Journal of Big Data

METHODOLOGY Open Access

Adaptive-Miner: an efficient distributed association rule mining algorithm on Spark

Sanjay Rathee[1*†] and Arti Kashyap[1,2†]

*Correspondence: sanjay_rathee@students.iitmandi.ac.in
†Sanjay Rathee and Arti Kashyap contributed equally to this manuscript.
[1] School of Computing and Electrical Engineering, IIT Mandi, Kamand Campus, Mandi, India
Full list of author information is available at the end of the article

Abstract
Extraction of valuable data from extensive datasets is a standout amongst the most vital exploration issues. Association rule mining is one of the highly used methods for this purpose. Finding possible associations between items in large transaction based datasets (finding frequent itemsets) is most crucial part of the association rule mining task. Many single-machine based association rule mining algorithms exist but the massive amount of data available these days is above the capacity of a single machine based algorithm. Therefore, to meet the demands of this ever-growing enormous data, there is a need for distributed association rule mining algorithm which can run on multiple machines. For these types of parallel/distributed applications, MapReduce is one of the best fault-tolerant frameworks. Hadoop is one of the most popular open-source software frameworks with MapReduce based approach for distributed storage and processing of large datasets using standalone clusters built from commodity hardware. But heavy disk I/O operation at each iteration of a highly iterative algorithm like Apriori makes Hadoop inefficient. A number of MapReduce based platforms are being developed for parallel computing in recent years. Among them, a platform, namely, Spark have attracted a lot of attention because of its inbuilt support to distributed computations. Therefore, we implemented a distributed association rule mining algorithm on Spark named as Adaptive-Miner which uses adaptive approach for finding frequent patterns with higher accuracy and efficiency. Adaptive-Miner uses an adaptive strategy based on the partial processing of datasets. Adaptive-Miner makes execution plans before every iteration and goes with the best suitable plan to minimize time and space complexity. Adpative-Miner is a dynamic association rule mining algorithm which change its approach based on the nature of dataset. Therefore, it is different and better than state-of-the-art static association rule mining algorithms. We conduct in-depth experiments to gain insight into the effectiveness, efficiency, and scalability of the Adaptive-Miner algorithm on Spark. Available: https://github.com/sanjaysinghrathi/Adaptive-Miner

Keywords: Association rule mining, Apache Spark, Hadoop, Distributed computing frameworks

Introduction

Data mining techniques like classification, recommendation systems, clustering, and association rule mining are highly used to extract the useful or relevant information from large datasets. Association rule mining is one of the best techniques to mine relevant information from large datasets. It produces relevant or interesting relations between different variables in the dataset in the form of association rules. For example, an association rule bread => butter (90%) states that nine out of ten customers that bought bread also bought butter. To generate such association rules, frequent patterns must be identified first. Therefore, frequent pattern mining forms the crux of any association rule mining process. To find frequent patterns, we need to get support for every itemset in the database. Itemsets with support more than minimum support are considered as frequent itemsets. In association rule example, 90% shows confidence or accuracy of the rule.

For example, I is a set of products (items). A set $X = \{i_1, i_2, ..., i_k\} \in I$ is called as *k-itemset*. A *transaction* over I is a pair $T = (tid, I)$ where *tid* is *transaction ID* and I is an itemset. A *transaction database* is a set of *transactions* having *itemsets* $\subseteq I$.

The *cover* of an itemset X in *transaction database D* contains the set of *transaction IDs* of *transactions* that support X.

$$cover(X, D) = \{tid | (tid, I) \in D, X \subseteq I\}$$

The total *support* of an *itemset* X in D is the total number of *transactions* in the cover of X in D:

$$support(X, D) := |cover(X, D)|$$

The total *frequency* of an *itemset* X in D is the total probability of occurance of X in a transaction $T \in D$:

$$frequency(X, D) := P(X) = support(X, D)/|D|$$

The *accuracy* or *confidence* of an *association rule* $X => Y$ in D is the probability (conditional) of having Y happening in a *transaction*, given that X is also exist in that *transaction*:

$$confidence(X => Y, D) = P(Y|X) = support(X \cup Y, D)/support(X, D)$$

Association rule mining has numerous applications in different areas. Historically, it is used in market basket analysis to mine products which were sold together frequently that in turn allowed industries to formulate better marketing plans to sell and manage their products and services. We illustrate with suitable examples the wide range of various applications that benefit from association rule mining.

- *Crime prevention/detection* Frequent pattern analysis of huge criminal databases that contain criminal events/activities can help to predict the most crime-prone areas in the city or predict the criminals/thieves that are most likely to be repeat offenders [1, 2].

- *Cyber security* Frequent pattern analysis of large network log files can help identify different IP addresses and ports that are highly susceptible to attacks [3]. This information can be used to block requests from these vulnerable ports or addresses.
- *Crowd mining* Extracting useful patterns from large databases of social sites allows a better comprehension of crowd (large group of people) behavior which in turn can improve possibilities of increasing sale and monetary gains [4, 5].
- *Bioinformatics* Analyzing sequence alignment results to find interesting information related to the human genome and motif discovery [6].

All the basic association rule mining algorithms work on sequential approach and they were efficient until the size of the dataset were small. As the size of datasets started increasing, their efficiency starts decreasing. Therefore, to handle large datasets, parallel algorithms were introduced [7–12]. Many cluster-based algorithms were capable of handling large datasets, but they were complex and had many issues like synchronization, replication of data etc. Hence parallel approach is replaced by MapReduce approach. MapReduce approach makes association rule mining process very fast because algorithms like Apriori have possibilities of high parallelism. Key-value pairs (MapReduce intermediate results) can be easily generated in case of Apriori algorithm. Many MapReduce based implementations of Apriori algorithm [13–16] were proposed which shows a high-performance gain as compared to the conventional Apriori algorithm. Hadoop [17] is one of the best platforms to implement Apriori algorithm as a MapReduce model. But still, there are some limitations in Hadoop based implementation of Apriori algorithm. On Hadoop platform, results are stored to HDFS after each iteration and input is taken from HDFS for next iteration, which decreases the performance due to input-output time. But Spark [18] platform tackle these problems by using its RDD (Resilient Distributed Datasets) architecture, which stores results at the end of an iteration in the local cache and provides them for next iterations. Apriori implementation on spark platform gives faster and efficient results on standard datasets which makes spark platform best for implementation of Apriori algorithm to mine frequent patterns and generate association rules later. Recently, Qiu [19] have reported nearly 18 times speedup on an average for various benchmarks for their yet another frequent itemset mining (YAFIM) algorithm. Their results with real-world data for medical application are observed to be many times faster than a MapReduce framework. We proposed a new algorithm, called R-Apriori [20], which was faster and efficient than standard Apriori on Spark for 2nd iteration. We replaced conventional Apriori approach with our reduced approach to reduce the number of computations in 2nd iteration. Our reduced approach outperforms conventional Apriori approach by 4–9 times for 2nd iteration. Use of our approach for all iterations of Apriori algorithm did not come out to be very promising as we found that our reduced approach will not be much fast and efficient for any iteration if a total number of the frequent itemset in the earlier iteration is less. Therefore, we tried to find out the threshold above which our approach will be efficient for any iteration.

The paper is organized as follows. After introducing the motivation in "Introduction" section, earlier work about frequent itemset mining, mainly MapReduce based Apriori algorithm is reported in "Related work" section. "Adaptive-Miner algorithm" section has detailed description of the Adaptive-Miner algorithm proposed in this paper. "Evaluation" section shows performance analysis of conventional Apriori, R-Apriori, and Adaptive-Miner on Spark. "Conclusion" section concludes the paper.

Mapreduce and Spark

With the availability of cloud computing technologies like Amazon EC2 cloud and Microsoft Azure, as evolutions to accommodate computing and storage service as a utility at very affordable prices, users can use these cloud services via internet from home or workplace by paying for resources consumed without worrying about availability, maintenance and flexibility issues. Cost is based on the type of service and time of service. Therefore, cloud computing has emerged as a very promising solution for demands of storage and computation in research eliminating the need for powerful and high computing capacity server. But to achieve efficiency, performance, and scalability for processing huge data, a highly parallel distributed computing model is required. Therefore, a parallel computing framework called MapReduce [21] was designed by Google which allows using thousands of commodity machines in parallel. MapReduce framework works on a basic idea of the flow of $\langle key, value\rangle$ pairs through the map and reduce phases. Input is split into fixed size chunks and distributed over available mappers. Every mapper processes its chunk of data and generates $\langle key, value\rangle$ pairs. These $\langle key, value\rangle$ pairs are shuffled or sorted to group values based on keys to generate intermediate $\langle key, value\rangle$ pairs where all values with the same key are grouped together. Reducers take the intermediate $\langle key, value\rangle$ pairs and combine all values for a key to generate final results. MapReduce framework can handle huge datasets because all map and reduce operations are executed concurrently on many machines. All map tasks need to finish to start reduce tasks. Many MapReduce framework based tools like Hadoop, Spark, and Flink are available to analyze huge datasets with ease. Hadoop is widely used MapReduce based model in bioinformatics in recent few years. Sequence alignment tools like IMR-Apriori [22], MR-Apriori [16] and BigFIM [23] used Hadoop for heavy analysis. Though Hadoop provides a highly parallel computing environment but has a limitation of high I/O time during various iterations. Apache Spark [18] overcomes this limitation of Hadoop by using its in-memory computing technique with the help of RDD storage. I/O operations on Spark RDD [18] are very efficient and fast due to which sometimes it outperforms Hadoop by 100 times. These advantages of Apache Spark ignited an interest in us to use Apache Spark for our association rule mining algorithm Adaptive-Miner.

Related work

During last three decades, a lot of association rule mining algorithms are proposed by researchers. These algorithms were fast and efficient until the evolution of Big Data in last decade. In an era of Big Data, data is produced at such high speed that these algorithms are unable to keep track with that. Therefore, association rule mining algorithms based on parallel and distributed computing has evolved as a solution. Most of these algorithms were based on MapReduce paradigm. Apriori is one of the simplest and easily parallelizable algorithms. Therefore most researchers use Apriori approach for implementing MapReduce-based association rule mining algorithms.

In 2008, Li [24] proposed an association rule mining algorithm called as PFP (Parallel Frequent Pattern). This algorithm is a parallel implementation of FP-Growth (Frequent Pattern-Growth) algorithm based on MapReduce paradigm. It eliminates the requirements of data distribution and load balancing by using MapReduce paradigm. It was highly scalable and quite suitable for web data mining. PFP search for top-k patterns instead of patterns fulfilling user specified minimum support criteria which make it effective for web data mining. Author apply it on query log for search recommendations. PFP load balancing technique was not so efficient, therefore Zhou [25] proposed a new algorithm called as BPFP (Balance Parallel FP-Growth). BPFP algorithm has better load balancing technique to make PFP faster and efficient.

In 2010, Yang [26] proposed a very simple MapReduce-based association rule mining algorithm. This algorithm was the very straightforward implementation of Apriori algorithm on Hadoop. It uses a single map and reduce phases to get frequent patterns. In 2011, Li [27] proposed a new cloud computing based association rule mining algorithm which uses one phase MapReduce implementation of Apriori. They used better data distribution techniques to improve efficiency and speed. An entirely different approach is used by Yu [28] for mining association rules. They replaced the original transactional database with the boolean matrix. Now many AND operations along with other logical operators are used on matrix to find frequent patterns. It uses Hadoop for parallel computation of matrix by dividing it into parts. They claim for better space and time efficiency.

Some researchers focused on using cloud computing platforms like EC2 and S3 for fast and efficient association rule mining.

In 2012, Li [13] proposed a MapReduce-based association rule mining algorithm and ran it on Amazon EC2 cluster. This algorithm uses basic apriori approach. It provides faster results due to high computation power available on EC2. It uses Amazon S3 for data storage. Later, lin [14] proposed three MapReduce-based association rule mining algorithms called as SPC (Single Pass Counting), FPC (Fixed Passes Combined-counting), and DPC (Dynamic Pass Counting). SPC algorithm is simple MapReduce implementation of Apriori. FPC algorithm works same as SPC for finding up to 2-itemsets. It combines candidate sets for remaining passes to get results in a single phase. It was useful in some cases where the size of candidate set is small after two iterations and many machines in Hadoop cluster remain idle during afterward steps. By default, FPC combines candidate set of 3 iterations like candidate set of 3-itemsets, 4-itemset and 5-itemset. FPC also has a drawback that it combines fix number of passes and have the possibility of crashing if candidate set for higher iterations is large. DPC algorithm resolve this issue quite efficiently by combining phases depending on candidate size and machines computation power. It provides better load balancing for increasing efficiency. In the same year, Li [15] and Yahya (MRApriori) [16] proposed another MapReduce paradigm based association rule mining algorithms which are a straightforward parallel implementation of Apriori algorithm.

Some researchers use random samples of the database to find approximate association rules. PARMA (Parallel Randomized Algorithm for Approximate association rule mining) [29] algorithm gives input of random samples of the database to various machines in the cluster. Every machine finds frequent patterns for its sample and reducer combines the results. This algorithm does not provide accurate results, but allow the user to define his choices for allowed error percentage. A random sample is fixed in size and depend on user-defined allowed error rate. It has a drawback of less accuracy as compared to existing MapReduce-based algorithms. In 2013, Kovacs [30] used a different approach that they calculate singleton and pair frequent itemsets in the first iteration of MapReduce using triangular matrix. Candidate set is generated in reduce phase instead of map phase in all existing MapReduce-based algorithms. Later, Oruganti [31] used parallel Apriori implementation using MapReduce paradigm to explore Hadoop capabilities. In the same year, Yong [32] proposed an association rule mining algorithm based on MapReduce paradigm which uses cloud computing to run parallel implementation of FP-Growth. It has two major benefits over conventional parallel FP-Growth algorithm. Firstly, it reduces database scans, and secondly, it reduces the cost of inter-processor communication in conventional parallel FP-Growth algorithm. Later, Moens [23] proposed two association rule mining algorithms called as Dist-Eclat (Distributed-Eclat) and BigFIM. Dist-Eclat works on the concept of Eclat algorithm. It is MapReduce paradigm based implementation of Eclat algorithm. It concentrated more on speed and suitable for faster results. BigFIM works on a hybrid approach. It uses both Apriori and Eclat algorithm mixed approach. It is more appropriate for handling large databases in an optimized way. In the same year, Farzanayar [22] came with IMRApriori (Improved MapReduce based Apriori) algorithm to analyze massive social network data.

In 2014, Lin [33] came with one more MapReduce paradigm based parallel implementation of Apriori. Lin uses most outdated cluster hardware (P4) for computations. Later, Barkhordari [34] proposed a MapReduce-based association rule mining algorithm called as ScaDiBino (scalable and distributable binominal association rule mining algorithm) which converts every row of input transactions to a binomial format. Binomial data can be processed more efficiently on MapReduce. This algorithm directly generates association rule without finding frequent patterns. They used this algorithm for recommending value added services to customers by analyzing network traffic of a mobile operator.

Some researchers use various data structures to improve the efficiency of association rule mining algorithms. Singh [35] tries to use a hash table, hash trie and hash table trie for candidate storage in Apriori MapReduce-based implementation. They find that hash table trie is most efficient than others in MapReduce context while it is not much efficient in a sequential approach.

On Hadoop platform, results are stored in HDFS (hadoop distributed file system) after every iteration, and these results are again sourced from HDFS as input for the next iteration, which decreases the performance due to the high I/O time. But Spark [18], a new in-memory, distributed data-flow platform, resolves this issue by using its RDD architecture. RDDs stores the results in main memory at the end of an iteration and make them available for the next iteration. Traditional Apriori implementation on Spark platform gives many times faster results on standard datasets which make Spark one of the best tool for implementation of Apriori. In 2014, Qiu [19] had reported speedups of more than 18 times on average for various benchmarks for YAFIM (yet another frequent itemset mining) algorithm based on Spark RDD framework. Their results on real-world medical data are observed to be many times faster than on the MapReduce framework. Later, Zhang [36] proposed an association rule mining algorithm called as DFIMA (Distributed Frequent Itemset Mining Algorithm) which is implemented on Spark. DFIMA algorithm uses matrix-based pruning technique to reduce candidate size. The author claims that it outperforms PFP algorithm when both are implemented on Spark. Recently, our new association rule mining algorithm called as R-Apriori [20] used a reduced approach for 2nd iteration to reduce computations. R-Apriori outperformed nearly all state-of-the-art association rule mining algorithms for 2nd iteration in terms of accuracy and performance. These results motivated us to come with innovative approaches for distributed association rule mining algorithms.

Adaptive-Miner algorithm

Adaptive-Miner is a MapReduce based parallel algorithm implemented on Apache Spark. It has two phases.

Phase I In the first Phase, all frequent singletons are mined from the dataset. This phase uses a single iteration of *Map* and *Reduce* to discover all frequent singletons. All frequent singletons are stored in *Bloom filter*. Apriori uses *Hash tree* to store and search candidate itemsets which have the limitation of false negatives. Therefore, *Hash tree* is replaced with *Bloom filter* to improve accuracy for association rule mining.

Phase II In the second Phase, algorithm finds all frequent itemsets of length 2, 3, 4 and so on. This phase uses many iterations of *Map* and *Reduce* until there is no frequent itemset for an iteration or maximum iteration limit is reached. In phase II, it makes execution plans for every iteration, compute the cost of execution for every plan and then select the best plan for that iteration. This algorithm uses a dynamic approach which makes it faster and gives efficient results for every iteration. There is some overhead of calculating the cost for every execution plan, but this overhead is negligible when we are working on large datasets.

Phase I—frequent singletone generation

Adaptive-Miner finds all frequent singletons from large transactional datasets during the first phase. The transaction dataset from HDFS is loaded into Spark RDD to make good use of cluster memory and also provide resilience to failures in the cluster. The detailed process of generating a frequent singleton itemset from the dataset is outlined in Algorithm 1.

Algorithm 1 Phase I- Singleton frequent items

```
Input: Transactional Dataset $D$
Output: Singleton frequent itemset $L_1$
 1: procedure SINGLETON–GEN
 2:     for each Transaction $T \in D$ do
 3:         flatMap(line offset, $T$)
 4:         for each item $I \in T$ do
 5:             Yield($I$, 1)
 6:         end flatMap
 7:         storeAtRDD1
 8:     RDD2=RDD1.reduceByKey
 9:     for each tuple $t \in$ RDD2 do
10:         flatMap($I$, $count$)
11:         if ($count \langle min - support$) then
12:             Yield($I$, $count$);
13:         end flatMap
14:         storeAtRDD3
```

The input file is broadcasted to every worker node to make it available locally to each worker. Subsequently, a *flatMap* function is applied on every transaction in dataset (line 2). For each transaction, the mapper generates $\langle key, value\rangle$ pairs where *key* denotes every item in transaction and *value* is an integer 1 (lines 4–6). Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out the results having *value* less than *minimum support* (lines 8–9). At last, results are stored in a *SparkRDD*. Lineage graph, showing the flow of data and control through various stages of phase I is presented in Fig. 1.

Example

Figure 2 shows an example of singleton frequent itemset generation step. A transactional database stored on HDFS is treated as the input. The mappers in the first round receive a set of transactions (e.g. mapper MAP-1 receives transaction T1–T2) and they split each transaction into items and generate corresponding $\langle key, value\rangle$ pairs. For example, transaction T1 (*S*, *R*, *M*) is mapped to $\langle S, 1\rangle$, $\langle R, 1\rangle$, and $\langle M, 1\rangle$. Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out pairs having *value* less than *min-support*. At last, results are stored in *Bloom filter*.

Phase II—frequent itemsets generation

Adaptive-Miner uses an adaptive approach and select conventional or reduced approach for every iteration based on the nature of the dataset. It iterates until all frequent itemsets of various length are discovered. The detailed process of finding frequent itemsets of *I* length for the transactional dataset is outlined in Algorithm 2.

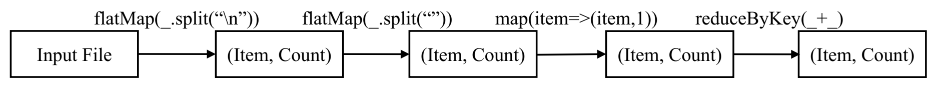
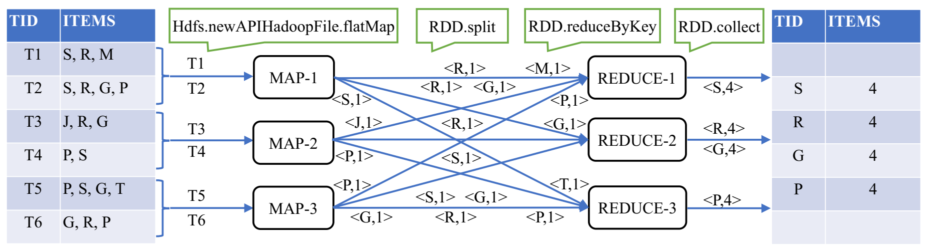


Fig. 1 Lineage graph for Phase I (frequent singleton generation)



TID	ITEMS
T1	S, R, M
T2	S, R, G, P
T3	J, R, G
T4	P, S
T5	P, S, G, T
T6	G, R, P

TID	ITEMS
S	4
R	4
G	4
P	4

Fig. 2 Mapreduce architecture for Phase I (frequent singleton generation)

Algorithm 2 Phase II- Frequent k-itemset generation

```
Input: Transactional Dataset $D$, Frequent itemsets in $k-1^{th}$ itearation $L_{k-1}$
Output: Frequent ($k$)-itemsets $L_k$
 1: procedure FREQUENT -GEN
 2:     if ($L_{k-1}$.size is large) then
 3:         $L_{k-1}$.storeInBloomFilter
 4:         for each Transaction $T \in D$ do
 5:             flatMap(line offset, $T$)
 6:             $P_T$ = Intersect($T$, $L_{k-1}$)
 7:             $C_k$ = pairs($P_T$)
 8:             for each pair $p \in C_k$ do
 9:                 Yield($p$, 1)
10:             end flatMap
11:             storeAtRDD1
12:     else if ($L_{k-1}$.size is less) then
13:         $C_k$ = candidate-gen($L_{k-1}$)
14:         for each Transaction $T \in D$ do
15:             flatMap(line offset, $T$)
16:             $C_T$ = subset($C_k$, $T$)
17:             for each candidate $c \in C_T$ do
18:                 Yield($c$, 1)
19:             end flatMap
20:             storeAtRDD1
21:     RDD2=RDD1.reduceByKey
22:     for each tuple $t \in$ RDD2 do
23:         flatMap($c$, $count$)
24:         if ($count \langle min - support$) then
25:             Yield($c$, $count$);
26:         end flatMap
27:         storeAtRDD3
```

For *I*th iteration, a basic condition (a large number of frequent itemsets in recent iteration) is checked which depends on the number of items frequent in (I-1) iteration. If the condition is satisfied, then Reduced approach is used. Singleton frequent items are stored in a *Bloom filter*. The input file is broadcasted to every worker node to make it available locally to each worker. Subsequently, a *flatMap* function is applied on every transaction in dataset (line 2). Mapper takes each transaction and prunes it so that it contains only items which exist in the *Bloom filter*. Mapper yields all possible pairs for the pruned transaction as ⟨*key*, *value*⟩ pairs where *key* denotes every pair in the pruned transaction and *value* is an integer 1 (lines 4–6). Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out the results having *value* less than *minimum support* (lines 8–9). At last, results are stored in a *SparkRDD*.

If the condition is not satisfied, then conventional Apriori approach is used. Initially, a candidate set (C_i) is generated for Ith iteration from last iteration frequent itemset (L_{k-1}). The input file is broadcasted to every worker node to make it available locally to each worker. Subsequently, a *flatMap* function is applied on every transaction to find possible (*k*)-itemsets for that transactions in hash tree (line 2). For each transaction, the mapper generates ⟨*key*, *value*⟩ pairs where *key* denotes every (*k*)-itemset for a transaction and *value* is an integer 1 (lines 4–6). Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out the results having *value* less than *minimum support* (lines 8–9). At last, results are stored in a *SparkRDD*. Lineage graph, showing the flow of data and control through various stages of phase I is presented in Fig. 3.

Example

Figure 4 shows an example of frequent pair itemsets generation step. For 2nd iteration, the approximate cost of reduced and conventional approach is calculated using the size of the frequent set in the first phase and dataset size. The reduced approach is used if the condition is satisfied. A transactional database stored on HDFS and singleton frequent itemsets stored in *Bloom filter* are treated as the input. The mappers in the first round receive a set of transactions (e.g. mapper MAP-1 receives transaction T1–T2) and they prune each transaction so that it contains only items which exist in the *Bloom filter*. For example, transaction T1 (S, R, M) is mapped to pruned transaction T1 (S, R).Now, mappers yield all possible $\langle key, value\rangle$ pairs for the pruned transaction. For example, pruned transaction T1 (S, R) is mapped to $\langle SR, 1\rangle$. Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out pairs having *value* less than two. At last, results are stored on *SparkRDD*.

If the cost of conventional approach is less than reduced, then conventional approach is used for pair generation. A transactional database stored on HDFS and

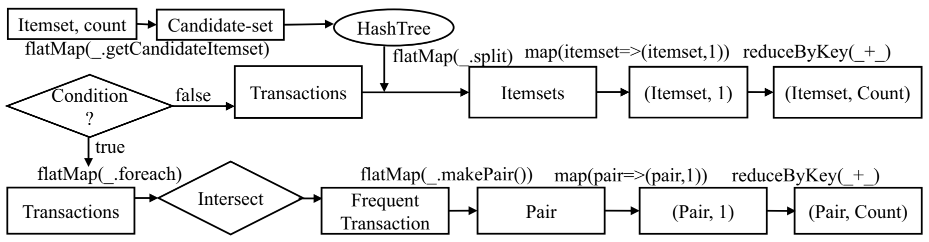


Fig. 3 Lineage graph for Phase II (frequent K-itemset generation)

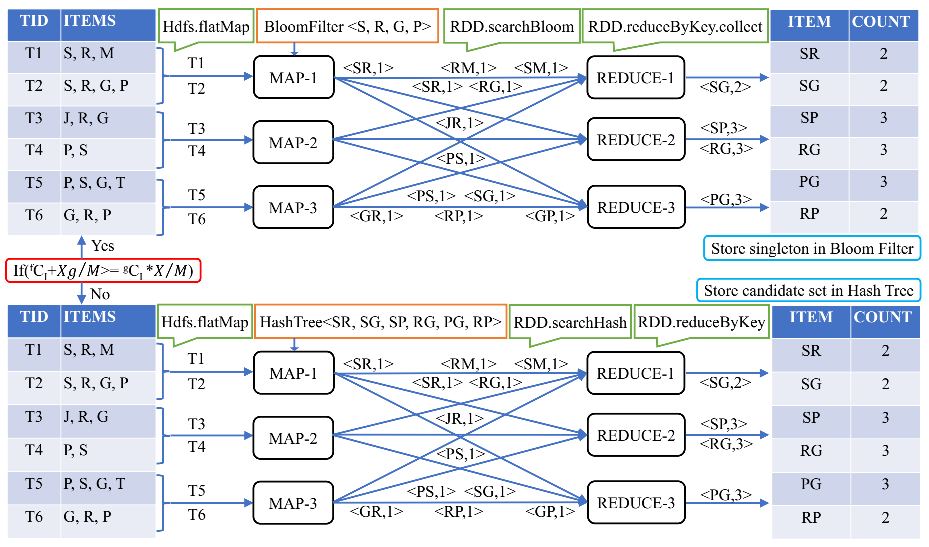


Fig. 4 Mapreduce architecture for Phase II (frequent pair generation)

a hash tree having candidate set ($\langle SR\rangle$, $\langle SG\rangle$, $\langle SP\rangle$, $\langle RG\rangle$, $\langle PG\rangle$, $\langle RP\rangle$) are treated as the input. The mappers in the first round receive a set of transactions (e.g. mapper MAP-1 receives transaction T1–T2) and they find each *k*-itemset for each transaction and generate corresponding $\langle key, value\rangle$ pairs. For example, transaction T1 (S, R, M) is mapped to $\langle SR, 1\rangle$. Subsequently, a reduce task combines all pairs according to the key (*reduceByKey* function) and filter out pairs having *value* less than two. At last, results are stored in *SparkRDD*.

Adaptiveness of the algorithm comes from the test for the condition as follows. Let us assume that there are X transactions, M mappers, g average number of elements in every transaction and f number of items frequent after the last iteration. Further, t is time to search an element in HashTree, and b is time taken to search an element in bloom filter.

For conventional approach, total time complexity of Ith Iteration is sum of time to generate candidate set (T_g), time to store candidate set in HashTree (T_h) and time taken by mapper (T_m). Here, time complexity to generate candidate set is the sum of total time for join and prune task.

- Time complexity to generate candidate set $= (I+1)C_I^f$
- Time complexity to store candidate set in HashTree $= I * C_I^f$
- Time complexity to search and yield pairs $= \frac{X}{M} * tg$
- Total time complexity for conventional approach $= (2I+1)(C_I^f) + \frac{X}{M} * tg$.

For Adaptive approach, time complexity of Ith iteration is sum of time complexity to store singleton frequent items in Bloom filter (T_{bf}), time complexity to make pruned transaction (T_{pr}) and time complexity to generate pairs (T_{mr}).

- Time complexity to store singleton frequent items in Bloom filter $= f$
- Time complexity to make pruned transaction $= \frac{X}{M} * g$
- Time complexity to generate pairs in worst case $= \frac{X}{M} * C_I^g$
- Total time complexity for reduced approach $= f + \frac{X}{M} * (C_I^g + g)$.

Adaptive-Miner will select reduced approach if time complexity of conventional approach will be greater than Reduced approach (time complexity of conventional approach − time complexity of reduced approach > 0)

$$(2I+1)(C_I^f) + \frac{X}{M} * tgf + \frac{X}{M} * (C_I^g + g) > 0$$

I, t and b are constant values mainly less than 10 and number of transactions are in millions for large datasets. Therefore, they are nearly negligible in term of complexity. After removing these final term will be $(C_I^f + \frac{Xg}{M}) >= (\frac{X}{M} * C_I^g)$

Number of frequent itemsets in last iteration (f) or candidate set size (candidate set size $= \frac{f(f-1)}{2}$) and average number of items (g) in a transaction are main attributes in above condition. Reduced approach will be more useful if either f is high or g is small.

For an iteration I, to use reduced approach the condition $(C_I^f + \frac{Xg}{M}) >= (\frac{X}{M} * C_I^g)$ must be satisfied. If the given condition is satisfied then complexity of conventional approach will be greater than reduced approach. Therefore, Adaptive-Miner will make decision to choose the reduced approach instead of conventional approach.

Evaluation

In this section, Adaptive-Miner's performance is evaluated in comparison to YAFIM and R-Apriori on spark. 1st iteration of all algorithms is same so computation time depends on the platform used. Remaining iterations are the main concern in perspective of performance. All experiments were executed four times, and average results were taken as a final result.

Cluster and dataset

The performance of Adaptive-Miner is evaluated on a cluster having 5 nodes where each node has 24 cores and 64 GB RAM. All computing nodes are running on Ubuntu 14.04 LTS operating system. *Oracle Java 8* is used to build the project. *Spark 1.5.2* and *Hadoop 2.6* are used to run Adaptive-Miner.

Experiments were done with six large datasets having different characteristics. Properties of these datasets are as shown in Table 1.

- T10I4D100K (artificial datasets generated by IBM's data generator) [37] have 10^5 transactions with 870 items in it.
- Retail dataset [37] was used for the market-basket model. It contains various transactions done by the customer in a shopping mall.
- Musroom dataset [37] is publically available musroom data.
- Kosarak dataset [37] was donated by Ferenc Bodon and contains the click-stream data of a hungarian on-line news portal.
- BMSWebView2 [38] is a dataset used for KDD cup 2000 competition. It has average length 4.62 and 6.07 standard deviation.
- T25I10D10K [38] is a synthetic dataset generated by random transaction database generator.

Performance evaluation

Adaptive-Miner is evaluated in terms of scalability and efficiency.

The scalability of the Adaptive-Miner is evaluated by increasing the number of compute cores and replicating the original datasets. Figure 5a shows that the execution

Table 1 Query datasets description

S. no.	Dataset	Number of items	Number of transactions
1	T10I4D100K	870	100,000
2	Retail	16,470	88,163
3	Musroom	119	8124
4	T25I10D10K	990	4900
5	Kosarak	41,270	9,90,002
6	BMSWebView2	3340	77,512

time decreases nearly linearly with increasing compute cores. Figure 5b shows that execution time increases nearly linearly with increasing dataset size. Therefore, we can say that Adaptive-Miner is a scalable algorithm.

Performances for all algorithms with different datasets were evaluated using two worker nodes. For all six datasets, the comparison is made between conventional Apriori, R-Apriori, and Adaptive-Miner on Spark.

- For T10I4D100K dataset, Adaptive-Miner performs better than Apriori and same as R-Apriori for 2nd iteration, but it performs same as both Apriori and R-Apriori for 3rd and 4th iterations (Fig. 6a).
- For T25I10D10K dataset, Adaptive-Miner performs better than Apriori and same as R-Apriori for 2nd iteration, but it outperforms R-Apriori also for 3rd iteration. For 4th iteration again all use same candidate set approach due to less number of frequent items in 3rd iteration (Fig. 6b).
- For Retail, Adaptive-Miner is better than Apriori but same as R-Apriori for all iterations because number of items after 2nd iteration is less and it uses candidate set approach when the number of frequent items in the last iteration is small (Fig. 7a).
- For Musroom dataset, Adaptive-Miner performs better than R-Apriori and same as conventional Apriori for 2nd iteration because number of singleton frequent itemsets are very less so Adaptive-Miner uses candidate set approach for 2nd iteration.

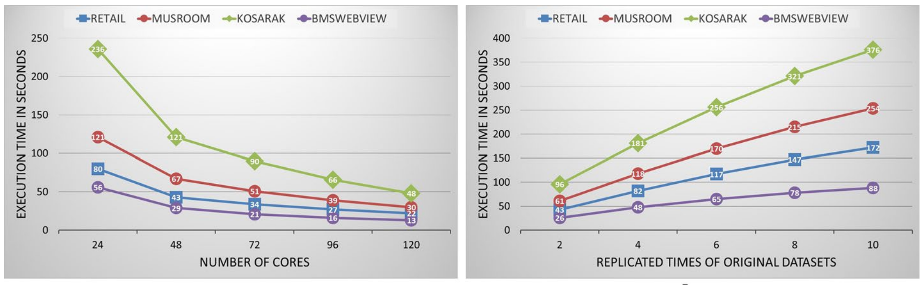


Fig. 5 **a** Adaptive-Miner execution time for different datasets with increasing computing nodes. **b** Adaptive-Miner execution time for datasets with increasing sizes by replication

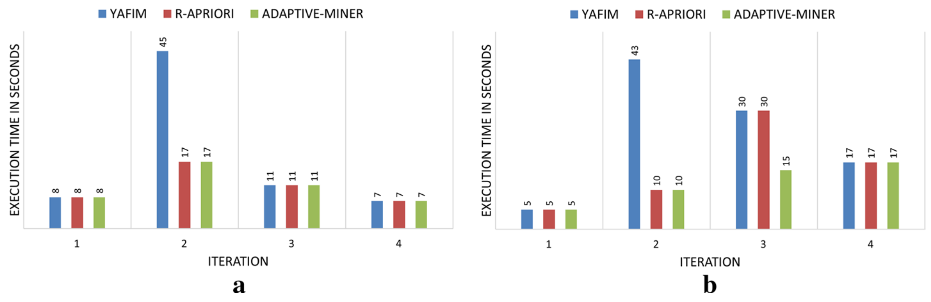


Fig. 6 Comparision of conventional Apriori, R-Apriori and Adaptive-Miner performance. **a** T10I4D100K dataset min sup = 0.15%. **b** T25I10D10K dataset min sup = 0.10%

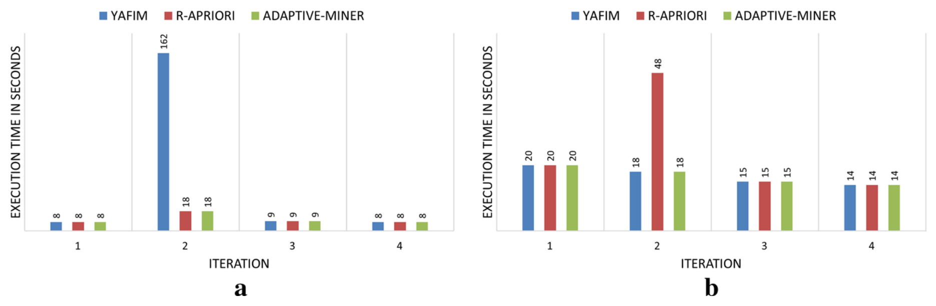


Fig. 7 Comparision of conventional Apriori, R-Apriori and Adaptive-Miner performance. **a** Retail dataset min sup = 0.15%. **b** Musroom dataset min sup = 30%

Hence Adaptive-Miner is always performing better or same as Apriori and R-Apriori (Fig. 7b).

- For Kosarak dataset, Adaptive-Miner performs better than Apriori and same as R-Apriori for 2nd iteration, but it outperforms R-Apriori also for 3rd and 4th iteration (Fig. 8a).
- For BMSWebView2 dataset, Adaptive-Miner performs better than Apriori and same as R-Apriori for 2nd iteration, but it performs same as both Apriori and R-Apriori for 3rd and 4th iterations (Fig. 8b).

Conclusions

A Spark based distributed algorithm Adaptive-Miner is implemented to mine frequent patterns from large datasets. It uses a new modified approach (Adaptive) as well as basic Apriori theorem that an itemset is frequent only if all its non-empty subsets are frequent. The reduced approach will be used when number of frequent itemsets in last iterations are large, otherwise basic Apriori approach will be used. Adaptive-Miner makes execution plans before every iteration and computes the cost for every execution plan. Execution plan with minimum cost is used to get results for that iteration. This dynamic approach of taking decision for every iteration during runtime makes Adaptive-Miner very fast and efficient. It is implemented on Apache Spark platform which provides it highly parallel and distributed computing environment. Spark is best suited for Adaptive-Miner because it has support for in-memory distributed computation. Results on

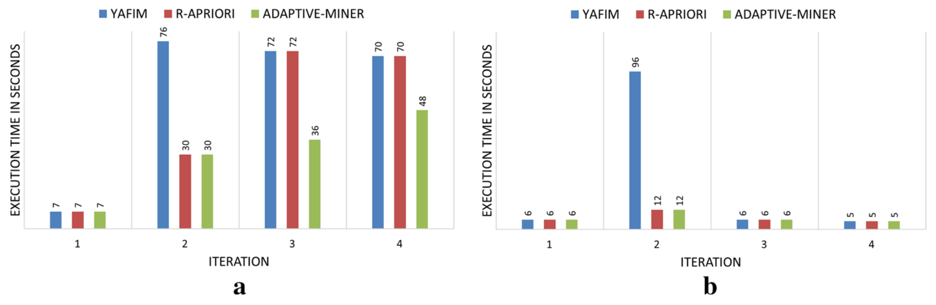


Fig. 8 Comparision of conventional Apriori, R-Apriori and Adaptive-Miner performance. **a** Kosarak dataset min sup = 0.60%. **b** BMSWebView2 dataset min sup = 0.10%

various standard datasets show that Adaptive-Miner outperforms existing MapReduce based distributed algorithms. Adaptive-Miner performs better or same as conventional Apriori and R-Apriori for every iteration for every large dataset. Adaptive-Miner is available on GitHub for download and use.

Future work

Apache Flink has shown great performance for iterative computations in recent few years. It provides native support for iterative computations. Adaptive-Miner is an iterative algorithm. Therefore, we will implement Adaptive-Miner on Flink to check performance in comparison to Spark implementation in future.

Authors' contributions

SR performed the literature review, implemented the proposed algorithm and conducted the experiments. AK advised SR all aspects of the paper development. Both authors read and approved the final manuscript.

Author details

[1] School of Computing and Electrical Engineering, IIT Mandi, Kamand Campus, Mandi, India. [2] School of Basic Sciences, IIT Mandi, Kamand Campus, Mandi 175005, India.

Authors' information

Sanjay Rathee received the B.Tech degree in computer engineering from Maharshi Dayanand University, Rohtak, Haryana, India, in 2011, and the M.Tech degree in computer engineering from Kurukshetra University, Haryana, India, in 2013. He is currently working toward the Ph.D. degree in computer engineering from Indian Institute of Technology, Mandi, India. He has developed several distributed algorithms related to business strategies and bioinformatics sector. His research interests include distributed computing algorithms and platforms, association rule mining and sequence alignment. Arti Kashyap received the B.Sc degree from Himachal Pradesh University, Shimla, H.P., India, in 1989, the M.Sc and Ph.D. degree from Indian Institute of Technology, Roorkee, India, in 1991 and 1996 respectively. She is currently working as associate professor at Indian Institute of Technology Mandi, India. Her research interests include distributed algorithms, big data analytics, sequence alignment and magnetic materials.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Not applicable.

Consent for publication

I allow journal to publish it.

Ethics approval and consent to participate

Not applicable.

Funding

Not applicable.

Publisher's Note

Received: 22 November 2017 Accepted: 4 January 2018
Published online: 20 February 2018

References

1. Wang T, Rudin C, Wagner D, Sevieri R. Learning to detect patterns of crime. In: European conference on machine learning and principles and practice of knowledge discovery in databases. 2013.
2. Buczak AL, Gifford CM. Fuzzy association rule mining for community crime pattern discovery. In: ACM SIGKDD workshop on intelligence and security informatics. ISI-KDD '10. New York: ACM; 2010. p. 2–1210. https://doi.org/10.1145/1938606.1938608.
3. Khan L, Awad M, Thuraisingham B. A new intrusion detection system using support vector machines and hierarchical clustering. VLDB J. 2007;16(4):507–21. https://doi.org/10.1007/s00778-006-0002-5.
4. Amsterdamer Y, Grossman Y, Milo T, Senellart P. Crowdminer: mining association rules from the crowd. Proc VLDB Endow. 2013;6(12):1250–3. https://doi.org/10.14778/2536274.2536288.

5. Amsterdamer Y, Grossman Y, Milo T, Senellart P. Crowd mining. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data. SIGMOD '13. New York: ACM; 2013. p. 241–52. https://doi.org/10.1145/2463676.2465318.
6. Naulaerts S, Meysman P, Bittremieux W, Vu TN, Vanden Berghe W, Goethals B, Laukens K. A primer to frequent itemset mining for bioinformatics. Brief Bioinform. 2015;16(2):216. https://doi.org/10.1093/bib/bbt074.
7. Zaki MJ, Parthasarathy S, Ogihara M, Li W. Parallel algorithms for discovery of association rules. Data Min Knowl Discov. 1997;1(4):343–73. https://doi.org/10.1023/A:1009773317876.
8. Cheung DW, Xiao Y. In: Wu X, Kotagiri R, Korb KB, editors. Effect of data skewness in parallel mining of association rules. Berlin: Springer; 1998. p. 48–60. https://doi.org/10.1007/3-540-64383-4_5.
9. Cheung DW, Han J, Ng VT, Fu AW, Fu Y. A fast distributed algorithm for mining association rules. In: Proceeding of fourth international conference on parallel and distributed information systems. 1996. p. 31–42. https://doi.org/10.1109/PDIS.1996.568665.
10. Cheung DW, Hu K, Xia S. Asynchronous parallel algorithm for mining association rules on a shared-memory multiprocessors. In: Proceedings of the tenth annual ACM symposium on parallel algorithms and architectures. SPAA '98. New York: ACM; 1998. p. 279–88. https://doi.org/10.1145/277651.277694.
11. Zaiane OR, El-Hajj M, Lu P. Fast parallel association rule mining without candidacy generation. In: Proceedings 2001 IEEE international conference on data mining. 2001. p. 665–8. https://doi.org/10.1109/ICDM.2001.989600.
12. Pramudiono I, Kitsuregawa M. Parallel fp-growth on pc cluster. Adv Knowl Discov Data Min. 2003:570.
13. Li J, Roy P, Khan SU, Wang L, Bai Y. Data mining using clouds: an experimental implementation of Apriori over Mapreduce. In: 12th international conference on scalable computing and communications (ScalCom'13). 2012. p. 1–8.
14. Lin M-Y, Lee P-Y, Hsueh S-C. Apriori-based frequent itemset mining algorithms on mapreduce. In: Proceedings of the 6th international conference on ubiquitous information management and communication. ICUIMC '12. New York: ACM; 2012. p. 76–1768. https://doi.org/10.1145/2184751.2184842.
15. Li N, Zeng L, He Q, Shi Z. Parallel implementation of apriori algorithm based on mapreduce. In: Proceedings of 13th ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing. 2012. p. 236–41. https://doi.org/10.1109/SNPD.2012.31.
16. Yahya O, Hegazy O, Ezat E. An efficient implementation of A-Priori algorithm based on Hadoop-Mapreduce model. Int J Rev Comput. 2012;12.
17. Apache Hadoop. Open-source software for reliable, scalable, distributed computing. Apache Hadoop. 2016. http://hadoop.apache.org/docs/r2.7.2/. Accessed 18 Mar 2016.
18. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2Nd USENIX conference on hot topics in cloud computing. HotCloud'10. Berkeley: USENIX Association; 2010. p. 10. http://dl.acm.org/citation.cfm?id=1863103.1863113
19. Qiu H, Gu R, Yuan C, Huang Y. Yafim: a parallel frequent itemset mining algorithm with spark. In: IEEE international parallel distributed processing symposium workshops. 2014. p. 1664–71. https://doi.org/10.1109/IPDPSW.2014.185.
20. Rathee S, Kaul M, Kashyap A. R-Apriori: an efficient apriori based algorithm on spark. In: Proceedings of the 8th workshop on Ph.D. workshop in information and knowledge management. PIKM 15. Melbourne: ACM; 2015. p. 27–34. https://doi.org/10.1145/2809890.2809893.
21. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on opearting systems design & implementation, vol. 6. OSDI'04. Berkeley: USENIX Association; 2004. p. 10. http://dl.acm.org/citation.cfm?id=1251254.1251264.
22. Farzanyar Z, Cercone N. Efficient mining of frequent itemsets in social network data based on mapreduce framework. In: Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining. ASONAM '13. New York: ACM; 2013. p. 1183–8. https://doi.org/10.1145/2492517.2500301.
23. Moens S, Aksehirli E, Goethals B. Frequent itemset mining for big data. In: Proceedings of IEEE international conference on big data. 2013. p. 111–8. https://doi.org/10.1109/BigData.2013.6691742.
24. Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM conference on recommender systems. RecSys '08. New York: ACM; 2008. p. 107–14. https://doi.org/10.1145/1454008.1454027.
25. Zhou L, Zhong Z, Chang J, Li J, Huang JZ, Feng S. Balanced parallel fp-growth with mapreduce. In: 2010 IEEE youth conference on information, computing and telecommunications. 2010. p. 243–6. https://doi.org/10.1109/YCICT.2010.5713090.
26. Yang XY, Liu Z, Fu Y. Mapreduce as a programming model for association rules algorithm on hadoop. In: Proceedings of the 3rd international conference on information sciences and interaction sciences. 2010. p. 99–102. https://doi.org/10.1109/ICICIS.2010.5534718.
27. Li L, Zhang M. The strategy of mining association rule based on cloud computing. In: Proceeding of international conference on business computing and global informatization. 2011. p. 475–8. https://doi.org/10.1109/BCGIn.2011.125.
28. Yu H, Wen J, Wang H, Jun L. An improved apriori algorithm based on the boolean matrix and hadoop. Procedia Eng. 2011;15:1827–31.
29. Riondato M, DeBrabant JA, Fonseca R, Upfal E. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In: Proceedings of the 21st ACM international conference on information and knowledge management. CIKM '12. New York: ACM; 2012. p. 85–94. https://doi.org/10.1145/2396761.2396776.
30. Kovács F, Illés J. Frequent itemset mining on hadoop. In: 2013 IEEE 9th international conference on computational cybernetics (ICCC). 2013. p. 241–5. https://doi.org/10.1109/ICCCyb.2013.6617596.
31. Oruganti S, Ding Q, Tabrizi N. Exploring hadoop as a platform for distributed association rule mining. In: FUTURE COMPUTING 2013-the fifth international conference on future computational technologies and applications. 2013. p. 62–7.
32. Yong W, Zhe Z, Fang W. A parallel algorithm of association rules based on cloud computing. In: Proceedings of 8th international conference on communications and networking in China (CHINACOM). 2013. p. 415–9. https://doi.org/10.1109/ChinaCom.2013.6694632.

33. Lin X. Mr-apriori: association rules algorithm based on mapreduce. In: Proceedings of IEEE 5th international conference on software engineering and service science. 2014. p. 141–4. https://doi.org/10.1109/ICSESS.2014.6933531.
34. Barkhordari M, Niamanesh M. Scadibino: an effective mapreduce-based association rule mining method. In: Proceedings of the sixteenth international conference on electronic commerce. ICEC '14. New York: ACM; 2014. p. 1–118. https://doi.org/10.1145/2617848.2617853.
35. Singh S, Garg R, Mishra P. Performance analysis of apriori algorithm with different data structures on hadoop cluster. 2015. arXiv preprint arXiv:1511.07017.
36. Zhang F, Liu M, Gui F, Shen W, Shami A, Ma Y. A distributed frequent itemset mining algorithm using spark for big data analytics. Clust Comput. 2015;18(4):1493–501.
37. FIMI. FIMI datasets. FIMI. 2017. http://fimi.ua.ac.be/data/. Accessed 2 Jan 2017.
38. SPMF. SPMF: a java open-source data mining library. SPMF. 2017. http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php. Accessed 2 Jan 2017.