YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark

Hongjian Qiu, Rong Gu, Chunfeng Yuan, Yihua Huang*
Department of Computer Science and Technology, Nanjing University
National Key Laboratory for Novel Software Technology, Nanjing University
Nanjing 210023, China
qiuhongjian_nju@hotmail.com, gurongwalker@gmail.com, cfyuan@nju.edu.cn, yhuang@nju.edu.cn

***Abstract*—The frequent itemset mining (FIM) is one of the most important techniques to extract knowledge from data in many real-world applications. The Apriori algorithm is the widely-used algorithm for mining frequent itemsets from a transactional dataset. However, the FIM process is both data-intensive and computing-intensive. On one side, large scale data sets are usually adopted in data mining nowadays; on the other side, in order to generate valid information, the algorithm needs to scan the datasets iteratively for many times. These make the FIM algorithm very time-consuming over big data. The parallel and distributed computing is effective and mostly-used strategy for speeding up large scale dataset algorithms. However, the existing parallel Apriori algorithms implemented with the MapReduce model are not efficient enough for iterative computation. In this paper, we proposed YAFIM (Yet Another Frequent Itemset Mining), a parallel Apriori algorithm based on the Spark RDD framework—a specially-designed in-memory parallel computing model to support iterative algorithms and interactive data mining. Experimental results show that, compared with the algorithms implemented with MapReduce, YAFIM achieved 18× speedup in average for various benchmarks. Especially, we apply YAFIM in a real-world medical application to explore the relationships in medicine. It outperforms the MapReduce method around 25 times.**

Keywords-Frequent Itemset Mining; Apriori Algorithm; Parallel Computing; Spark; Medical Application

I. INTRODUCTION

Frequent itemset mining (FIM) is a kind of important data analysis and data mining applications. It aims to extract information from databases based on frequently occurring events. The Apriori algorithm [1], in which the frequency counting is achieved by scanning the dataset repeatedly for different sizes of candidate itemsets, is one of the well-known algorithms for FIM.

Many single-node techniques have been put forward to mine databases for frequent events [1, 2, 3]. These techniques work well on small datasets. However, as the datasets grow larger and larger, due to the limit memory capacity and computation capability of a single node, these methods become inefficient to mine frequent itemsets over big data. The memory requirements for handling the complete set of candidate itemsets blow up fast and the computational cost can be expensive on a single machine.

On the other side, parallel programming is more and more used to accelerate processing of the massive amount of data. Some researchers have implemented several parallelized Apriori algorithms with the MapReduce framework and achieved certain speedup compared with the single-node methods. However, the MapReduce framework [4, 5] is not suited for the FIM algorithm like the Apriori algorithm with intensive iterated computation.

In this paper, we put forward YAFIM (Yet Another Frequent Itemset Mining), a parallel Apriori algorithm based on the Spark, an in-memory-based and iterative computing model and framework. First we load transactional datasets from HDFS into the Spark RDDs (Resilient Distributed Datasets), the memory-based data objects in Spark. Then we iteratively use k-frequent itemset to generate (k+1)-frequent itemset. This way we can make good use of the cluster memory. The experimental results show that YAFIM outperforms the methods with MapReduce with speedup of 18× in average on many benchmarks. Further, we also applied YAFIM in a real-world medical application to mine the relationships among medical entities. The result shows that YAFIM is 25 times faster than the MapReduce-based methods.

The rest of this paper is organized as follows. Section II briefly describes the concepts of frequent itemset mining, the Apriori algorithm and Spark. Section III discusses related work. Section IV introduces the YAFIM algorithm in detail. Section V evaluates the performance of YAFIM and compares it with current methods. In this section, we also apply our YAFIM algorithm in a real-world medical case data. Section VI concludes the paper.

II. BACKGROUND

A. Frequent Itemset Mining and the Apriori Algorithm

Let $I = \{i_1, i_2, i_3, \cdots, i_n\}$ be a set of items and a transaction is defined as $T = (tid, X)$, where tid is a transaction identifier and X is a subset of items from I. A transactional database D consists of a set of such transactions. The support of an itemset Y is defined as the number of transactions that contain the itemset. Formally,

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.185

$$\sup(\mathrm{Y}) = \left|\left\{tid \mid \mathrm{Y} \subseteq \mathrm{X}, (\mathrm{tid}, \mathrm{X}) \in \mathrm{D}\right\}\right| \qquad (1)$$

An itemset is said to be frequent if its support is greater than a given threshold σ , which is called *minimum support* or *MinSup* in short.

The Apriori algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset property that all nonempty subsets of a frequent itemset must also be frequent [6]. Based on this fact, we will start the FIM process from finding the frequent itemsets with 1 item first (denoted as 1-frequent itemset). Then we will find k-frequent itemsets based on (k-1)-frequent itemsets.

Algorithm 1 [1] describes the pseudocode for the Apriori algorithm. First of all, it starts finding the frequent items by making a pass over D . Then, the algorithm iteratively continues to extend k-length candidate itemsets and counts their supports by making another pass over D to check whether these candidate itemsets are frequent. Exploiting the monotonic property, Apriori prunes those candidates for which a subset is known to be infrequent. Depending on the minimum support threshold used, this greatly reduces the search space of candidate itemsets.

Algorithm 1 Apriori Algorithm

```
$Apriori(\mathrm{T}, \sigma)$
  $L_1 \leftarrow \{\mathrm{large1-itemsets}\}$
  $k \leftarrow 2$
    **while** $L_k \neq emptyset$
      $C_k \leftarrow \left\{a \cup \{\mathrm{b}\} \mid \mathrm{a} \in L_{k-1} \wedge b \in \bigcup L_{k-1} \wedge b \notin a\right\}$
      **for** $transactions\ t \in T$
        $C_t \leftarrow \left\{c \mid c \in C_k \wedge c \subseteq t\right\}$
        **for** $candidates\ c \in C_t$
          $count\left[c\right] \leftarrow \mathrm{count}\left[c\right] + 1$
      $L_k \leftarrow \left\{c \mid c \in C_k \wedge count\left[c\right] \geq \sigma\right\}$
      $k \leftarrow k + 1$
    **return** $\bigcup_k L_k$
```

B. The Spark Parallel Computing Framework

Spark [7] is a distributed computing framework developed at the Berkeley AMPLab. It offers a number of features to make big data processing fast. The fundamental feature that especially benefits the iterative computations is its in-memory parallel execution model in which all data will be loaded into memory. The second key feature is that, deferring from the fixed two-stage data flow model in MapReduce, Spark can provide very flexible DAG-based (directed acyclic graph) data flow. These two features can significantly speedup the computation for those iterative algorithms such as the Apriori algorithm and some of other machine learning algorithms. This is how Spark can achieve 1-2 orders of magnitude faster than MapReduce.

The programming model of Spark is upon a new distributed memory abstraction called *resilient distributed datasets (RDDs)* which is proposed for in-memory computations on large cluster. An RDD is an immutable collection of data records. It can provide a variety of built-in operations to transform one RDD to another RDD. Spark will cache the contents of the RDDs in memory on the worker nodes, making data reuse substantially faster. RDDs can achieve fault-tolerance based on lineage information rather than replication. Spark tracks enough information to reconstruct RDDs when a node fails.

III. RELATED WORK

A number of existing algorithms on FIM have been proposed in past decades [9-14], including the MapReduce-based parallel FIM algorithms emerging in recent years to deal with large scale datasets.

In general, we can classify the existing parallel FIM algorithms into two categories: one-phase and k-phase algorithms. One-phase algorithms need only one phase (e.g., a MapReduce job) to find all frequent k-itemsets [15]. The one-phase algorithm needs to generate many redundant itemset during processing, which may lead memory overflow and too much execution time for large data sets. The k-phase algorithm (k is maximum length of frequent itemsets) needs k phases (e.g., k iterations of a MapReduce job) to find all frequent k-itemsets [16, 17, 18]: phase one to determine 1-frequent itemsets, phase two the 2-freuent itemsets, and so on.

Lin et al. proposed three algorithms that are adaptations of Apriori on MapReduce: SPC, FPC and DPC. SPC is one-phase and two others are k-phase. These algorithms distribute the datasets to mappers to perform the counting step in parallel [17]. The PApriori algorithm proposed by Li et al. is very similar to SPC. The map function performs the procedure of counting each occurrence of all the candidates in a parallel way and then the reduce function sums up the occurrence [16]. Nguyen et al. aimed at mining high dimensional correlated subspaces for knowledge discovery in multi-dimensional data in Apriori processing through theoretical analysis [23]. Sandy Moens et al. introduced two new methods for mining large datasets: Dist-Eclat and BigFIM. Dist-Eclat is a MapReduce implementation of the well known Eclat algorithm which focuses on speed while BigFIM is optimized to deal with truly Big Data by using a hybrid algorithm with both Apriori and Eclat on MapReduce [24]. S. Hammoud proposed a method to iteratively switch between vertical and horizontal database layouts to mine all frequent itemsets. At each iteration the database is partitioned and distributed across mappers for frequency counting [19].

All the above work was intended to parallelize the Apriori algorithm in MapReduce framework. Actually, the iterative and level-wise structure of the Apriori-based algorithms does not fit well into the MapReduce framework because in each round a new MapReduce job needs to read the data from and write back to HDFS [25], which will create a lot of overhead on I/O and thus heavily increase the time cost. Unlike these methods, the YAFIM algorithm we propose can parallelize the FIM processing over the Spark RDD and make better use of memory and avoid many overheads.

IV. The YAFIM Algorithm

A. Design

YAFIM is a parallel Apriori algorithm based on the Spark RDD model. It contains with two phases in processing workflow.

1) *Phase I :* we load transaction datasets from HDFS into the Spark RDD and make good use of cluster memory.

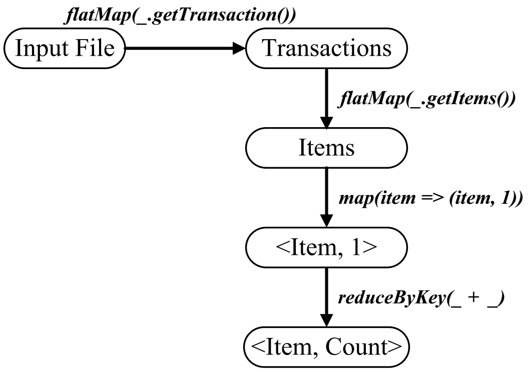


Figure 1. Lineage graph for the RDDs in YAFIM of Phase I.

Algorithm 2 Phase I of YAFIM

```
Foreach $transaction\ t\ in\ S_i$
  flatMap ($line\ offset, t$)
    Foreach $item\ I\ in\ t$
      $Out(I,1)$
    End Foreach
  End flatMap
End Foreach
reduceByKey ($I, count$)
  $sum = 0$
  While (Item I in partition)
    $sum += count$
  End While
  If (sum ≥ min_sup)
    $Out(I, sum)$
  End if
End reduceByKey
```

As illustrated in Fig. 1, first, the transactional dataset input file will be applied a *flatMap()* function (executed by workers) to read all transactions and put them into the Transactions RDD. Then a *flatMap()* function will be applied on each of transactions to get all items from them. When the flatMap process is finished, the *map()* function is applied to transform all items to the <Item, 1> key/value pairs. Finally, the *reduceByKey()* function starts to count the frequency of each items in the transactional dataset and also prunes the items which frequency are less than the minimum support σ . All items above the threshold will fall into 1-frequent itemset. The Algorithm 2 describes the pseudocode for the process.

2) *Phase II :* in this phase, we iteratively using k-frequent itemsets to generate (k+1)-frequent itemsets.

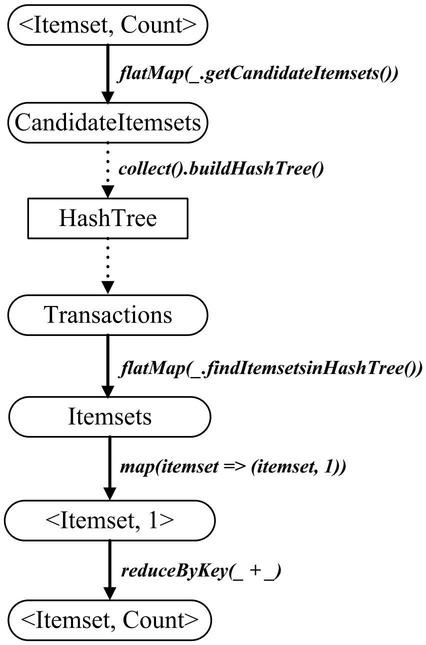

<Itemset, Count>
flatMap(_.getCandidateItemsets())
CandidateItemsets
collect().buildHashTree()
HashTree
Transactions
flatMap(_.findItemsetsinHashTree())
Itemsets
map(itemset => (itemset, 1))
<Itemset, 1>
reduceByKey(_ +_)
<Itemset, Count>

Figure 2. Lineage graph for the RDDs in YAFIM of Phase II.

As illustrated in Fig. 2 and described in Algorithm 3, we read the k-frequent itemsets L_k and store them as RDDs in the form of <itemset, count>. Then we will get candidate itemsets C_{k+1} from them. To speed up the searching speed of finding (k+1)-frequent itemsets from the candidate itemsets, we store C_{k+1} in a hash tree. Then we scan the existing Transactions RDD to list all occurances of each of candidate itemsets by applying a *flatMap()* function. Further we apply a *map()* function to emit the <itemset, 1> key/value pairs for each of occurring itemsets. Finally the *reduceByKey()* function collects all the support counts of each of candidate itemsets and output the <itemset, count> pairs as (k+1)-frequenct itemsets when the count is no less than the minimum support count σ . The Algorithm 3 describes the pseudocode.

Algorithm 3 Phase II of YAFIM

```
Read $L_{k-1}$ from RDD
$C_k = ap\_gen(L_{k-1})$
Foreach transaction t in $S_i$
    flatMap $(line\ offset, t)$
        $C_t = subset(C_k, t)$
        Foreach candidate c in $C_t$
            $Out(c,1)$
        End Foreach
    End flatMap
End Foreach
reduceByKey $(c, count)$
    $sum = 0$
    While (Item c in partition)
        $sum += count$
    End While
    If $(sum \geq min\_sup)$
        $Out(c, sum)$
    End if
End reduceByKey
```

B. Memory Utilization

As discussed in the previous subsection, the YAFIM algorithm consists of two phases: the phase I is to get frequent 1-itemsets and the phase II is to generate (k+1)-frequent itemsets from k-frequent itemsets iteratively. Each iteration of the YAFIM algorithm will reuse the same dataset of original transactional data. In phase I, the code thus starts by loading the transactional dataset into an in-memory cached RDD, by passing a text file through a *getTransaction()* function that reads each line of text into a class representing the transaction. We then reuse this RDD to generate frequent itemsets. Phase II consists of the join and prune steps. These steps can readily be expressed as a MapReduce computation, which we implement in Spark using the *flatMap()* and *reduceByKey()* operations.

Our proposed algorithm takes the advantage of the Spark RDD model. We store all data as RDDs in distributed workers of the cluster and achieve in–memory computation. In this way, the transactional dataset only needs to be loaded in the very beginning and will be held in the memory as much as possible and the computations for later iterations will be much faster.

C. Share Data With Boradcast

In YAFIM, we need to share some data to the worker nodes at each iteration. To achieve this, we need to broadcast the transactional data to all the nodes at the start of the job and when frequent itemsets are generated after each iteration.

A naïve way is to package the transactional data needed for each task and send to the cluster, which was the default behavior in Spark. However, this will result in poor performance for YAFIM as the master node's bandwidth becomes a bottleneck, capping the rate at which tasks could be launched and further limiting the scalability. For larger dataset even reading the data once per node from a distributed file system is a bottleneck. To mitigate the problem of large datasets, we adopt an advanced feature called *broadcast variables abstraction* in the Spark, which allows programmers to send a piece of shared data to each slave only once, rather than with every task that uses the data [8]. We use the broadcast variables to send static data that are used throughout the job. This can improve the speed of the overall job processing.

V. PERFORMANCE EVALUATION

In this section, we evaluated the performance for YAFIM by comparing it with MRApriori [16]. YAFIM is the first implementation of the Apriori algorithm in the Spark framework and MRApriori is a typical implementation of the parallel Apriori algorithm based on MapReduce [22] with good performance. Other MapReduce implementations of the Apriori algorithm are the same as MRApriori in solution that in each round a new MapReduce job reads the data from and writes back to disk iteratively. As a result, the performance of MRApriori is almost the same as that of other MapReduce implementations.

A number of benchmark datasets were used in our experiments. All experiments were executed three times and the average was taken as experimental results. We implemented the YAFIM algorithm in Spark-0.7.3 and run MRApriori in Hadoop-1.0.4. All the data were stored on the same HDFS cluster. We made our experiments on a cluster consisting of 12 nodes, where each node has two Intel Quad Core Xeon CPUs running at 2.4 GHz, 24 GB memory and a 2TB disk. The computing nodes are all running at the RedHat Enterprise Linux Server 6.0 and Java 1.6.0.

Further, we also evaluated the scalability performance of YAFIM in terms of the sizeup and speedup that are important factors of a parallel algorithm. The sizeup analysis holds the number of cores in the system constant and grows the size of the datasets to measure whether a given algorithm can deal with larger datasets. The speedup measures how much faster a parallel algorithm is than a corresponding sequential algorithm by growing the number of cores in the system while keeping the size of datasets constant. The correctness of YAFIM is also evaluated and all the experimental results of YAFIM are exactly same as MRApriori.

A. Datasets

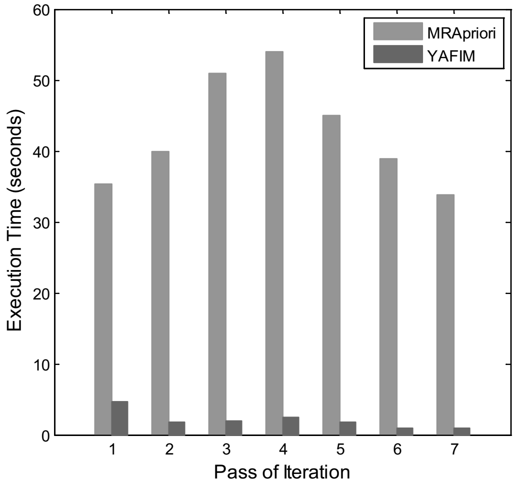
We ran experiments with both programs on four large scale data sets with different characteristics. The data sets we used include MushRoom (a data set describing poisonous and edible mushrooms by different attributes [21]), T10I4D100K (an artificial data set generated with IBM's data generator [20]), Chess (a data set listing chess end game positions for king vs. king and rook) and Pumsb_star [26]. The first three data sets are available from the UCI machine learning repository [21]. Properties of these datasets are given in Table I.

TABLE I. PROPERTIES OF DATASETS FOR OUR EXPERIMENTS

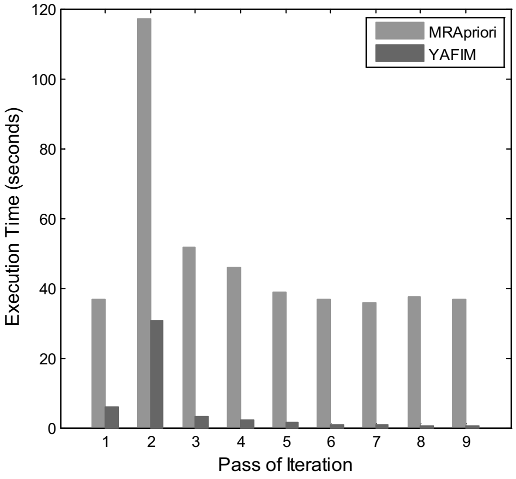
Dataset	Number of Items	Number of Transactions
MushRoom	119	8,124
T10I4D100K	870	100,000
Chess	75	3,196
Pumsb_star	2,113	49,046

B. *Speed Performance Analysis*

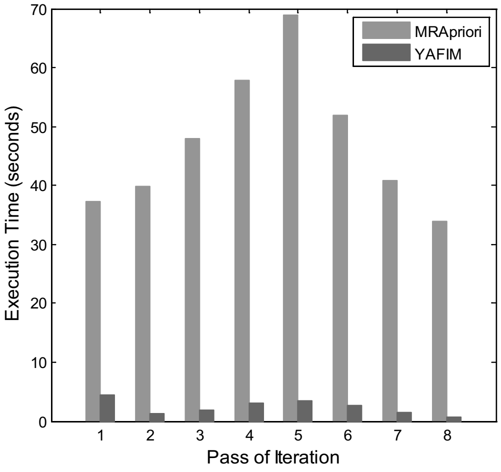
Fig. 3 shows the performance of two algorithms with different data sets in each iteration. We can see that in each iteration, YAFIM takes much less time than MRApriori in all different datasets. For the MushRoom dataset, the total execution time cost is about 14s for YAFIM while 297s for MRApriori, which is about 21 times faster. Especially, in the last pass of iterations for the MushRoom dataset, it takes 0.9s for YAFIM and 34s for MRApriori, which is about 37 times faster. In the Chess dataset, the total execution time cost is 18s for YAFIM and 378s for MRApriori, which is about 20 times faster. Moreover, in the last pass of iterations for the Chess dataset, YAFIM takes 0.6s and MRApriori takes 34s, achieving nearly 55× speedup. For the T10I4D100K dataset, the total execution time for YAFIM is about 10 times faster than that for MRApriori. In the Pumsb_star dataset, it is approximately 21 times faster.



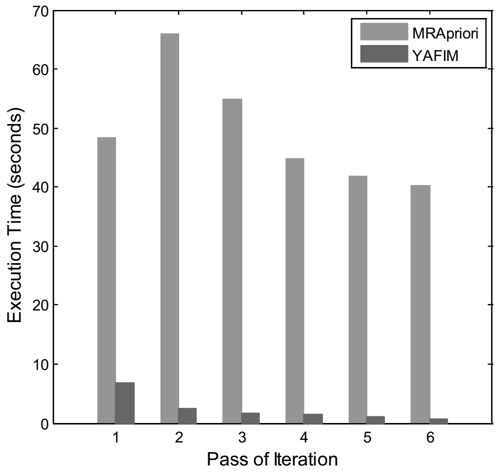
(a) MushRoom: Sup = 35%.



(b) T10I4D100K: Sup = 0.25%.



(c) Chess: Sup = 85%.



(d) Pumsb_star: Sup = 65%.

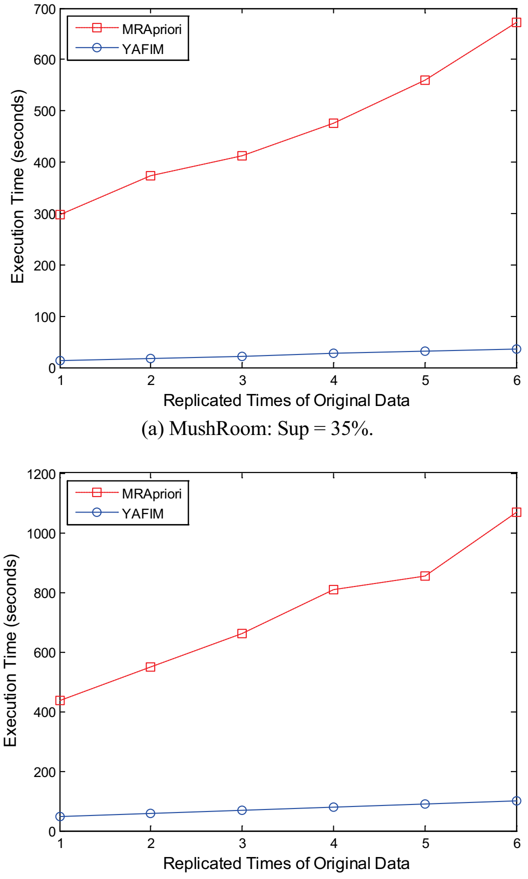
Figure 3. Performance evaluation of each pass in YAFIM and MRApriori. X-axis is the pass of the iteration, Y-axis is the execution time.

As shown from Fig. 3, YAFIM outperforms MRApriori about 18× in average for four different benchmarks in speed performance.

C. *Scalability Performance Analysis*

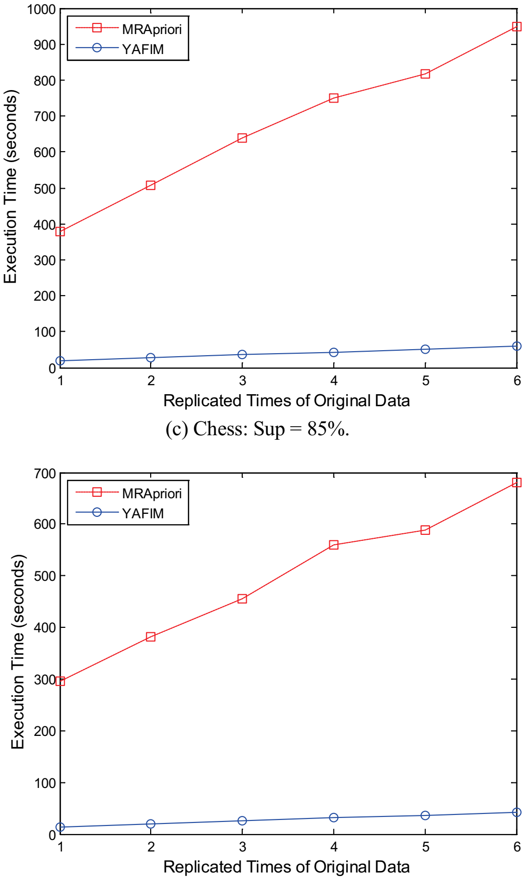
To measure the performance of sizeup, we fix the number of cores to 48. We replicate four datasets to 2, 3, 4, 5 and 6 times in size to get larger datasets in order to make experiments to examine the data scalability of YAFIM. Fig. 4 shows the data scalability performance. The x-axis is the replicated times of data size of original datasets. As shown in Fig. 4, the execution time cost for MRApriori increases sharply and almost grows linearly when the size of the datasets increases. In contrast, our YAFIM algorithm grows slowly and keeps nearly flat for all benchmarks. This is the result of memory computing and broadcasting that reduces the overall time spent in I/O and network communication.

The results show better performance for YAFIM than that for MRApriori.



(a) MushRoom: Sup = 35%.

(b) T10I4D100K: Sup = 0.25%.

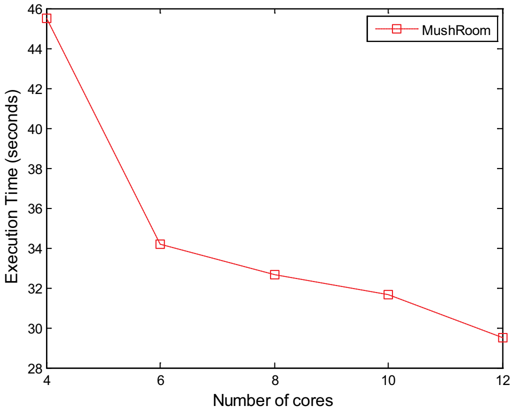


(c) Chess: Sup = 85%.

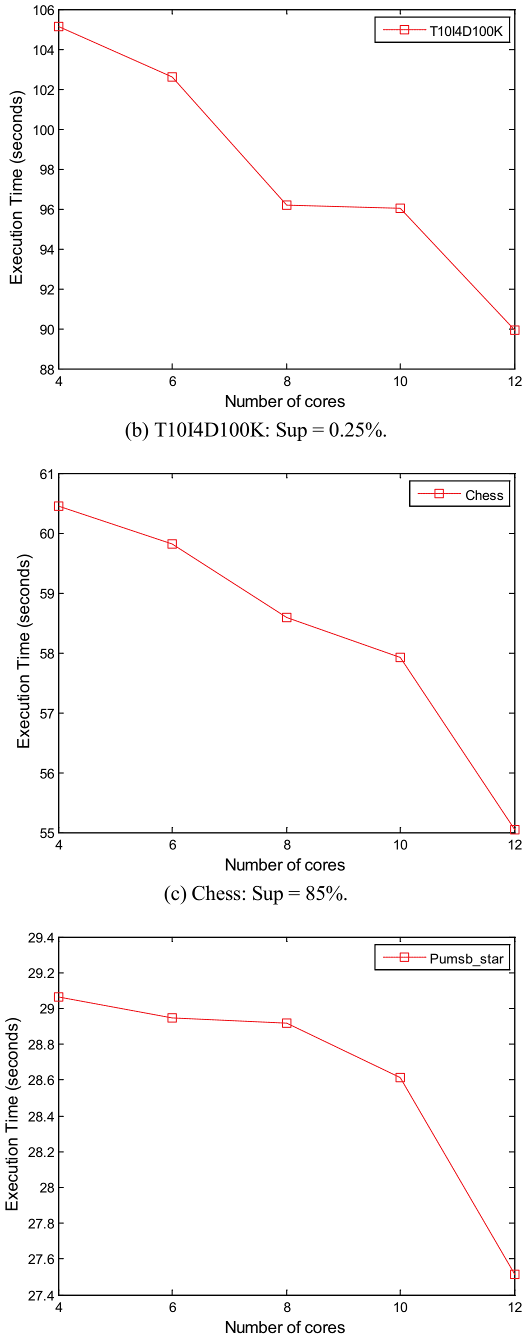
(d) Pumsb_star: Sup = 65%.

Figure 4. Sizeup performance evaluation of different datasets.

To measure the node salability, we kept the dataset fixed and varied the number of running nodes. The number of nodes varies from 4, 6, 8, and 10 to 12. Fig. 5 shows the speedup for different datasets. The x-axis is the number of the cores. We can see that as the number of running nodes increases, the time cost for YAFIM goes near-linear. The result indicates that YAFIM can achieve near-linear scalability in performance.



(a) MushRoom: Sup = 35%.



(b) T10I4D100K: Sup = 0.25%.

(c) Chess: Sup = 85%.

(d) Pumsb_star: Sup = 65%.

Figure 5. Speedup performance evaluation of different datasets in YAFIM.

D. Application in Healthcare and Medicine

We also apply the YAFIM algorithm into a medical text semantic analysis application to evaluate its performance. Healthcare is a typical domain with big data. Hospitals and other health related organizations have collected huge volumes of data, thus attracting data mining researchers to explore and find more facts and knowledge. FIM is initially used in sales-purchase domain and now has been used in various fields such as healthcare and medical applications. Resemblance between medical case and sales-purchase bill is the motivation of using FIM in this research work. We apply our YAFIM algorithm in medical case data to find the relationship in medicine.

Compared with the MRApriori algorithm, our YAFIM algorithm is 25 times faster for medical case dataset analysis. In every pass of iterations, the execution time of YAFIM is far less than MRApriori. Moreover, the execution time of each iteration becomes less and less with the increase of iterations. The first reason is that YAFIM just reads the dataset from disk into memory at the beginning and then computes data in the memory later. The second reason is that with the increase of iterations, the size of frequent itemsets decreases. In contrast, the MRApriori algorithm needs to scan data from disk in every job. Fig. 6 shows the comparison between YAFIM and MRApriori. The x-axis is the pass of the iterations.

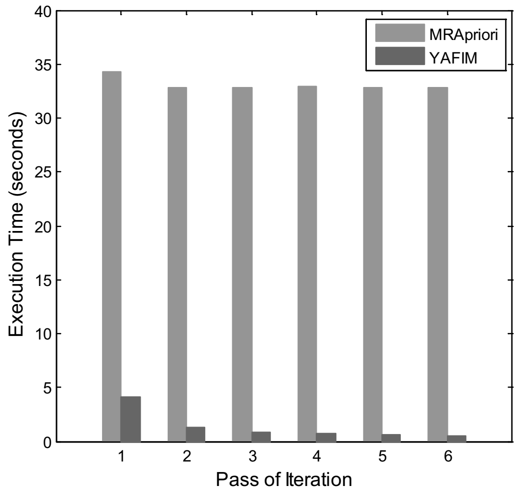


Figure 6. Comparison between YAFIM and MRApriori in real-word medical case data (Sup = 30%)

VI. Conclusion

FIM is an important technique to extract knowledge in data mining and Apriori is one of the most typical algorithms for this task. However, FIM is both data-intensive and computing-intensive and it is a challenge to develop fast and efficient algorithm that can handle large scale datasets.

In this paper, we proposed YAFIM, a parallel FIM algorithm implemented with the Spark framework. It contains with two phases in processing workflow. In Phase I, we load the transactional datasets from HDFS into the Spark RDD and generate 1-frequent itemset. In Phase II, we iteratively use k-frequent itemsets to generate (k+1)-frequent itemsets. We store all the data as RDDs in Spark and make good use of memory. Our work affirmed the advantage of in-memory computation model from Spark for iterative algorithms compared with MapReduce-based algorithms.

Our experiments show that YAFIM is about 18× faster than Apriori algorithms implemented in MapReduce framework. Furthermore, we can achieve a better performance in both sizeup and speedup for different datasets. In addition, we also evaluated YAFIM for medical application and revealed that YAFIM outperforms MRApriori about 25× speedup.

ACKNOWLEDGMENT

This work is funded in part by China NSF Grants (No. 61223003), and the USA Intel Labs University Research Program.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In Proc. VLDB, pages 487–499, 1994.

[2] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. SIGMOD Rec., pages 85–93, 1998.

[3] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. Data Min. and Knowl. Disc., pages 343–373, 1997.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Proc. OSDI. USENIX Association, 2004.

[5] Apache hadoop. http://hadoop.apache.org/, 2013.

[6] Jiawei Han and Micheline Kamber. Data Mining, Concepts and Techniques. Morgan Kaufmann, 2001

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.

[9] J. Han, J. Pei, and Y. Yin: Mining Frequent Patterns without Candidate Generation. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 29(2):1-12, 2000.

[10] M. J. Zaki. Parallel and distributed association mining: A survey. IEEE Concurrency, pages 14–25, 1999.

[11] J. Li, Y. Liu, W.-k. Liao, and A. Choudhary. Parallel data mining algorithms for association rules and clustering. In Intl. Conf. on Management of Data, 2008.

[12] E. Ozkural, B. Ucar, and C. Aykanat. Parallel frequent item set mining with selective item replication. IEEE Trans. Parallel Distrib. Syst., pages 1632–1640, 2011.

[13] B.-H. Park and H. Kargupta. Distributed data mining: Algorithms, systems, and applications. 2002.

[14] L. Zeng, L. Li, L. Duan, K. Lu, Z. Shi, M. Wang, W. Wu, and P. Luo. Distributed data mining: a survey. Information Technology and Management, pages 403–409, 2012.

[15] Li L. & Zhang M. (2011). The Strategy of Mining Association Rule Based on Cloud Computing. Proceeding of the 2011 International Conference on Business Computing and Global Informatization (BCGIN '11). Washington, DC, USA, IEEE: 475- 478.

[16] Li N., Zeng L., He Q. & Shi Z. (2012). Parallel Implementation of Apriori Algorithm Based on MapReduce. Proc. of the 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD '12). Kyoto, IEEE: 236 – 241.

[17] Lin M., Lee P. & Hsueh S. (2012). Apriori-based Frequent Itemset Mining Algorithms on MapReduce. Proc. of the 16th International Conference on Ubiquitous Information Management and Communication (ICUIMC '12). New York, NY, USA, ACM: Article No. 76.

[18] Yang X.Y., Liu Z. & Fu Y. (2010). MapReduce as a Programming Model for Association Rules Algorithm on Hadoop. Proc. of the 3rd International Conference on Information Sciences and Interaction Sciences (ICIS '10). Chengdu, China, IEEE: 99 – 102.

[19] S. Hammoud. MapReduce Network Enabled Algorithms for Classification Based on Association Rules. Thesis, 2011.

[20] Synthetic Data Generation Code for Associations and Sequential Patterns. Intelligent Information Systems, IBM Almaden Research Center. http://www.almaden.ibm.com/software/quest/Resources/index.shtml.

[21] C.L. Blake and C.J. Merz. UCI Repository of Machine Learning Databases. Dept. of Information and Computer Science, University of California at Irvine, CA, USA. 1998. http://www.ics.uci.edu/mlearn/MLRepository.html.

[22] HadoopApriori. https://github.com/solitaryreaper/HadoopApriori.

[23] H.V. Nguyen, E. Muller, K. Bohm. 4S: Scalable Subspace Search Schema Overcoming Traditional Apriori Processing. 2013 IEEE International Conference on Big Data. 2013.

[24] S. Moens, E. Aksehirli and Goethals. Frequent Itemset Mining for Big Data. University Antwerpen, Belgium. 2013 IEEE International Conference on Big Data. 2013.

[25] Y. Bu et al . HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 3(1-2):285–296, 2010.

[26] Frequent itemset mining dataset repository. http://fimi.us.ac.be/data. 2004