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. 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.

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

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
```

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 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

```
function REDUCER(Set<Integer> itemset, Value value)
    produce(itemset, 1)
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.

Algorithm 4 Second Map Task

```
function MAPPER(String data_chunk)
    List<Set<Integer>> candidate_itemsets = READ_CANDIDATE_LIST()
    List<String> basket_list = SPLIT_INTO_BASKETS(data_chunk)
    HashMap<Set<Integer>, Integer> candidate_counts = new HashMap<Set<Integer>, Integer>
    for each String basket ∈ basket_list do
        Set<Integer> temp_basket = CONVERT_TO_SET(basket)
        for each Set<Integer>> candidate ∈ candidate_itemsets do
            Boolean basket_contains_candidate = true
            for each Integer item ∈ candidate do
                if NOT temp_basket.CONTAINS(item) then
                    basket_contains_candidate = false
                    break
                end if
            end for
            if basket_contains_candidate = true then
                candidate_counts(candidate)=candidate_counts(candidate)+1
            end if
        end for
    end for
    for each candidate ∈ candidate_itemsets do
        produce(candidate, candidate_counts(candidate))
    end for
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 (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.

```java
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 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*. 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 accessable 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 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 in Hadoop were:

- setting the threshold parameter for the data chunks in the Mapper class of the Hadoop job. Each mapper needs to calculate the lower threshold dynamically according to the total threshold and the total number of transactions. 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 threshold parameter

As mentioned in 4.1 the support threshold and the total number of transactions were written to Hadoop accessable 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];

    FSDataInputStream 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.transaction_count = 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 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. 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. 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 the Appendix.

```
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.

5 Evaluation

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

6.2 Java Modules