MINING ASSOCIATION RULES IN CLOUD

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Pallavi Roy

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Department/Program: Computer Science/ Software Engineering

August 2012

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Mining Association Rules in Cloud

By

Pallavi Roy

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Juan (Jen) Li

Chair

Dr. Kenneth Magel

Dr. Simone Ludwig

Dr. Samee Khan

Approved:

07-26-2012	Dr. Brian Slator
Date	Department Chair

ABSTRACT

The association rule mining was implemented in Hadoop. An association rule mining helps in finding relation between the items or item sets in the given data. The performance of the algorithm was evaluated by testing it in the cloud (EC2) by increasing the number of nodes in the testing set up. The association rules are developed on the basis of the frequent item set generated from the data. The frequent item set were generated following the Apriori algorithm. As the input data and number of distinct items in the data set is large, lots of space and memory is required, so Hadoop was used, as Hadoop provide parallel, scalable, robust framework in the distributed environment.

ACKNOWLEDGEMENTS

I wish to express my deep sense of respect, and indebtedness to my major adviser, Dr. Jen Juan Li, for her valuable suggestions, and ceaseless encouragement during the project work. I would also like to thank the committee member. I have had a great pleasure in expressing my sincere thanks to my advisory committee members, for their valuable support throughout the project.

Last but not least, words run short to express my heartfelt gratitude to my beloved husband, parents and my family members for their inspiration, encouragements, everlasting blessing, and abundant love for me for the successful completion of this achievement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1. INTRODUCTION

Data mining is the process of analyzing data from different angles and getting useful information from the data. The data mining can help in predicting a trend or value, classifying, categorizing the data, and in finding correlations, patterns from the data set. Data can be mined irrespective of storage format in which it is stored. The data can be stored in in flat files, spreadsheets, database tables, or some other storage format. The storage format is not important, but its applicability to the problem to be solved is more important.

Association rule mining is a kind of data mining process. Association rule mining is done to extract interesting correlations, patterns, associations among items in the transaction database or other data repositories. For example an association rule fruit => milk generated from the transaction database of a grocery store can help in formulating marketing strategy around the rule. Association rules are widely used in various areas such as telecommunication networks, marketing and risk management, and inventory control etc. Many companies and firms keep large quantities of their day to day transaction data. These data could be analyzed to learn the purchasing trend of the customer. Such valuable insight can be used to support variety of business-related applications such as marketing and promotion of the products, inventory management etc. Besides markets based data analysis, association rules can also be mined for the field of bioinformatics, medical diagnosis, web mining and scientific data analysis. All the above fields deal with the huge amount of input data, whose locations could be distributed. Processing such huge data requires lots of resources and time.

The algorithm generates an extremely large number of association rules in many cases and sometimes the association rules are very large. It becomes almost impossible for the users to comprehend or validate such large number of complex association rules, thereby limiting the usefulness of the data mining results. Thus we reach at the concept of generating only “interesting” rules, generating only “non-redundant” rules, or generating only those rules which satisfies certain criteria. The criteria could be confidence, coverage, leverage, lift or strength.

The parallel association rule mining can be categorized in two sections [1]. The first is data parallelism in which the input data set could be divided among the participating node to

generate the rules. The second method is of dividing the task among the nodes so that each node will access the whole input data set for generating the rules.

The input data size is usually quite large and distributed in nature for association rule mining so clouds could be used for generating rules. Cloud computing allows consumers and businesses to use applications without any installation and access their files at any computer with internet access. It lets us do efficient computing by centralizing storage, memory, processing and bandwidth. Further, in cloud you pay have to pay only for services which you use and according to the duration of usage, thus making it a very good and inexpensive option for using it for association rule mining.

Hadoop can provide much needed robustness and scalability option to a distributed system. As Hadoop provides inexpensive and reliable storage and also tools for analyzing structured and unstructured data. MapReduce and HDFS of Hadoop use simple, robust techniques on inexpensive computer systems to deliver very high data availability and to analyze enormous amounts of information quickly. However, converting all the sequential algorithms to the parallel form, which could be converted to the map/reduce format may not be possible. There could arise situation that the algorithms may not be effectively implemented in the map/reduce format, or implementing could require greater overheads and not compensate for the advantages Hadoop provides, thus making implementation on Hadoop a bad choice.

1.1. Motivation

This paper can show how Hadoop can be seen as a solution to various distributed systems related problems. The problems like failure of nodes, communication among the nodes of the system, bursty nature of the network. A programmer does not have to worry about these kinds of framework related problems in the distributed system. All a programmer has to do is focus actual requirement of the software. In most of the situation providing reliable system and an efficient (fast) system cannot be attained at the same time. Here a programmer has to work on the efficient part of the problem while the reliable part is taken care by the Hadoop framework. Thus the association rule mining is done on Hadoop and cloud so that outcome is a reliable and efficient product.

1.2. Problem Statement

The importance of database mining is growing at an extremely fast pace due to the increasing use of computing for various applications. Mining association rules may require iterative scanning of large transaction or relational distributed databases, which is quite costly in processing. Developing a robust system which would be immune to multiple systems failures could cause greater overhead on developing the module. In this project the association rule mining algorithm is implemented in Hadoop and its efficiency is tested by varying the number of nodes in the cluster.

It aims to show that Hadoop along with cloud computing can be considered as an option for association rule mining. As with increase of data set and increase of items in the candidate set all the data cannot be kept and managed on a single computer. It also shows that the scalability of Hadoop system with respect to increase in number of candidate sets and input data.

1.3. Objective

The objectives of this project were:

- To develop Association Rule algorithm on Hadoop by implementing the Apriori algorithm for generation of the frequent Item set.
- Compare the performance of algorithm for different data set on single node.
- Compare the performance of algorithm for different data set on single node Hadoop system with algorithm on non-Hadoop system.
- The evaluate performance of algorithm as the number of nodes are varied.

1.4. Paper Organization

The paper is divided in following 5 sections:

- Chapter 1: consists of introduction of the topic, motivation and problem statement and the objective of paper
- Chapter 2: provides details of the aspects related to the project and researches on the related field.
- Chapter 3: describes the implantation of the association rule mining on Hadoop, cloud.

- Chapter 4: deals with the result and analysis. It provides details regarding the performance of the application.
- Chapter 5: provides the summary, limitations and future work.

CHAPTER 2. BACKGROUND AND RELATED PAPERS

2.1. Background Study

2.1.1. Association rule mining

Association rule mining (ARM) is a popular method of data mining method for discovering interesting relations between items in the dataset. The concept of strong rules was used by Agarwal et al [1] to find association rules in items sold for large scale transaction data base recorded by point of sale systems in supermarkets. An association rule defines relation between two set of items for e.g.

{A, B} => {C}.

In a purchase relation this would indicate if a person buys A and B together, he/she is more likely to also buy C. Mining association rule consists of following two steps:

- Finding the itemset which are frequent in the data set: The frequent item sets are set of those items whose support (sup (item)) in the data set is greater than the minimum required support (min_sup). Considering the above example all three A, B and C belongs to frequent itemset and sup {A, B} and sup {C} would be greater than the min_sup. The support of an itemset is defined as proportion of transactions which contains the itemset.
- Generating association rule from frequent itemset: Generating the interesting rules from the frequent itemsets on the basis of confidence (conf). The confidence of the above rule will be sup {A, B} divided by sup{C}. If the confidence of the rule is greater than the required confidence, the rule can be considered as an interesting one.

The performance of the association rule depends on the first step i.e. generation of the frequent itemset. As is evident the algorithm does not require the details to be specified like the number of dimensions for the tables, or the number of categories for each dimension, as each item of transaction is considered. Hence, this technique is particularly well suited for data and text mining of huge databases.

2.1.2. Apriori algorithm

The frequent itemset required for generation of association rule can be generated using Apriori algorithm. The algorithm is designed to run on database containing transactions. It is a 'bottom up' approach as candidate items are first generated and then database is scanned to count the support for candidate item to exceed minimum support required. The number of items in candidate subsets is increased one at a time, with iterations. These candidate sets are converted to frequent subsets once their support count is matched with minimum required. The iteration would stop when no frequent subset or candidate set could be generated. The Apriori Algorithm as described in the [2].

Input: D, (Data set), min_sup (minimum support threshold), Output: L, (frequent itemset)
Method:

(1) L1 =find_frequt_1-itemset (D);

(2) for (k=2; Lk-1 ≠ Φ ; k++) {

(3) Ck = Apriori_gen (Lk-1,min_sup); // Ck has candidate set

(4) for each transaction t ε D { //scan Database for counts for count of items

(5) Ct = subset (Ck, t); //get the subsets of t

(6) for each candidate c ε Ct

(7) c.count++;

(8) }

(9) LK = {c ε Ck | c.count > min_sup}

(10) }

(11) return L= L ∪ Lk ;

The function of Apriori_gen (Lk-1, min_sup) as presented in the paper [2]. In the join step Lk-1 is joined with Lk-1. The function takes argument Lk-1, number of items as pattern is k-1. It returns the superset of candidate item set with number of items as k. Ck the generated candidate set as:

Insert into Ck

Select p.item1, p.item2, ..., p.itemk-1, q.itemk-1 // p and q are a pattern from the frequent item set Lk-1

From Lk-1 p, Lk-1 q

Where p.item1 = q.item1, . . ., p.itemk-2 = q.itemk-2, // all the items are same for p and q other than the last item

p.itemk-1 ≠ q.itemk-1; // the last item is not the same for p and q

The prune step consists of

for all itemsets c ε Ck do

for all (k-1)-subsets s of c do

if(s ∩ Lk-1 = Φ) then,

delete c from Ck;

Thus the Apriori property is followed in the algorithm. The Apriori property is:

- If an itemset I does not satisfy the minimum support threshold, min_sup, then, I is not frequent, that is, sup (I) < min_sup.
- If an item A is added to the itemset I(where I is set of frequent itemset), then the resulting itemset (i.e., I ∪ A) cannot occur more frequently than I. Also, I ∪ A is not frequent if I or A is not frequent, that is, if sup(I) < min_sup or sup(A) < min_sup then sup(I ∪ A) < min_sup.

2.1.3. Map/reduce

A MapReduce job usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The map, maps the job into key and value. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. The input of reduce and output of map must have same type. Typically both the input and the output of the job are stored in a file-system. The output from the 'map' is stored in the temporary file in the HDFS, after completion of the all the map reduce task the file is converted into the permanent one. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

The MapReduce framework consists of a single master Job Tracker and one slave Task Tracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The number of times a failed job is tried to be executed is configurable. The slaves execute the tasks as directed by the master.

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job. It is not necessary that the output of map and output of reduce both be of the same type. They can be of different data types. Input and Output types of a MapReduce job:

(Input) <k1, v1> -> map -> <K2, v2> -> combine -> <K2, v2> -> reduce -> <k3, v3> (output)

There are various input and output format defined. A programmer can also override these, to have input and output formats, according to the requirement.

2.1.4. Hadoop

Hadoop is a free, Java-based programming framework that supports the processing of large data sets in a distributed computing environment. Hadoop is designed to run on a large number of machines that don't share any memory or disks. It creates clusters of machines and coordinates work among them. Hadoop consists of two key services: reliable data storage using the Hadoop Distributed File System (HDFS) and high-performance parallel data processing using a technique called Map/Reduce.

HDFS splits user data across servers in a cluster. It uses replication to ensure that even multiple node failures will not cause data loss. It also keeps track of location of the replicated file. In addition, Hadoop includes Map/Reduce, a parallel distributed processing system that is different from most similar systems on the market. It was designed for clusters of commodity, shared-nothing hardware. No special programming techniques are required to run analyses in parallel using Map/Reduce, most existing algorithms work without changes. Map/Reduce takes advantage of the distribution and replication of data in HDFS to spread execution of any job across many nodes in a cluster.

Even if a machine fails, Hadoop continues to operate the cluster by shifting work to the remaining machines. It automatically creates an additional copy of the data from one of the replicas it manages. As a result, clusters are self-healing for both storage and computation without requiring intervention by systems administrators.

2.2. Related Work

2.2.1. Association rule mining

In 1994 Agarwal et.al [2] gave the Apriori and AprioriTid algorithm for generating frequent itemset and then mining association rule from these frequent itemset. The performance of Apriori and AprioriTid was much better than the earlier performance of SETM or AIS (artificial immune system) algorithm for mining association rules. The SETM and AIS algorithm both generated candidate set and compared them with frequent set. The disadvantage of SETM and AIS was that they generated too many candidate sets which later turned out to be in frequent.

Various algorithms were defined on the basis of Apriori algorithm and different methods were formed for reducing the time to generated frequent itemset. Some of these papers are [3, 4, 5]. In these papers hashing was used to prune the candidate set and hence speed up the generation of frequent item set. The faster the less frequent itemset are removed the quicker would be the mining. However, the trimming and pruning properties caused some problems that made it impractical in many cases.

Yet another approach was used [6], in which candidate set is not generated for getting the frequent item set, rather it creates a relatively compact tree-structure that alleviates the multi-scan problem and improves the candidate itemset generation. The algorithm requires only two full I/O scans for the dataset. It first creates a FP (frequent pattern) tree using the frequent length-1 item as nodes of the tree; storing quantitative information about frequent patterns, pattern growth is obtained by concatenating suffix patterns with the ones generated from the conditional FP tree. The search used in this is divide and conqueror for mining frequent pattern. This has come up as a fast method for generation of frequent patterns.

2.2.2. Association rule in distributed systems

Development of distributed algorithm for association rule is yet another important angle to address the large centralized data and distributed databases, for mining relations. Many large databases are distributed in nature. The idea of local and global frequent item set was introduced [7], it finds the local support counts and prunes all infrequent local support counts. After completing local pruning, each site broadcasts messages containing all the remaining candidate sets to all other sites to request their support counts. It then decides whether large itemsets are globally frequent and generates the candidate itemsets from those globally frequent itemsets. The [1] paper focused on communication and synchronization of the systems to reduce the number of candidate itemsets in a distributed data base system. The parallel data mining [8] collected information from each node on each itemset from its local database by hashing method. The information discovered by each node is next shared with other nodes via some communication schemes. It later employed a technique, called clue-and poll, to address the uncertainty due to the partial knowledge collected at each node by selecting a small fraction of the itemsets for the exchange of count information among nodes.

The hashing and Trie data structures (Fig 2.1 and 2.2) were also used for reducing time taken for the generation of frequent item set in the distributed systems [8,9]. In [8] developed a new distributed Trie-based algorithm (DTFIM) to find frequent itemsets. In second phase, FDM (Fast Distributed Mining) algorithm was used for candidate generation step. Experimental evaluations on different sort of distributed data showed it to be very effective algorithm.

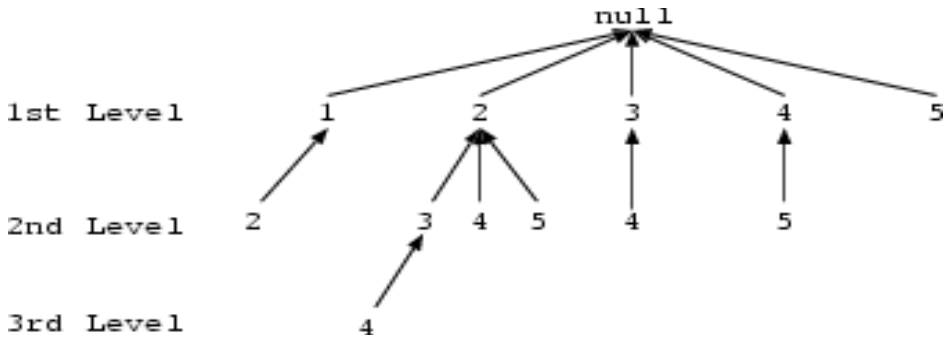

Figure 2.1.Trie: each node represents a frequent itemset [14]

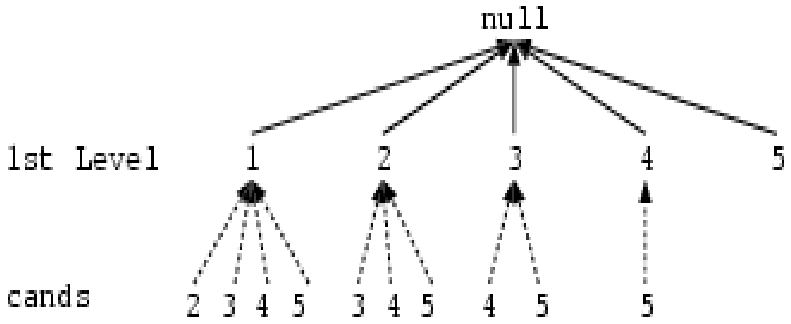

Figure 2.2. Trie and candidate itemsets on the 2nd-level [14]

CHAPTER 3. SYSTEM IMPLEMENTATION

3.1. Introduction

The project implements the association rule mining on Hadoop. The input data set is saved on HDFS as it can store huge chunks of data and also because it help in data localizations for the map/reduce task. The input data is divided into parts and allotted to the mapper. The output from the mapper is item (key) and its count as 1(value). This output is taken in by the combiner. It combines all the count value related to a particular item (key). The result from this is taken in by the reducer for further, combining and summing up of the values corresponding to an item (key). After getting the sum of values for an item, in the reduce stage it is checked whether the value exceeds the given threshold value. If the value exceeds the item along with the item is written as the output. The value is discarded if it is less than the minimum support threshold value. This would generate frequent-1 itemset. Frequent -1 itemset consists of set of one item pattern.

The candidate-2 item set is generated using the frequent -1 item set in the mapper. The count of each pattern within the candidate -2 is checked with the input data assigned to the mapper. The output of the data reaches the combiner, the combiner accumulates and sums the of candidate item set with the corresponding value. Reducer further reduces and combines the data. In the end only those candidate set is written on the file whose value exceeds the value of minimum threshold. The same process is continued. The candidate set is generated using previous iteration's frequent item set. The iteration stops when no more candidate set can be generated for iteration, or the frequent item set of the previous iteration cannot be found.

Once the frequent-n itemset is generated and saved as output, the association rules are generated. The association rule confidence calculation requires the support count of item set which forms the rule. The itemset along with its support count is written in the output file. Each line in the file gives a different frequent item set and is tab separated with the support count. The frequent-n item set resides in the output folders 'n' named sub folder. These are used to calculate. Once confidence is found to be 100% the rule is generated.

The Hadoop0.20.0 framework was used for implementation. The single node Hadoop was run on the virtual machine to mimic the Hadoop environment. The time function used for measuring was system time, it returned time in millisecs. Fig 3.1 shows the flow data for two iterations in the project. The Fig 3.2 shows the Map data is written in temporary files in HDFS which is taken in by combine and processed by it. Later the output of the Combine to be required by reduce is written in temporary files and which is further processed by the Reduce. Reduce also saves the data in a temporary file while it is processing the data. Once all the data is processed by the Reduce for that stage the resultant temporary file is converted into a permanent file and is stored in the specified output path. The details of passing the correct path of temporary files to Combine and Reduce are taken care by the Hadoop framework.

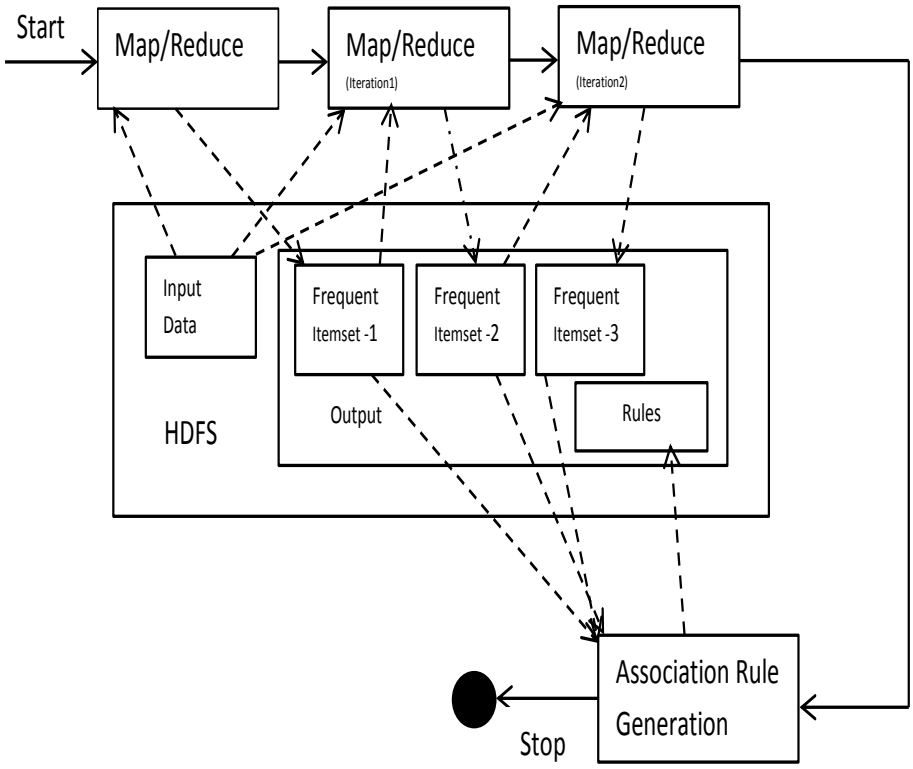

Figure 3.1. Data flow diagram showing two iterations.

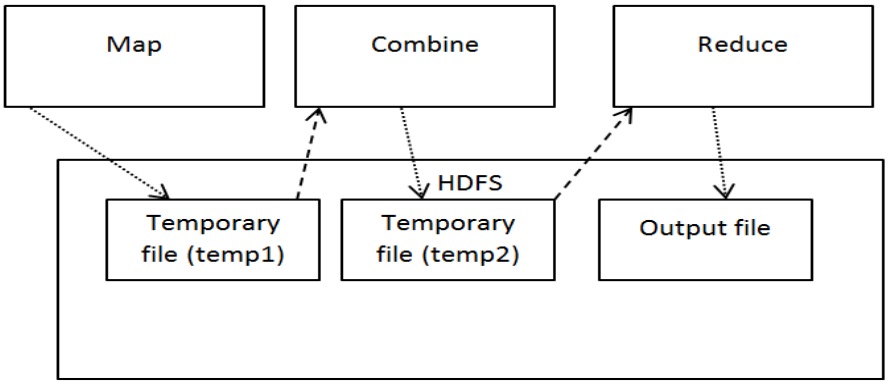

Figure 3.2. Shows data transfer among the methods Map, Combine and Reduce in distributed system.

Apriori is a traditional algorithm for association rule generation and is effective to operate on database containing transactions. The idea behind Apriori is producing candidate items, and then scanning the database so as to decide whether they the candidate items meet the required minimum support count. It uses a "bottom up" approach, frequent subsets are extended one item at a time (a step known as candidate generation), and groups of candidates are tested against the data. The algorithm terminates when no further successful. In the candidate generation, an additional pruning step is used, to check that all the subsets of the candidate are frequent. This helps in reducing the size of candidate set before scanning the data base. In this method, the number of times the input file will be read will depend on the number of iteration is required to get the maximum items in the frequent itemset. For example if the maximum number of items in the frequent itemset is 15, then the whole input file will be read minimum of 15 times.

3.2. Splitting the Data Set

The “Input Format” splits the input file(s) into logical “Input Splits”, each of which is then assigned to an individual Mapper. The Hadoop does the logical splitting of the data set to be fed to Map/Reduce on its own with the value assigned to “mapred.min.split.size” parameter in “hadoop-site.xml”. By default, the “FileInputFormat” and its descendants break a file up into 64 MB chunks. In the project the size of input split is being set by overriding the parameter in the “JobConf” object used to submit a particular MapReduce job. This will help several map tasks to operate on a single file in parallel. If the file is very large, this can improve performance

significantly through parallelism. As various blocks that make up the file may be spread across different nodes in the cluster, it allows tasks to be scheduled on each of these different nodes. Thus blocks can be operated on locally, instead of needing to be transferred from one node to another.

The number of logical splits in which the File would be split can be specified at run time, in case nothing is specified the default number of logical splits is 5. While splitting the file on bytes care is taken that a transaction is not split between two different splits. In the scenario where a transaction might get split the whole transaction resides in the part where the beginning of the transaction is and the pointer for next split starts from the next transaction. The “Input Format” defines the list of tasks that make up the mapping phase; each task corresponds to a single input split. The tasks are then assigned to the nodes in the system based on where the input file chunks are physically resident.

3.3. Apriori Algorithm in Map/Reduce

The split chunk of the data file is fed to Mapper. The Map reads a line at a time and assigns each item as a key and the value associated with the key is 1 for the finding frequent-1 itemset. This <key, value> combination is taken in by the combiner. The combiner combines the key with the value and generates the list of values associated and also sums up the list, for the respective key. The data from the mapper is written in the temporary files in HDFS to be used by combiner so combiner from any system can be allotted to work on it. Later the data written by combiner is written in temporary file. This is then passed to reducer. The reducer accumulates the count of values associated with the respective key. It sums up the values associated with the key and writes the value in the output file in the increasing order of the keys. Only those items are written in the output file whose value equals or exceeds the minimum support required.

In the later iteration of Map/Reduce phase the Mapper takes in the frequent item set from the previous iteration generates the candidate itemset. This candidate itemset forms the key, if it is present in the transaction read by the Map and is passed to Combiner with the value as 1. The rest is similar to frequent -1 itemset generation step. The candidate itemset is developed by using

Fast Apriori Algorithm. The Fast Apriori helps in reducing the size of candidate itemset as, it removes those itemset whose subset were absent in the previous iteration's output file.

The algorithms stops if candidate itemset could not be generated or the output file from the previous iteration is not present. The presence of output file is checked before starting the next iteration. Once the frequent item sets are generated, association rules are developed.

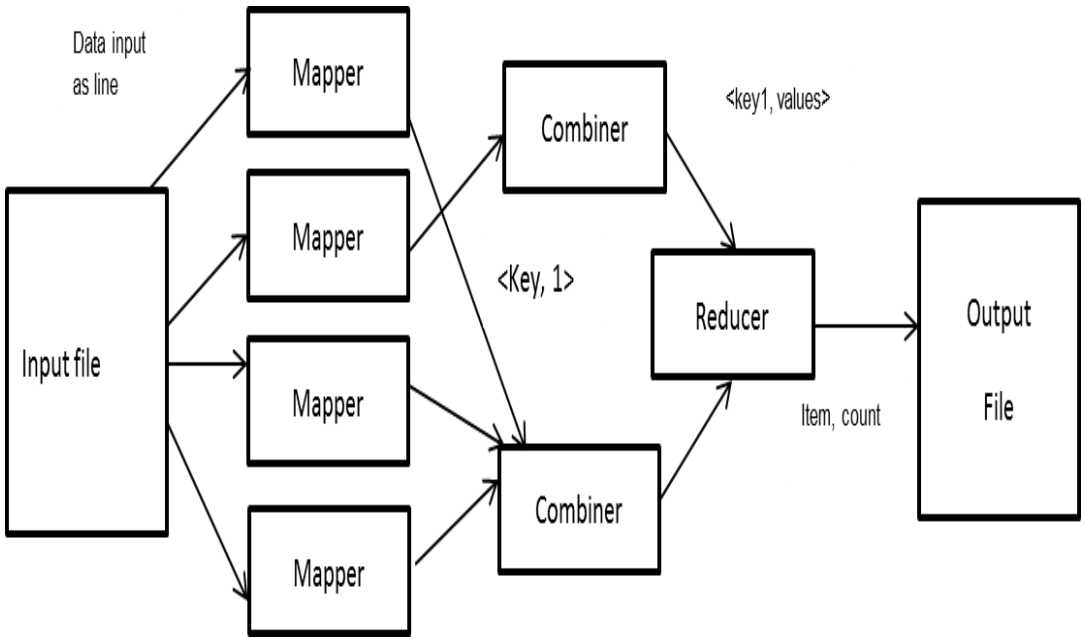

Figure 3.3. Map /Reduce with Apriori for the project.

3.4. HDFS for Algorithm

The Hadoop Distributed File System (HDFS) will split large data files into chunks which are managed by different nodes in the cluster. In addition to this each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable. The number of machines in which the data are replicated can be configured. Always the number of replication should be chosen such that the Hadoop can withstand multiple node failures. An active monitoring system then re-replicates the data in response to system failures which can result in partial storage. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so their contents are universally accessible. The input

file and the output files are stored on the HDFS. The files are required to be copied into HDFS from the local system so that it can be read and used by Hadoop. Further, an output file in the Hadoop has to be copied back to local system to read the output.

The input files are saved in the HDFS in the Input folder. This path can be specified during the run time, if not specified the application will not start and will give an error. The output of frequent item set is saved in HDFS in output folder under the iteration number sub- folder. This also forms an input file for the next iteration for generating candidate frequent item set. It also comes useful in generating association rule as they contain the frequent item set with the support count, as support count at each level of frequent item set is sufficient to calculate confidence for the corresponding association rule.

3.5. Rule Generation from the Frequent Item Set

The output folder contains frequent item with their count after the successful run of that iteration. The outputs from the two successive iterations are read at a time. The confidence is calculated by using the count written along with the successive frequent itemset. If the confidence is 100% the rule is considered to be an interesting one. For example the frequent itemset {A,B,C} from iteration 2 has support_count 50 and support_count of {B,C} from iteration 1 is 50. The relation would be {B,C} => {A}, as the confidence is 100% and support_count > min_sup.

The items in the frequent itemset are comma separated. A frequent itemset is read and its support count saved. The subsets of the frequent itemset are found. The support count of the subsets is read from the previous iterations output files. The confidence is calculated. If the confidence is found to be 100% the rule is generated as an interesting rule. The total 20 of such association rule is generated. The association rule for later iteration frequent itemset is given more importance than the earlier ones.

3.6. Implementation on EC2 (Elastic Cloud Computing)

The project jar file was run on EC2 for evaluating the performance by increasing the number of nodes in the set up. The data input files were saved on S3. Amazon S3 is a data storage service. Transfer between S3 and Amazon EC2 is free. This makes use of S3 attractive for Hadoop users who run clusters on EC2. The details related like path and permission to S3 is

configured. The output data are also written back in the bucket of S3 at the end. The temporary data is written in the HDFS files so as to optimize data locality for the nodes to process in the project. Amazon EMR takes care of provisioning a Hadoop cluster, running the job flow, terminating the job flow, moving the data between Amazon EC2 and Amazon S3, and optimizing Hadoop. Amazon EMR removes most of the difficulties associated with the Hadoop configuration like setting up the hardware and networking required by the Hadoop cluster, such as monitoring the setup, configuring Hadoop, and executing the job flow.

Hadoop on the cloud can be shown in Fig 3.4. The request is sent to the EMR model, to start the job with all the details required for the job, such as path to S3 etc. The Hadoop cluster with master and slave instances is created. The Hadoop cluster works on the job and finishes the job. The temporary created during the run of job files can be stored in HDFS or S3. Storing in S3 may not be the right choice for all the cases as it may add to communication overhead. The output after the job is stored in S3. Only the error or fatal messages are written on the screen during the entire run of job. Once the job is complete response is sent indicating the completion of the job.

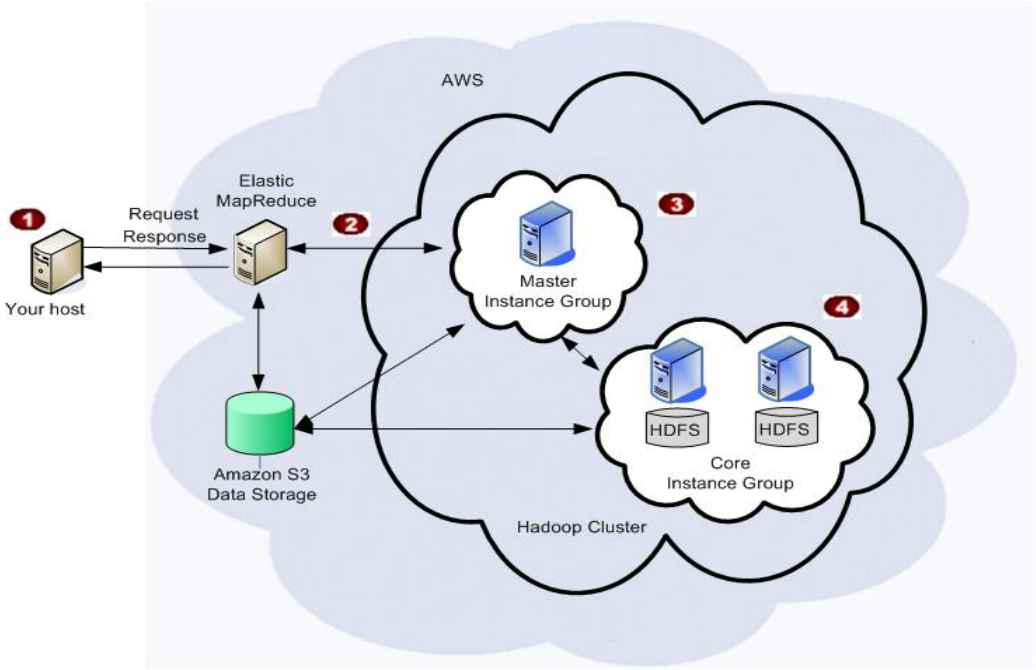

Figure 3.4. Job flow in the cloud for Hadoop [12]

CHAPTER 4. RESULT AND ANALYSIS

4.1. Result

4.1.1. Description of the data set

The program was tested on the chess, connect, mushroom and T10I4D100K data set. The input data set were not compressed. The data set T10I4D100K is synthetically generated data set. The generator is from the IBM Almaden Quest research group. However, the generator cannot be downloaded from their site anymore. The data set was got from [11]. The chess, connect and mushroom data set are real data set and downloaded from UCI data set repository. The chess, mushroom and connect data set can be found in the UCI repository [15]. Some of the details regarding the data sets are in table 4.1.

The data items are space separated in the input file. Each line in the input file gives the new tuple for the transaction. The chess, connect, mushroom data set are comma (,) separated files (.csv). The items are converted to numeric value, with each unique item denoting an integer. For example won or no-win in chess could be converted integers ‘1’ and ‘2’ respectively. A typical transaction in the data set of chess looks like f,f,f,f,f,f,f,f,f,f,f,f,l,f,n,f,f,t,f,f,f,f,f,f,t,f,f,f,f,f,f,f,t,t,n,won”. The “f” would be counted so many times in the same transaction, unless the data would be imported as a matrix or in the table. This can be avoided if “f” and “t” pertaining to each column is given a unique numeric value. This unique numeric value can be replaced in for “f” and “t” for the whole column. The idea can be extended to cases in which a column has more than two parameters. Each string parameter can be mapped to a unique numeric value and this can be replaced. This will also add uniformity to the data set as now all the values are of the same type that is numeric. The items within a transaction get automatically arranged in the increasing order, if the values assigned to the columns are in increasing order.

Table 4.1. Details of the data sets.

Data set	Total instances	Total attributes
Chess	3196	36
Mushroom	8124	22
Connect	67557	42
T10I4D100K	100000	26

4.1.2. Results of experiment

In the table 4.2 and all the graphs (Figure 4.1, 4.2, 4.3 and 4.4) the threshold is the minimum support (min_sup) taken divided by the total number of the transactions in the data set. The value of the threshold is in decimal lies between 0 and 1. The threshold value .95 for chess data set implies the minimum support considered is 3037 (threshold * total number of transactions in the data set). The threshold is taken so that time taken can be compared among the data sets each having different number of total number of transactions(as they will have different minimum support count).The time output is in mill- seconds which is converted to seconds for comparing the performance.

The performance of different data sets is tested on a single node Hadoop system. This gives an idea on how different data set would perform for the algorithm. The data set Connect takes the maximum time even when the data set T10I4D100K has maximum number of transactions. The reason is number of items are denser in Connect and the number iterations in Connect data set to find the frequent item set is more than that of T10I4D100k. Even though the number of iteration for chess and mushroom data go till 17-18 iterations, still they take very less time for generating association rules. The figure 4.1 shows the performance of different data sets when run on a single node Hadoop system.

Table 4.2. Time taken in secs for all data set.

Data set	Threshold= .95	Threshold= .90	Threshold= .85	Threshold= .80	Threshold= .75
Single Hadoop node (T10I4D100k)	50.26	68.529	88.559	133.621	220.631
Single Hadoop node (chess)	108.18	156.326	220.122	392.927	878.156
Single Hadoop node (mushroom)	65.746	71.005	86.53	91.251	111.547
Single Hadoop node (connect)	84.612	241.172	681.576	1552.194	2934.792

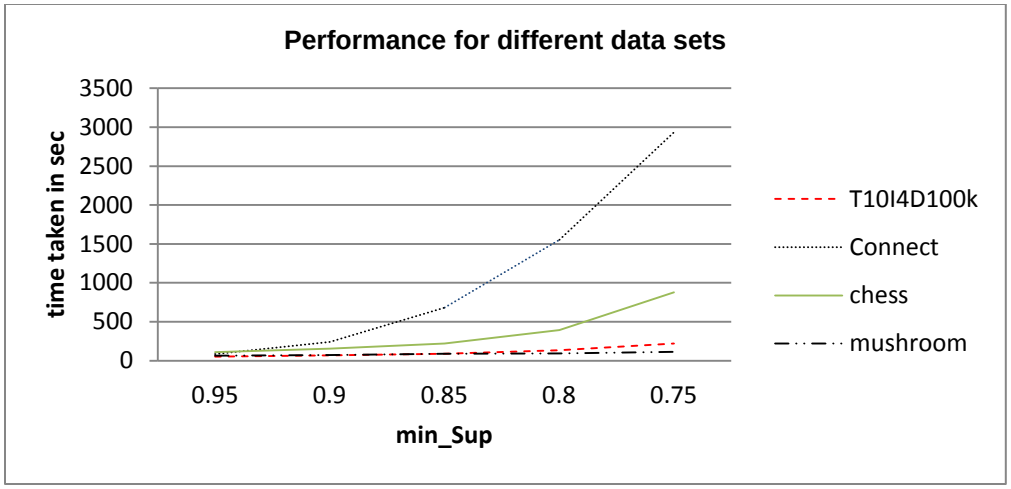

Figure 4.1. Graph Showing performance of different data set by varying the minimum support on single node Hadoop.

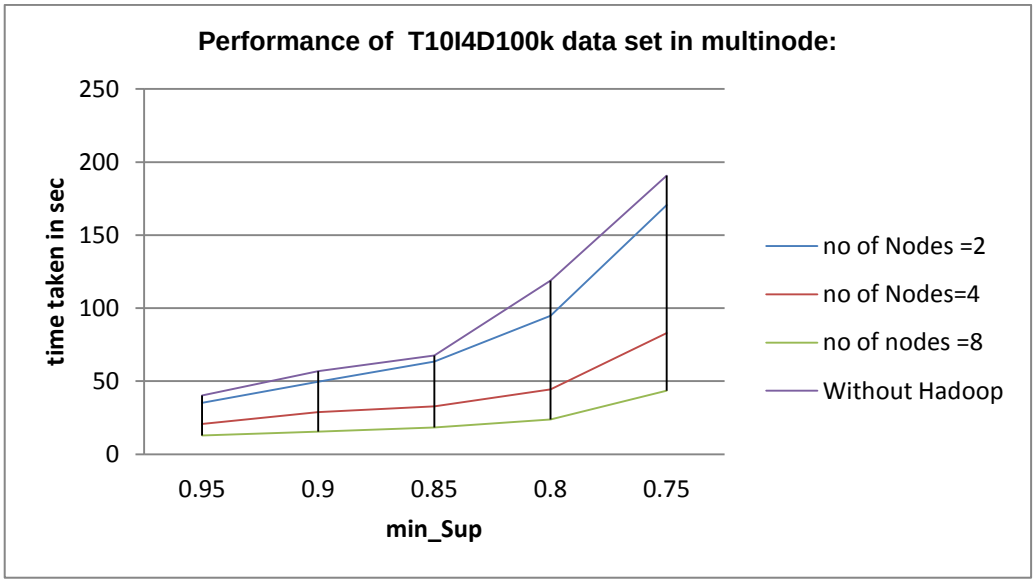

Figure 4.2. Graph showing the performance of algorithm for the synthetic data set T10I4D100k on multi- node Hadoop system and on single node non Hadoop system.

We can contrast the performance of different data sets on single non Hadoop machine for the same algorithm implementation. The performance of the algorithm in single node is better than Hadoop on single node as, much of the Hadoop time span is spent providing the reliability to the system. The tasks like dividing data and writing result, so that it is available to all the participating nodes maps, the mapped key value to the reducer, etc.

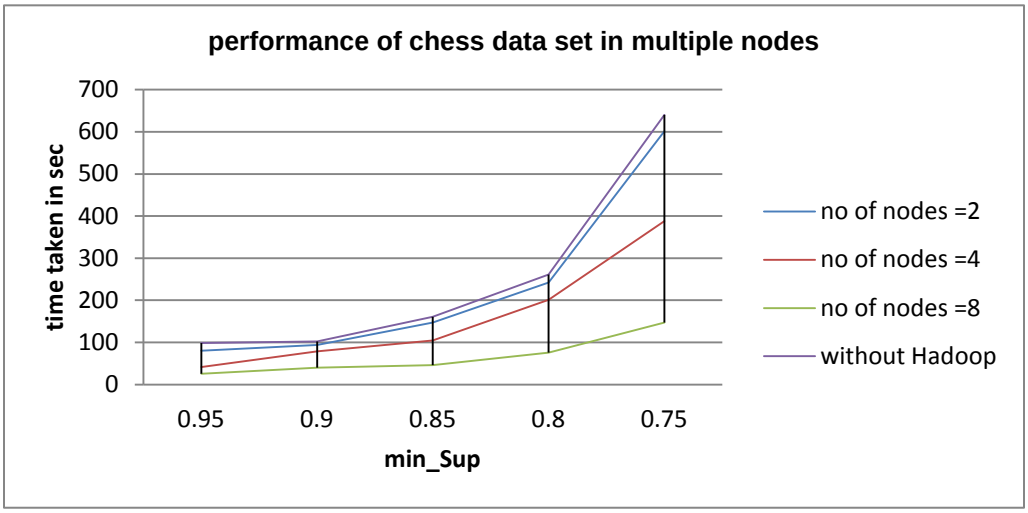

Figure 4.3. Graph showing the performance of algorithm for the dense data set chess on multi- node Hadoop system and on single node non Hadoop system.

4.2. Analysis

The performance of data set on the multi node Hadoop cloud can be seen in the Fig 4.2, 4.3, 4.4. In these figures we can see that connect data set reduces the run time to a great extent. The performance of the connect data set is much better compared to the other data set. The reduction of run time is more evident for those data sets which have long run time for example the reduction in run time from 10 to 2 sec may go unnoticed but a reduction from 100 to 20 sec will be called a very efficient system. As the size of the data set increases, the performance on Hadoop also gets better. Even the other data sets does show improvement in their run time as, the number of nodes in the cluster is increased. This reinstates the fact that Hadoop will give a much better performance for large data sets. The performance also depends on the number of nodes in the cluster. The performance gets almost 1.3 in better in two node set up from a single node. While if we compare the performance in 8 nodes set up, in spite, of all the increase in traffic of data from one node to another, the performance gets better to the count of 5 times to the single node system.

The range for time taken in Connect varies from 0-2500 while in T10I4D100k lies in the 0-250 which is even less than the time taken range for chess 0-750. The number of iterations for generation of frequent item sets is a very important factor along with the size of the data set. The

data set is read in iterations, the number of times data set would be read will depend on the number of iteration the program will run before it comes to stop. Thus number of iterations and size of data set would play important factor in affecting the run time.

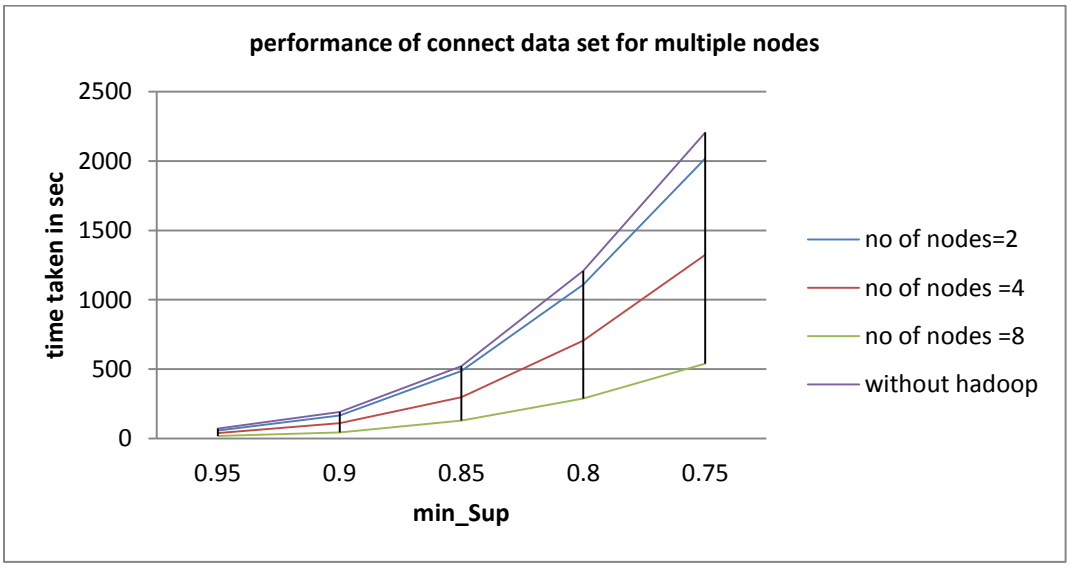

Figure 4.4. Graph showing the performance of algorithm for the synthetic dense and large data set connect on multi- node Hadoop system and on single node non Hadoop system.

CHAPTER 5. CONCLUSION

5.1. Summary

The implementation of association rule in the distributed systems can be efficiently done on Hadoop. It can scale up to large data set with comparatively less cost and provide good performance. It can also cater to the distributed nature of the input data. The input data is divided among the nodes. Further, the data transfer among the nodes and the situations like a storage node dies or, what would happen if some nodes in the cluster does not run, are taken care by Hadoop. This adds a great deal of robustness and scalability to the system. The performance of small data set is better but the performance is not as good as for the dense and large data sets.

5.2. Limitation

The present algorithm can be made more effective by experimenting on number of splits, so that each map runs fully on the given time allotted to it. It shouldn't happen that some maps are idle while other is over worked because of huge data being allotted to it. Also not all the association rules are generated. There can be too many association rule generated.

The algorithm in its own is not a very fast one. Other faster variation to it has been found, but to extend them to parallel, distributed environment is required. The implementation is being considered by converting categorical data to numeric ones so that arranging the results after each step is easier.

A small data set may not give good performance in the given set up. It might give worse run time than the non-Hadoop system for the same algorithm. In the present project the infrequent items were not removed from the input data set to be considered for next iterations. The removal of infrequent items can reduce the size of data set but time might be taken for creating new input data set at end of iterations.

5.3. Future Research Recommendation

The implemented algorithm can be made faster by using hashing and so that more of items are screened out of candidate items in the initial stages. The other single node faster algorithm can be extended to the distributed systems. The ones like Trie etc. so that the time of

computation for n maps with in nodes is reduced and better performance can extracted out of the distributed system.

The algorithm can also be developed with for the items with some constraint. There may be a requirement to get the association rule for items which have minimum support count much less in number. For example mining rules for the sale of television from the transaction data base which has items like chocolate in their database can be difficult. In those cases we have to fine for items individual threshold and confidence. Customizing algorithm to address this issue can also be done.

REFERENCES

[1] Ashrafi, M. Z., Taniar, D., & Smith, K. (2004). ODAM: An optimized distributed association rule mining algorithm. *Distributed Systems Online, IEEE, 5*(3)

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. 1994 Int. Conf.Very Large Data Bases, pages 487-499, Santiago, Chile, September 1994

[3] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. 1995. An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD international conference on management of data* (SIGMOD '95), Michael Carey and Donovan Schneider (Eds.). ACM, New York, NY, USA, 175-186. DOI=10.1145/223784.223813 http://doi.acm.org/10.1145/223784.223813

[4] Ozel, S. A. and Guvenir, H. A. 2001. An algorithm for mining association rules using perfect hashing and database pruning. In 10th Turkish Symposium on Artificial Intelligence and Neural Networks, Gazimagusa, T.R.N.C., A. Acan, I. Aybay, and M. Salamah, Eds. Springer, Berlin, Germany, 257--264.

[5] Karam Gouda , Mohammed Javeed Zaki, Efficiently Mining Maximal Frequent Itemsets, Proceedings of the 2001 IEEE International Conference on Data Mining, p.163-170, November 29-December 02, 2001

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In ACM-SIGMOD, Dallas, 2000

[7] D.W. Cheung , et al., "A Fast Distributed Algorithm for Mining Association Rules," *Proc.Parallel and Distributed Information Systems,* IEEE CS Press, 1996,pp. 31-42;http://csdl.computer.org/comp/proceedings/pdis/1996/7475/00/74750031abs.htm.

[8] Ansari E, Dastghaibifard G, Keshtkaran M, Kaabi H. Distributed frequent itemset mining using trie data structure. IAENG International Journal of Computer Science. 2008;35(3):377-381.

[9] Bodon F. Surprising results of trie-based FIM algorithms. . 2004;90.

[10] Park, J. S., Chen, M. S., & Yu, P. S. (1995). Efficient parallel data mining for association rules. Paper presented at the *Proceedings of the Fourth International Conference on Information and Knowledge Management,* pp. 31-36.

[11] http://miles.cnuce.cnr.it/~palmeri/datam/DCI/datasets.php

[12] http://docs.amazonwebservices.com/ElasticMapReduce/latest/DeveloperGuide/Introduction_EMRArch.html

[13] Frank, A. & Asuncion, A. (2010). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[14] http://wis.cs.ucla.edu/wis/atlas/doc/atlas-manual/sec5-4.html

APPENDIX. CODES IN MAP/REDUCE

```
public static class ItemCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> { //  mapper for frequent-1 itemset,  the key is text
type and the value is IntWritable type
        private final IntWritable one = new IntWritable(1); // value of the key is 1
    private Text word = new Text();   // key
    public void map(LongWritable key, Text value,  OutputCollector<Text,IntWritable> output,
Reporter reporter) throws IOException {
        String line = value.toString(); // input file being read line by line
        StringTokenizer itr = new StringTokenizer(line.toLowerCase());
      while(itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);// key and value being mapped to be used by
   combiner                                    }
     }
}
public static class ItemCountCombiner extends MapReduceBase  implements
Reducer<Text, IntWritable, Text, IntWritable> {  // combiner
      public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            IntWritable value = values.next(); // values with the same key combined
            sum += value.get(); // process value
        }
            output.collect(key, new IntWritable(sum)); // values to be sent to reduce
     }
}
```

```
public static class ItemCountReducer  extends MapReduceBase  implements Reducer<Text,
IntWritable, Text, IntWritable> { // reducer
        private static long perc;
        public void configure(JobConf conf){
            perc=Long.parseLong(conf.get("percent")); // the value of threshold got from main
        }
        public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException { // reduce
            int sum = 0;
            while (values.hasNext()) {
                IntWritable value = values.next();
                sum += value.get(); // process value
            }
           if(sum >=perc ) // checking the support count before writing to output file
                output.collect(key, new IntWritable(sum));
        }
    }
public static class ItemIteratorMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> { // mapper for the iteration step
        private static Integer filepathCount;
       private final IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public List<String> frequentItemList=new ArrayList<String>(); // list for candidate
itemset
        public void configure(JobConf conf) {
            Integer perc=0;
            frequentItemList.clear();
            System.out.println("inside frequentItemSet");
```

```
perc=Integer.parseInt(conf.get("percent"));
String outputPath = conf.get("iterOutputPath");
Integer n=0;
try{
    n=conf.getInt("iterationCount",1);
    System.out.println(n + "the value of n");
    Path pt=new Path(outputPath+"/" + n.toString() +"/part-00000");
   FileSystem fs = FileSystem.get(conf);
   List<String> items= new ArrayList<String>();
   if (fs.exists( pt)) {
        BufferedReader br=new BufferedReader(new
InputStreamReader(fs.open(pt)));
        String line="";
        while ((line = br.readLine()) != null){
            StringTokenizer itr = new StringTokenizer(line.toLowerCase(),"\t");
            String itrString= itr.nextToken();
                items.add(itrString);
        }
   }
   else{
        return;
  }
   Integer arraylength=items.size();
    if(arraylength<1)
    {
     fs.close();
    return;
    }
```

```
Integer i,count=0;
Iterator<String> iterator12= items.iterator();
while(iterator12.hasNext()){
    String listItem= iterator12.next();
    if(listItem.indexOf(",") > 0) {
        for(i=count+1;i<arraylength; i++) {
            String temp= items.get(i);
            String input=" ";
            String str1 = listItem.substring(0, listItem.lastIndexOf(","));
            String temp1= temp.substring(0, temp.lastIndexOf(","));
             if(str1.equals(temp1)) {
                String lasttoken1 =
listItem.substring(listItem.lastIndexOf(",")+1);
                String lasttoken2=
temp.substring(temp.lastIndexOf(",")+1);
                 if(!(lasttoken1.equals(lasttoken2))){
                    if(Integer.parseInt(lasttoken1)>
Integer.parseInt(lasttoken2))
                        input= str1 + "," +lasttoken2 + "," + lasttoken1;
                    else
                        input=str1 + "," + lasttoken1 + "," +lasttoken2;
                    frequentItemList.add(input);
                     input=" ";
                }
             } else{
                input="  ";
                i= arraylength;
             }
```

```
                    }
                } else {
                    for(i=0; i < arraylength && i!=count; i++) {
                        String temp= items.get(i);
                        String input=" ";
                        if(Integer.parseInt(listItem) > Integer.parseInt(temp))
                            input= temp + "," + listItem;
                        else
                            input=listItem + "," +temp;
                        frequentItemList.add(input);
                    }
                }
                count++;
            }
        } catch (IOException ioe) {
            System.err.println("IOException during operation: " + ioe.toString());
            ioe.printStackTrace();
            return;
        }
    if(frequentItemList.isEmpty()){
        System.out.println("Init Error");
        return ;
    }
  }
  public void map(LongWritable key, Text value,  OutputCollector<Text,IntWritable> output,
Reporter reporter) throws IOException {
        String line = value.toString();
```

```
                if (frequentItemList == null || frequentItemList.isEmpty())  throw new
IOException("reducer output is null");
            Iterator<String> itrList=frequentItemList.iterator();
            while(itrList.hasNext()){
                String freqItem =itrList.next();
                  StringTokenizer itrfre=new StringTokenizer(freqItem,",");
                  boolean result = true;
             while(itrfre.hasMoreTokens() && result) {
                        String tempfreq1= itrfre.nextToken();
                        String start= tempfreq1 + " ";
                        String end = " " + tempfreq1;
                        String mid = " " + tempfreq1 + " ";
                        if( line.startsWith(start) || line.endsWith(end) || (line.indexOf(mid) >0 ))
                              result=true;
                        else
                              result=false;
                }
                if(result) {
                        word.set(freqItem);
                        output.collect(word, one);
                }
            }
        }
    }
```