1 Introduction

2 High Level Approach

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 of 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. Each mappers task is to find all itemsets that occur at least in *ps* of the baskets in the processed chunk. To get these itemsets, which are frequent for a data chunk the A-Priori Algorithm was utilized which is described in more detail in 1.1.1. The pseudocode given in Algorithm 1 describes the function of the first set of mappers.

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 *ps*) in the data chunk that is given as input by using the A-Priori Algorithm. For this step an own implementation of the A-Priori Algorithm was

Algorithm 1 First Map

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

utilized. From a top level perspective the key function of the A-Priori algorithm is listed in Algorithm 2. Appendix 6.2 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 data chunk that contains all items in the itemset *candidate*. Algorithm 4 gives pseudocode for the mappers that run in the second Map Task.

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
```

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.

Algorithm 5 Second Reduce Task

```
1: function REDUCE(Set<Integer> itemset, List<Integer> count_list, Integer support_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
```

3 Test Data

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). 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. The complete code is listed in Appendix 6.2.

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

int total_number_of_transactions =
            get_number_transactions(file_path);
int transactions_per_chunk = total_number_of_transactions/k;

/*
    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 by the subscript

copy_ chunks_ to_ hadoop.sh path_ from path_ to

where the path names *path_ from* and *path_ to* are generated by the calling parent script *find_ frequent_ itemsets.sh*. Before executing the two MapReduce passes it is necessary to set the lower threshold for the mappers in the first pass and the total threshold for the reducers in the second pass. This is done by the two subscripts

set_ chunk_ threshold.sh p
set_ total_ threshold.sh s

which set the vaules p and s in configuration files on the hadoop site, where they are readable by the mapper and reducer classes (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 process a data chunk and find frequent itemsets using A-Priori. 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. 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 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 the Mapper Task in hadoop were:

- setting the threshold parameter for the data chunks in the Mapper class of the hadoop job. Because of the distributed architecture of the hadoop framework the threshold parameter can not simply be set as a static variable. Passing the threshold parameter to the mapper functions 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 threshold parameter

As mentioned in 4.1 the threshold parameters for the mappers are written to configuration files on the hadoop site. The hadoop class *Mapper* from which the implemented Mapper class inherits contains a *setup* method which was utilized to read the parameter from the configuration file. By default the *setup* method is called at the beginning of a Mapper (Reducer) Task and is capable of setting the parameter as a class variable and passing

it to each run of the *map* (*reduce*) method. Assuming that the threshold parameter for a data chunk is written to the file *config/threshold_ chunk.txt* on the hadoop machine in the format *threshold:p* it can be obtained as listed in Listing 2.

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

private int threshold;
public void setup(Context context) throws IOException {
    Path path = context.getWorkingDirectory();
    String path_as_string= path.toString();
    String config_file_path =
        path_as_string.concat("/config/threshold_chunk.txt");
    Path configFilePath = new Path(config_file_path);
    FileSystem fs = FileSystem.get(context.getConfiguration());
    FSDataInputStream in = fs.open(configFilePath);
    String line;
    String[] temp = new String[2];
    String thresholdString = null;
    while ((line = in.readLine()) != null) {
        temp = line.split(":");
        thresholdString = temp[1];
    }
    in.close();
    this.threshold = Integer.parseInt(thresholdString);
}
/*
actual code for the map function goes here (see Appendix)
*/
}
```

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 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. 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. To be exact each line is in the format $[itemset, 1]$ as a result of the key-value format of the first MapReduce run. 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. For example the first couple of lines of the candidate itemset file could look like this:

$$[[23, 42, 17], 1]$$
$$[[1, 9, 109, 22, 3, 16,], 1]$$
$$[[4], 1]$$

In order for the second set of mappers to read the candidate itemset file it was copied to a temporary directory by the parent script *find_frequent_itemsets.sh* before the hadoop job of the second pass is started. The default name of this file is *part-r-00000*. We utilized the inherited *setup* method of the Mapper class to read the candidate itemsets. Listing 4 lists the main loop to obtain the candidates. The complete code is listed in the Appendix.

```
public static class SecondPassMapper extends
            Mapper<Object, BytesWritable, Text, IntWritable> {
public ArrayList<Set<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 Set<Integer>();
    for (String item : itemset) {
        tempSet.add(Integer.parseInt(item));
    }
    candidateItemsets.add(tempSet);
}
in.close();
}
```

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. The last step in the pipeline is to transform the Hadoop generated format into the desired output format. Therefore we need to read in the complete list of frequent itemsets and sort them subject to their counts. This was done by a simple run of bubble/quick/... sort.

5 Evaluation

The pipeline was evaluated on the given test data file example.dat for x different choices for the number of data chunks and y different support thresholds. The total running time of each choice of parameters is listed in table t. Interestingly for the parameter choices x,y,z the mapper functions in the first pass failed due to timeouts.

6 Appendix

6.1 Shell Scripts

6.2 Java Modules