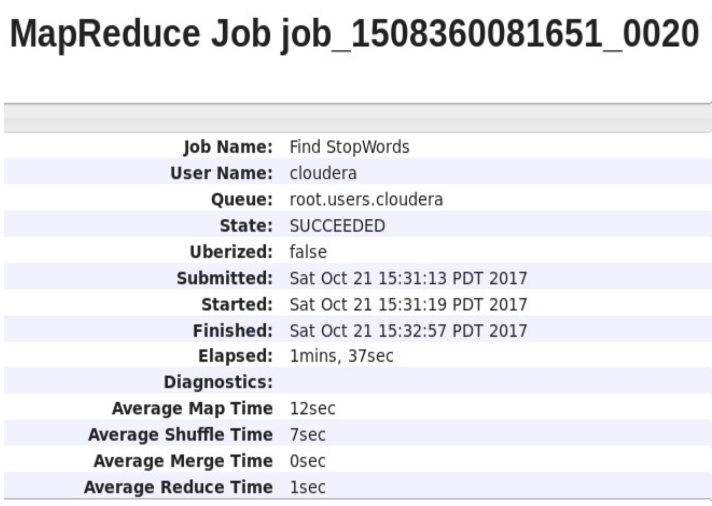
IntComparator.class extends the class WritableComparator and overrides its method compare. It compares two integer values v1 and v2 and outputs the higher one. This way keys are sorted from higher to lower before being passed to reducer.

Exercise2

Exercise2a

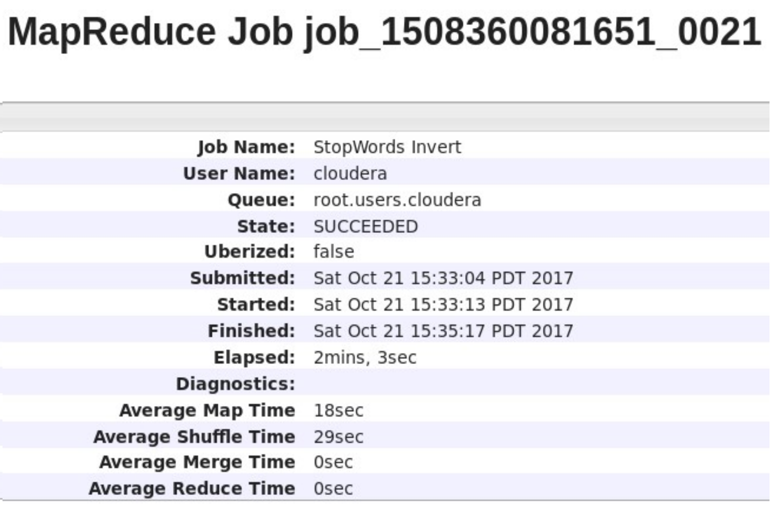
Applying different settings to the set of jobs of Exercise 1, the execution time is measured:

Setting 1 (Total time elapsed: 3mins, 40sec) | 10 Reducers , No Combiner |



MapReduce Job job_1508360081651_0020

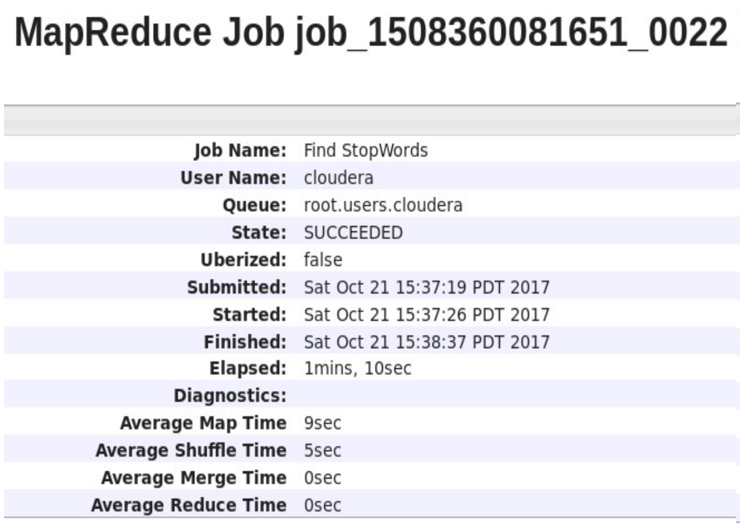
Job Name:	Find StopWords
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:31:13 PDT 2017
Started:	Sat Oct 21 15:31:19 PDT 2017
Finished:	Sat Oct 21 15:32:57 PDT 2017
Elapsed:	1mins, 37sec
Diagnostics:	
Average Map Time	12sec
Average Shuffle Time	7sec
Average Merge Time	0sec
Average Reduce Time	1sec



MapReduce Job job_1508360081651_0021

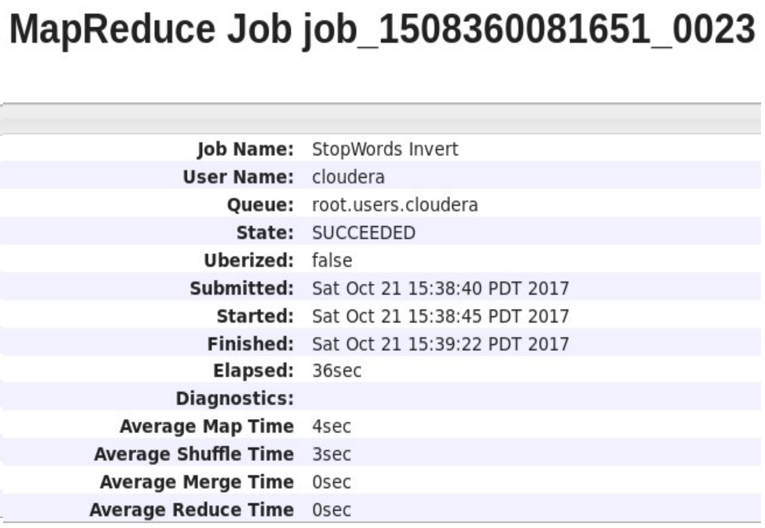
Job Name:	StopWords Invert
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:33:04 PDT 2017
Started:	Sat Oct 21 15:33:13 PDT 2017
Finished:	Sat Oct 21 15:35:17 PDT 2017
Elapsed:	2mins, 3sec
Diagnostics:	
Average Map Time	18sec
Average Shuffle Time	29sec
Average Merge Time	0sec
Average Reduce Time	0sec

Setting 2 (Total time elapsed: 1mins, 46sec) | 10 Reducers , With Combiner |



MapReduce Job job_1508360081651_0022

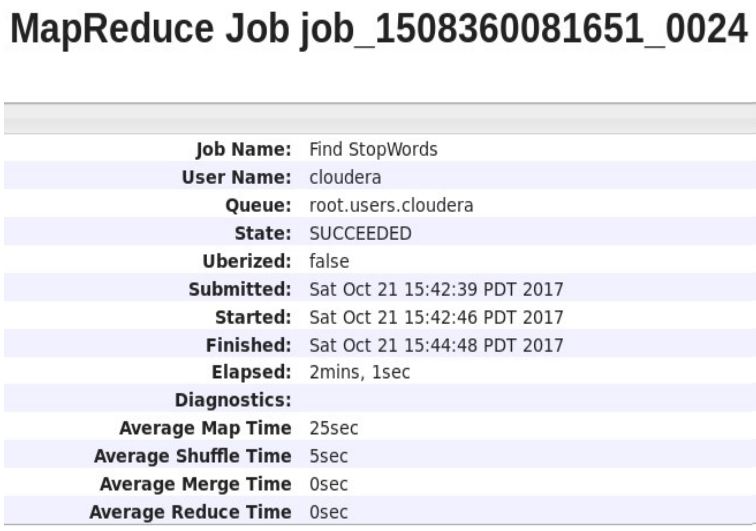
Job Name:	Find StopWords
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:37:19 PDT 2017
Started:	Sat Oct 21 15:37:26 PDT 2017
Finished:	Sat Oct 21 15:38:37 PDT 2017
Elapsed:	1mins, 10sec
Diagnostics:	
Average Map Time	9sec
Average Shuffle Time	5sec
Average Merge Time	0sec
Average Reduce Time	0sec



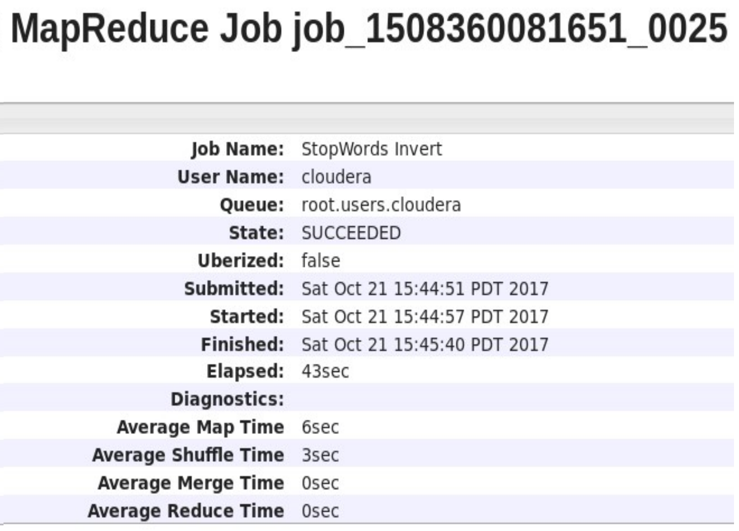
MapReduce Job job_1508360081651_0023

Job Name:	StopWords Invert
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:38:40 PDT 2017
Started:	Sat Oct 21 15:38:45 PDT 2017
Finished:	Sat Oct 21 15:39:22 PDT 2017
Elapsed:	36sec
Diagnostics:	
Average Map Time	4sec
Average Shuffle Time	3sec
Average Merge Time	0sec
Average Reduce Time	0sec

Setting 3 (Total time elapsed: 2mins, 44sec) | 10 Reducers , With Combiner, With map compression |

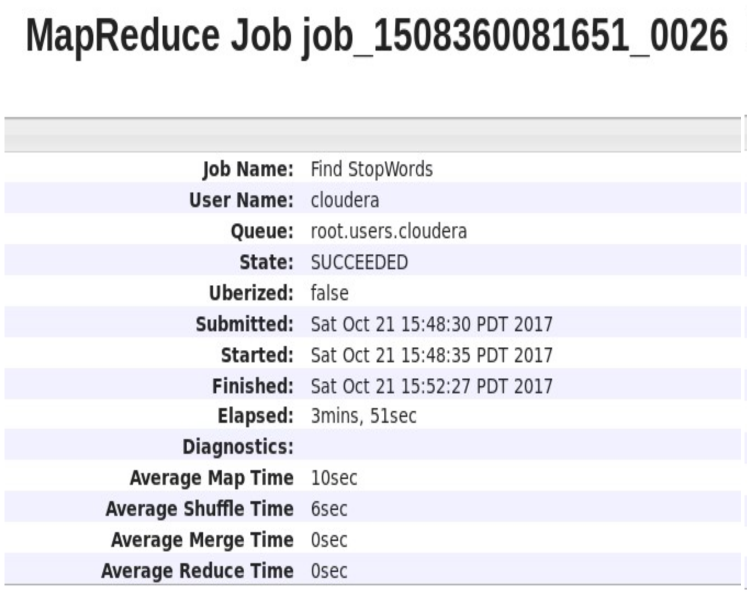

MapReduce Job job_1508360081651_0024

Job Name:	Find StopWords
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:42:39 PDT 2017
Started:	Sat Oct 21 15:42:46 PDT 2017
Finished:	Sat Oct 21 15:44:48 PDT 2017
Elapsed:	2mins, 1sec
Diagnostics:	
Average Map Time	25sec
Average Shuffle Time	5sec
Average Merge Time	0sec
Average Reduce Time	0sec

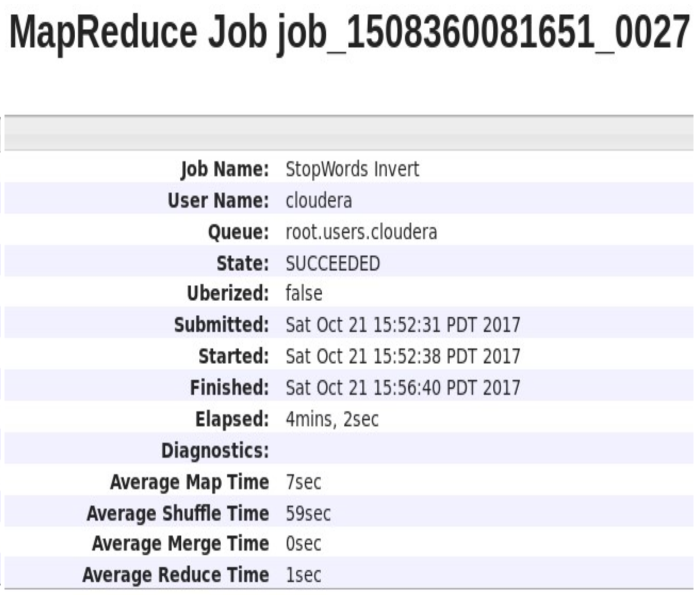

MapReduce Job job_1508360081651_0025

Job Name:	StopWords Invert
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:44:51 PDT 2017
Started:	Sat Oct 21 15:44:57 PDT 2017
Finished:	Sat Oct 21 15:45:40 PDT 2017
Elapsed:	43sec
Diagnostics:	
Average Map Time	6sec
Average Shuffle Time	3sec
Average Merge Time	0sec
Average Reduce Time	0sec

Setting 4 (Total time elapsed: 7mins, 53sec) | 50 Reducers , With Combiner, With map compression |


MapReduce Job job_1508360081651_0026

Job Name:	Find StopWords
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:48:30 PDT 2017
Started:	Sat Oct 21 15:48:35 PDT 2017
Finished:	Sat Oct 21 15:52:27 PDT 2017
Elapsed:	3mins, 51sec
Diagnostics:	
Average Map Time	10sec
Average Shuffle Time	6sec
Average Merge Time	0sec
Average Reduce Time	0sec


MapReduce Job job_1508360081651_0027

Job Name:	StopWords Invert
User Name:	cloudera
Queue:	root.users.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Oct 21 15:52:31 PDT 2017
Started:	Sat Oct 21 15:52:38 PDT 2017
Finished:	Sat Oct 21 15:56:40 PDT 2017
Elapsed:	4mins, 2sec
Diagnostics:	
Average Map Time	7sec
Average Shuffle Time	59sec
Average Merge Time	0sec
Average Reduce Time	1sec

Results

Setting 1 -> Setting 2

Transitioning from Setting 1 to Setting 2, execution time is **decreased by a total of 1 minute 54 seconds**. The use of a combiner for grouping the mapped values before passing it to the reducers greatly minimizes the data size that will be passed therefore reducing the execution time in each phase.

Setting 2 – Setting 3

Transitioning from Setting 2 to Setting 3, execution time is **increased by a total of 58 seconds**. Although compressing the intermediate map output should minimize the size of data in need for transfer to reducers and thus decreasing the total execution time, in this case it actually increases the execution time. This could be happening as the jobs are running in pseudo-distributed mode.

Setting 3 – Setting 4

Using 50 reducers has an incredibly negative outcome on execution time. The execution time is **increased by 5 minutes 9 seconds**. This happens as using 50 reducers causes an overhead to the managing framework.

Exercise 2b

1.In this exercise a implemented a single job with a mapper and a reducer class. The purpose of this exercise is to create a simple inverted index of the document corpus displaying all words, excluding stop words, as keys and the documents they appear in as values.

2.As a second requirement the job counts the number of unique words that exist in each document. That is accomplished by using a custom enumeration type with predefined values. The values represent the document corpus.

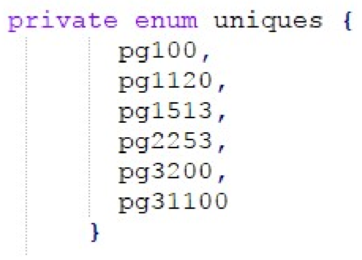
The job takes four arguments/directories:

[0] – document corpus input [1] – output of inverted index [2] – stopwords.csv input directory
[3] -output of "unique words count" file

Eg. [cloudera@quickstart ~]$ hadoop jar exercise2b.jar **docs/pg*.txt ex2/ ex1/stopwords.csv ex2/uniquewords.txt**

2. Unique words

Private enum uniques



```
private enum uniques {
        pg100,
        pg1120,
        pg1513,
        pg2253,
        pg3200,
        pg31100
    }
```

It is a custom enumeration type with predefined values corresponding to the document corpus filenames.

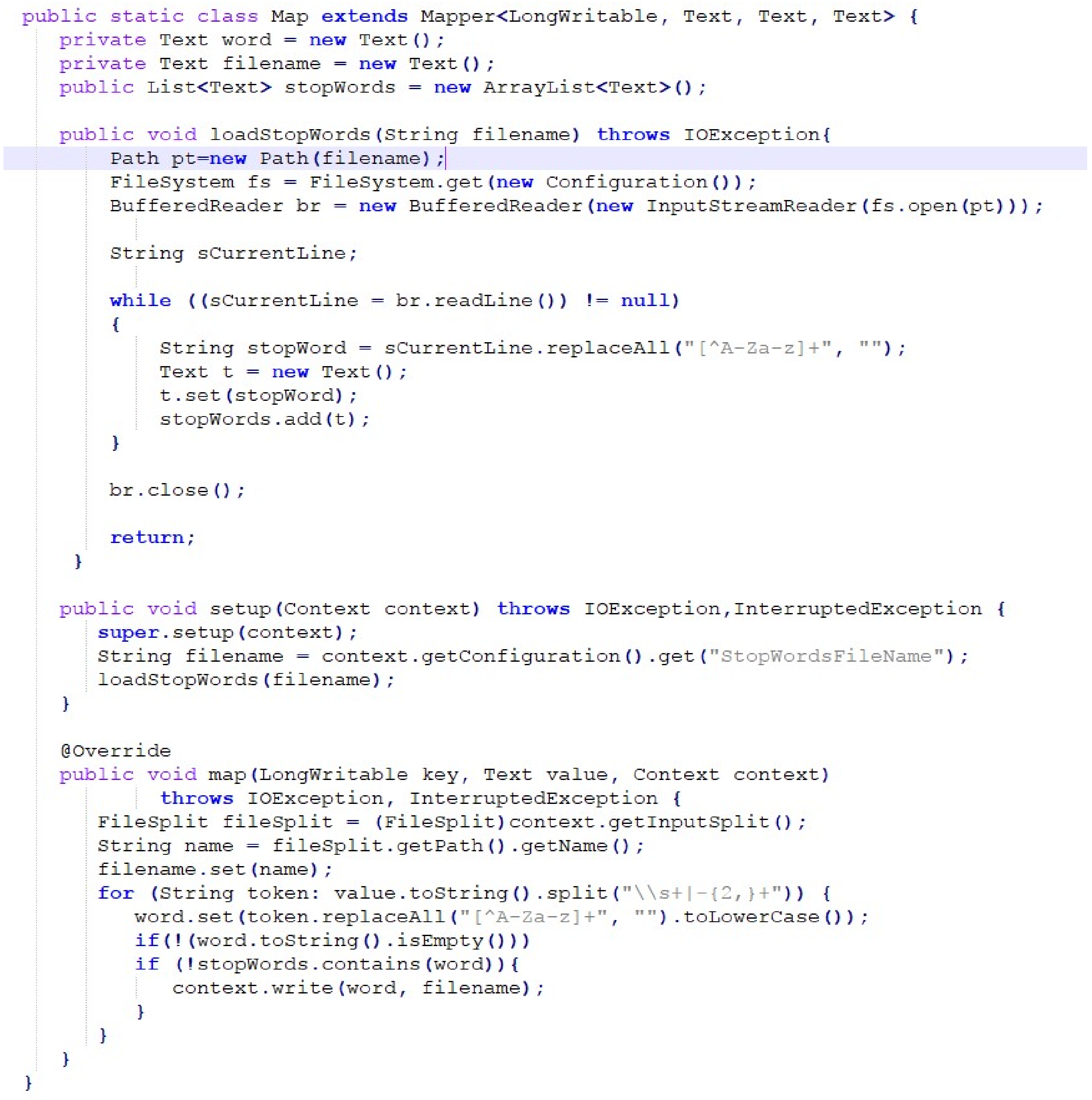
PrintWriter – unique words

```
Integer i;
PrintWriter writer = new PrintWriter(args[3], "UTF-8");

i = (int) job.getCounters().findCounter(uniques.pg100).getValue();
writer.println("pg100: "+i.toString()+"\n");
i = (int) job.getCounters().findCounter(uniques.pg1120).getValue();
writer.println("pg1120: "+i.toString()+"\n");
i = (int) job.getCounters().findCounter(uniques.pg1513).getValue();
writer.println("pg1513: "+i.toString()+"\n");
i = (int) job.getCounters().findCounter(uniques.pg2253).getValue();
writer.println("pg2253: "+i.toString()+"\n");
i = (int) job.getCounters().findCounter(uniques.pg3200).getValue();
writer.println("pg3200: "+i.toString()+"\n");
i = (int) job.getCounters().findCounter(uniques.pg31100).getValue();
writer.println("pg31100: "+i.toString()+"\n");
writer.flush();
writer.close();
```

An object of the class PrintWriter is used to write in a user defined destination file the amount of unique words per document that are found. For each case a counter is initialized that gets the value that is incremented during reduce phase described later. Results are also printed on the terminal.

1.Inverted Index – Job

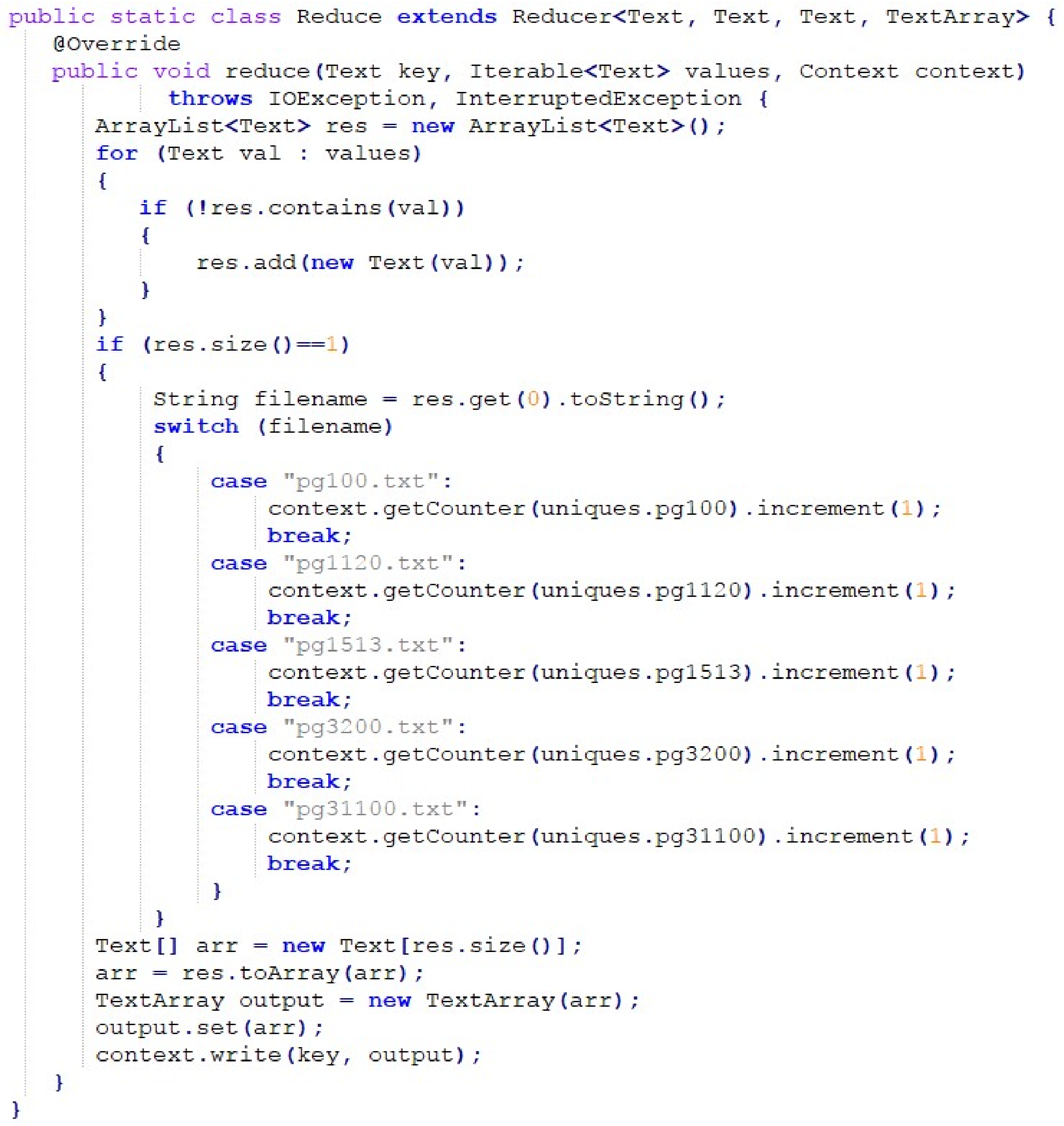
Map.class



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

    public void loadStopWords(String filename) throws IOException{
        Path pt=new Path(filename);
        FileSystem fs = FileSystem.get(new Configuration());
        BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));

        String sCurrentLine;

        while ((sCurrentLine = br.readLine()) != null)
        {
            String stopWord = sCurrentLine.replaceAll("[^A-Za-z]+", "");
            Text t = new Text();
            t.set(stopWord);
            stopWords.add(t);
        }

        br.close();

        return;
     }

    public void setup(Context context) throws IOException,InterruptedException {
       super.setup(context);
       String filename = context.getConfiguration().get("StopWordsFileName");
       loadStopWords(filename);
    }

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
       FileSplit fileSplit = (FileSplit)context.getInputSplit();
       String name = fileSplit.getPath().getName();
       filename.set(name);
       for (String token: value.toString().split("\\s+|-{2,}+")) {
          word.set(token.replaceAll("[^A-Za-z]+", "").toLowerCase());
          if(!(word.toString().isEmpty()))
          if (!stopWords.contains(word)){
             context.write(word, filename);
          }
       }
    }
 }
```

Map.class includes a loadStopWords function that loads all stop words from the stopwords.csv file and adds them to a text ArrayList . Then in the map function this ArrayList is used to compare each word of the document corpus(filtered similarly to exercise 1) to the stop words. The output is key,value pairs where keys are the words and values are the documents where they appear in.

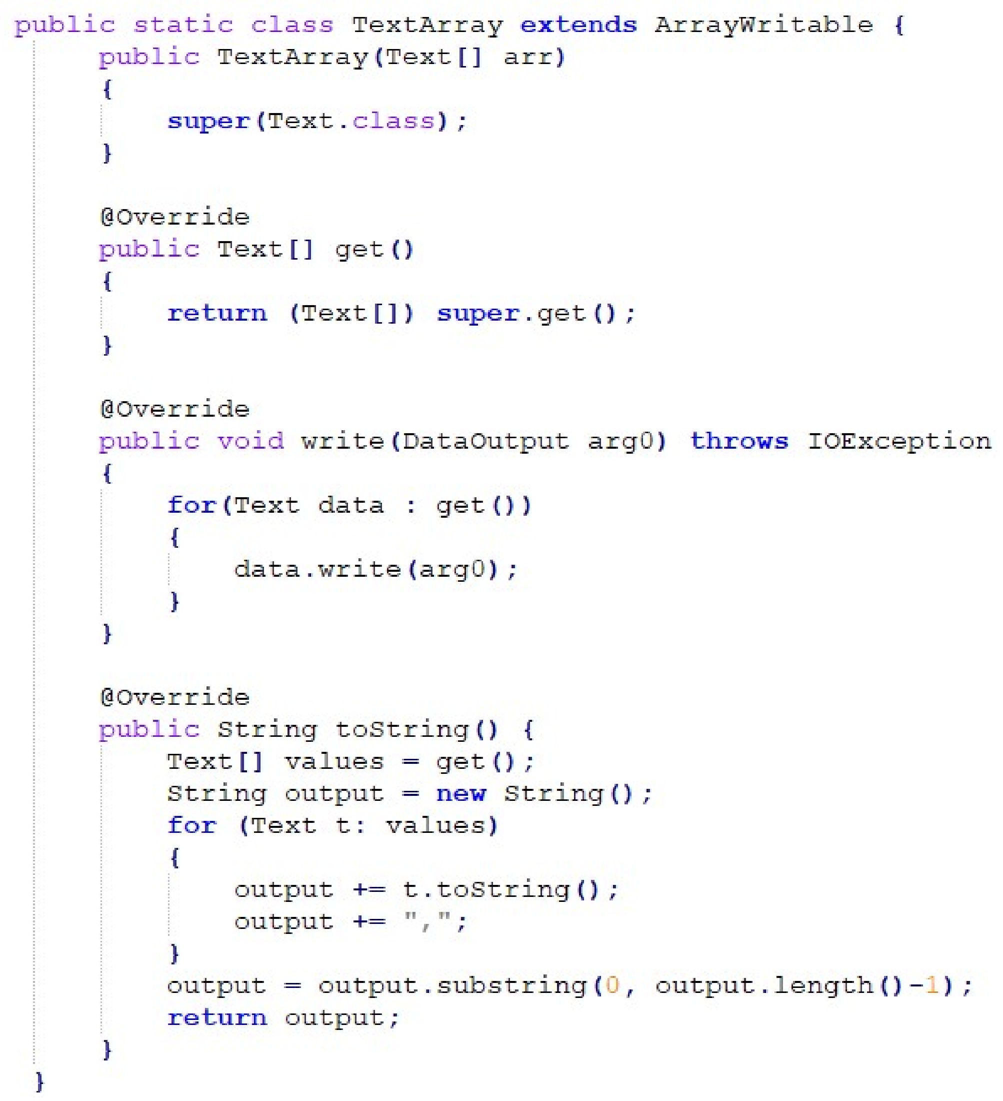
Reducer.class



```
public static class Reduce extends Reducer<Text, Text, Text, TextArray> {
   @Override
   public void reduce(Text key, Iterable<Text> values, Context context)
           throws IOException, InterruptedException {
      ArrayList<Text> res = new ArrayList<Text>();
      for (Text val : values)
      {
         if (!res.contains(val))
         {
             res.add(new Text(val));
         }
      }
      if (res.size()==1)
      {
          String filename = res.get(0).toString();
          switch (filename)
          {
              case "pg100.txt":
                  context.getCounter(uniques.pg100).increment(1);
                  break;
              case "pg1120.txt":
                  context.getCounter(uniques.pg1120).increment(1);
                  break;
              case "pg1513.txt":
                  context.getCounter(uniques.pg1513).increment(1);
                  break;
              case "pg3200.txt":
                  context.getCounter(uniques.pg3200).increment(1);
                  break;
              case "pg31100.txt":
                  context.getCounter(uniques.pg31100).increment(1);
                  break;
              }
          }
      Text[] arr = new Text[res.size()];
      arr = res.toArray(arr);
      TextArray output = new TextArray(arr);
      output.set(arr);
      context.write(key, output);
   }
}
```

The reducer class "counts" the unique words in each document by incrementing the value of data type enum uniques by 1 every time a case is matched. The cases consider only words that appear in a single document. The reducers output is a key,value pair where key is the word and value is an Array of texts(output) which contain the names of documents that the given word appears in.

TextArray.class



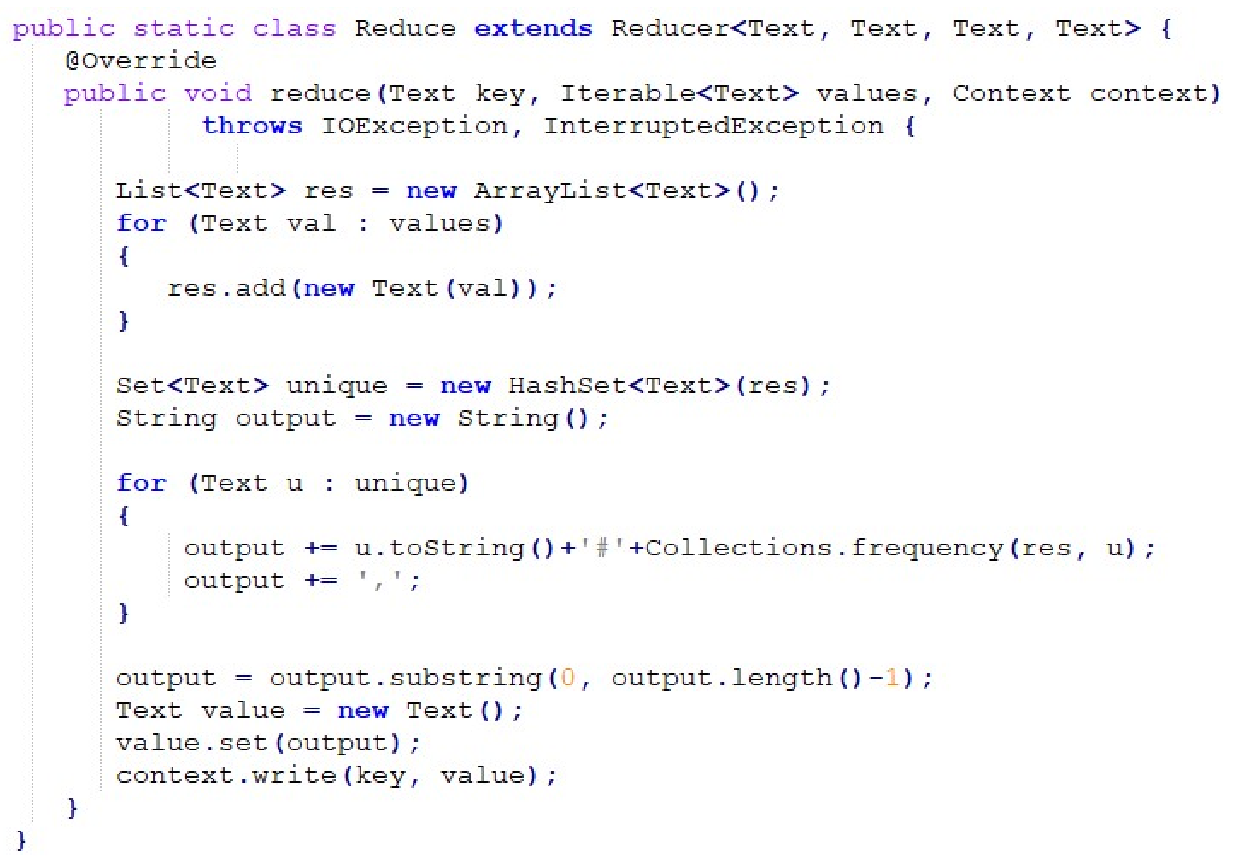
```java
public static class TextArray extends ArrayWritable {
    public TextArray(Text[] arr)
    {
        super(Text.class);
    }

    @Override
    public Text[] get()
    {
        return (Text[]) super.get();
    }

    @Override
    public void write(DataOutput arg0) throws IOException
    {
        for(Text data : get())
        {
            data.write(arg0);
        }
    }

    @Override
    public String toString() {
        Text[] values = get();
        String output = new String();
        for (Text t: values)
        {
            output += t.toString();
            output += ",";
        }
        output = output.substring(0, output.length()-1);
        return output;
    }
}
```

TextArray.class is an extension to ArrayWritable class. The toString method is overridden to suit the needs of reducer class, inputing a “,” between the document names.

Exercise 3

In Exercise 3 a job is implemented that is an extension to the inverted table of Exercise 2. The job includes a mapper, a reducer/combiner class. Purpose is to find the number of times a word appears in each document. The map.class is identical to the one used in Exercise 2 and will not be described below.

Reducer.class



```
public static class Reduce extends Reducer<Text, Text, Text, Text> {
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        List<Text> res = new ArrayList<Text>();
        for (Text val : values)
        {
            res.add(new Text(val));
        }

        Set<Text> unique = new HashSet<Text>(res);
        String output = new String();

        for (Text u : unique)
        {
            output += u.toString()+'#'+Collections.frequency(res, u);
            output += ',';
        }

        output = output.substring(0, output.length()-1);
        Text value = new Text();
        value.set(output);
        context.write(key, value);
    }
}
```

The reducer takes the output of the mapper described in Exercise 2 and makes a text Hashset consisting of the key (word) and value (the names of documents for a given key-word). Then for every element of the hash set add the element in string format to a string output plus a '#' character plus the frequency of the element appearing in the ArrayList res. The output is a key(word) and a value (document#times a word appears,)