2017

A PARALLEL IMPLEMENTATION OF APRIORI ALGORITHM FOR MINING FREQUENT ITEMSETS IN HADOOP MAPREDUCE FRAMEWORK

Gokarna Neupane

A PARALLEL IMPLEMENTATION OF APRIORI ALGORITHM FOR MINING FREQUENT ITEMSETS IN HADOOP MAPREDUCE FRAMEWORK

by

GOKARNA NEUPANE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree, of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2017

Acknowledgements

The completion of this project would not have been possible without the constant help and support of various individuals. I would like to express a great deal of gratitude to all the people involved in motivating and guiding me to accomplish the targets of my research.

Firstly, I am very grateful to Dr. Leonidas Fegaras for agreeing to supervise my project and helping me in every step with my queries and issues. He provided me with vital comments and remarks to further improve my work and complete the thesis in time. Additionally, I would like to thank my committee members Dr. Ramez Elmasri and Mr. David Levine.

I would like to thank my friends for constant encouragement and moral support to achieve my goals. I would like to mention Achyut Paudel, Gaurav Thapa, Nigesh Shakya and Ishwor Timilsina for their continued support.

Lastly, I would like to thank Ms. Camille Costabile for helping me in the process of final report submission and presentation.

April 20, 2017

Abstract

A PARALLEL IMPLEMENTATION OF APRIORI ALGORITHM FOR MINING FREQUENT ITEMSETS IN HADOOP MAPREDUCE FRAMEWORK

GOKARNA NEUPANE, MS

The University of Texas at Arlington, 2017

Supervising Professor: Dr. Leonidas Fegaras

Committee Member: Dr. Ramez Elmasri

Committee Member: Mr. David Levine

With explosive growth of data in past few years, discovering previously unknown, frequent patterns within the huge transactional data sets has been one of the most challenging and ventured fields in data mining. Apriori algorithm is widely used and one of the most researched field for frequent pattern mining. The exponential increase in the size of the input data has adverse effect on the efficiency of the traditional or centralized implementation of this algorithm. Thus, various distributed Frequent Itemset Mining(FIM) algorithms have been developed. MapReduce is a programming framework that allows the processing of large datasets with a distributed algorithm over a distributed cluster. During this research, I have implemented a parallel Apriori algorithm in Hadoop MapReduce framework with large volumes of input data and generate frequent patterns based on user defined parameters. I have implemented hash tree data structure to represent the candidate itemsets which aids in faster search for those candidates within a transaction. These experiments were conducted in real-life datasets and varying parameters. Based on various evaluations, the proposed algorithm turns out to be scalable and efficient method to generate frequent item-sets from a large dataset over a distributed network.

Table of Contents

Acknowledgements iii

Abstract iv

List of Illustrations viii

Chapter 1 Introduction 1

Chapter 2 Apriori Algorithm 3

2.1 Problem Statement 3

2.2 The Apriori Algorithm 3

2.3 Apache Hadoop MapReduce Framework 5

2.4 Parallel Implementation of Apriori Algorithm 8

2.4.1 First Mapper Setup 11

2.4.2 Rest Mapper Setup 12

2.4.3 Combiner Setup 13

2.4.4 Reducer Setup 14

2.5 Systems Used for Evaluation 15

2.6 Input Data 15

2.7 Output Data 17

2.8 Bottleneck 18

2.9 Execution 19

Chapter 3 Performance Evaluation 21

3.1 Scaleup 21

3.2 Size-up 22

3.3 Speed-up 23

3.4 Threshold 24

Chapter 4 Source Code 25

Chapter 5 Application and Future Work 43

5.1 Application 43

5.2 Future Work 44

Chapter 6 Conclusion 45

References 46

List of Illustrations

Figure 1: Hadoop MapReduce Framework 7

Figure 2: Parallel Implementation of Apriori Algorithm 10

Figure 3: Input Data Format 17

Figure 4: Frequent 3-itemsets 17

Figure 5: Frequent 2-itemsets 18

Figure 6: Frequent 1-itemsets 18

Figure 7: Execution of 1.48GB Webdocs Data 19

Figure 8: Execution of 1.04GB Webdocs data 20

Figure 9: Execution of 1.93GB Webdocs data 20

Figure 10: Scale-up 21

Figure 11: Size-up 22

Figure 12: Speed-up 23

Figure 13: Varying threshold 24

Chapter 1

Introduction

Today, there are countless number of businesses running side by side competing with each other for success. Most of these might be from different regions, design different products and target different customers but all share one common outcome which is data. With today's consumption rate, there must be huge amount of data being generated on daily basis for all organizations. The proper processing and analysis of this data to spawn useful information is vital in efficient operation of these businesses. So, data mining plays key role in generation of such valuable information. In addition to top business organizations, various scientific researches are known to adopt data mining techniques to achieve their targets.

Frequent Itemset Mining(FIM) is a data mining technique used to explore interesting connections within a huge transactional data. FIM aims to find information based on how frequent the interesting events occur in the database[1]. The most typical application of FIM is in Market Basket Analysis(MBA) which analyzes the purchasing behaviour of customers[3]. For example, in Consumer Package Goods(CPG) industry, FIM is used to identify patterns in customer purchases, mainly, which items tend to be bought together[2].

FIM is one of the most researched field in data mining and various algorithms have been developed and studied for this purpose. With the astronomical increase in the size of data, the traditional centralized FIM algorithms failed to prove effeciency regardless of any strategy. The processing and storage capacity of a single machine was not enough to efficiently generate frequent item sets from large volume of data. Thus, parallel versions of these algorithms were developed that ran in a distributed cluster which appeared to improve the mining performance than the centralized version of the algorithm. Even though the mining performances were improved, distributed computation added overhead of managing the distributed system as described in [3]. These issues can be overcome by the MapReduce framework developed by Google [3]. MapReduce is a scalable and highly efficient distributed programming model for processing and analyzing big data.

During this research, I have implemented the parallel Apriori algorithm for mining frequent item-sets on Hadoop MapReduce Framework and reviewed the performance with varying user defined parameters based on real-life dataset. In addition, I have highlighted the limitations of the parallel Apriori algorithm with a discussion of future improvements that can be brought in this field.

Chapter 2

Apriori Algorithm

2.1 Problem Statement

The problem of mining frequent itemsets over basket data was introduced by R. Agrawal in [4]. As described in [5], the problem can be formally stated as follows. Let T = {T1, T2, T3, ..., Tn} be a transaction database containing n transactions. Each transaction Ti = {i1, i2, i3, ..., im} contains a set of items from I = {i1, i2, i3, ..., ip} where p≥m such that Ti ⊆ I. An itemset X with k items from I is called a k-itemset. A transaction Ti contains an itemset X if and only if X ⊆ Ti. The support of an itemset X in T denoted as supp(X) indicates the number of transactions in TDB containing X. An itemset is frequent if its support, supp(X) is greater than a support threshold called minimum support.

2.2 The Apriori Algorithm

Apriori is an influential algorithm for generating frequent item-sets from a transactional database. It uses a bottom-up approach where frequent subsets are extended one at a time and those item-sets satisfying the minimum support threshold and Apriori property are kept and rest are pruned [6]. The Apriori property states that, “Any subset of a frequent item-set must also be frequent”.

For example, let us suppose a transactional database D with n items and m transactions. To extract k-frequent item-sets, this algorithm scans the transaction database k times. This algorithm uses a pre-defined minimum support threshold to filter out the frequent item-sets. Firstly, the algorithm scans the database to generate frequent 1 item set. For rest of the process, the algorithm seeks to join two k-1 frequent item-sets to generate possible k item-sets combination. Since determining the support for every possible combination of the previous collection of item-sets is expensive, this method prunes all the item-sets which does not satisfy the Apriori property. The item-sets obtained after pruning are called candidate item-sets. Then the algorithm computes the support of each candidate item-sets and those having support lower than given threshold are discarded and rest are extracted as frequent item-sets. The algorithm ends when there are no candidate item-sets.

Algorithm (Serial Apriori)**:**

D = Transactional database

t = A transaction in D

min_support = Minimum threshold support

L_k = Frequent k item-sets

C_k = Candidate k item-sets

1. Calculate L_1 from the database from the first scan.

2. For k>2, do.

3. Join L_{k-1} with itself to create possible k item-sets.

4. Calculate C_k by pruning the item-sets not satisfying the Apriori property.

5. For each transaction t in the database D, do.

6. Calculate and increase the count of all the candidates C_k present in t.

7. Extract L_k from the candidate sets which satisfy the minimum support threshold.

8. End do.

9. Increase k to k+1 and go to step 2 until C_k = ø.

2.3 Apache Hadoop MapReduce Framework

Hadoop is a large-scale distributed batch processing infrastructure for parallel processing of big data on large cluster of commodity computers [7]. Hadoop is an open source project of Apache [8] which implemented Google's File System [9] as Hadoop Distributed File System (HDFS) and Google's MapReduce [10] as Hadoop MapReduce programming model.

Hadoop Distributed File System (HDFS) is distributed file system that holds a large volume of data in terabytes or petabytes scale and provides fast and scalable access to such data [7]. It stores files in a replicated manner across different machine to provide fault tolerance and high availability during execution of parallel applications [7].

HDFS is a block-structured file system and breaks a file into fixed size blocks (default block size is 64MB) to store across several machines. Hadoop uses two types of machine working in a master-worker fashion, a NameNode as master machine and a number of DataNodes as worker machines. The NameNode assigns block ids to the blocks of a file and stores metadata (file name, permission, replica, location of each block) of the file system in its main memory providing fast access to this information. DataNodes are the individual machines in the clusters which store and retrieve the replicated blocks of multiple files [7].

MapReduce is a parallel programming model designed for parallel processing of large volumes of data by breaking the job into independent tasks across a large number of machines[7]. According to [11], a MapReduce program is composed of a procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a Reduce() method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy.

The model is a specialization of the split-apply-combine strategy for data analysis.[12] It is inspired by the map and reduce functions commonly used in ,[13] although their purpose in the MapReduce framework is not the same as in their original forms.[14] The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine. As such, a single-threaded implementation of MapReduce will usually not be faster than a traditional (non-MapReduce) implementation; any gains are usually only seen with multi-threaded implementations.[15] The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play. Optimizing the communication cost is essential to a good MapReduce algorithm.[15]

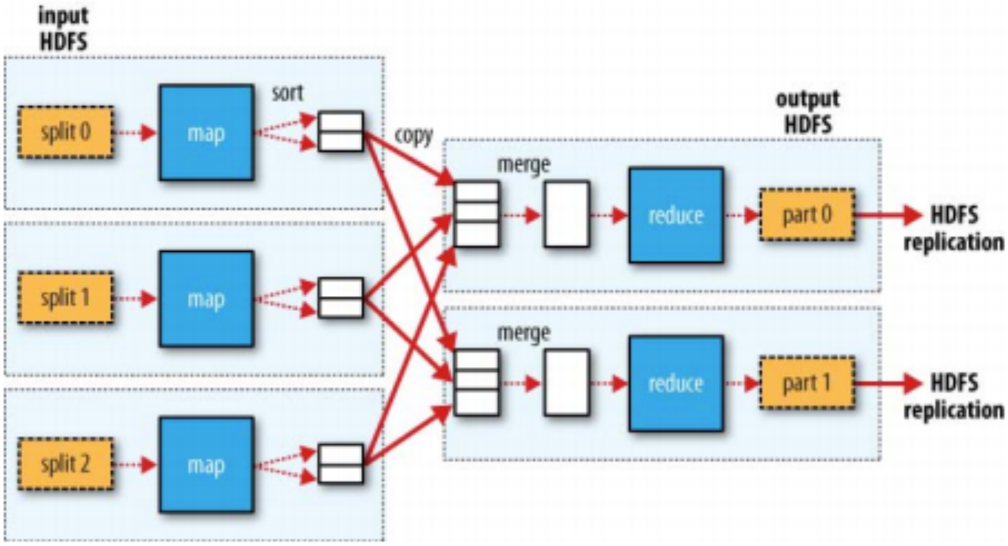


Figure 1: Hadoop MapReduce Framework

Following explains the MapReduce core functionality according to [15]:

- Input Reader: Divides the input file into appropriate size splits (default 64MB) which gets assigned to a Map function for processing.
- Map Function: Maps the file data to smaller, intermediate (key, value) pairs.
- Partition Function: finds the correct reducer: given the key number of reducers, returns the desired reduce node
- Compare Function: input for the reduce is pulled from the Map intermediate output and sorted according to the compare function.
- Reduce Function: takes intermediate values and reduces to a smaller solution handledback to the framework.
- Output writer: writes file output.

2.4 Parallel Implementation of Apriori Algorithm

As discussed earlier, when the size of data increases, the I/O cost of centralized Apriori algorithm increases exponentially because of iterative scan of the large database and high memory consumption. Hence, the need for parallel Apriori algorithm to process huge transactional data. To implement any parallel algorithm on MapReduce framework, it is vital to design map and reduce function and convert the datasets in the form of (key, value) pairs. These maps and reduce functions run on different machines in parallel and produce aggregated output from each reducer. Since, Apriori algorithm is an iterative process, it requires multiple execution of the map-reduce jobs to extract k frequent item-sets. During this research, we implement the parallel Apriori Count Distribution(CD) algorithm to generate k frequent item-sets.

Count Distribution algorithm is categorized under data parallelism where the data is split into various blocks and transmitted to various machines to process in parallel. It reduces the overhead of synchronization and processor communication [3]. According to [3], following is the algorithm that describes the Count Distribution.

D = Original Large Transactional database

D_i = Local data partition

P_i = Local Processor

t = A transaction in D

min_support = Minimum threshold support

L_k = Frequent k item-sets

C_k = Candidate k item-sets

1. Candidate 1-item-sets, C_1 = I, the set of all items.

2. To generate C_k, where k >= 1 join two of $L_{k-1,}$ generating k item-sets and prune these item-sets according to apriori property.

3. To generate L_k, where k >= 1 processor P_i scan the local partition D_i and find the local support counts of candidates in C_k

4. Processor P_i exchange the local counts with other processors to find global counts of candidates C_k.

5. Find the frequent k-item-sets L_k from C_k as L_k = all candidates of C_k with minimum support threshold.

6. Repeat from step 2 until C_k is empty.

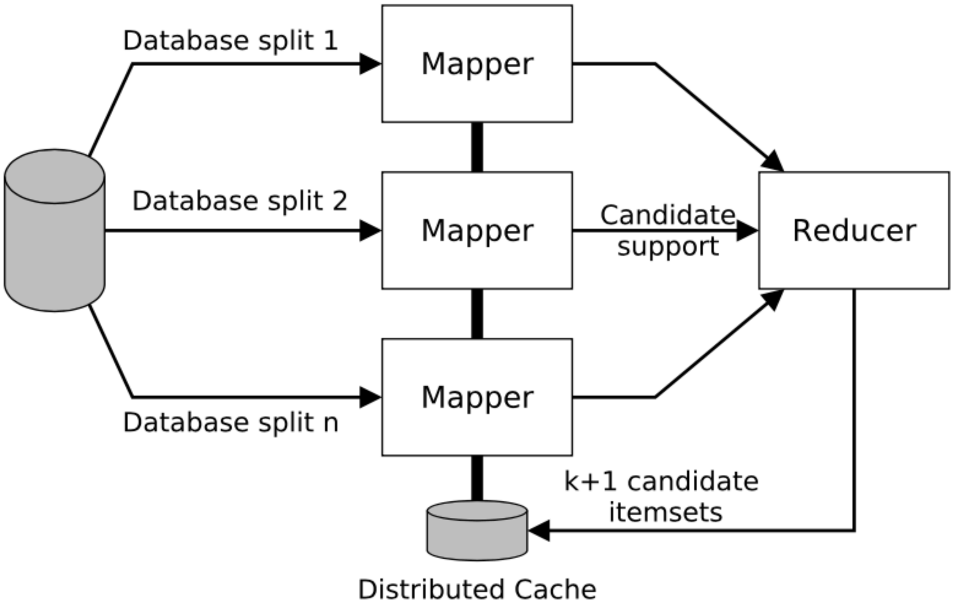


Figure 2: Parallel Implementation of Apriori Algorithm

Figure 2, describes the components of the parallel implementation of the apriori algorithm and shows the data flow in the process. At, first we take the source data which is to be processed and format it in HDFS. We use two separate mappers, one combiner and one reducer for this entire task.

2.4.1 First Mapper Setup

The primary purpose of the first mapper is to find out frequent 1 item-sets. In the first mapper job, the input database D is split into smaller chunks of D_i(default is 64MB) and distributed to i processors. The mapper reads the input line by line and reads each item separated by a delimiter. Then each item is set as key and the value is set to 1. 1 represents the frequency for that item. Thus obtained (key, value) pair is then passed to subsequent combiner / reducer. Following is the pseudo code for the first mapper.

```
•   for each transaction t in the chunk $D_i$ do
    o   for every item I in the transaction t do
        ▪   set key = item and value = 1
        ▪   send (key, value) as output
    o   end for
•   end for
```

2.4.2 Rest Mapper Setup

After the generation of 1 frequent item-sets, to generate rest of the item-sets, the mapper phase adopts different approach.

Starting from the process to generate 2 or more item-sets, each mapper gets chunk of the dataset and L_{k-1} frequent item-sets generated by previous reducers as the distributed cache. Thus, obtained database from the distributed cache is joined with itself to create k item-sets. Now the k item-sets is passed to a pruning function that prunes all the item-sets that does not satisfy the apriori property, i.e. every subset of the item-set must also be frequent. This creates a set of candidates' item-sets.

Now the rest mapper reads all available candidates as distributed cache and counts the presence of each candidate within each transaction in the database chunk. If a candidate is present in the transaction, the mapper writes (candidate, 1) as key-value pair and send it to the combiner. Following is the pseudo code for this mapper:

```
•   $L_{k-1}$ = read from HDFS as distributed cache
•   Join $L_{k-1}$ with itself to create k item-sets
•   Prune thus obtained item-sets to generate candidate k item-sets
        o   for each k item-set calculate all k-1 subsets do
                ▪   check if all subsets present in $L_{k-1}$ obtained from
            distributed cache
                        ▪   if yes, store it as a candidate
                ▪ else, discard the item-set
            ▪ end for
• for each transaction t in $D_i$ do
    o if candidate k item-set present in t, set key = candidate, value = 1
    o output (key, value)
• end for
```

2.4.3 Combiner Setup

A Combiner, also known as a semi-reducer**,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class [16]. The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual reducer task as input [16].

For our algorithm, the combiner class takes the (key, value) pair from the mapper output and aggregates the count of each key and sets the value as the aggregated count then sends the output to the reducer as (key, value) pair. Following is the pseudo code for the combiner class:

```
• for each key do
    o aggregate the count for each key
• end for
• set value = aggregated count
• output new (key, value) pair
```

2.4.4 Reducer Setup

The reducer is common for all iterations. Like the combiner, the reducer class takes the (key, value) pair from the combiner output and again aggregates the count and checks if the count satisfies the minimum support threshold. If yes then the reducer outputs the key and the aggregated count as value and discard the rest nor satisfying the condition. Following is the pseudo code for the reducer class:

```
• for each key do
    o aggregate the count for each key
• end for
• check if the aggregate count of each key is >= minimum support threshold
    o if yes do
```

        - set value = aggregated count
        - output (key, value) pair
    - else, continue

2.5 Systems Used for Evaluation

We have used two different system for the performance evaluation purposes:

- UTA’s Hadoop Cluster:
    - Cluster of 8 Linux Servers
    - Each server has 4 Xeon cores at 3.2 GHz with 4GB Memory

- Personal Computer
    - Single server with 4 cores at 2.5 GHz with 8GB memory.

2.6 Input Data

WebDocs is a huge real-life transactional dataset publicly available to the Data Mining community through the FIMI (Frequent Itemset Mining Dataset) repository. It was built from a spidered collection of about 1.7 million web html documents. It is processed and stemmed to create a transaction dataset with distinct terms appearing within the document as items.

Resulting dataset of size 1.48GB

- Total transaction: 1,692,082
- Total items: 5,267,656
- Max length of a transaction: 71,472

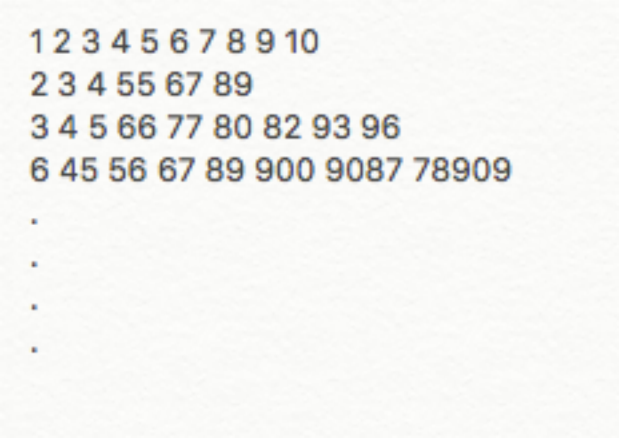
In addition to WebDocs, we created two more transactional datasets using the WebDocs.

WebDocs1GB

- Size: 1.04GB
- Number of Transactions: 1,200,000

WebDocs2GB

- Size: 1.93GB
- Number of Transactions: 2,184,163

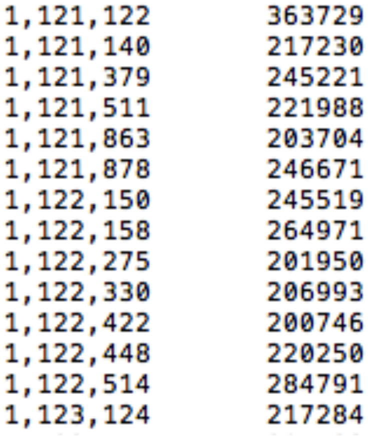


```
1 2 3 4 5 6 7 8 9 10
2 3 4 55 67 89
3 4 5 66 77 80 82 93 96
6 45 56 67 89 900 9087 78909
.
.
.
.
```

Figure 3: Input Data Format

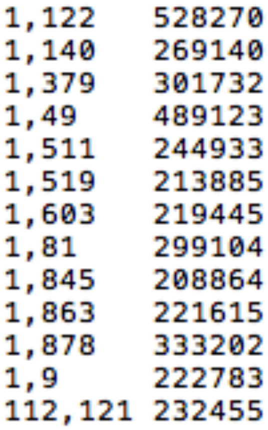
2.7 Output Data

Each iteration of a MapReduce job produces a definite output. In this case after each iteration, the first column of the output represents the frequent item-set combination and the second and the final column represents the support count of those frequent item-sets.



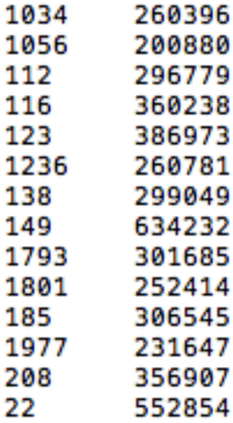
```
1,121,122       363729
1,121,140       217230
1,121,379       245221
1,121,511       221988
1,121,863       203704
1,121,878       246671
1,122,150       245519
1,122,158       264971
1,122,275       201950
1,122,330       206993
1,122,422       200746
1,122,448       220250
1,122,514       284791
1,123,124       217284
```

Figure 4: Frequent 3-itemsets



```
1,122   528270
1,140   269140
1,379   301732
1,49    489123
1,511   244933
1,519   213885
1,603   219445
1,81    299104
1,845   208864
1,863   221615
1,878   333202
1,9     222783
112,121 232455
```

Figure 5: Frequent 2-itemsets



```
1034    260396
1056    200880
112     296779
116     360238
123     386973
1236    260781
138     299049
149     634232
1793    301685
1801    252414
185     306545
1977    231647
208     356907
22      552854
```

Figure 6: Frequent 1-itemsets

Above figures 4, 5 and 6 demonstrate the output format for 3, 2 and 1 item-sets respectively.

2.8 Bottleneck

As the size of the database increases, the number of candidate generated are increased given a certain threshold. Because of increased number of candidate-sets, the number of scans required to generate count of those frequent item-sets increase. This requires excess resources i.e. memory and storage and hence increasing the computational costs.

2.9 Execution

We ran the program in UTA's Hadoop cluster with varying parameters for all three datasets. We used different threshold for all three datasets to evaluate the execution. In all conditions, the results showed that the time required for generation of frequent item-sets increases as the number of candidate item-sets increase. The graphical representation of the execution time against threshold parameter is shown in following figures:

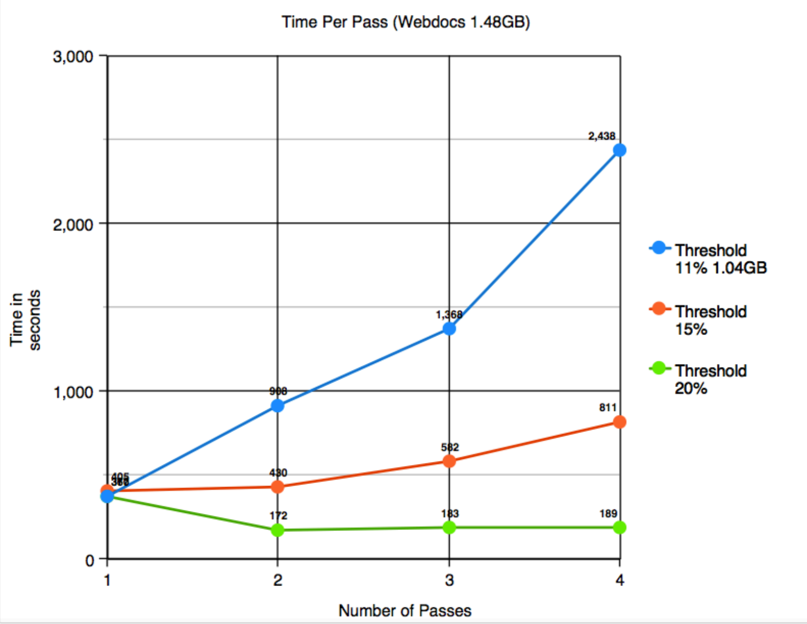


Figure 7: Execution of 1.48GB Webdocs Data

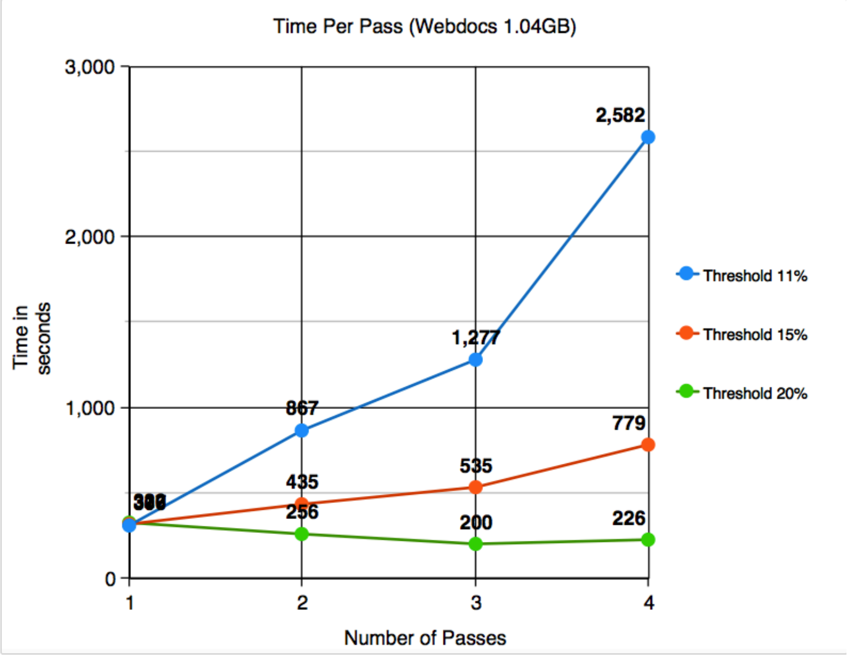


Figure 8: Execution of 1.04GB Webdocs data

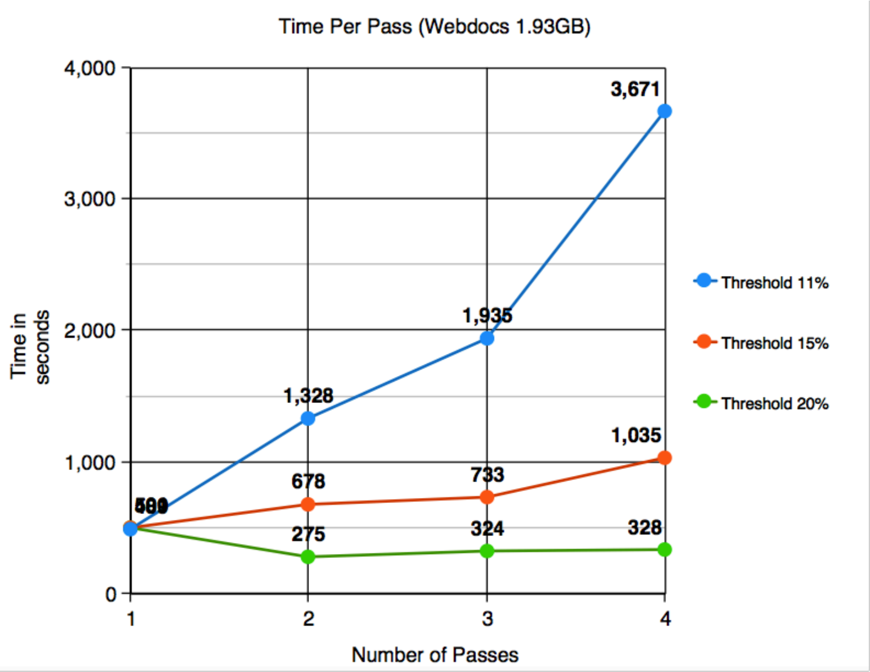


Figure 9: Execution of 1.93GB Webdocs data

Chapter 3

Performance Evaluation

3.1 Scaleup

Evaluates the ability of the algorithm to grow both the system and dataset size. It is defined by ability of a n-times larger system to perform a n-times larger job in similar time. We used a single server system with 4 cores and UTA's Hadoop cluster with 32 cores for this purpose. We took 1/8th of input data to run on single server. We performed scale up performance for 3 different thresholds (11%, 15% and 20%).

The results showed that our algorithm scales well and maintains around 70% and above scalability for the dataset.

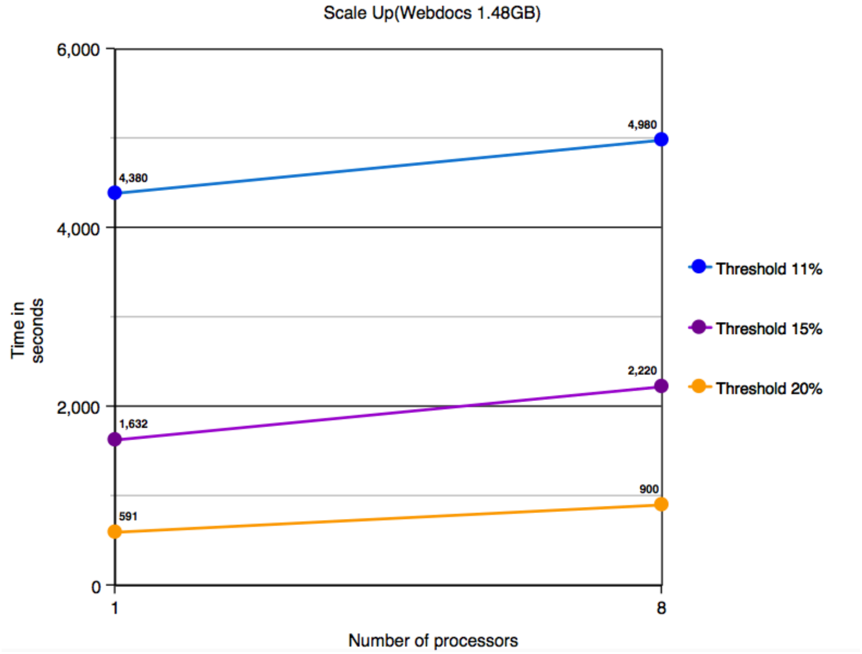


Figure 10: Scale-up

3.2 Size-up

By holding the number of cores in the system constant we grow the size of the dataset. This measure how much longer it takes on a given system, when dataset is n times larger than original dataset.

We tested with three different datasets for three different thresholds.

Our experiment showed that as the threshold increases, the execution time difference drops drastically even though there is significant size difference between the datasets.

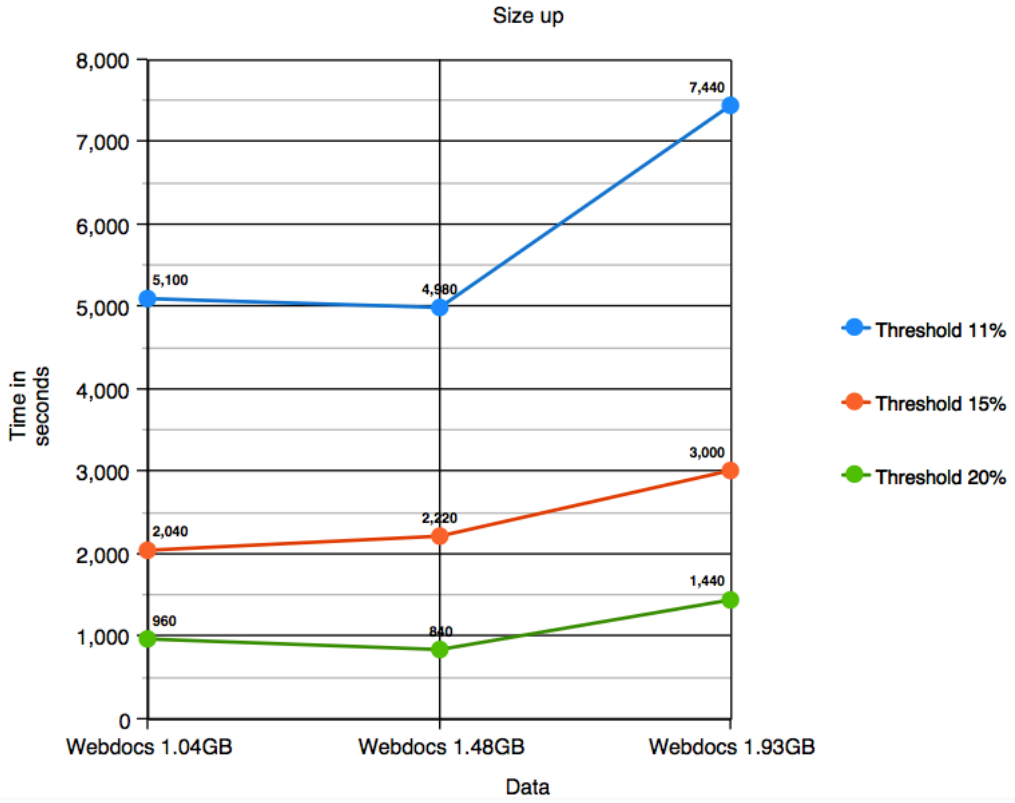


Figure 11: Size-up

3.3 Speed-up

It measures the performance of the algorithm varying the number of cores for parallel execution. For that we varied the number of reducers and kept the dataset constant. We tested for five different number of reducers. The performance improved as we increased the number of reducers. But the progress was stagnant after 4 reducers as the communication cost plays part among numerous processors in increasing the execution time. The following figure illustrates the experiment.

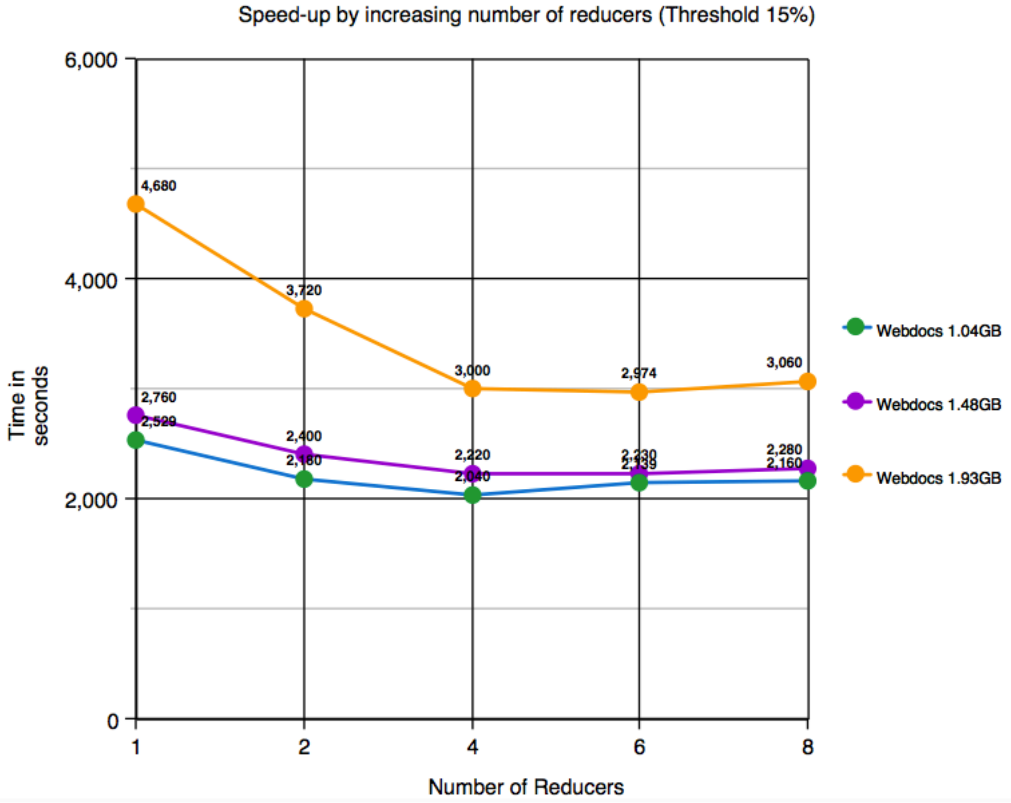


Figure 12: Speed-up

3.4 Threshold

It measures the performance of algorithm for varying minimum support threshold. We used three different threshold values (11%, 15% and 20%). The algorithm was quickest when the threshold was higher. I.e. 20%, and slowest when the threshold was lower. i.e. 11%. Higher the threshold, lower the number of candidate item-sets and hence the faster execution of the program.

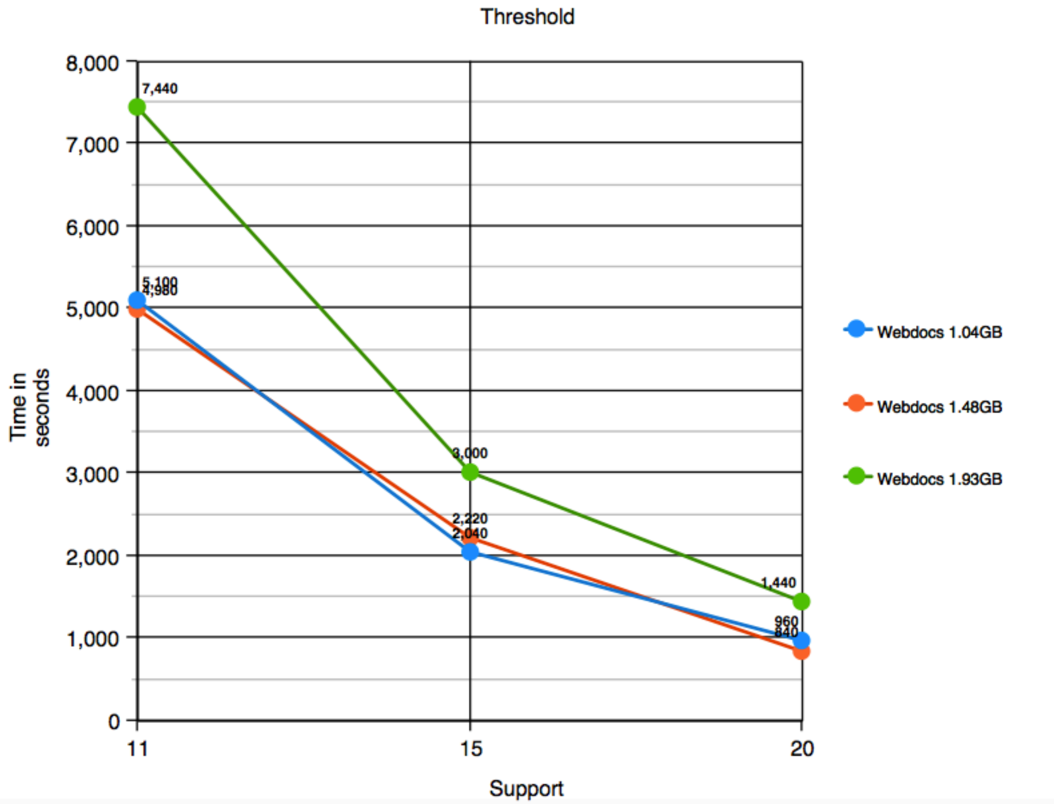


Figure 13: Varying threshold

Chapter 4

Source Code

Driver Class:

```
public static void main(String[] args) throws Exception, IOException, InterruptedException
{

int sys = ToolRunner.run(new myApriori(), args);
System.exit(sys);

}
@Override
public int run(String[] args) throws Exception, IOException, InterruptedException {

String inputF = args[0];
String candiP = args[1];
String outputF = args[2];
Integer minS = Integer.parseInt(args[3]);
long startTime = System.currentTimeMillis();
int numF = 1;
int numOfReducers = 4;
Configuration newConf = new Configuration();
newConf.setInt("minSupport", minS);
newConf.setInt("numF", numF);
```

```
Job aprioriJob = Job.getInstance(newConf, "Apriori");
aprioriJob.setJarByClass(myApriori.class);
aprioriJob.setMapperClass(firstItemSetMapper.class);
aprioriJob.setNumReduceTasks(numOfReducers);
aprioriJob.setReducerClass(firstItemSetReducer.class);
aprioriJob.setOutputKeyClass(Text.class);
aprioriJob.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(aprioriJob, new Path(inputF));
//System.out.println(outputF + numF);
FileOutputFormat.setOutputPath(aprioriJob, new Path(outputF + numF));
boolean isJob1done = (aprioriJob.waitForCompletion(true) ? true : false);


for(int i = 2; i<=4; i++)
                              {

numF = i;
int cacheNum = numF-1;



Configuration confCand = this.getConf();
confCand.setInt("minSupport", minS);
confCand.setInt("numF", numF);
confCand.setInt("numOfReducers", numOfReducers);
```

```
//confName.setBoolean("mapreduce.map.output.compress", true);
//confName.set("mapreduce.map.output.compression.type",
CompressionType.BLOCK.toString());
//confName.setClass("mapreduce.map.output.compression.codec",       GzipCodec.class,
CompressionCodec.class);
//confName.setBoolean("mapreduce.output.fileoutputformat.compress", false);
Job candGen = Job.getInstance(confCand, "candGen"+i);
candGen.setJarByClass(myApriori.class);
candGen.setMapperClass(candMapper.class);
//myAprioriJob.setNumReduceTasks(1);
//myAprioriJob.setNumReduceTasks(numOfReducers);
candGen.setReducerClass(candReducer.class);
candGen.setOutputKeyClass(Text.class);
candGen.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(candGen, new Path(candiP));
FileOutputFormat.setOutputPath(candGen, new Path(outputF + "Candidate" + numF));
for(int j = 0; j<numOfReducers; j++)
                                        {
candGen.addCacheFile(new Path(outputF + cacheNum +"/part-r-0000"+j).toUri());
                                        }
boolean isCandGendone = (candGen.waitForCompletion(true) ? true : false);
Configuration confName = this.getConf();
confName.setInt("minSupport", minS);
confName.setInt("numF", numF);
confName.setInt("numOfReducers", numOfReducers);
```

```
//confName.setBoolean("mapreduce.map.output.compress", true);
 //confName.set("mapreduce.map.output.compression.type",
CompressionType.BLOCK.toString());
//confName.setClass("mapreduce.map.output.compression.codec", GzipCodec.class,
CompressionCodec.class);
 //confName.setBoolean("mapreduce.output.fileoutputformat.compress", false);
Job myAprioriJob = Job.getInstance(confName, "Apriori"+i);
myAprioriJob.setJarByClass(myApriori.class);
myAprioriJob.setMapperClass(restMapper.class);
myAprioriJob.setCombinerClass(combiner.class);
myAprioriJob.setNumReduceTasks(numOfReducers);
myAprioriJob.setReducerClass(firstItemSetReducer.class);
myAprioriJob.setOutputKeyClass(Text.class);
myAprioriJob.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(myAprioriJob, new Path(inputF));
FileOutputFormat.setOutputPath(myAprioriJob, new Path(outputF + numF));
myAprioriJob.addCacheFile(new Path(outputF + "Candidate" + numF +"/part-r-
00000").toUri());

boolean isJobdone = (myAprioriJob.waitForCompletion(true) ? true : false);


                              }


                            return 1;
                    }
```

First Mapper:

```
public static class firstItemSetMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
        }
        }
        }
```

Combiner:

```
public static class combiner extends Reducer < Text, IntWritable, Text, IntWritable > {
          private IntWritable result = new IntWritable();
          public void reduce(Text key, Iterable < IntWritable > values,
             Context context
          ) throws IOException, InterruptedException {
             int sum = 0;
             Integer minSup =
Integer.parseInt(context.getConfiguration().get("minSupport"));
             for (IntWritable val: values) {
               sum += val.get();
             }
             result.set(sum);
             context.write(key, result);
          }
}
```

Reducer:

```
public static class firstItemSetReducer extends Reducer < Text, IntWritable, Text,
IntWritable > {
        private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable < IntWritable > values, Context context) throws
IOException, InterruptedException {
          int sum = 0;
          Integer minSup =
Integer.parseInt(context.getConfiguration().get("minSupport"));
          for (IntWritable val: values) {
            sum += val.get();
          }
          result.set(sum);
```

```
                    if (result.get() >= minSup) {
                        context.write(key, result);
                    }
                }
}
```

Candidate Mapper:

```
public static class candMapper extends Mapper < Object, Text, Text, IntWritable > {
private List < List < Integer >> previousFreqItemsets = new ArrayList < List < Integer >>
();
    private List < List < Integer >> candidates = new ArrayList < List < Integer >> ();;
    private List < Integer > uniqueItemsFromPrev = new ArrayList < Integer > ();
    private final static IntWritable one = new IntWritable(1);
    private static HashMap < String, Integer > prevItemSetsHashMap = new HashMap < >
();
    private static HashMap < String, Integer > twoSetHashMap = new HashMap < > ();

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        Integer itemSetNum = Integer.parseInt(context.getConfiguration().get("numF"));
        Integer                    numberOfReducers                    =
Integer.parseInt(context.getConfiguration().get("numOfReducers"));
        FileSystem fileSystem = FileSystem.get(context.getConfiguration());
        for (int k = 0; k < numberOfReducers; k++) {
            URI mappingFileUri = context.getCacheFiles()[k];
            Path p = new Path(mappingFileUri);
            InputStreamReader inputRead = new InputStreamReader(fileSystem.open(p));
            BufferedReader buffRead = new BufferedReader(inputRead);

            String line = null;
            while ((line = buffRead.readLine()) != null) {
                line = line.trim();
                String[] tokens = line.split("[\\s\\t]+");
                List < Integer > items = new ArrayList < Integer > ();

                if (tokens.length < 2) {
                    items.add(Integer.parseInt(tokens[0]));
                } else {
                    String[] firstToken = tokens[0].split(",");
                    prevItemSetsHashMap.put(tokens[0], 1);
                    for (int i = 0; i < firstToken.length; i++) {
                        String itm = firstToken[i];
                        items.add(Integer.parseInt(itm));
                    }
                }
                previousFreqItemsets.add(items);

            }
        }
```

```
    uniqueItemsFromPrev = uniqueItems.getUniqueItems(previousFreqItemsets);

candidates = getCandidates.getAllCandidates(previousFreqItemsets,
uniqueItemsFromPrev, prevItemSetsHashMap, itemSetNum);
    //System.out.println(candidates);

  }

  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    for (List < Integer > newCand: candidates) {
      String s = "";
      int size = newCand.size();
      for (int x: newCand) {
        if (x != newCand.get(size - 1)) {
          s = s + x + ",";
        } else
          s = s + x;
      }
      Text data = new Text(s);
      context.write(data, one);
    }
  }
}
```

Candidate Reducer:

```
public static class candReducer
extends Reducer < Text, IntWritable, Text, IntWritable > {
    private final static IntWritable one = new IntWritable(1);


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


    }
}
```

Rest Mapper:

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


    private List < List < Integer >> candidates = new ArrayList < List < Integer >> ();;
    private final static IntWritable one = new IntWritable(1);
    htNode root = new htNode();


    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        Integer itemSetNum = Integer.parseInt(context.getConfiguration().get("numF"));
```

```
FileSystem fileSystem = FileSystem.get(context.getConfiguration());


URI mappingFileUri = context.getCacheFiles()[0];
Path p = new Path(mappingFileUri);
InputStreamReader inputRead = new InputStreamReader(fileSystem.open(p));
BufferedReader buffRead = new BufferedReader(inputRead);


String line = null;
while ((line = buffRead.readLine()) != null) {
    line = line.trim();
    String[] tokens = line.split("[\\s\\t]+");
    List < Integer > items = new ArrayList < Integer > ();


    if (tokens.length < 2) {
        items.add(Integer.parseInt(tokens[0]));
    } else {
        String[] firstToken = tokens[0].split(",");
        for (int i = 0; i < firstToken.length; i++) {
            String itm = firstToken[i];
            items.add(Integer.parseInt(itm));
        }
    }
    candidates.add(items);


}
```

```
/*
 * Build a hash tree to store all the candidate itemsets with root as "null"
 * */

for (List < Integer > itemSet: candidates) {
    htNode parent = null;
    htNode child = root;
    for (int i = 0; i < itemSetNum; i++) {
        parent = child;
        HashMap < Integer, htNode > nextNode = child.nextNode;
        Integer item = itemSet.get(i);

        if (nextNode.containsKey(item)) {
            child = nextNode.get(item);
        } else {
            child = new htNode();
            nextNode.put(item, child);
        }

        parent.nextNode = nextNode;
    }

    child.isLeaf = true;
    child.itemset.add(itemSet);
```

```
        }

    }

    public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
        String mline = (value.toString()).trim();
        String[] toks = mline.split("[\\s\\t]+");
        List < Integer > newItems = new ArrayList < Integer > ();
        List < List < Integer >> allCandsInInputLine = new ArrayList < List < Integer >> ();
        Integer itemSetNum = Integer.parseInt(context.getConfiguration().get("numF"));
        HashMap < String, Integer > allSubs = new HashMap < > ();
        SortedSet newItSet = new TreeSet();

        for (int i = 0; i < toks.length; i++) {
            newItSet.add(Integer.parseInt(toks[i]));
        }

        newItems.addAll(newItSet);

        if (newItems.size() >= itemSetNum) {
            allCandsInInputLine = recursiveSearchTree.searchTree(root, 0, newItems);

            for (List < Integer > newCand: allCandsInInputLine) {
                String s = "";
                int size = newCand.size();
                for (int x: newCand) {
                    if (x != newCand.get(size - 1)) {
                        s = s + x + ",";
                    } else
                        s = s + x;
                }
                Text data = new Text(s);
                context.write(data, one);

            }
        }
    }
}
```

Candidate Generation Class:

```
public static List < List < Integer >> getAllCandidates(List < List < Integer >> prevFrequentSets, List < Integer > uniqueSet, HashMap < String, Integer > prevItemSetsHashMap, int itemSetSize) {

List<List<Integer>> newCandidates = new ArrayList<List<Integer>>();
List<List<Integer>> afterPruningCandidates = new ArrayList<List<Integer>>();

  for (List < Integer > itemS: prevFrequentSets) {
        for (int itm: uniqueSet) {
            List < Integer > x = new ArrayList(itemS);
            if (itm > Collections.max(itemS)) {
                x.add(itm);
                newCandidates.add(x);
            }
        }
    }
    if (itemSetSize > 2) {
        afterPruningCandidates = pruneCandidates(newCandidates,
prevItemSetsHashMap, itemSetSize);
        return afterPruningCandidates;
    } else
        return newCandidates;
}

public static List < List < Integer >> pruneCandidates(List < List < Integer >> cand,
HashMap < String, Integer > prevItemSetsHashMap, int itemSetSize) {

    List < List < Integer >> prunedCandidates = new ArrayList < List < Integer >> ();
    for (List < Integer > x: cand) {
        if (ifAllSubsFrequent(x, prevItemSetsHashMap, itemSetSize)) {
            prunedCandidates.add(x);
        } else
            continue;
    }

    return prunedCandidates;

}

public static boolean ifAllSubsFrequent(List < Integer > candSet, HashMap < String,
Integer > prevItemSetsHashMap, int itemSetSize) {

    for (int i = 0; i < itemSetSize; i++) {
        List < Integer > temp = new ArrayList < Integer > ();
        temp.addAll(candSet);
        temp.remove(i);
        int size = temp.size();
        String s = "";
        for (int x: temp) {
            if (x != temp.get(size - 1)) {
                s = s + x + ",";
            } else
                s = s + x;
        }
        if (!prevItemSetsHashMap.containsKey(s)) {
            return false;
        }
    }
    return true;
}
```

Tree Searching Class:

```
public static class recursiveSearchTree {
    public static List < List < Integer >> searchTree(htNode root, int index, List < Integer >
newItems) {
        List < List < Integer >> txCandItemSet = new ArrayList < List < Integer >> ();

        if (root.isLeaf) {
            return root.itemset;
        }

        for (int i = index; i < newItems.size(); i++) {
            int itm = newItems.get(i);
            HashMap < Integer, htNode > nwNode = root.nextNode;

            if (nwNode.containsKey(itm)) {
                htNode temp = nwNode.get(itm);
                List < List < Integer >> newset = searchTree(temp, i + 1, newItems);
                txCandItemSet.addAll(newset);
            }
        }
        return txCandItemSet;
    }
}
```

Class to get unique items from k-1 frequent Itemset:

```
public static class uniqueItems {

    public static List < Integer > getUniqueItems(List < List < Integer >> prevFrequentSets)
{

 List<Integer> unItems = new ArrayList<Integer>();
SortedSet<Integer> temp = new TreeSet<Integer>();

        for (List < Integer > itemS: prevFrequentSets) {
            for (int itm: itemS) {
                temp.add(itm);
            }
        }
        unItems.addAll(temp);
        return unItems;
    }

}
```

Chapter 5

Application and Future Work

5.1 Application

Following are few areas of application for our algorithm:

- Supermarkets:
    - Develop combo offers based on product purchased together.
    - Organize and place associated products nearby inside the store.
    - Control inventory based on product demands and what products sell together.
- Internet Service Providers:
    - Analysis of web browsing patterns.
- Financial Services Company:
    - Analysis of cash, credit and debit card purchases.

In addition to above mentioned examples, this algorithm can be implemented to find frequent patterns in any kind of transactional database.

5.2 Future Work

Multiple scans of the database is one of the major hinderance in better performance in this algorithm. We can work on some way to either decrease or eliminate multiple database scans. In addition to multiple scans of database, another major performance issue of this algorithm is candidate generation. In order to generate candidates of k-itemsets, this algorithm requires to perform a cartesian product of k-1 frequent itemset with itself. It causes in high usage of resources and causing the computational cost to increase. In future, we could work on techniques to eliminate the need for creating candidate item-sets.

Chapter 6

Conclusion

Frequent Itemset Mining (FIM) is one of the most important and ventured field in data mining. Almost every large or small, business or research organizations use various FIM techniques to find frequent patters to aid in extracting knowledge from huge amount to data. Apriori algorithm is one of the primitive and effective algorithm to do so. During this research, we parallelized the algorithm to handle big data over a distributed network and performed various experiments with huge real-life data. These experiments were performed in different environments with varying parameters. We found that our implementation is scalable, efficient and performed better with increase in the database size.

References

[1] Janos Illes and Istvan Vajk, “Performance Evaluation of Apriori Algorithm on a Hadoop”,

[2] Ning Li, Li Zeng, Qing He and Zhongzhi Shi, “Parallel Implementation of Apriori Algorithm Based on MapReduce”

[3] Sudhakar Singh, Rakhi Garg, PK Mishra, “Review of Apriori Based Algorithms on MapReduce Framework”,

[4] R. Agrawal, T. Imielinski, and A. Swami, “Mining Association Rules between Sets of Items in Large Database,”

[5] Yanbin Ye, Chia-Chu Chiang, “A Parallel Apriori Algorithm for Frequent Itemsets Mining”

[6] Dao-I Lin, Zvi M. Kedem, “Pincer-Search: An Efficient Algorithm for Discovering the Maximum Frequent Set”

[7] Yahoo! Hadoop Tutorial,

[8] Apache Hadoop,

[9] S. Ghemawat, H. Gobioff and S. Leung, “The Google File System,” *in ACM SIGOPS Operating Systems Review,* vol. 37, no. 5, pp. 29-43, 2003.

[10] J. Dean, and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters” *in ACM Commun.,* vol. 51, pp. 107-113, 2008.

[11] MapReduce,

[12] Wickham, Hadley (2011). "The split-apply-combine strategy for data analysis". *Journal of Statistical Software*.

[13] Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages." -, by Jeffrey Dean and Sanjay Ghemawat; from Google Research

[14] Lämmel, R. (2008). "Google's Map *Reduce* programming model — Revisited". *Science of Computer Programming*. **70**: 1–30.

[17] Eui-Hong Han, George Karypis, Vipin Kumar. “Scalable Parallel Data Mining for Association rules.”

[18] F. Bodon and L. Ronyai, “Trie: An Alternative Data Structure for Data Mining Algorithms.”