Yao Song (301266041)
Zhenjian Lu (301172144)

CMPT-741 Project Report

1. Description

In this project, we implemented the SON (Savasere, Omiecinski, and Navathe) algorithm to count frequent itemsets using MapReduce.

The task of counting frequent itemsets is to find sets of items that appear frequently among a collection of "market baskets". As the dataset (number of baskets) becomes large, the task becomes challenging for commodity computer with limited resources due to the huge number of possible itemsets.

The idea of the SON algorithm is to divide a large dataset into many small subsets that can be processed efficiently on a machine with small main memory. In its first phase, it generates a set of *local frequent* itemsets for each subset, and combines them to form the set of candidate itemsets. In the second phase, it counts the supports for each candidate among each subset and sums the results to obtain their actual supports among the baskets and hence find the *global frequent* itemsets.

MapReduce is a programming model for parallel computation by breaking a large computational task into small tasks and distributing them to a set of nodes in the cluster. Each task is performed individually (Map). Their outputs are combined at the end to compute the final result (Reduce).

In this project, we implemented the two-phase SON algorithm as a MapReduce-MapReduce process to take advantage of the parallel feature offered by the MapReduce model in order to gain performance improvement.

2. Implementation

2.1 Overview

We used Java and Hadoop, which is an open source MapReduce framework. The program can be divided into the following four parts in high-level view:

1. Processing Input
    - Copy input file to Hadoop Distributed File System (HDFS), gather input information and configure Hadoop to split the input file into a set of sub files, each of which is a subset of the baskets.

2. Finding Candidate Itemsets
    - Find the itemsets that are locally frequent in at least one subset of the baskets.

3. Finding Frequent Itemsets.
    - Find the itemsets that are globally frequent by counting the support for each candidate itemset produced in previous step.

4. Processing Output
    - Output and store the global frequent itemsets result to a local file.

2.2 Description of Key Steps

2.2.1 Processing Input

The input data is given as a file in local file system that contains all baskets, one basket per line. The program first copies this file to a special directory in HDFS, which will be specified as the input path for the MapReduce jobs described in 2.2.2 and 2.2.3.

It also gathers necessary information about the input, such as the total number of baskets and the size of the file, in order to find out the support threshold and the size of a sub file.

Splitting is achieved by setting the input split size, which is a parameter for a MapReduce job, to be the value *s*/*k*, where *s* is the size of the input file and *k* is the number of subsets. In Java, this can be done by the following statement:

```
FileInputFormat.setMaxInputSplitSize(job, subFileSize);
```

With this configuration, Hadoop will automatically spit the single input file into *k* splits and send each split as a map task to a mapper.

2.2.2 Finding Candidate Itemsets

This step is done as a MapReduce job with the mapper class and reducer class described below.

Map

The mapper takes an input split described in 2.2.1 that contains a subset of the baskets as input value. It then performs the A-Priori algorithm on this subset to find all itemsets that are locally frequent against the specified threshold percentage. Finally, it outputs all the pairs (c, null), where c is an itemset found to be locally frequent in this subset.

```
map (key, value):
   // value: a subset of the baskets
   s: the number of baskets in this subset
   p: the threshold as percentage
   k = 0
   do:
      generate candidates of size k+1 from the list of frequent itemsets of size k
      count the support for each candidate
      for each candidate in candidates of size k+1:
         if count of candidate >= ceiling(p*s):
            add candidate to the list of frequent itemsets of size k+1
            emit (candidate, null)
      k = k + 1
   while there is more than one element in the list frequent itemsets of size k+1
```

Note:

- The threshold percentage is distributed to all mappers as a cache variable using Hadoop's feature.
- Since Hadoop will send one record in the input split at a time to the mapper, the mapper simply collects all records during the map function, and performs the actual task in the cleanup function, which will be called after all the records are sent.
- The generation of candidates of size k+1 from the list of frequent itemsets of size k is done in the following way:

```
for each (f1, f2) in all possible pairs from the list of frequent itemsets of size k:
   if the first k-1 items of f1 = the first k-1 items of f2:
      last1 = last item of f1
      last2 = last item of f2
      if last1 < last2:
         candidate = first k-1 items of f1 + last1 + last2
      else:
         candidate = first k-1 items of f1 + last2 + last1

      add candidate to the list of candidates of size k+1
```

Because the algorithm compares the prefixes (i.e., the first k-1 items) of two itemsets, every newly created itemset is sorted in ascending order to make sure they can be used to generate new itemset in the future. Generating candidates in this way can effectively reduce the number of candidates to be considered and hence increase memory efficiency. For k = 0, the candidates of size 1 are generated as needed during the process of counting support.

Reduce

The reducer simply outputs the keys, which are local frequent itemsets found by the mappers.

```
reduce (key, values)
   // key: a local frequent itemset
   emit (key, null)
```

Since there can be more than one reducer, after the MapReduce job finishes, the program merges the sequence of output files produced by all reducers into a single file that contain all local frequent itemsets, which will be used later as the candidates for finding global frequent itemsets.

2.2.3 Finding Frequent Itemsets

This step is also done using MapReduce.

Map

The mapper takes a split of the input as in previous step. I also accesses to the candidate itemsets produced by previous step, which are the itemsets that are locally frequent in at least one subset. Then it counts the support for each of these candidates among the subset of the baskets that it receives. Finally, it outputs a set of pair (c, v), where c is a candidate itemset and v is its support.

```
map (key, value)
   // value: a subset of the baskets
   candidates: the set of all local frequent itemsets produced by previous step

   for each basket in baskets:
      for each candidate in candidates:
         if candidate appears in basket:
            increment the count for candidate

   for each candidate in candidates:
         emit (candidate, count of candidate)
```

Note:

- As in 2.2.2, the mapper first collects all the records of the split in the map function, and then performs the actual counting task in the cleanup function.

- The file containing the candidates is distributed as a cache file to all mappers before the MapReduce job executes. This is a feature of Hadoop to allow efficient access of large, read-only files.
- The counting process is a nested loop that compares every basket to every candidate. In this case, a basket is implemented as a hash set so that the checking whether the candidate is contained in the basket can be more efficient.

Reduce

The reducer takes the itemsets as keys and their supports in each subset as values. It then sums the associated supports of each itemset as its total support and outputs this result if the support is larger than the threshold support.

```
reduce (key, values)
   // key: an itemset
   // values: the counts of this itemset in each subset of baskets
   sum = 0
   for each count in values:
      sum = sum + count

   if sum >= support threshold:
      emit (key, sum)
```

Note:

- The support threshold is pre-calculated as (threshold percentage * total number of baskets) in the step described in 2.2.1, and is distributed as a cache variable to all reducers.

As in 2.2.2, after the MapReduce job finishes, the program merges the results of all reducers into a single file, which will be the file that contains all global frequent itemsets.

2.2.4 Processing Output

When the MapReduce job in 2.2.3 finishes, the program reads the output file in HDFS containing the frequent itemsets, sorts them in descending order according to their supports, and writes to a file in local file system. It also outputs this result to the screen.

2.3 Additional Details

Empty lines in the input file are discarded. Since the threshold is given as percentage, this means that the minimum support number is the total number of non-empty baskets multiplied by the threshold percentage.

3. Testing

3.1 Test Cases & Result

To evaluate the correctness of the program, we ran it on a Hadoop cluster using datasets of various sizes and with different parameters. Specifically, we tried the following test cases:

Case	Number of Baskets	Number of Splits	Threshold	Reason
1	10000	10	0.05	Normal case
2	10000	1	0.05	Only one split
2	10000	10	0.005	Small threshold
3	20000	10	0.5	Large threshold
4	20000	99	0.05	Large number of splits
5	100000	20	0.05	Relatively large number of records

We first found the results for each test case using another program that was implemented to run on a single machine. Then we compared these results with the results produced by our program that ran on the Hadoop cluster to see if they agree with each other. In all test cases, they agreed with each other.

3.2 Sample Test & Result

The following is a sample test result on a dataset of 10 records. The number of splits is 2 and the threshold is 0.3.

Data:

```
1 2 5 7 9
1 7 9
3 4 6 9
2 8
5 10
1 4 5 7 8
```

```
5
9
3 7 8
2 5 6 7 8 10
```

Output:

```
9
5          (5)
7          (5)
8          (4)
9          (4)
1          (3)
1 7        (3)
2          (3)
5 7        (3)
7 8        (3)
```

4. Discussions

4.1 Performance

For the test cases described in 3.1, we measured the elapsed time of the program. They varied depending on the size of dataset and the specified parameters in an expected way (e.g., decreasing the threshold will increase the elapsed time). All of them were able to finish in a reasonable amount of time.

Case	Number of Baskets	Number of Splits	Threshold	Elapsed Time (seconds)
1	10000	10	0.05	58.99
2	10000	1	0.05	54.13
2	10000	10	0.005	142.28
3	20000	10	0.5	65.11
4	20000	99	0.05	294.01
5	100000	20	0.05	131.57

4.2 Impacts of Number of Splits on Performance

As we were testing the program, we noticed that the number of splits for the MapReduce jobs can have a big impact on the performance. For example, for a dataset of 10000 records and with threshold 0.05, setting the number of splits to be 10 led to an elapsed time of 58.99 seconds, while setting this number to be 20 gave 83.39 seconds, which is 40% slower. This implies an optimal number of splits for a fixed task. The reason may be that when the number of splits is too small, the task is not divided into enough small tasks to take full advantage of parallelism; when the number is too large, it causes a large overhead due to communication and allocating resources for mappers.

4.3 Potential Future Work for Improvement

In the processing input step described in 2.2.1, the program needs to scan the input file once in order to get the total number of baskets, which is necessary in order to calculate the required number of supports. For large file, this process can be a factor of slowing down the program. A possible solution is to turn this process into a MapReduce job that counts the number of baskets in a set of subsets in parallel and sums the results to get the total number of baskets.

References

J. Leskovec, A. Rajaraman and J. Ullman. *Mining of Massive Datasets. Cambridge University Press, 2011.*

K.Wang Lecture notes for *Data Mining*. Fall 2014.

P. Tan, M. Steinbach and V. Kumar. *Introduction to Data Mining. Addison Wesley, 2006.*

T. White. *Hadoop:The definitive Guide. O'Reilly Media, 2012.*

Appendix

```java
package org.myorg;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StringReader;
import java.net.URI;
import java.text.SimpleDateFormat;
import java.util.*;

import org.apache.commons.io.FileUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileUtil;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FrequentCount {
    // Constants
    private final static String HDFS_APP_DIR_NAME = "FrequentCount";
    private final static String HDFS_RUN_DIR_NAME = "run";
    private final static String HDFS_INPUT_DIR_NAME = "input";
    private final static String HDFS_CANDIDATE_DIR_NAME = "candidate";
    private final static String HDFS_OUTPUT_DIR_NAME = "output";
    private final static String CANDIDATES_FILE_NAME = "candidates";
    private final static String RESULT_FILE_NAME = "result";

    private final static String ITEM_SEPARATOR = " ";
    private final static String OUTPUT_SEPARATOR = ";";

    private final static String THRESHOLD_VARIABLE_NAME = "THRESHOLD";

    // First Mapper Class
    public static class Mapper1 extends
            Mapper<LongWritable, Text, Text, NullWritable> {
        private Text frequentSet = new Text();
        private double thresholdPer;
        private int numOfBaskets;
        private ArrayList<HashSet<String>> baskets;

        @Override
        protected void setup(Context context) throws IOException,
                InterruptedException {
            super.setup(context);
            thresholdPer = context.getConfiguration().getDouble(
                    THRESHOLD_VARIABLE_NAME, 0);
            baskets = new ArrayList<HashSet<String>>();
        }

```

```
63          @Override
64          public void map(LongWritable key, Text value, Context context)
65                  throws IOException, InterruptedException {
66              String basket = value.toString().trim();
67
68              if (!basket.isEmpty()) {
69                  numOfBaskets++;
70                  baskets.add(toItemset(basket));
71              }
72          }
73
74          @Override
75          protected void cleanup(Context context) throws IOException,
76                  InterruptedException {
77              // A-prior algorithm to generate frequent sets that are locally
78              // frequent in current sub file
79
80              int threshold = (int) (Math.ceil(numOfBaskets * thresholdPer));
81              ArrayList<String> frequents = new ArrayList<String>();
82              Map<String, Integer> candidates = new HashMap<String, Integer>();
83              int size = 1;
84
85              do {
86                  candidates.clear();
87                  getCandidates(frequents, candidates);
88                  countSupport(candidates, size, baskets);
89
90                  frequents.clear();
91                  for (Map.Entry<String, Integer> entry : candidates.entrySet()) {
92                      if (entry.getValue() >= threshold) {
93                          String fs = entry.getKey();
94                          frequents.add(fs);
95
96                          frequentSet.set(fs);
97                          context.write(frequentSet, NullWritable.get());
98                      }
99                  }
100
101                 size++;
102             } while (frequents.size() > 1);
103
104             super.cleanup(context);
105         }
106
107         private static void getCandidates(ArrayList<String> previousFrequents,
108                 Map<String, Integer> candidates) {
109             for (int i = 0; i < previousFrequents.size() - 1; i++) {
110                 for (int j = i + 1; j < previousFrequents.size(); j++) {
111                     String candidate = createNewCandidate(
112                             previousFrequents.get(i), previousFrequents.get(j));
113
114                     if (candidate != null)
115                         candidates.put(candidate, 0);
116                 }
117             }
118         }
119
120         private static String createNewCandidate(String frequent1,
121                 String frequent2) {
122             String prefix1 = getPrefix(frequent1);
123             String prefix2 = getPrefix(frequent2);
124
```

```
125             if (!prefix1.equals(prefix2))
126                 return null;
127
128             String last1 = getLast(frequent1);
129             String last2 = getLast(frequent2);
130
131             prefix1 = prefix1.isEmpty() ? prefix1 : prefix1 + ITEM_SEPARATOR;
132
133             if (last2.compareToIgnoreCase(last1) > 0)
134                 return prefix1 + last1 + ITEM_SEPARATOR + last2;
135
136             return prefix1 + last2 + ITEM_SEPARATOR + last1;
137         }
138
139         private static String getPrefix(String set) {
140             int index = set.lastIndexOf(ITEM_SEPARATOR);
141             return index != -1 ? set.substring(0, index) : "";
142         }
143
144         private static String getLast(String set) {
145             return set.substring(set.lastIndexOf(ITEM_SEPARATOR) + 1);
146         }
147
148         private static void countSupport(Map<String, Integer> candidates,
149                 int size, ArrayList<HashSet<String>> baskets)
150                 throws IOException, InterruptedException {
151             for (HashSet<String> bk : baskets) {
152                 if (size == 1) {
153                     for (String item : bk) {
154                         if (candidates.containsKey(item)) {
155                             candidates.put(item, candidates.get(item) + 1);
156                         } else {
157                             candidates.put(item, 1);
158                         }
159                     }
160                 } else {
161                     for (Map.Entry<String, Integer> entry : candidates
162                             .entrySet()) {
163                         if (contains(bk, entry.getKey())) {
164                             entry.setValue(entry.getValue() + 1);
165                         }
166                     }
167                 }
168             }
169
170         }
171
172         private static HashSet<String> toItemset(String basket) {
173             HashSet<String> result = new HashSet<String>();
174
175             String[] items_temp = basket.split(ITEM_SEPARATOR);
176             for (String item : items_temp) {
177                 item = item.trim();
178                 if (!item.isEmpty())
179                     result.add(item);
180             }
181
182             return result;
183         }
184
185         private static boolean contains(HashSet<String> basketItems,
186                 String candidateSet) {
```

```
187             List<String> cItems = Arrays.asList(candidateSet
188                     .split(ITEM_SEPARATOR));
189             return basketItems.containsAll(cItems);
190         }
191
192     }
193
194     // First Reducer Class
195     public static class Reducer1 extends
196             Reducer<Text, NullWritable, Text, NullWritable> {
197         @Override
198         public void reduce(Text key, Iterable<NullWritable> values,
199                 Context context) throws IOException, InterruptedException {
200             // output union of the frequents from each sub file
201             context.write(key, NullWritable.get());
202         }
203     }
204
205     // Second Mapper Class
206     public static class Mapper2 extends
207             Mapper<LongWritable, Text, Text, IntWritable> {
208         private String candidatesStr = "";
209         private Text frequentSet = new Text();
210         private ArrayList<HashSet<String>> baskets;
211
212         @Override
213         protected void setup(Context context) throws IOException,
214                 InterruptedException {
215             super.setup(context);
216             candidatesStr = FileUtils.readFileToString(new File(
217                     CANDIDATES_FILE_NAME));
218             baskets = new ArrayList<HashSet<String>>();
219         }
220
221         @Override
222         public void map(LongWritable key, Text value, Context context)
223                 throws IOException, InterruptedException {
224             // collect baskets
225             String basket = value.toString().trim();
226             if (!basket.isEmpty()) {
227                 baskets.add(toItemset(basket));
228             }
229         }
230
231         @Override
232         protected void cleanup(Context context) throws IOException,
233                 InterruptedException {
234             // count support of each candidates in current sub file
235             ArrayList<FrequentSet> candidates = readCandidates();
236             if (!candidates.isEmpty()) {
237                 for (HashSet<String> bk : baskets) {
238                     for (FrequentSet set : candidates) {
239                         if (contains(bk, set.getItems())) {
240                             set.incrementSupport();
241                         }
242                     }
243                 }
244             }
245
246             // output count result
247             for (FrequentSet fs : candidates) {
248                 frequentSet.set(fs.getItems());
```

```
249                 context.write(frequentSet, new IntWritable(fs.getSupport()));
250             }
251
252             super.cleanup(context);
253         }
254
255         private ArrayList<FrequentSet> readCandidates() throws IOException {
256             ArrayList<FrequentSet> result = new ArrayList<FrequentSet>();
257             BufferedReader br = new BufferedReader(new StringReader(
258                     candidatesStr));
259             String line = null;
260             while ((line = br.readLine()) != null) {
261                 result.add(new FrequentSet(line.trim(), 0));
262             }
263             br.close();
264
265             return result;
266         }
267
268         private static HashSet<String> toItemset(String basket) {
269             HashSet<String> result = new HashSet<String>();
270
271             String[] items_temp = basket.split(ITEM_SEPARATOR);
272             for (String item : items_temp) {
273                 item = item.trim();
274                 if (!item.isEmpty())
275                     result.add(item);
276             }
277
278             return result;
279         }
280
281         private static boolean contains(HashSet<String> basketItems,
282                 String candidateSet) {
283             List<String> cItems = Arrays.asList(candidateSet
284                     .split(ITEM_SEPARATOR));
285             return basketItems.containsAll(cItems);
286         }
287     }
288
289     // Second Reducer Class
290     public static class Reducer2 extends
291             Reducer<Text, IntWritable, Text, IntWritable> {
292         private IntWritable count = new IntWritable();
293         private int threshold;
294
295         @Override
296         protected void setup(Context context) throws IOException,
297                 InterruptedException {
298             threshold = context.getConfiguration().getInt(
299                     THRESHOLD_VARIABLE_NAME, 0);
300         }
301
302         @Override
303         public void reduce(Text key, Iterable<IntWritable> values,
304                 Context context) throws IOException, InterruptedException {
305             // sum supports in each sub file and output those that are actually
306             // frequent
307             int sum = 0;
308             for (IntWritable val : values) {
309                 sum += val.get();
310             }
```

```
311
312             if (sum >= threshold) {
313                 count.set(sum);
314                 context.write(key, count);
315             }
316         }
317     }
318
319     public static class FrequentSet implements Comparable<FrequentSet> {
320         private String items;
321         private Integer support;
322
323         public FrequentSet(String items, Integer support) {
324             this.items = items;
325             this.support = support;
326         }
327
328         public String getItems() {
329             return this.items;
330         }
331
332         public Integer getSupport() {
333             return this.support;
334         }
335
336         public void incrementSupport() {
337             this.support++;
338         }
339
340         @Override
341         public int compareTo(FrequentSet o) {
342             // sort by support, then by item alphabetic
343             int cmp = o.getSupport().compareTo(this.getSupport());
344             return cmp == 0 ? this.getItems().compareToIgnoreCase(o.getItems())
345                     : cmp;
346         }
347     }
348
349     private static long[] processInput(String inputFile, int k,
350             String hdfsInputDir) throws IOException {
351         // count total number of lines
352         File input = new File(inputFile);
353         long totalNumOfBaskets = 0;
354         BufferedReader br = new BufferedReader(new FileReader(input));
355         String line = null;
356         while ((line = br.readLine()) != null) {
357             if (!line.trim().isEmpty())
358                 totalNumOfBaskets++;
359         }
360         br.close();
361
362         // get size of sub file in order to split the file
363         Long subFileSize = (Long) (input.length() / k);
364
365         // copy to hdfs input directory
366         FileSystem fs = FileSystem.get(new Configuration());
367         fs.copyFromLocalFile(new Path(inputFile), new Path(hdfsInputDir));
368         fs.close();
369
370         System.out
371                 .println("Process Input File Finished. Total Number of Baskets="
372                         + totalNumOfBaskets);
```

```
373         return new long[] { totalNumOfBaskets, subFileSize };
374     }
375
376     private static void firstMR(String input, String output,
377             double thresholdPer, long subFileSize) throws Exception {
378         Configuration conf = new Configuration();
379         conf.setDouble(THRESHOLD_VARIABLE_NAME, thresholdPer);
380         conf.set("mapreduce.output.textoutputformat.separator",
381                 OUTPUT_SEPARATOR);
382         Job job = Job.getInstance(conf, "find_candidates");
383         job.setJarByClass(FrequentCount.class);
384         job.setMapperClass(Mapper1.class);
385         job.setReducerClass(Reducer1.class);
386         job.setOutputKeyClass(Text.class);
387         job.setOutputValueClass(NullWritable.class);
388
389         // set the input split size to be the sub file size so that hadoop can
390         // split the input into k sub inputs
391         FileInputFormat.setMaxInputSplitSize(job, subFileSize);
392         FileInputFormat.addInputPath(job, new Path(input));
393         FileOutputFormat.setOutputPath(job, new Path(output));
394
395         System.out.println("First MapReduce Started. Threshold percentage is "
396                 + thresholdPer);
397         job.waitForCompletion(true);
398         System.out.println("First MapReduce Finished.");
399
400         // when there are more than one reducer
401         // merge output files into a single file as the candidates
402
403         FileSystem hdfs = FileSystem.get(conf);
404         FileUtil.copyMerge(hdfs, new Path(output), hdfs, new Path(output + "/"
405                 + CANDIDATES_FILE_NAME), false, conf, null);
406     }
407
408     private static void secondMR(String candidatesDir, String input,
409             String output, int threshold, long subFileSize) throws Exception {
410         Configuration conf = new Configuration();
411         conf.setInt(THRESHOLD_VARIABLE_NAME, threshold);
412         conf.set("mapreduce.output.textoutputformat.separator",
413                 OUTPUT_SEPARATOR);
414         Job job = Job.getInstance(conf, "find_frequents");
415
416         // add output of first MapReduce as cache
417         String canadidates_output = candidatesDir + "/" + CANDIDATES_FILE_NAME
418                 + "#" + CANDIDATES_FILE_NAME;
419         job.addCacheFile(new URI(canadidates_output));
420
421         job.setJarByClass(FrequentCount.class);
422         job.setMapperClass(Mapper2.class);
423         job.setReducerClass(Reducer2.class);
424         job.setOutputKeyClass(Text.class);
425         job.setOutputValueClass(IntWritable.class);
426
427         // set the input split size to be the sub file size so that hadoop can
428         // split the input into k different sub inputs
429         FileInputFormat.setMaxInputSplitSize(job, subFileSize);
430         FileInputFormat.addInputPath(job, new Path(input));
431         FileOutputFormat.setOutputPath(job, new Path(output));
432
433         System.out.println("Second MapReduce Started. Threshold is "
434                 + threshold);
```

```
435         job.waitForCompletion(true);
436         System.out.println("Second MapReduce Finished.");
437 
438         // merge output files into a single file as the final result
439         FileSystem hdfs = FileSystem.get(conf);
440         FileUtil.copyMerge(hdfs, new Path(output), hdfs, new Path(output + "/"
441                 + RESULT_FILE_NAME), false, conf, null);
442     }
443 
444     private static void processOutput(String resultDir) throws IOException {
445         ArrayList<FrequentSet> result = new ArrayList<FrequentSet>();
446 
447         // Writer to local result file
448         BufferedWriter bw = new BufferedWriter(new FileWriter(RESULT_FILE_NAME));
449 
450         // Reader to hdfs result file
451         Configuration conf = new Configuration();
452         String filePath = resultDir + "/" + RESULT_FILE_NAME;
453         Path path = new Path(filePath);
454         FileSystem hdfs = path.getFileSystem(conf);
455         FSDataInputStream inputStream = hdfs.open(path);
456 
457         BufferedReader br = new BufferedReader(new InputStreamReader(
458                 inputStream));
459         String line = null;
460         while ((line = br.readLine()) != null) {
461             String[] kv = line.split(OUTPUT_SEPARATOR);
462             result.add(new FrequentSet(kv[0].trim(), Integer.parseInt(kv[1]
463                     .trim())));
464         }
465 
466         Collections.sort(result);
467 
468         int size = result.size();
469         System.out.println(size);
470         bw.write(size + "");
471         bw.newLine();
472 
473         String record = "";
474         for (FrequentSet kv : result) {
475             record = String.format("%-20s%-10s", kv.getItems(),
476                     "(" + kv.getSupport() + ")");
477             System.out.println(record);
478             bw.write(record);
479             bw.newLine();
480         }
481         System.out.println();
482 
483         bw.close();
484         br.close();
485         hdfs.close();
486     }
487 
488     // Main
489     public static void main(String[] args) throws Exception {
490         // create directories to store files during executing
491         Date date = new Date();
492         SimpleDateFormat dateFormat = new SimpleDateFormat(
493                 "yyyy-MM-dd-HH-mm-ss");
494         String homeDir = FileSystem.get(new Configuration()).getHomeDirectory()
495                 .toString();
496         String runDir = homeDir + "/" + HDFS_APP_DIR_NAME + "/"
```

```
497                 + HDFS_RUN_DIR_NAME + "_" + dateFormat.format(date);
498         String inputDir = runDir + "/" + HDFS_INPUT_DIR_NAME;
499         String candidatesDir = runDir + "/" + HDFS_CANDIDATE_DIR_NAME;
500         String outputDir = runDir + "/" + HDFS_OUTPUT_DIR_NAME;
501 
502         // execute
503         int subFileNum = Integer.parseInt(args[1]);
504         long[] inputInfo = processInput(args[0], subFileNum, inputDir);
505 
506         double threshold = Double.parseDouble(args[2]);
507         int support_threshold = (int) Math.ceil(threshold * inputInfo[0]);
508 
509         // using threshold percentage
510         firstMR(inputDir, candidatesDir, threshold, inputInfo[1]);
511         // using support threshold
512         secondMR(candidatesDir, inputDir, outputDir, support_threshold,
513                 inputInfo[1]);
514         processOutput(outputDir);
515     }
516 }
```