CMPT 741, Fall 2014

Data Mining - Project Report

MapReduce Programming using Hadoop

Marten Heidemeyer * Rafael Bradley†

November 20, 2014

1 Introduction

The goal of this project is to find all frequent itemsets in a given list of transactions. The intuitive solution to this problem would be to count all possible itemsets. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of items in the input data, the set of possible itemsets is the power set over I and has size 2^{n-1}. Although the size of the powerset grows exponentially in the number of items n in I, efficient search is possible using the downward-closure property of support (also called anti-monotonicity) which guarantees that for a frequent itemset, all its subsets are also frequent and thus for an infrequent itemset, all its supersets must also be infrequent. However, if memory is too small to hold the complete list of transactions and the count for all itemsets, efficient methods are needed to process large databases. In this regard, this project focuses on implementing the algorithm of Savasere, Omiecinski, and Navathe (called also SON algorithm after the authors) which uses parallel computing to find frequent itemsets. Assume the Input:

- text file with a large set of transactions (baskets)
- desired number of subfiles k to split the text file in.
- support threshold s.

The output we want to produce is:

- all itemsets that occur in at least s of the given baskets.

Note: the amount of items per transaction is unknown as well as the amount of transactions and the global amount of the items.

*mheideme@sfu.ca
†rafaelb@sfu.ca

2 High Level Approach

The input file is manually partitioned into k data chunks. In a first pass of MapReduce, each chunk is passed to a map task in charge of returning all the frequent itemsets (subject to a lower chunk threshold, see 2.3) in this particular chunk using the A-priori Algorithm. The frequent itemsets generated from all the chunks are then merged by the reduce tasks. The output of the first pass is the candidate itemsets. In a second pass of MapReduce, each mapper again processes one data chunk and returns how many times every candidate appears in its chunk. Then, the reduce tasks summarize the results and discard all itemsets in which the global count is less than the support threshold s. The output gives all the frequent itemsets.

2.1 The MapReduce-MapReduce sequence

All in all the frequent itemsets are found in two runs of MapReduce. This section gives pseudocode-level details about the algorithms that run in the first mappers and first reducers as well as the second mappers and second reducers. Before going over the two passes section 2.2 gives details about two important data structures that are used in the passes.

2.2 Data Structures

In the first pass each mapper in the Map Task is running the A-Priori Algorithm over the input data chunk (see section 2.3). The A-Priori Algorithm first finds all frequent single items, then all frequent itemsets of size two and so on (see section 2.3.1). The data structure that is used in the A-Priori Algorithm is the following:

- For simplicity and in agreement with the given test data (see section 3) we assume that items are represented by integers. A set of items is then stored as *Set<Integer>*. Frequent itemsets can have size $1, 2, \ldots$. For each size we get one list of itemsets (*List<Set<Integer>>*). The frequent itemsets data structure contains one list of frequent itemsets for each size of which frequent itemsets were found. Thus we get the data structure *List<List<Set<Integer>>>*.

The second run of mappers count for each candidate itemset how many of the baskets of a data chunk contain the candidate (see section 2.5). Therefore they need to store a count for each candidate. The data structure that is used to store the counts is the following:

- Each itemset is an instance of *Set<Integer>*. Therefore we need a hashtable that maps from *Set<Integer>* to counts (Integer). This gives us the data structure *HashMap<Set<Integer>, Integer>*.

2.3 First Map Task

In the first run of mapper functions each mapper is processing one of the data chunks. If the fraction of the initial file that is given to the mapper is k it is its task to find all

itemsets that occur at least ks times in the chunk (s = support threshold for complete file). To get the candidate itemsets of a data chunk the A-Priori Algorithm was utilized which is described in more detail in 2.3.1. The pseudocode given in Algorithm 1 describes the function of the first set of mappers.

Algorithm 1 First Map

```
1: function MAPPER(String data_chunk)
2:     List<String> basket_list = SPLIT_INTO_BASKETS(data_chunk)
3:     int chunk_threshold = COMPUTE_CHUNK_THRESHOLD(basket_list.length)
4:     List<List<Set<Integer>>> frequent_itemsets = RUN_APRIORI(data_chunk,
   chunk_threshold)
5:     for each Set<Integer> itemset ∈ frequent_itemsets do
6:         produce(itemset, 1)
7:     end for
8: end function
```

2.3.1 Finding frequent itemsets in first phase - the A-Priori Algorithm

Each mapper in the first run of mappers finds all itemsets that are frequent (subject to the lower threshold ks) in the input data chunk by using the A-Priori Algorithm. For this step an own implementation of the A-Priori Algorithm was utilized. From a top level perspective the key function of the A-Priori Algorithm is listed in Algorithm 2. Appendix 6.2.2 contains well documented Java code of a complete implementation.

Algorithm 2 A-Priori

```
1: function RUN_APRIORI(List<String>> baskets, int chunk_threshold)
2:     all_frequent_itemsets = new List<List<Set<Integer>>>
3:     frequent_itemsets = GET_FREQUENT_SINGLE_ITEMS(baskets, chunk_threshold)
4:     if frequent_itemsets not empty then
5:         all_frequent_itemsets.ADD(frequent_itemsets)
6:         candidates = COMPUTE_CANDIDATES_FOR_NEXT_PASS(frequent_itemsets)
7:     end if
8:     while candidates not empty AND frequent_itemsets not empty do
9:         frequent_itemsets = GET_FREQUENT_CANDIDATES(baskets, candidates,
   chunk_threshold)
10:        if frequent_itemsets not empty then
11:            all_frequent_itemsets.ADD(frequent_itemsets)
12:            candidates = COMPUTE_CANDIDATES_FOR_NEXT_PASS(frequent_itemsets)
13:        end if
14:    end while
15:    return all_frequent_itemsets
16: end function
```

2.4 First Reduce Task

The Reducer Task in the first pass merges together all the itemsets that were detected in the separate data chunks by the first Map Task. Each reducer gets a key-value pair where the key is an itemset that was detected as frequent in at least one of the data chunks. The reducers ignore the value part and simply produce the itemset as $< itemset, 1 >$. The combined output of all reducers form the list of candidate itemsets for the second pass. The pseudocode given in Algorithm 3 describes the reducers which run in the first Reduce Task:

Algorithm 3 First Reduce Task

```
1: function REDUCER(Set<Integer> itemset, Value value)
2:     produce(itemset, 1)
3: end function
```

2.5 Second Map Task

In the second run of mapper functions each mapper is again processing one of the data chunks. Before processing the data chunk the mapper reads in the list of candidate itemsets that was produced in the first MapReduce pass. The task of each mapper is to produce a key-value pair of the form $< candidate, count >$ for each *candidate* where *count* is the number of baskets in the chunk that contains all items in the itemset *candidate*. Algorithm 4 gives pseudocode for the mappers that run in the second Map Task.

2.6 Second Reduce Task

Each reducer in the second run of reducers gets a key-value pair where the key is a candidate itemset and the value is a list of counts. The count list contains one value for each data chunk which is the number of baskets in that chunk that contains all the items of the candidate. The reducer sums up all the counts and gets the number of total baskets that contain the candidate. If this number is at least s then the candidate is a frequent itemset for the whole file and the reducer produces it. Algorithm 5 describes the function of the reducers which run in the second Reduce Task.

3 Test Data

In this project, the test data consists of a plain text file where:

- Each transaction (basket) is separated by the end of line character.
- Each item is separated by the space character.
- Each item is a positive integer.

Algorithm 4 Second Map Task

```
1: function MAPPER(String data_chunk)
2:     List<Set<Integer>> candidate_itemsets = READ_CANDIDATE_LIST()
3:     List<String> basket_list = SPLIT_INTO_BASKETS(data_chunk)
4:     HashMap<Set<Integer>, Integer> candidate_counts = new HashMap<Set<Integer>,
   Integer>
5:     for each String basket ∈ basket_list do
6:         Set<Integer> temp_basket = CONVERT_TO_SET(basket)
7:         for each Set<Integer>> candidate ∈ candidate_itemsets do
8:             Boolean basket_contains_candidate = true
9:             for each Integer item ∈ candidate do
10:                if NOT temp_basket.CONTAINS(item) then
11:                    basket_contains_candidate = false
12:                    break
13:                end if
14:            end for
15:            if basket_contains_candidate = true then
16:                candidate_counts(candidate)=candidate_counts(candidate)+1
17:            end if
18:        end for
19:    end for
20:    for each candidate ∈ candidate_itemsets do
21:        produce(candidate, candidate_counts(candidate))
22:    end for
23: end function
```

Algorithm 5 Second Reduce Task

```
1: function REDUCE(Set<Integer> itemset, List<Integer> count_list, Integer sup-
   port_threshold)
2:     Integer total_count = 0
3:     for each Integer count ∈ count_list do
4:         total_count = total_count+count
5:     end for
6:     if total_count≥ support_threshold then
7:         produce(itemset, total_count)
8:     end if
9: end function
```

It is assumed that there is no repeated items within a transaction. An example input file could look like the following:

52 164 240 274 328 368 448 538 561 630 687 730 775 825 834
39 120 124 205 401 581 704 814 825 834
35 249 674 712 733 759 854 950
39 422 449 704 825 857 895 937 954 964
15 229 262 283 294 352 381 708 738 766 853 883 966 978
26 104 143 320 569 620 798
7 185 214 350 529 658 682 782 809 849 883 947 970 979
227 390
71 192 208 272 279 280 300 333 496 529 530 597 618 674 675 720 855 914 932
183 193 217 256 276 277 374 474 483 496 512 529 626 653 706 878 939
161 175 177 424 490 571 597 623 766 795 853 910 960
125 130 327 698 699 839
392 461 569 801 862

4 Implementation Details

For the implementation of the two pass MapReduce pipeline the Hadoop Framework was used. Hadoop is an open-source implementation of a distributed file system (HDFS) with a high level API that allows to easily run the Mapper and Reducer Tasks each in parallel on distributed nodes. All the code for the Mapper and Reducer Tasks as well as preprocessing and postprocessing steps was written in Java. In the scope of this project a pipeline was created which in sequential order consists of the following modules:

1. preprocessing of the input file to create data chunks
2. setting of the threshold parameters so that they are accessible by the Mapper and Reducer classes of the Hadoop API
3. running the first pass of MapReduce to create itemset candidates
4. running the second pass of MapReduce to find frequent itemsets
5. postprocessing of the output of the second MapReduce pass

A user executes this pipeline by calling the implemented script

find_frequent_itemsets.sh data.dat k s

where *data.dat* is a transaction file in the format described in section 3, k is the desired number of data chunks that *data.dat* is split into and s is the support threshold in

percentage (see Appendix 6.1.1). For each module this section gives insight into how it is processing the given input. Section 4.1 describes the splitting of the test data into chunks and the parameter setting so that they are accessible by the Hadoop Modules. Section 4.2 and 4.3 give details about the first and second run of MapReduce. These two sections give Hadoop specific implementation details. 4.4 describes how the output of the second MapReduce run is postprocessed.

4.1 Data Preprocessing and Parameter Setting

As a first step in the pipeline the complete list of transactions is split into k chunks. In order to find out how many transactions of the complete list go into each chunk it is required to know how many transactions l the complete list contains. If l and k are known it is possible to go over the complete list and write $\frac{l}{k}$ transactions into each chunk. To find l we go over the complete list once, then we go over the list again and write $\frac{l}{k}$ transactions into each file. The main loop to get the chunks is listed in Listing 1 (Note that we round up for $\frac{l}{k}$, if the initial file can not be split into k parts equally). The complete code is listed in Appendix 6.2.1.

```
public void create_chunks(String file_path, int k,
    String output_directory) throws IOException{

int total_number_of_transactions =
            get_number_transactions(file_path);
float transactions_per_chunk_float =
        (float) total_number_of_transactions/k;
int transactions_per_chunk =
        (int) Math.ceil(transactions_per_chunk_float);

/*
    initialize all counters, file_writers,..., see Appendix 5.3
*/

while ((transaction = file_reader.readLine())!=null){
    transactions_written_to_chunk++;
    total_line_counter++;
    chunk_writer.println(transaction);

    if (transactions_written_to_chunk ==
            transactions_per_chunk){

        chunk_writer.close();
        chunk_counter++;
        output_chunk =
                get_filename_for_next_chunk(chunk_counter);
        if (total_line_counter!=total_number_of_transactions)
        {
            chunk_writer = get_chunk_writer(output_directory,
                                output_chunk);
        }
        transactions_written_to_chunk = 0;
    }

}
chunk_writer.close();
```

```
file_reader.close();
}
```

Listing 1: writing the data chunks

The above code writes the data chunks to a local directory from which they are copied to the Hadoop machine into a directory:

$$data\_chunks\_k\_\$k$$

by the script *find_frequent_itemsets.sh* (see Appendix 6.1.1 for details). Finally the first pipeline step writes the total threshold s and the number of lines l_{total} of the initial file to the configuration files *config/support_threshold.txt* and *config/number_lines.txt* on the Hadoop machine where they are accessible to the following runs of MapReduce (see 4.2 and 4.3).

4.2 The First MapReduce Pass

After the data chunks are copied to the hadoop machine and the parameters for the thresholds are set the parent script *find_frequent_itemsets.sh* executes the first run of MapReduce. The first run of mappers each run A-Priori over the given data chunk to find candidates. This section does not examine the implemented A-Priori algorithm as it would exceed the scope of this report. Well documented code for the A-Priori function is listed in Appendix 6.2.2. Also the actual code for the mappers and reducers is very similar to the pseudocode given in 2.3, 2.5 and thus not examined here (see Appendix 6.2.4 for well documented code of the mappers and reducers). On the other hand this section covers technical details about the use of the Hadoop framework. The two aspects that appeared challenging to implement in Hadoop were:

- setting the parameters which allow each mapper to calculate the lower chunk threshold dynamically. Let l_{chunk} be the number of transactions in a data chunk, l_{total} be the number of transactions in the initial file and s be the total support threshold. The lower threshold p is computed as $\frac{l_{chunk}}{l_{total}} * s$ where we round to the next lower number if we get a value $p \notin N$. How to obtain l_{total} and s in each mapper is explained in 4.2.1

- for our application of the Hadoop framework it was required that each mapper processes the complete content of a data chunk. This is not the standard use of Hadoop. In the given *WordCount.java* for example each mapper only processes one line of a file at a time. How to configure Hadoop to process complete data chunks in each mapper is explained in 4.2.2.

4.2.1 Getting the chunk threshold

As mentioned in 4.1 the support threshold and the total number of transactions were written to Hadoop accessible configuration files. The hadoop class *Mapper* from which

the implemented Mapper class inherits contains a *setup* method which was utilized to read these parameters from the configuration files. By default the *setup* method is called at the beginning of a Mapper (Reducer) Task and is capable of setting the parameters as class variables where they are readable by the *map* (*reduce*) method. Assuming that the support threshold was written to the file *config/support_threshold.txt* and the total number of transactions to *config/number_transactions.txt* on the Hadoop machine in the format *parameter_name:p* they can be obtained as listed in Listing 2.

```
public static class Mapper extends
            Mapper<Object, BytesWritable, Text, IntWritable> {

private int total_support_threshold;
private int total_number_transactions;

public void setup(Context context) throws IOException {
    Path path = context.getWorkingDirectory();
    String path_as_string= path.toString();
    String threshold_file_path_string =
        path_as_string.concat("/config/support_threshold.txt");
    String transactioncount_file_path_string =
        path_as_string.concat("/config/number_transactions.txt");
    Path threshold_file_path = new Path(threshold_file_path_string);
    Path transaction_count_path =
             new Path(transactioncount_file_path_string);

    FileSystem fs = FileSystem.get(context.getConfiguration());
    FSDataInputStream in = fs.open(threshold_file_path);
    String line;
    String[] temp = new String[2];
    line = in.readLine();
    in.close();
    temp = line.split(":");
    String threshold_string = temp[1];

    in = fs.open(transaction_count_path);
    line = in.readLine();
    in.close();
    temp = line.split(":");
    String transactioncount_string = temp[1];

    this.total_support_threshold = Integer.parseInt(threshold_string);
    this.total_number_transactions
             = Integer.parseInt(transactioncount_string);
}
public void map(Object key, BytesWritable value, Context context)
                throws IOException, InterruptedException {

    /*
    convert key to Array of Strings;
    number of Strings in Array gives us
            the number of transactions in data chunk;
    compute the lower threshold for the data chunk, using
            total_support_threshold (set in setup method),
            total_number_transactions (set in setup method) and
            number of transactions in data chunk;
    run A-Priori with lower threshold;
```

```
    */
}
}
```

Listing 2: setting the threshold parameter

4.2.2 Passing complete file content to mappers

The manner in which Hadoop passes the content of an input path to the individual mapper functions can be configured by creating a class which inherits from the Hadoop class *FileInputFormat* and setting it as the *InputFormatClass* of the Hadoop job. Listing 3 shows the implemented Format Class.

```
public class WholeFileInputFormat
        extends FileInputFormat<NullWritable, BytesWritable>{
@Override
protected boolean isSplitable(JobContext context,
                              Path filename) {
return false;
}
@Override
public RecordReader<NullWritable, BytesWritable>
        createRecordReader(InputSplit split,
                TaskAttemptContext context)
                              throws IOException {
return new WholeFileRecordReader();
}
}
```

Listing 3: InputFormatClass to pass complete data chunk content to mappers

The two methods that are overwritten from the parent class *FileInputFormat* are *isSplitable* and *createRecordReader*. *isSplitable* just returns false which assures that a data chunk is not split further and handed to separate mappers. *createRecordReader* returns an instance of the class *WholeFileRecordReader* which inherits from *RecordReader*. The class *WholeFileRecordReader* defines in the method *nextKeyValue* how to break the file splits into key-value pairs for input to the mappers. In our case we want the file splits to be passed to the mappers as one. Therefore the method *nextKeyValue* was overwritten to read complete file splits without breaking them further. Appendix 6.2.3 lists the implemented class *WholeFileRecordReader*.

4.3 The Second MapReduce Pass

The first MapReduce pass created a list of candidate itemsets. Each mapper in the second run processes one data chunk and counts how often each candidate occurs in it. Again the Java code for the second run of mappers and reducers is very similar to the pseudocode given in 2.5 and 2.6. Well documented code for the second run of mappers

and reducers can be found in Appendix 6.2.5. To not exceed the volume of this report this section rather focusses on the technical detail of how to pass the candidate itemsets to each mapper in the second pass which is explained in 4.3.1

4.3.1 Passing the candidate itemsets to mappers

The output of the first pass of MapReduce is a text file which contains one candidate itemset per line. Each line is in the key-value format as shown below. The value part of these pairs is not of interest and can be discarded. We are only interested in the itemset part of each line. The first few lines of the candidate itemset file, which by default is called *part-r-00000* will look similar to this:

[0]	1
[100, 362]	1
[100]	1
[104]	1
[11, 793]	1

In order for the second set of mappers to read the candidate file it gets copied to a temporary directory by the parent script *find_frequent_itemsets.sh* before starting the Hadoop job for the second pass. Then we utilized the inherited *setup* method of the Mapper class to read the candidate file. Listing 4 lists the main loop to obtain the candidates. The complete code is listed in Appendix 6.2.5.

```
public static class SecondPassMapper extends
            Mapper<Object, BytesWritable, Text, IntWritable> {

public ArrayList<HashSet<Integer>> candidateItemsets =
                new ArrayList<HashSet<Integer>>();

public void setup(Context context) throws IOException {
/*
    initialize hadoop file reader for candidate itemset file
            (see Apendix)
*/
while ((line = in.readLine()) != null) {
    temp = line.split("\\t");
    candidatesString = temp[0].substring(1, temp[0].length());
    candidatesString =
        candidatesString.substring(0, candidatesString.length() - 1);
    itemset = candidatesString.split(", ");
    tempSet = new HashSet<Integer>();
    for (String item : itemset) {
        tempSet.add(Integer.parseInt(item));
    }
    candidateItemsets.add(tempSet);
}
in.close();
}
/*
```

```
actual code for the map function goes here (see Appendix)
*/
}
```

Listing 4: reading the candidate itemset file

4.4 Output Postprocessing

As a result of the second MapReduce pass we get a textfile which contains all frequent itemsets in the format generated by Hadoop. In this format we get an unsorted list of itemset-count pairs. In the last pipeline step we transform this format into the desired format of sorted itemsets by sorting it respective to the itemsets count. Then we simply write the sorted list into a final output file *frequent_ itemsets* where the first line contains the total number of frequent itemsets and the following lines each contain one itemset count pair in descending order regarding the itemsets count. Complete code for this module can be found in Appendix 6.2.6.

5 Evaluation

As a reference value for the following evaluation of the MapReduce implementation, note that without using MapReduce it took the implemented A-Priori Algorithm about 23 minutes to find frequent itemsets with a support threshold of 1% in the transaction list *example.dat* on a 3.00 GHz machine.

The pipeline was tested on the given SFU cluster (see project description) for the given test data file *example.dat* for different choices of subfiles k and support thresholds s. The following tables lists the calculation time for the two MapReduce passes as well as the number of found candidates and number of frequent itemsets:

k	s (in %)	t first pass (in sec.)	candidates	t second pass (in sec.)	frequent itemsets	t total (in sec.)
5	1	timeout		-		-
10	1	420	454	21	385	441
25	1	296	616	45	385	341
50	1	290	930	74	385	364
100	1	370	2415	145	385	515
200	1	timeout		-		-

For the choice of 5 and 200 file splits Hadoop could not run the first MapReduce successfully due to timeouts in the mappers. We assumed that for small chunks ($k = 200$) the lower support threshold for the chunk becomes so low that the A-Priori runs in the mappers have to deal with too many candidates and thus the mappers time out. We assumed that the same applies for big chunks ($k = 5$): Even though the lower threshold is larger than for smaller chunks, A-Priori has to deal with too many candidates when finding frequent itemsets. Therefore we ran the pipeline for $k = 200, k = 5$ again with a higher threshold s. For $s = 2$ the two passes executed successfully:

k	s (in %)	t first pass (in sec.)	candidates	t second pass (in sec.)	frequent itemsets	t total (in sec.)
5	2	61	170	21	155	82
200	2	257	637	327	155	412

6 Appendix

6.1 Shell Scripts

6.1.1 find_frequent_itemsets.sh

```
#!/bin/sh
numlines=$(wc -l $1 | awk '{print $1;}')
whole_file_threshold=$((($numlines/100)*$3))
rm -r ./data_chunks_p_$2
mkdir ./data_chunks_p_$2
./SplitFile.jar $1 data_chunks_p_$2 $2
hadoop fs -rm -r data_chunks_p_$2
hadoop fs -mkdir data_chunks_p_$2
hadoop fs -copyFromLocal ./data_chunks_p_$2/* data_chunks_p_$2/
hadoop fs -rm -r config
hadoop fs -mkdir config
./set_total_number_baskets.sh $numlines
./set_threshold_whole.sh $(($whole_file_threshold/$2))
./remove_output_directory.sh $2
hadoop jar MapReduce.jar hadoop_project.FirstPass data_chunks_p_$2 \
                                                  candidates_p_$2
./copy_first_pass_output_to_temp_directory.sh $2
hadoop jar MapReduce.jar hadoop_project.SecondPass data_chunks_p_$2 \
                                                   frequent_itemsets_p_$2
hadoop fs -copyToLocal frequent_itemsets_p_$2/part-r-00000 ./
./Postprocess.jar part-r-00000
rm part-r-00000
```

Listing 5: Connection of Pipeline Modules

6.2 Java Modules

6.2.1 Preprocessing

```
package filesplit;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;


public class FileSplit {

    //returns total number of lines in "filepath"
    public static int get_number_transactions(String filepath){
```

```
        int lineCounter=0;
        try {
            FileReader fr = new FileReader(filepath);
            BufferedReader textReader = new BufferedReader(fr);
            while (textReader.readLine()!=null){
                lineCounter++;
            }
            textReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return lineCounter;
    }

    //assembles the next name of a data chunk according
    //to fileCounter
    public static String get_filename_for_next_chunk(int fileCounter){

        String filename;
        if (fileCounter<10){
            filename="file";
            filename=filename.concat("00"+fileCounter+".dat");
        }
        else if (fileCounter<100){
            filename="file";
            filename=filename.concat("0"+fileCounter+".dat");
        }
        else{
            filename="file";
            filename=filename.concat(fileCounter+".dat");
        }

        return filename;
    }

    //returns a filereader for "path_to_file"
    public static BufferedReader get_file_reader(String path_to_file){
        BufferedReader textReader=null;
        try {
            FileReader fr = new FileReader(path_to_file);
            textReader = new BufferedReader(fr);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return textReader;

    }

    //returns a filewriter for "output_chunk", which
    //is put into "directory"
    public static PrintWriter get_chunk_writer(
                String directory, String output_chunk){

        String fullpath = directory.concat("/"+output_chunk);
        PrintWriter writer=null;
        try {
            writer = new PrintWriter(fullpath, "UTF-8");
```

```
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return writer;
    }
    //creates the actual data chunks
    public static void create_chunks(String file_path, int k,
                String output_directory) throws IOException{
        int total_number_of_transactions
                = get_number_transactions(file_path);
        float transactions_per_chunk_float
                = (float) total_number_of_transactions/k;
        int transactions_per_chunk
                = (int) Math.ceil(transactions_per_chunk_float);

        int chunk_counter = 1;
        String output_chunk = get_filename_for_next_chunk(chunk_counter);
        PrintWriter chunk_writer
                 = get_chunk_writer(output_directory, output_chunk);
        BufferedReader file_reader = get_file_reader(file_path);
        String transaction;
        int transactions_written_to_chunk = 0;
        int total_line_counter = 0;
        while ((transaction = file_reader.readLine())!=null){
            transactions_written_to_chunk++;
            total_line_counter++;
            chunk_writer.println(transaction);
            if (transactions_written_to_chunk == transactions_per_chunk){
                chunk_writer.close();
                chunk_counter++;
                output_chunk = get_filename_for_next_chunk(chunk_counter);
                if (total_line_counter!=total_number_of_transactions)
                    chunk_writer
                        = get_chunk_writer(output_directory, output_chunk);
                transactions_written_to_chunk = 0;
            }
        }
        chunk_writer.close();
        file_reader.close();
    }
    //runs the method create_chunks to create
    //the data chunks
    public static void main(String[] args) throws IOException{
```

```
        int numberFiles = Integer.parseInt(args[2]);
        create_chunks(args[0], numberFiles, args[1]);

    }

}
```

Listing 6: Creating Data Chunks

6.2.2 A-Priori

```
package hadoop_project;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Set;

public class Apriori {

    ArrayList<HashSet<Integer>> candidateItemsets
                = new ArrayList<HashSet<Integer>>();
    ArrayList<ArrayList<HashSet<Integer>>> allFrequentItems
                 = new ArrayList<ArrayList<HashSet<Integer>>>();
    Hashtable<HashSet<Integer>, Integer> itemsetCount
                = new Hashtable<HashSet<Integer>, Integer>();
    String[] allBaskets;
    Integer numberBaskets;
    Integer threshold;
    Integer itemCount = 1;

    //this method counts all single items in a transaction "line".
    //It either creates a new count for an item in itemsetCount
    //or increases an existing counter if it was seen before.
    private void processBasketInFirstPass(String line) {

        String[] items = line.split(" ");
        HashSet<Integer> tempList = new HashSet<Integer>();

        for (String item : items) {

            tempList = new HashSet<Integer>();
            if (!item.equals("")) {
                tempList.add(Integer.parseInt(item));
                if (itemsetCount.containsKey(tempList)) {
                    itemsetCount.
                        put(tempList, itemsetCount.get(tempList) + 1);
                } else {
                    itemsetCount.put(tempList, 1);
                }
            }

        }

    }
```

```
//this method expects that candidates were counted in the last
//pass and that their counts are now in itemsetCount.
//It checks each count and puts those itemsets that have a count
//>= threshold into the list allFrequentItems
private boolean filterOutFrequentItemsets() {

    Set<HashSet<Integer>> keys = itemsetCount.keySet();

    ArrayList<HashSet<Integer>> tempList
        = new ArrayList<HashSet<Integer>>();

    for (HashSet<Integer> key : keys) {

        if (itemsetCount.get(key) >= threshold) {
            tempList.add(key);
        }

    }
    if (tempList.size() > 0) {
        allFrequentItems.add(tempList);
        return true;
    }
    return false;

}

//this method merges together the itemsets in "list"
//and returns them as a HashSet.
private HashSet<Integer> getAllOccuringItemsFromList(
        ArrayList<HashSet<Integer>> list) {

    HashSet<Integer> allOccuringItems = new HashSet<Integer>();

    for (HashSet<Integer> itemset : list) {

        for (Integer item : itemset) {

            if (!allOccuringItems.contains(item)) {
                allOccuringItems.add(item);
            }

        }

    }

    return allOccuringItems;
}

//this function computes all candidate itemsets which
//were added in the last pass. documentation for this
//method is given line by line
private ArrayList<HashSet<Integer>>
        getAllCandidatesFromPreviousListOfFrequentItems(
            ArrayList<HashSet<Integer>> previousList) {

    //this is the number of frequent itemsets
    //that were found in the last pass.
    Integer sizeOfPreviousList = previousList.size();
    //this is the size of the itemsets that were found
```

```
//in the last pass.
Integer itemsInPreviousList = previousList.get(0).size();
//this is the size of the candidate itemsets
//that will be computed.
Integer numberItemsInNewList = itemsInPreviousList + 1;
//this array contains indexes which will refer to
//frequent itemsets
//out of which the candidates will be computed
Integer[] selectedLists = new Integer[numberItemsInNewList];
//this is the index that will be increased in the next iteration.
//its always the last one in the list until the last one reaches
//the last itemset. Then we try to increase the one
//before that and so on
Integer pointerToUpdate = numberItemsInNewList - 1;
//at first we check the first numberItemsInNewList itemsets to see
//if they can produce a candidate
for (int i = 0; i < numberItemsInNewList; i++) {
    selectedLists[i] = i + 1;
}

Integer counter;
ArrayList<HashSet<Integer>> tempLists
        = new ArrayList<HashSet<Integer>>();
HashSet<Integer> tempSet;
//contains all candidates
ArrayList<HashSet<Integer>> allCandidates
        = new ArrayList<HashSet<Integer>>();
//allListsChecked becomes true when all
//combinations of frequent itemsets were checked
// to see if they can produce a candidate
boolean allListsChecked = false;

//if previousList.size() < numberItemsInNewList
//we can't produce new candidates
//because we need at least numberItemsInNewList
//to produce a candidate
if (previousList.size() < numberItemsInNewList)
    return allCandidates;

while (!allListsChecked) {
    //tempLists contais all frequent itemsets
    //which we check to see
    //if they can produce a new candidate
    tempLists.clear();
    for (int i = 0; i < numberItemsInNewList; i++) {
        tempLists.add(previousList.get(selectedLists[i] - 1));
    }
    //tempSet is the content of tempLists merged into one list
    //if tempSet contains exactly numberItemsInNewList items
    //then the itemsets in tempSet produce a new candidate and
    //tempSet is added to the list of candidates
    tempSet = new HashSet<Integer>();
    tempSet = getAllOccuringItemsFromList(tempLists);
    if (tempSet.size() == numberItemsInNewList) {
        allCandidates.add(tempSet);
    }
```

```
            //we update the indexes in selectedLists
            //to give us a new combination
            //of itemsets from the last pass. If no index
            //can be updated we checked
            //all combinations.
            while (selectedLists[pointerToUpdate] == sizeOfPreviousList
                    - (numberItemsInNewList - (pointerToUpdate + 1))) {
                pointerToUpdate--;
                if (pointerToUpdate == -1)
                    break;
            }
            if (pointerToUpdate >= 0) {
                selectedLists[pointerToUpdate]++;
                counter = 1;
                for (int i = pointerToUpdate + 1; i <
                        numberItemsInNewList; i++) {
                    selectedLists[i] = selectedLists[pointerToUpdate]
                                                    + counter;
                    counter++;
                }
                pointerToUpdate = numberItemsInNewList - 1;
            } else {
                allListsChecked = true;
            }
        }

        return allCandidates;
    }

    //this method runs over all baskets and calls the method
    //processBasketInFirstPass on each one. processBasketInFirstPass
    //creates a new counter for each single item in the
    //HashTable itemsetCount if the item was not hashed
    //before or increases the counter by one
    //if the item was hashed before.
    private boolean firstPass() throws IOException {

        boolean itemsetAdded = false;
        Integer counter = 0;
        while (counter < numberBaskets) {

            processBasketInFirstPass(allBaskets[counter]);
            counter++;
        }
        itemsetAdded = filterOutFrequentItemsets();

        return itemsetAdded;
    }

    //this method checks for all candidate itemsets in candidateItemsets
    //if it occurs in the transaction "line". If it occurs in the
    //transaction the counter for the candidate in
    //the HashTable itemsetCount is increased by one.
    private void processBasketForCandidateItemsets(String line) {

        HashSet<Integer> itemsAsInt = new HashSet<Integer>();
        String[] items = line.split(" ");
        for (String item : items) {
```

```
            itemsAsInt.add(Integer.parseInt(item));
        }

        for (HashSet<Integer> candidate : candidateItemsets) {
            boolean allItemsOfSetInBasket = true;
            for (Integer item : candidate) {

                if (!itemsAsInt.contains(item)) {
                    allItemsOfSetInBasket = false;
                    break;
                }

            }
            if (allItemsOfSetInBasket) {
                if (itemsetCount.containsKey(candidate)) {
                itemsetCount
                    .put(candidate, itemsetCount.get(candidate) + 1);
                } else {
                    itemsetCount.put(candidate, 1);
                }
            }

        }

    }

    //nextpass() expects that candidates for the next pass
    //were already computed and stored in the List candidateItemsets.
    //The method runs over all transactions and counts in each
    //transaction how often each candidate occurs in it by calling
    //the method processBasketForCandidateItemsets. After all candidates
    //were counted, it calls the function filterOutFrequentItemsets which
    //adds the frequent candidates to the complete
    //list of frequent itemsets
    public boolean nextPass() throws IOException {

        boolean itemsetAdded = false;

        itemsetCount.clear();
        int counter = 0;

        while (counter < numberBaskets) {
            processBasketForCandidateItemsets(allBaskets[counter]);
            counter++;
        }

        itemsetAdded = filterOutFrequentItemsets();

        return itemsetAdded;

    }

    //this function just reads in the baskets into a local Array and sets
    //the variable numberBaskets for other methods.
    private void readBaskets(String[] baskets) throws IOException {

        numberBaskets = baskets.length;

        allBaskets = new String[numberBaskets];
        for (int i = 0; i < numberBaskets; i++) {
```

```
            allBaskets[i] = baskets[i];
        }

    }

    //this is the main function which finds all frequent itemsets, subject
    //to "threshold" in the Array of transactions "baskets".
    public ArrayList<ArrayList<HashSet<Integer>>> aprioriCalculation(
            String[] baskets, Integer threshold) throws IOException {


        //store all transaction into the local array allBaskets
        readBaskets(baskets);

        //set the threshold for the algorithm which defines
        //if an itemset is frequent or not
        this.threshold = threshold;

        //firstPass() runs over the transactions and counts all
        //single items, if frequent single items are found
        //then it returns true, else false
        boolean itemsetAdded;
        itemsetAdded = firstPass();
        //if frequent single items were found, find candidate pairs
        //in which both items are frequent single items
        if (itemsetAdded) {

            candidateItemsets
        = getAllCandidatesFromPreviousListOfFrequentItems(allFrequentItems
                    .get(0));

        }

        //itemCount defines of which size the itemsets are that
        //we are detecting at the moment. nextPass() runs over
        //all transactions and counts how often each candidate
        //itemset occurs, then it filters out the frequent candidates
        //and adds them to the List allFrequentItems.
        //If frequent candidates were found, candidate itemsets are
        //computed again over the list of frequent itemsets
        //that were found in the last iteration. These steps are
        //repeated until no more candidates were found in the last
        //iteration. Also the loop runs only, if itemsets where found
        //in the first pass (single itemsets).

        while (itemsetAdded && candidateItemsets.size() > 0) {
            itemCount++;
            itemsetAdded = nextPass();

            if (itemsetAdded) {

                if (allFrequentItems.get(itemCount - 1).size() > 0) {
                    candidateItemsets =
        getAllCandidatesFromPreviousListOfFrequentItems(allFrequentItems
                            .get(itemCount - 1));

                }
            }
```

```
        }
        return allFrequentItems;
    }
}
```

Listing 7: implemented A-Priori Algorithm Java Code

6.2.3 WholeFileRecordReader

```
package hadoop_project;
package hadoop_project;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

class WholeFileRecordReader
    extends RecordReader<NullWritable, BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private boolean processed = false;

    private NullWritable key = NullWritable.get();
    private BytesWritable value = new BytesWritable();


    public void initialize(InputSplit inputSplit,
            TaskAttemptContext taskAttemptContext)
                throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) inputSplit;
        this.conf = taskAttemptContext.getConfiguration();
    }

    public boolean nextKeyValue() throws IOException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];

            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);

            FSDataInputStream in = null;
            try {
                in = fs.open(file);
```

```
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

}
```

Listing 8: WholeFileRecordReader Java Code

6.2.4 First Pass MapReduce

```
package hadoop_project;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.io.InputStream;
import java.io.ByteArrayInputStream;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
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;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.mortbay.log.Log;

public class FirstPass {

    //This class is responsible for the first Mapper Task.
    //An initialization the setup method is called then for
    //each data chunk the method "map" is called.
    public static class ChunkMapper extends
            Mapper<Object, BytesWritable, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        private int total_support_threshold;
        private int total_number_transactions;

        //this method is called when a new Mapper Task is initialized.
        //it is responsible to set the parameters total_support_threshold
        //and total_number_transactions by reading their values from the
        //config files.
        public void setup(Context context) throws IOException {
            //get the Hadoop working path
            Path path = context.getWorkingDirectory();
            String path_as_string= path.toString();
```

```
        //set the path to the config file support_threshold.txt
        String threshold_file_path_string =
            path_as_string.concat("/config/support_threshold.txt");
        //set the path to the config file number_transactions.txt
        String transactioncount_file_path_string
            = path_as_string.concat("/config/number_transactions.txt");
        //create Hadoop Path variables for the two config files
        Path threshold_file_path
            = new Path(threshold_file_path_string);
        Path transaction_count_path
            = new Path(transactioncount_file_path_string);

        //for both config files, open them using the Hadoop
        //classes FileSystem and FSDataInputStream, read the
        //parameters from the file and store them locally
        FileSystem fs = FileSystem.get(context.getConfiguration());
        FSDataInputStream in = fs.open(threshold_file_path);
        String line;
        String[] temp = new String[2];
        line = in.readLine();
        in.close();
        temp = line.split(":");
        String threshold_string = temp[1];

        in = fs.open(transaction_count_path);
        line = in.readLine();
        in.close();
        temp = line.split(":");
        String transactioncount_string = temp[1];

        this.total_support_threshold
            = Integer.parseInt(threshold_string);
        this.total_number_transactions
            = Integer.parseInt(transactioncount_string);
    }

    //This method is called for each data chunk.
    //The data chunk is given as parameter "value".
    //The "key" parameter is never used. First
    //the "value" parameter in format BytesWritable is cast to
    //an array of Strings. Then the lower threshold for this mapper
    //is computed according to the total threshold, the total number
    //of transactions and the number of transactions in the chunk.
    //Finally Apriori is processed on the chunk with the lower
    //threshold as parameter which gives all itemsets that are frequent
    //in the chunk. These itemsets are produces as key-value
    //pairs in format <itemset,1>
    public void map(Object key, BytesWritable value, Context context)
            throws IOException, InterruptedException {

        String str;
        int size = value.getLength();
        InputStream is = null;
        byte[] b = new byte[size];
        is = new ByteArrayInputStream(value.getBytes());
```

```
            is.read(b);
            str = new String(b);
            String[] baskets = str.split("\n");
            is.close();
            //compute chunk threshold
            //if not a normal number round down
            int threshold =
                (int)(((float)baskets.length/total_number_transactions)
                    *total_support_threshold);
            Apriori apriori = new Apriori();

            ArrayList<ArrayList<HashSet<Integer>>> frequentItemsets
                    = apriori
                    .aprioriCalculation(baskets, threshold);

            for (ArrayList<HashSet<Integer>> itemsetSize:frequentItemsets){
                for (HashSet<Integer> itemset : itemsetSize) {
                    Log.info(itemset.toString());
                    word.set(itemset.toString());
                    context.write(word, one);
                }
            }

        }
    }

    //This class is responsible for the first Reduce Task
    //The variable result just represents a "1" for the
    //output key-value pair. Each call of reduce gets
    //an itemset in the parameter "key" which is produces as
    //as candidate itemset.
    public static class ChunkReducer extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

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

            result.set(1);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "first pass");
        job.setJarByClass(FirstPass.class);
        //Here we set the implemented InputFormat which
        //assures that each mapper gets a whole data chunk
        //to process
        job.setInputFormatClass(WholeFileInputFormat.class);
        //set the Mapper class
        job.setMapperClass(ChunkMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        //set the Reducer Class
```

```
        job.setReducerClass(ChunkReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        //set the input and output format
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //run the first pass of MapReduce.
        job.waitForCompletion(true);
    }
}
```

Listing 9: First Pass MapReduce Java Code

6.2.5 Second Pass MapReduce

```
package hadoop_project;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
import java.io.InputStream;
import java.io.ByteArrayInputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IntWritable;
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;
import org.mortbay.log.Log;

public class SecondPass {

    //This class is responsible for running the second Mapper Task
    public static class SecondPassMapper extends
            Mapper<Object, BytesWritable, Text, IntWritable> {

        private static IntWritable count;
        private Text word = new Text();
        public ArrayList<HashSet<Integer>> candidateItemsets
            = new ArrayList<HashSet<Integer>>();

        //this method is called when the second Mapper Task is
        //initialized. It reads in the candidates file which was
        //output by the first pass of MapReduce. The candidates
        //are stored in the class variable candidateItemsets.
        public void setup(Context context) throws IOException {

            //Initialize the file reader for the candidate file
```

```
        Path path = context.getWorkingDirectory();
        String firstPassOutputFile = path.toString();
        firstPassOutputFile = firstPassOutputFile
                .concat("/temp_candidates/part-r-00000");
        FileSystem fs = FileSystem.get(context.getConfiguration());
        Path inFile = new Path(firstPassOutputFile);
        FSDataInputStream in = fs.open(inFile);

        String line;
        String[] temp = new String[2];
        String tempString;
        String[] itemset;
        HashSet<Integer> tempSet;
        //read the candidate file line by line
        while ((line = in.readLine()) != null) {

            //split each line by "tab"
            //temp[0] will contain an itemset in format [1,4,6,12]
            //temp[1] will contain "1" which is never used
            temp = line.split("\\t");

            //tempString transforms [1,4,6,12] into 1,4,6,12
            tempString = temp[0].substring(1, temp[0].length());
            tempString
                = tempString.substring(0, tempString.length() - 1);

            //itemset gets the single items of tempString
            //tempSet gets the items as integers
            itemset = tempString.split(", ");
            tempSet = new HashSet<Integer>();
            //each item is transformed to an integer
            //and added to tempSet
            for (String item : itemset) {
                tempSet.add(Integer.parseInt(item));
            }
            //finally tempSet is added to the class variable
            //candidateItemsets
            candidateItemsets.add(tempSet);

        }
        in.close();
    }

    //one call to this map method is made for each data chunk
    //the parameter "value" contains the data chunk. "value" is
    //transformed into an Array of Strings. Each map method keeps
    //a counter for each candidate which is initialized with all 0's.
    //Then we go over all transactions in the data chunk. For each
    //transaction we go over all candidates and check whether all
    //items of the candidate are in the transaction basket. If
    //so, we increase the count for that candidate. Finally we produce
    //an itemset-count pair for all candidates.
    public void map(Object key, BytesWritable value, Context context)
            throws IOException, InterruptedException {

        //used to store a single transaction
```

```
HashSet<Integer> currentBasket = new HashSet<Integer>();
//stores the candidate counts
HashMap<HashSet<Integer>, Integer> candidateCounts
     = new HashMap<HashSet<Integer>, Integer>();

//initialize counts
for (HashSet<Integer> candidate : candidateItemsets) {
    candidateCounts.put(candidate, 0);
}

//transform "value" to String Array.
String str;
int size = value.getLength();
InputStream is = null;
byte[] b = new byte[size];
is = new ByteArrayInputStream(value.getBytes());
is.read(b);
str = new String(b);
is.close();

String[] baskets = str.split("\n");
Log.info(baskets.length + " baskets handed to this map task");

//contains all single items of one transaction
String[] itemsAsString;

//go over all baskets in data chunk
for (String basket : baskets) {

    //transform the transaction into
    //HashSet<Integer>. currentBasket
    //contains the transaction afterwards
    itemsAsString = basket.split(" ");
    currentBasket = new HashSet<Integer>();
    for (String item : itemsAsString) {
        currentBasket.add(Integer.parseInt(item));
    }

    //for all candidates check if the candidate occurs
    //in the current basket
    boolean basketContainsCandidate = true;
    for (HashSet<Integer> candidate : candidateItemsets) {

        basketContainsCandidate = true;
        for (Integer item : candidate) {

            if (!currentBasket.contains(item)) {
                basketContainsCandidate = false;
                break;
            }

        }
        if (basketContainsCandidate) {
            candidateCounts.put(candidate,
                    candidateCounts.get(candidate) + 1);
        }

    }
```

```
            }

            //produce all candidate counts of the data chunk
            Set<HashSet<Integer>> keys = candidateCounts.keySet();

            for (HashSet<Integer> key2 : keys) {
                word.set(key2.toString());
                count =
                    new IntWritable(candidateCounts.get(key2).intValue());
                context.write(word, count);
            }
        }
    }

    //This class is responsible for running the second Reducer Task.
    public static class SecondPassReducer extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();
        private int threshold = 0;

        //this method is called at the initialization of the Reducer Task
        //it just reads in the threshold for the complete list of
        //transactions which is used in the following calls to reduce.
        public void setup(Context context) throws IOException {

            Path path = context.getWorkingDirectory();
            String pathAsString = path.toString();
            pathAsString = pathAsString
                    .concat("/config/threshold_whole_file.txt");
            FileSystem fs = FileSystem.get(context.getConfiguration());
            Path inFile = new Path(pathAsString);
            FSDataInputStream in = fs.open(inFile);
            String line;
            String[] temp = new String[2];
            String thresholdAsString = null;
            while ((line = in.readLine()) != null) {
                temp = line.split(":");
                thresholdAsString = temp[1];
            }
            in.close();
            threshold = Integer.parseInt(thresholdAsString);

        }

        //This method is responsible for running the second Reducer Task
        //each call to reduce gets a key-value pair, where key
        //contains an itemset and value is a list of all counts for
        //that itemset from all previous mappers. The reduce method
        //sums up the values in that list and checks if the sum is
        //greater than the threshold. If so, the itemset is produced
        //in the format <itemset, sum>.
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws
                     IOException, InterruptedException {
            int sum = 0;
```

```
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            if (sum >= threshold)
                context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(SecondPass.class);
        //Here we set the implemented InputFormat which
        //assures that each mapper gets a whole data chunk
        //to process
        job.setInputFormatClass(WholeFileInputFormat.class);
        //set the Mapper class
        job.setMapperClass(SecondPassMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        //set the Reducer class
        job.setReducerClass(SecondPassReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        //set input path for data chunks
        FileInputFormat.addInputPath(job, new Path(args[0]));
        //set output path for file of frequent itemsets
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);

    }

}
```

Listing 10: Second Pass MapReduce Java Code

6.2.6 Postprocessing

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
```

```
import java.util.Set;

public class Main {

    //customComparator which allows to compare two HashMaps
    //The Comparator compares the two hashmaps by their
    //value part.
    public static class CustomComparator implements
            Comparator<HashMap<Set<Integer>,Integer>> {
        @Override
        public int compare(HashMap<Set<Integer>,Integer> o1,
                           HashMap<Set<Integer>,Integer> o2) {
            Set<Set<Integer>> itemset1 = o1.keySet();
            Set<Set<Integer>> itemset2 = o2.keySet();

            Iterator<Set<Integer>> iterator1 = itemset1.iterator();
            Iterator<Set<Integer>> iterator2 = itemset2.iterator();

            return o2.get(iterator2.next()).
                    compareTo(o1.get(iterator1.next()));
        }
    }

    //returns the total number of sets
    //in filepath
    public static int get_number_of_sets(String filepath){

        int lineCounter=0;
        try {
            FileReader fr = new FileReader(filepath);
            BufferedReader textReader = new BufferedReader(fr);
            while (textReader.readLine()!=null){
                lineCounter++;
            }
            textReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return lineCounter;
    }

    //returns an outputwriter for "output_path"
    public static PrintWriter get_output_writer(String output_path){

        PrintWriter writer=null;
        try {
            writer = new PrintWriter(output_path, "UTF-8");
        } catch (FileNotFoundException e) {

            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {

            e.printStackTrace();
        }

        return writer;

    }
```

```
//returns a file reader for "path_to_file"
public static BufferedReader get_file_reader(String path_to_file){
    BufferedReader textReader=null;
    try {
        FileReader fr = new FileReader(path_to_file);
        textReader = new BufferedReader(fr);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return textReader;

}

//reads in all itemset count pairs and runs "sort" on the list
//with the customComparator (see above).
public static ArrayList<HashMap<Set<Integer>,Integer>>
        get_itemsets_count_pairs(String filepath){

    ArrayList<HashMap<Set<Integer>,Integer>> itemsets_count_pairs
             = new ArrayList<HashMap<Set<Integer>,Integer>>();


    BufferedReader reader = get_file_reader(filepath);
    String line;
    String[]  itemset_and_count = new String[2];
    String itemsetstring;
    String[] items;
    Set<Integer> itemsetints;
    Integer count;
    HashMap<Set<Integer>, Integer> itemset_count_pair;

    try {
        while ((line = reader.readLine())!=null){

            itemset_and_count = line.split("\\t");

            itemsetstring = itemset_and_count[0].
                    substring(1, itemset_and_count[0].length());
            itemsetstring = itemsetstring.
                    substring(0, itemsetstring.length() - 1);
            items = itemsetstring.split(", ");

            itemsetints = new HashSet<Integer>();
            for (String item : items) {
                itemsetints.add(Integer.parseInt(item));
            }

            count = Integer.parseInt(itemset_and_count[1]);

            itemset_count_pair = new HashMap<Set<Integer>, Integer>();
            itemset_count_pair.put(itemsetints, count);

            itemsets_count_pairs.add(itemset_count_pair);


        }
    } catch (IOException e) {
        e.printStackTrace();
```

```
        }

        Collections.sort(itemsets_count_pairs, new CustomComparator());

        return itemsets_count_pairs;

    }
    //this method first runs over the file "filename" and counts the
    //number of lines. This value is written to "outputfile" as a first
    //line.Then it reads in all lines from "filename" and casts them to
    //ArrayList<HashMap<Set<Integer>,Integer>>. get_itemsets_count
    //returns the list in the desired, sorted order. This list is
    //written to "outputfile" line by line.
    public static void sort_and_print(String filename, String outputfile){

        PrintWriter outputwriter = get_output_writer(outputfile);
        Integer number_sets = get_number_of_sets(filename);
        outputwriter.println(number_sets.toString());
        ArrayList<HashMap<Set<Integer>,Integer>> itemsets_count_pairs
                = get_itemsets_count_pairs(filename);

        for (HashMap<Set<Integer>,Integer> itemset_count_pair
                 : itemsets_count_pairs){

            Set<Set<Integer>> itemset = itemset_count_pair.keySet();
            Iterator<Set<Integer>> iterator = itemset.iterator();

            Set<Integer> temp = iterator.next();


            outputwriter.println(temp+"
                    ("+itemset_count_pair.get(temp)+")");

        }

        outputwriter.close();

    }

    public static void main(String[] args){

        sort_and_print(args[0], "frequent_itemsets");

    }
}
```

Listing 11: Postprocessing