Data Mining Using Clouds: An Experimental Implementation of Apriori over MapReduce

Juan Li[1], Pallavi Roy[1], Samee U. Khan[1], Lizhe Wang[2], Yan Bai[3]
[1]North Dakota State University, Fargo, USA
[2]Chinese Academy of Sciences, Beijing, China,
[3]University of Washington Tacoma, Tacoma, USA
{j.li, pallavi.roy,same.khan}@ndsu.edu, lwang@ceode.accn, yanb@uw.edu

Abstract— **Cloud computing has become a viable mainstream solution for data processing, storage and distribution. It promises on demand, scalable, pay-as-you-go compute and storage capacity. To analyze "big data" on clouds, it is very important to research data mining strategies based on cloud computing paradigm from both theoretical and practical views. For this purpose, we study a strategy of data mining on cloud using association rule mining as an example. In particular, we redesign and convert an existing sequential association rule algorithm, Apriori, to support MapReduce parallel computing platform. We implement and evaluate the proposed algorithm on Amazon EC2 MapReduce platform. The efficiency of our approach is manifested by the preliminary experimental results documented in this paper.**

Keywords- cloud computing, data mingling, association rule

I. INTRODUCTION

The increasing ability to generate vast quantities of data brings potentials to discover and utilize valuable knowledge from data. Data mining has been an effective tool to analyze data from different angles and getting useful information from data. It can also help in predicting trends or values, classification of data, categorization of data, and to find correlations, patterns from the dataset. On the other hand, utilizing the vast amount of data presents technical challenges as data storage and transfer approaches needs to deal with prohibitive amounts of data. The management of data resources and dataflow between the storage and compute resources is becoming the main bottleneck. Analyzing, visualizing, and disseminating these large data sets has become a major challenge and data intensive computing is now considered as the "fourth paradigm" in scientific discovery after theoretical, experimental, and computational science [15].

To shift to this fourth paradigm, researchers have to plan strategically to perform data analysis. Cloud computing, a business model containing a pool of resources, provides an effective paradigm for this purpose. In this context, cloud computing is a distributed computing paradigm that enables large datasets to be sliced and assigned to available computer nodes where the data can be processed locally, avoiding network-transfer delays. This makes it possible for people to understand and utilize the trillions of rows of information in a data center. We strongly believe that cloud computing can be an effective platform for data mining. To gain more experience in cloud-assisted data mining, in this paper, we use association rule based algorithm, Apriori [5], as an example to study how data mining algorithms can be adjusted to fit the increasing demand for parallel computing environment of cloud.

Association rule mining aims to extract interesting correlations, patterns, associations among sets of items in the transaction database or other data repositories. The Apriori algorithm [5] is the most widely used algorithm for association rule mining. The input data size of Apriori is usually quite large and distributed in nature. Therefore, a cloud could be an ideal platform for this algorithm. However, the classical Apriori algorithm was not designed to be performed in the parallel environment of cloud because the iterative approach to get the frequent sets causes repeated scans of the disk. This high I/O overhead makes running the algorithm in clouds impractical. Although there have been some paralleled association rule algorithms [16, 17, 18, 19], implementing these algorithms is difficult because programmers have to deal with challenges of process communication and synchronization [20]. It is especially difficult to deal with failures that occur frequently in data intensive computing.

In our work, we utilize MapReduce, a programming paradigm commonly used in clouds for developing and executing parallel algorithms to process massive datasets. The MapReduce model has also been used in machine learning applications [21]. A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer [11]. We choose to use Hadoop as the implementation, as it provides inexpensive and reliable storage, and tools for analyzing structured and unstructured data. However, converting a sequential algorithm to the parallel MapReduce format as required by Hadoop needs a considerabl effort. There could arise situations 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. In our work, we have not only revised Apriori to the MapReduce format, but also improved its parallel performance at all stages.

The remainder of this paper is organized as follows: In Section II we detail the background knowledge and related

work. Section III describes the design of the enhanced Apriori algorithm for MapReduce. In Section IV, we evaluate the proposed methods and show their effectiveness with a comprehensive set of simulations. Concluding remarks are provided in Section V.

II. Background and Related Work

A. Background

1) Association rule mining and Apriori: Association rules are widely used in various areas, such as telecommunication networks, marketing and risk management, and inventory control. Many companies keep large quantities of their day to day transactions, which can be analyzed to learn the purchasing trend or pattern of customers. Such valuable insight can be used to support a variety of business-related applications, such as marketing, promotion of products, and inventory management. Besides market-based data analysis, association rules are also mined in the field of bioinformatics, medical diagnosis, web mining and scientific data analysis.

The concept of strong rules was used by Agarwal et al [1] to find association rules in items sold in supermarkets. An association rule defines a relation between two set of items. For example, *{A, B}=>{C}* in a purchase relation, would indicate if a person buys *A* and *B* together, he/she is more likely to also buy *C*.

Mining association rule consists of the following two steps: (1) finding the item sets that 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 of the three items *A*, *B* and *C* belongs to the frequent item set and *sup(A, B)* and *sup(C)* would be greater than *min_sup*. The support of an item set is defined as proportion of transactions that contains the item set. (2) generating interesting rules from the frequent item sets 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, then the rule can be considered as an interesting one. The performance of the association rule mining mostly depends on the first step i.e. generation of the frequent item set. As is evident, the algorithm does not require the details to be specified, such as like the number of dimensions for the tables or the number of categories for each dimension, as each item of transaction is considered. Therefore, this technique is particularly well-suited for data and text mining of huge databases.

Many times, an association rule mining algorithm generates extremely large number of association rules. In most cases, it is impossible for users to comprehend and validate a large number of complex association rules. Therefore, it is important to generate only "interesting" rules, "non-redundant" rules, or rules satisfying certain criteria, such as coverage, leverage, lift, or strength.

The Apriori algorithm is one of the most wildly used algorithms to generate association rules [2]. Apriori algorithm was designed to run on databases containing transactions. It is a 'bottom up' approach as candidate items are first generated and then the database is scanned to count the support for candidate items exceeding minimum support. The number of items in candidate subsets is increased one at a time with iteration. These candidate sets are converted to frequent subsets once their support count is matched with the required minimum support. The iteration would stop when no frequent subset can be generated. In the candidate generation phase, 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 a minimum of 15 times.

The Apriori Algorithm is based on the Apriori property:

- If an item set *I* does not satisfy the minimum support threshold (i.e., $P(I) < min_sup$), then *I* is not frequent.
- If an item *A* is added to the item set *I*, then the resulting item set (i.e., $I \vee A$) cannot occur more frequently than *I*, i.e., $P(I \vee A) < min_sup$.

The detailed Apriori Algorithm can be found in [2].

2) MapReduce and Hadoop: MapReduce is a programming paradigm and an associated parallel and distributed implementation for developing and executing parallel algorithms to process massive datasets on clusters of commodity machines [11]. Algorithms are specified in MapReduce using two functions: *map* and *reduce*. A MapReduce job usually splits the input data-set into independent chunks that are processed by the map task in a completely parallel manner. The map task maps the job into key and value. The framework sorts the outputs of the map tasks, which are then inputted to the reduce tasks. The input of reduce and output of map must have same data type. Typically both the input and the output of a job are stored in a file-system.

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

The MapReduce framework operates exclusively on <key, value> pairs. That is to say that the framework views the input of a job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job. The output of map and the output of reduce do not need to be of the same type. The output of reduce can be used as input for another (different) map function to develop MapReduce algorithms that complete in multiple rounds.

Hadoop is a free, Java-based programming framework that supports the processing of large data sets in a distributed computing environment [22]. Hadoop is designed to run on a large number of machines that donnt share any memory or disks. It creates clusters of machines and coordinates works.

Hadoop consists of two key services: reliable data storage using the Hadoop Distributed File System (HDFS) and high-performance parallel data processing using Map/Reduce.

HDFS splits user data across servers in a cluster. It uses replication to avoid data loss even when multiple nodes fail. It also keeps track of locations of replicated files. Moreover, Hadoop implements MapReduce, the aforementioned distributed parallel processing system. Hadoop was designed for clusters of commodity, shared-nothing hardware. No special programming techniques are required to run analyses in parallel using Hadoop MapReduce as most existing algorithms can work without many changes. MapReduce 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 some machines fail, 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.

B. *Related work*

Various methods were proposed to improve Apriori's efficiency of generating frequent item set [3, 4, 5]. Most of these works use hash to prune the candidate set. The earlier the frequent item sets are removed, the faster would be the mining. However, the trimming and pruning algorithms themselves are very difficult, which made them impractical in many cases [24]. An alternative approach was proposed in [6], in which candidate set is not generated for getting the frequent item set, rather a relatively compact tree-structure was created to alleviate the multi-scan problem and improve the candidate item set generation. The algorithm proposed in [6] requires only two full I/O scans of the dataset. It first creates a frequent-pattern tree (FP tree) using the frequent length-1 item as nodes of the tree. It also stores quantitative information about frequent patterns. The pattern growth is obtained by concatenating suffix patterns with the ones generated from the conditional FP tree. The algorithm of [6] uses divide and conqueror strategy to mine frequent patterns. This approach is efficient and widely used for generation of frequent patterns.

Researchers expect parallelism to relieve current association rule mining methods from the sequential bottleneck, providing scalability to massive datasets and improving response time [25]. In general, the parallel association rule mining can be categorized in two sections: 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 [27].

Ref. [7] first introduced the idea of local and global frequent item set. 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 item sets are globally frequent and generates the candidate item sets from those globally frequent item sets.

The Count Distribution (CD) algorithm is most direct parallelization of the Apriori algorithm [26]. It uses the sequential Apriori algorithm in a parallel environment and assumes datasets are horizontally partitioned among different sites. The global candidate item sets and frequent item sets are stored in each node. In each step, the algorithm counts the support of candidate item sets of the local data using the Apriori algorithm. At the end of each scanning, it exchanges local support counts with all other nodes to generate global support counts. The Count Distribution algorithm's main advantage is that it does not exchange data tuples between processors as it only exchanges the counts.

Ashrafi et al. focused on the communication and synchronization of the systems to reduce the number of candidate item sets in a distributed database system [1]. In another work, the authors collected information from each node on each item set from its local database by using a hashing method [8]. The information discovered by each node is then 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 item sets for the exchange of count information among nodes.

Hashing and Trie data structures were also used for reducing time taken for the generation of frequent item set in distributed systems [8, 9]. In [8], Ansari et al. developed a new distributed Trie-based algorithm (DTFIM) to find frequent itemsets. In the second phase, FDM algorithm was used for candidate generation step. Experimental evaluations showed it to be very effective algorithm.

III. METHODOLOGY

A. *Overview*

In this section, we present how to redesign the Apriori algorithm so that it can be effectively executed on MapReduce. We also illustrate how to implement the revised Apriori on Hadoop.

First, the input data set is saved on the Hadoop Distributed File System (HDFS) because HDFS can hold huge chunks of data and HDFS can help data localization for MapReduce task. The input data is divided into parts and allotted to the mapper. The output from the mapper is items (keys) and each item is count as value. Thereafter, the output is taken in by the combiner. The combiner combines all the count values related to a particular item (key). The result from the combiner is then taken in by the reducer for further combining and summing up of the values corresponding to an item (key). After getting the sum of all of the values of an item, the reducer checks whether the sum value exceeds a given threshold value; if so, the sum value along with the item are written to the output. The sum value of an item is discarded if it is less than the minimum support threshold value *min_sup*. This process would generate

frequent-1 item set. Frequent-1 item set 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 item set is checked using the input data assigned to the mapper. Again, the output is given to the combiner, which accumulates and sums the of candidate item set with the corresponding value. The "Reducer" further reduces and combines the data. In the end only those candidate sets whose value exceeds the value of minimum threshold are written to the output file. The same process is repeated. The candidate set is generated using previous iteration's frequent item set. The iteration stops when no more candidate set can be generated for an iteration, or the frequent item set of the previous iteration cannot be found.

Once the frequent-n item set is generated and saved as output, the association rules are generated. The association rule confidence calculation requires the support count of item set that forms the rule. The item set along with its support count is stored in the output file, each line with a different frequent item set tab-separated with the support count. The frequent-n item set resides in the output folder's 'n' named sub-folder. Once confidence is found to be 100% the rule is generated.

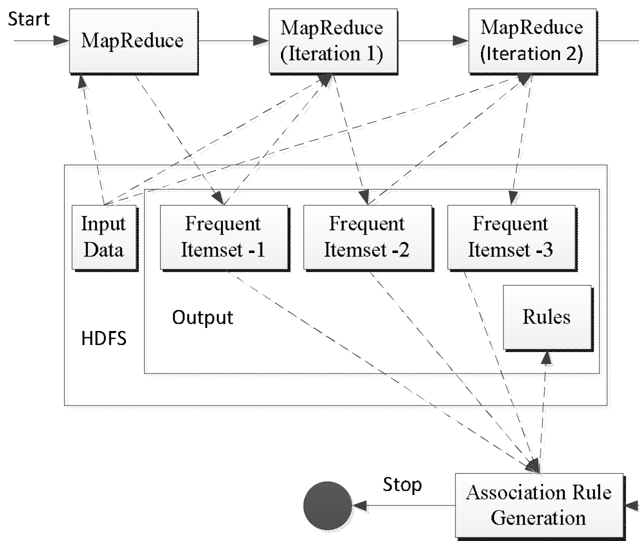

Figure 1. Data flow diagram showing two iterations

The Hadoop0.20.0 framework was used for implementation. Figure 1 shows the flow data for two iterations of the algorithm. Figure 2 shows the distributed data flow between methods Map, Combine, Reduce and HDFS. First, the map data is written in temporary files in HDFS that 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 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 of by the Hadoop framework.

In the following sections, we explain the major steps of the proposed algorithm.

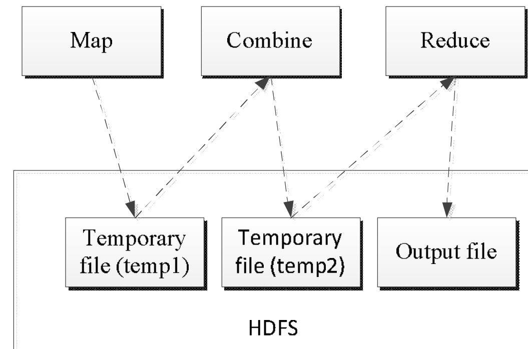

Figure 2. Distributed data flow between methods Map, Combine, Reduce and HDFS

B. Splitting the data set

The "*InputFormat*" method of Hadoop 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, which is fed to MapReduce 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 this 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 be operated on a single file in parallel. If the file is very large, then 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. Therefore, blocks can be operated locally instead of being transferred from one node to another.

Although the default number of logical splits is 5, the number of logical splits can be specified at run time. When splitting the file on the byte basis, we must make sure that a transaction is not split between two different splits. In the scenario when a transaction gets 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 "*InputFormat*" 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.

C. Revising Apriori for MapReduce

The split chunk of the data file is fed to Mapper. The Map reads one line at a time and assigns each item a key and the value associated with the key is 1 to find frequent-1 item set. 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 with the key. The combiner also sums up the list for each key. The output data from the mapper is written in the temporary files in HDFS to be used by combiner, so that combiner from any system can be allotted to work on it. Later the data written by combiner in the temporary file is then passed to the reducer. The reducer accumulates the count of values associated with respective keys. 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 whose value equal or exceeds the minimum support are written to the output file.

In the later iteration of MapReduce phase, the Mapper takes in the frequent item set from the previous iteration and generates the candidate item set. This candidate item set 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 item set generation step. The candidate item set is developed by using the Fast Apriori Algorithm [2]. The Fast Apriori helps in reducing the size of candidate item set as, it removes those item sets whose subset were absent in the previous iteration's output file.

The algorithms stops if candidate an item set could not be generated or the output file from the previous iteration is not present. The presence of the output file is checked before starting the next iteration. Once the frequent item sets are generated, association rules are developed. Figure 3 illustrates the execution of the revised Apriori algorithm on Hadoop MapReduce.

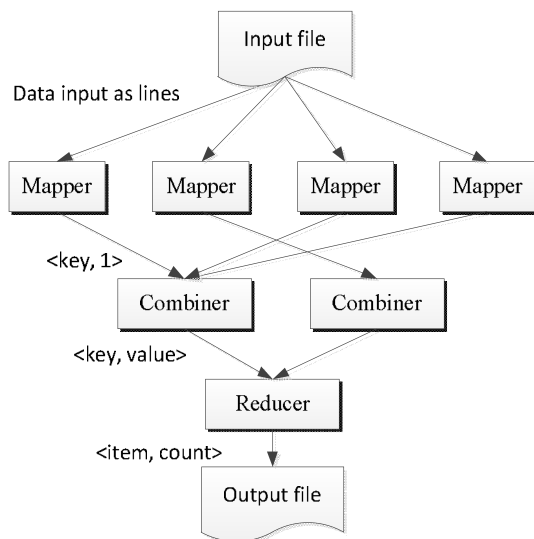

Figure 3. Excecution of converted Apriori on Hadoop MapReduce

D. Supporting from HDFS

The Hadoop Distributed File System (HDFS) will split large data files into chunks that are managed by different nodes in the cluster. Moreover, each chunk is replicated across several machines, so that a single machine failure would not cause data loss. The number of machines in which the data are replicated can also be configured. The number of replication should also 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 that can result in partial storage. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so that their contents are universally accessible. The input files 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. An output file in the Hadoop has to be copied back to local system to read the output.

The input files are saved within the Input folder of HDFS. The path of the input files can be specified during the run time. The output of frequent item set is saved within output folder under a sub-folder with an iteration number. The output file forms an input file for the next iteration for generating candidate frequent item set. It is also useful in generating association rule because they contain the frequent item set with the support count. Support count at each level of frequent item set is sufficient to calculate confidence for the corresponding association rule.

E. Generating rules from frequent item sets

The output folder contains the frequent items with their count after the successful run of certain iterations. 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 item set. If the confidence is 100% the rule is considered to be an interesting one. For example, the frequent item set $\{A,B,C\}$ from iteration 2 has *support_count* 50 and *support_count* of $\{B,C\}$ from iteration 1 is 50. The relation would be $\{B,C\} \Rightarrow \{A\}$, as the confidence is 100% and *support_count* > *min_sup*.

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

IV. IMPLEMENTATION ON EC2

We deployed the revised Apriori algorithm on Amzon EC2 (Elastic cloud computing) to evaluate the performance. The data input files were saved on S3. Amazon S3 is a data storage service. Data transference between Amazon S3 and Amazon EC2 is free. This makes S3 attractive for Hadoop EC2 users. Parameters, such as paths and permissions of S3 are configurable. The output data is also written back in buckets 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 Elastic MapReduce (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, such as setting up the hardware and networking required by the Hadoop cluster including monitoring the setup, configuring Hadoop, and executing the job flow.

Hadoop job flow on the EC2 cloud is shown in Figure 4. To start the job, a request is sent to the EMR model with parameters, such as path to S3. The Hadoop cluster with master and slave instances is created. The Hadoop cluster works on the job and finishes the job. The temporary files created during the execution of the job can be stored either on HDFS or on S3. Storing files on S3 would not be wise for our work because it adds communication overhead. The final output is stored in S3. Only the error or fatal messages are written on the screen during the entire execution of a job. Once the job is completed, a message is sent to the user indicating the completion of the job.

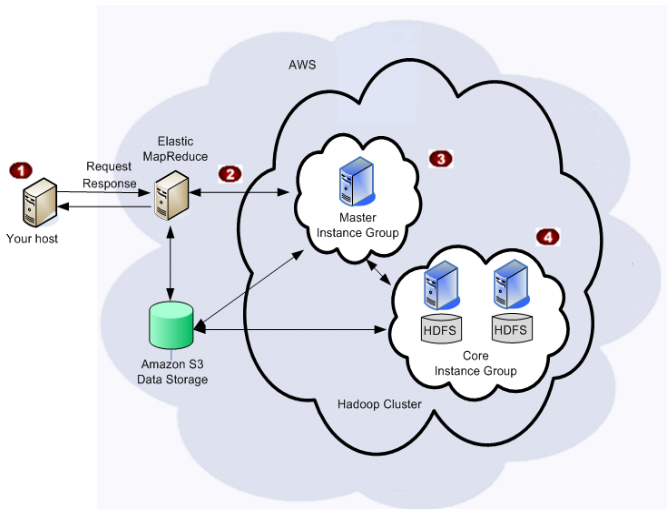

Figure 4. Job flow in the EC2 cloud for Hadoop [12]

V. EXPERIMENT

The revised Apriori algorithm was evaluated using well-known dataset available online. For frequent item sets, a benchmark of four datasets is commonly used [28]. In particular, the dataset of *chess*, *connect*, and *mushroom* are real-life datasets. They are downloaded from UCI data set repository [15]. *T10I4D100K* was a synthetically generated data set. It is created using the generator from the IBM Almaden Quest research group. However, the generator cannot be downloaded from their web site anymore. We downloaded the dataset from [11]. For all these datasets, we have studied the distributions of frequent item sets for many representative minimum support thresholds with respect to an item set size. The input data set were not compressed. Some of the details regarding the data sets are in Table I.

TABLE I. DETAILS OF THE DATA SETS

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

The data items are space separated in the input file. Each line in the input file gives a new tuple of the transaction. The *chess*, *connect*, *mushroom* data set are comma (,) separated files (.csv). The items are converted to numeric values, with each unique item denoting an integer. For example win 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,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. This will also add uniformity to the data set as now all the values of the same type are numeric. The items within a transaction get automatically arranged in the increasing order, if the values assigned to the columns are in increasing order.

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 millisecond.

TABLE II. EXECUTION TIME (IN SECS) FOR ALL DATA SETS

data set |threshold	.95	.90	.85	.80	.75
T10I4D100k	50.26	68.529	88.559	133.62 1	220.63 1
chess	108.18	156.32 6	220.12 2	392.92 7	878.15 6
mushroom	65.746	71.005	86.53	91.251	111.54 7
connect	84.612	241.17 2	681.57 6	1552.1 94	2934.7 92

Table II illustrates the execution time of our revised Apriori algorithm on different data sets. The threshold is the minimum support (*min_sup*) divided by the total number of the transactions in the data set. The value of the threshold lies between 0 and 1. For example, the threshold value .95 for *chess* dataset implies the minimum support considered is 3037 (threshold × total number of transactions in the data set). Threshold is used here because the execution time can be compared among the datasets each of which has different number of transactions (as they will have different minimum support count).

The performance of different data sets is tested on a single node Hadoop system. This gives us an idea on how different data set would perform for the algorithm. The data set *Connect* takes the maximum time although the *T10I4D100K* dataset has maximum number of transactions. The reason is that the number of items are denser in *Connect* and the number of 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.

Figure 5 shows the performance of Apriori algorithm on a single-node Hadoop system. We also compared the performance of different datasets on a single non-Hadoop machine for the same algorithm implementation. The performance of the algorithm in a single non-Hadoop node is better than Hadoop node. This is because much of the Hadoop time span is spent providing the reliability to the system, and other tasks such as dividing data and writing result.

The performances of the revised Apriori algorithm over different dataset on multi-node Hadoop cloud are illustrated in Figure 6, Figure 7, and Figure 8 respectively.

The execution time range of *Connect* varies from 0-2500 while the execution of range of T10I4D100k lies in 0-250 and 0-750 for chess. 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 multiple iterations, the number of times data set would be read will depend on the number of iteration the program will run before it terminates. Therefore, the number of iterations and the size of dataset would play important factor in affecting the run time.

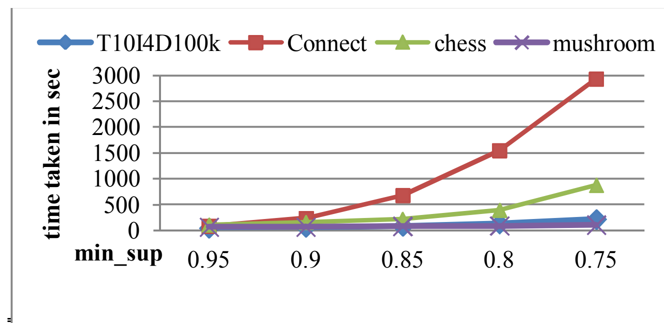

Figure 5. Execution time of the revised Apriori Algorithm on different datasets with varying minimum support on single-node Hadoop.

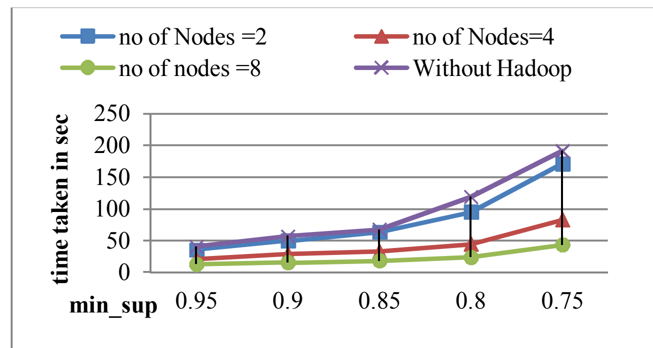

Figure 6. Execution time of the revised Apriori Algorithm on on the synthetic dataset *T10I4D100k* with varying minimum support

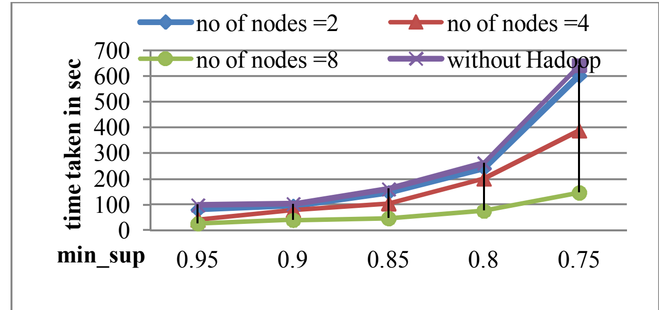

Figure 7. Execution time of the revised Apriori Algorithm on the dense dataset *chess* with varying minimum support

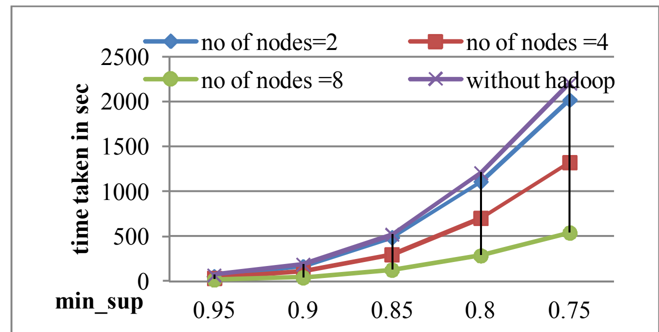

Figure 8. Execution time of the revised Apriori Algorithm on the dense and large dataset *Connect* with varying minimum support

Comparing Figure 6, Figure 7 and Figure 8, we found that when applying the revised Apriori algorithm on multiple Hadoop nodes, the performance over the connect dataset is better than all other datasets. The reduction of execution time is more evident for those datasets which have longer execution time. Moreover, we can see that when the size of the dataset increases, the performance on multi-node Hadoop also gets better. This reinstates the fact that Hadoop will give a much better performance for larger datasets. The performance also depends on the number of nodes in the cluster. In our experiments, with the increase of the number of nodes in the cluster, the performance improves.

VI. CONCLUSIONS

Cloud computing provides cost-efficient solutions of storing and analyzing mass data. In this paper, we have studied parallel association rule mining strategy in the cloud computing environment. In particular, we implemented the improved Apriori Algorithm on MapReduce programming model on the Hadoop platform. The improved algorithm can scale up to large data set with comparatively less cost. Moreover, this distributed algorithm can also cater to the distributed nature of the input data. Furthermore, details of the management of the distributed systems, such as data transferring among nodes and node failures are taken care by Hadoop, which adds a great deal of robustness and scalability to the system. Based on our evaluation, we found that the improved Apriori algorithm achieves best performance for dense and large data sets.

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/Introduct ion_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.

[15] T. Hey, S. Tansley, and K. Tolle. The Fourth Paradigm: Data-Intensive Scientic Discovery. Microsoft Research, 2009.

[16] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: an architecture-conscious solution. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP ’07, pages 2–12, New York, NY, USA, 2007. ACM.

[17] M. El-Hajj and O. Zaiane. Parallel leap: large-scale maximal pattern mining in a distributed environment. In Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on, volume 1, page 8 pp., 0-0 2006.

[18] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang. Parallel data mining on graphics processors. Technical Report 07, The Hong Kong University of Science & Technology, 2008.

[19] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP ’05, pages 255–265, New York, NY, USA, 2005.

[20] PIETRACAPRINA, A., RIONDATO, M., UPFAL, E., AND VANDIN, F. 2010. Mining top-K frequent itemsets through progressive sampling. Data Mining and Knowledge Discovery 21, 310–326. ACM.

[21] D. Gillick, A. Faria, and J. Denero, “Mapreduce: Distributed computing for machine learning,” 2008. [Online]. Available:

[22] http://citeseerx.ist.psu.edu/viewdoc/summary10.1.1.111.9204

[23] http://hadoop.apache.org/

[24] J.D. Holt and S.M. Chung, Mining association rules using inverted hashing and pruning, Information Processing Letters 83(4) (2002).

[25] ZAKI, M. J. 1999. Parallel and distributed association mining: A survey. IEEE Concurrency (Special Issue on Parallel Mechanisms for Data Mining.) 7, 4, 14–25.

[26] R. Agrawal and J.C. Shafer , "Parallel Mining of Association Rules,"IEEE Tran. Knowledge and Data Eng. , vol. 8, no. 6, 1996,pp. 962-969.

[27] Kamath C. and Musik R., Scalable Data Mining through Fine-Grained Parallelism, in Advances in Distributed and Parallel Knowledge Discovery, P.C. H. Kargupta, Editor. 2000, AAAI Press / The MIT Press.

[28] Grahne, G., & Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. In R. J. Bayardo Jr., B. Goethals, & M. J. Zaki (Eds.), Journal of Intelligent Information *Systems*.