ABSTRACT
Association rule mining remains a very popular and effective method to extract meaningful information from large datasets. It tries to find possible associations between items in large transaction based datasets. In order to create these associations, frequent patterns have to be generated. The "Apriori" algorithm along with its set of improved variants, which were one of the earliest proposed frequent pattern generation algorithms still remain a preferred choice due to their ease of implementation and natural tendency to be parallelized.

While many efficient single-machine methods for Apriori exist, the massive amount of data available these days is far beyond the capacity of a single machine. Hence, there is a need to scale across multiple machines to meet the demands of this ever-growing data. MapReduce is a popular fault-tolerant framework for distributed applications. Nevertheless, heavy disk I/O at each MapReduce operation hinders the implementation of efficient iterative data mining algorithms, such as Apriori, on MapReduce platforms.

A newly proposed in-memory distributed dataflow platform called Spark overcomes the disk I/O bottlenecks in MapReduce. Therefore, Spark presents an ideal platform for distributed Apriori. However, in the implementation of Apriori, the most computationally expensive task is the generation of candidate sets having all possible pairs for singleton frequent items and comparing each pair with every transaction record. Here, we propose a new approach which dramatically reduces this computational complexity by eliminating the candidate generation step and avoiding costly comparisons.

We conduct in-depth experiments to gain insight into the effectiveness, efficiency and scalability of our approach. Our studies show that our approach outperforms the classical Apriori and state-of-the-art on Spark by many times for different datasets.

Keywords: Apriori, Mapreduce, Spark, Hadoop, R-Apriori, Frequent itemset mining.

1. INTRODUCTION
Data mining techniques cover a wide range of techniques like clustering, classification, and association rule mining to extract meaningful information from large datasets. In this paper, we focus on association rule mining, which produces interesting relations between variables in the dataset in the form of association rules. In order to create such association rules, frequent patterns must first be generated. Therefore, frequent pattern mining forms the crux of any association rule mining process.

Association rule mining finds numerous applications in various areas. Historically, it was used in market basket analysis to find products that were sold together frequently which in turn allowed companies to formulate better business plans to arrange and sell their products. We illustrate with examples the wide range of applications that benefit from association rule mining.

- Crime Detection/Prevention: Frequent pattern analysis of large criminal databases that contain criminal activities/events can help predict the most crime-prone areas in a city or predict the criminals that are most likely to be repeat offenders [20,1].
- Cyber Security: Frequent pattern analysis in large network log files can help identify various ports and IP addresses that are most susceptible to attacks [11]. This information can then be used to block requests from these vulnerable addresses or ports.
- Crowd Mining: Extracting useful patterns from large databases of social information allows a better comprehension of crowd behavior which in turn can improve possibilities of increasing monetary gains [22,23].

Apriori, Éclat, and FP-Growth are the top few amongst the large variety of association rule mining algorithms proposed in research literature. Most of these algorithms sequentially scan the dataset to generate frequent patterns, which can later be used for rule generation.

The Apriori algorithm proposed by R. Agarwal et al. [18], works on the fundamental observation that an itemset is frequent only if all its non-empty subsets are also frequent. It uses an iterative approach where results of the previous iterations are used to find the frequent itemsets in the current iteration. It starts with finding singleton frequent items where an item occurring more times than a user-specified minimum threshold (support count) is termed frequent. Then K-frequent itemsets are found using K-1 frequent itemsets based on the Apriori approach. After finding singleton frequent items, a candidate set is generated. The candidate set for the Kth iteration has all K size combinations of items whose all
possible subsets are frequent in \((K-1)\)th iteration. To find \(K\)-frequent itemsets, we check only itemsets whose all possible subsets are present in \(K\)-1 frequent itemsets. Now these frequent itemsets of the \(K\)th iteration are used for the generation of \(K+1\) phase candidates and it iterates until all frequent patterns are found.

A major drawback of the Apriori algorithm is that the candidate set for the 2nd phase is too huge if the number of items is large and the entire dataset is scanned several times in every iteration that follows. \(\text{Eclat}\) \cite{6} tackles this problem of multiple passes over the entire dataset using a horizontal approach. In this approach, a list \(L\) is created where every row contains: 1) the item \(i\) and 2) the corresponding list of transactions that contain item \(i\). Singleton frequent items are found by counting the number of transactions in column 2 that contain item \(i\). After creating the candidate set for the \(K\)th iteration, intersection of transactions in the 2nd column for all items in \(L\) can be used to find the support of an itemset. This approach eliminates making multiple scans over the entire dataset in each iteration, unlike in \(\text{Apriori}\). \(\text{Eclat}\) makes a single pass of the dataset, which makes it much faster than Apriori, but it does not solve the case where there are too many items generated in the 2nd phase. To resolve this huge candidate set problem \(\text{FP-growth}\) \cite{8} algorithm was proposed by J. Han et al. In \(\text{FP-Growth}\), the candidate generation step is eliminated. Initially, singleton frequent itemsets are found and a \(\text{FP-Tree}\) is created. This \(\text{FP-Tree}\) is used to find frequent patterns of all sizes. This algorithm is faster than Apriori because it avoids the candidate generation step. The aforementioned algorithms work in a sequential manner on relatively small datasets. As the size of datasets increases, the efficiency of these algorithms drops. Therefore, to handle large datasets, parallel algorithms are introduced. Initially, cluster based parallel algorithms\cite{17} are implemented. These cluster-based algorithms were capable of handling large datasets but they were complex and had many issues like synchronization, replication of data etc. These parallel approaches were thus replaced by \(\text{MapReduce}\) approaches. \(\text{MapReduce}\) approach makes association rule mining process very fast because algorithms like Apriori can achieve higher parallelism. Key-value pairs can be easily generated in case of Apriori algorithm. Many \(\text{MapReduce}\) based implementations of Apriori Algorithm\cite{12, 7} have been proposed that show high performance gains as compared to conventional Apriori. Hadoop\cite{2} is one of the best platforms to implement Apriori as a \(\text{MapReduce}\) model. Hadoop distributed file system (HDFS) is used to store the dataset, which is then scanned to find frequent patterns. Later, every mapper receives an input from HDFS and produces a Key-value pair where the item is the key and the item’s occurrences form the payload of the value.Reducers combine different values of a Key to calculate the total count for any key. If the count exceeds the minimum support then the item is deemed frequent otherwise it is removed from the list. This filtering process is repeated in each iteration after generating the candidate set. It makes the process of finding frequent patterns highly parallel and fast. However there are still some limitations in Hadoop based implementations of Apriori. On Hadoop platform, results are stored to HDFS after each iteration and input is sourced from HDFS for the next iteration, which decreases the performance due to the high I/O time. But \(\text{Spark}\)\cite{15}, a new in-memory, distributed data-flow platform, tackles these problems by using its RDD architecture, which stores results at the end of an iteration and provides them for the next iteration. Apriori implementation on \(\text{Spark}\) platform gives many times faster results on standard datasets which makes \(\text{Spark}\) the best tool for implementation of Apriori. Recently, Qiu et al. \cite{5} have reported speedups of nearly 18 times on average for various benchmarks for the \textit{yet another frequent itemset mining (YAFIM)} algorithm based on \(\text{Spark RDD}\) framework. Their results on real world medical data is observed to be many times faster than on the MapReduce framework. We consider YAFIM to be the current state-of-the-art Apriori implementation. These results sparked our interest to understand the flow of the algorithm on \(\text{Spark}\) architecture and triggered us to eliminate what seems to be the time consuming step in the whole flow of the algorithm, namely, the second pass where unnecessary computations were happening in YAFIM.

In this paper, we propose \textit{Reduced-Apriori} (\textit{R-Apriori}), a parallel Apriori algorithm based on the \(\text{Spark RDD}\) framework. Furthermore, we speedup the 2nd round of candidate set generation by eliminating candidate generation altogether, in order to achieve higher speedups in comparison to YAFIM (i.e., the state-of-the-art). This algorithm is a part of PhD work of 1st author. 1st author is working on a Crowd Mining framework which analyze high volumes of Crowd data and predict the behaviour of people. \(\text{R-Apriori}\) is used in 1st phase of Crowd Mining framework to analyze big datasets to extract useful patterns which can be used to make rules to show Crowd behaviour.

The paper is organized as follows. After introducing the subject and the motivation in Section I, earlier work about frequent itemset mining, mainly Apriori algorithm is reported in Section II. Section III introduces our proposed \textit{R-Apriori} algorithm in detail. Section IV evaluates the performance of \textit{R-Apriori} and compares it to the YAFIM, which we consider as the state-of-the-art solution. Section V shows the use of \textit{R-Apriori} algorithm for crowd mining and section VI concludes the paper.

2. EARLIER WORK

\textit{Apriori} was implemented by R. Agrawal et al.\cite{18}. Two algorithms \textit{AprioriTid} and \textit{AprioriTid} were introduced by R. Agrawal et al. to discover all significant association rules for transaction datasets. Later, many improvements were done to make Apriori better, efficient and faster. Ning Li et al.\cite{17} proposed a parallel implementation of Apriori based on \textit{MapReduce}. They used \textit{<Key, Value>} pairs to find frequent itemsets in parallel and evaluate the performance for parallel Apriori algorithm and showed that their method was more efficient as the size of datasets increased. Zahra Farzanyar et al.\cite{16} reported a time efficient algorithm based on \textit{MapReduce} framework, called \textit{CMR-Apriori}, to mine frequent itemset from large scale social network datasets and showed that their algorithm achieved better performance in terms of execution time as compared to MRApriori algorithm by using an efficient pruning technique. \textit{CMR-Apriori} \cite{9} by Jian Guo et al., was an improved Apriori algorithm with improvements using coding and Mapreduce. They applied it to a book recommendation service system and found that \textit{CMR-Apriori} algorithm outperformed traditional Apriori. Xin Yue Yang et al.\cite{20} implemented Apriori on \textit{Hadoop} using cloud computing where the underlying cloud computing framework, can easily handle issues like concurrency control, network communication and fault tolerance. Ming-Yen Lin et al.\cite{14} proposed three new methods based on the parallel implementation of Apriori called \textit{SPC}, \textit{FPC} and \textit{DPC} and tested them for different cluster and database sizes. Hongjian Qiu et al.\cite{5} provided a Mapreduce based implementation of Apriori algorithm(YAFIM) on \textit{Spark} platform and showed substantial
improvements in runtime in comparison to Hadoop based implementations of Apriori. YAFIM was many times faster than all Hadoop based algorithms, but for the 2nd phase when the number of candidate pairs was too much, it was not as efficient.

3. THE R-APRIORI ALGORITHM

Apriori is an algorithm, which uses an iterative approach to find frequent itemsets in a transactional database and generate association rules from these frequent itemsets. It is based on an observation that an itemset is frequent only if all its non-empty subsets are frequent. Each kth iteration of Apriori generates frequent patterns of length k. The 1st iteration finds all frequent items of length 1. Now a candidate set is generated having all possible combinations of length 2 whose every possible non-empty subsets are frequent. In the 2nd iteration, all frequent itemsets of length 2 are discovered. Now these frequent itemsets of length 2 are used to create candidate set having all possible combinations of length 3 whose all non-empty subsets are frequent. This procedure iterates until there is no frequent itemset left. Apriori implementation by R. Agarwal et al.[18] has a single phase where every iteration initially generates a candidate set from results of the earlier iteration and scans the dataset to find occurrences of itemsets in the candidate set and yields itemsets which have occurrences that exceed the user-specified minimum support, as frequent itemsets. There exist MapReduce implementations of Apriori on Hadoop. Hadoop makes Apriori more scalable and reliable by providing a parallel storage and computation environment.

YAFIM (Yet Another Frequent Itemset Mining)[5], which is an Apriori implementation on Spark significantly outperforms the Hadoop implementations of Apriori. Initially transactional datasets from HDFS are loaded into Spark RDDs (Resilient Distributed Datasets), which are memory-based data objects in Spark, to make good use of the available cluster memory. It uses two phases: In the 1st phase, it generates all singleton frequent items and in the 2nd phase it iteratively uses k-frequent itemsets to generate (k+1)-frequent itemsets.

Our proposed R-Apriori is a parallel Apriori implementation on Spark. It adds an additional phase to YAFIM. R-Apriori has three phases in the processing workflow model. Our modifications to the 2nd phase reduces the number of computations for generating pairs in Apriori. Finding frequent pairs are the most time and space consuming step in Apriori which is greatly simplified by our approach.

In R-Apriori, the 1st phase is similar to YAFIM. We further divide the 2nd phase by first tackling the 2-itemset generation differently from YAFIM followed by generating all subsequent k-itemsets in the same manner as YAFIM. The next section explains all the three phases of R-Apriori in detail.

3.1 Implementation

3.1.1 Phase I

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 input file is broadcasted to every worker node to make it available locally to each worker. Subsequently, a flatMap() function is applied on the dataset to read transactions and then again a flatMap() function is used on each transaction to read every item from it. After the flatMap() process is completed, a map() function converts each Item into (item, 1) key/value pairs. Finally, reduceByKey() function is invoked to calculate the frequency of each item and prunes items having frequency lesser than minimum support count. Remaining items are stored in the Singleton-frequent itemset. Figure 1(a) presents the algorithm used to compute singleton frequent items in phase I. Lineage graph, showing flow of data and control through various stages of phase I is presented in Figure 1(b). Figure 1(c), is the graphical depiction of the input and output flow of mappers and reducers.

3.1.2 Phase II

In the 2nd iteration of classical Apriori on Spark, a candidate set is generated from singleton frequent items. This candidate set is stored in a hash tree to make searches faster. Every mapper takes a transaction and yields all possible pairs for that transaction which exists in the candidate set. Reducers combine all occurrences of a pair and yield it if its count is more than minimum support. The 2nd iteration is the most time and space consuming iteration for Apriori algorithm due to its generation of a huge candidate set. For example, for 100 frequent singletons, there are nearly 106 candidate pairs. To check such a huge number of candidates for every transaction is time consuming.

In R-Apriori, we improved performance of the 2nd iteration by: 1) removing the candidate set generation step and 2) using a bloom filter in place of a hash tree.

In R-Apriori, singleton frequent items are stored in a bloom filter because bloom filters are faster than hash trees and can easily store items of length 1. Mapper takes every transaction and computes the total count for each pair and yields all the pairs having total count more than the user-specified minimum support. The output of mapper and reducer, as expected, is similar for R-Apriori as it is for classical Apriori on Spark. However, elimination of the candidate set generation step and the use of bloom filters makes R-Apriori more time and space efficient.

Observation 1: R-Apriori is faster than classical Apriori on Spark if the number of singleton frequent items is large.

The Reducer receives the same input and produces the same output as in case of conventional Apriori, so the time complexity is the same for the reducers in both algorithms. Time complexity during mapper execution is to be looked at. 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 lth iteration. Further, t is the time to find an element in a hash tree and b is the time taken to search an element in the bloom filter.

For Apriori total time of 2nd iteration is the sum of the following:

(i) Time to generate candidate set (Tg)
(ii) Time to store candidate set in hash tree (Ts)
(iii) Time taken by mapper (Tm)

Time to generate the candidate set is the sum of the following:

(i) Time to join
(ii) Time to prune
The total time to generate candidate set then equals:

$$\frac{f(f-1)}{2} + \frac{2f(f-1)}{2} = \frac{3f(f-1)}{2}$$

Time complexity to store a candidate set in a hash tree equals:

$$\frac{3f(f-1)}{2} \approx O(f^2)$$

Time complexity to search and yield pairs equals:

$$\frac{X}{M} \times \frac{tg(g-1)}{2} \approx O(X/M \cdot g^2)$$

For R-Apriori the time of the 2<sup>nd</sup> iteration is the sum of:

(i) Time to store singleton frequent items in the Bloom filter ($T_{bd}$)

(ii) Time to make pruned transaction($T_{ps}$)

(iii) Time to generate pairs ($T_{m}$)

Time to store singleton frequent items in the Bloom filter= $f$

Time complexity to make pruned transaction equals:

$$\frac{X}{M} \times g \approx O(X/M \cdot g)$$

Time complexity to generate pairs in worst case equals:

$$\frac{X}{M} \times \frac{bg(g-1)}{2} \approx O(X/M \cdot b \cdot g^2)$$

Thus, the overall time complexity for R-Apriori equals:

$$f + \frac{X}{M} \times \frac{bg(g-1)}{2} \approx O(f + X/M \cdot b \cdot g^2)$$

Next, we analyze the time complexity of YAFIM to show that R-Apriori performs better theoretically.

Time Complexity of YAFIM - Time Complexity of R-Apriori

$$= \frac{5f(f-1)}{2} + \frac{X}{M} \times \frac{tg(g-1)}{2} - f \times \frac{X}{M} \times \frac{bg(g-1)}{2}$$

$$= \frac{f(5f-7)}{2} + \frac{X}{M} \times \frac{tg(g-1)}{2} \times (t - b) \approx O(f^2 + X/M(t-b) \cdot g^2)$$

The analysis shows that YAFIM incurs an additional quadratic cost in $f$, while R-Apriori is still linear in $f$.

As time to search an element in a bloom filter (b) is one cycle, therefore for the situation where $t$ is more than one cycle, R-Apriori will be faster. Hence, large datasets where $t>1$ will be dealt more efficiently by R-Apriori.

**Observation 2:** Space complexity of R-Apriori is lower than the classical Apriori algorithm on Spark.

The Reducer receives the same input and produces the same output for R-Apriori as in the case of conventional Apriori. Therefore, space complexity is the same for reducers of both algorithms. However, space complexity during mapper execution matters most.

For Apriori, space complexity of the 2<sup>nd</sup> iteration is the sum of:

(i) Space complexity to store the candidate set

(ii) Space complexity to create the hash tree to store the candidate set.

If there are $f$ frequent items remaining after the $f$<sup>th</sup> iteration. Additionally, let $b$ and $c$ be the bytes required to store a pair of candidate set and a single item in the hash tree, respectively.
Figure 2. Phase 2 of R-Apriori. (a) Algorithm used in Phase 2 (b) Lineage graph for RDDs of Phase 2 (c) Graphical Diagram for Phase 2 where $T = \{T_1, T_2, T_3, \ldots\}$ is a set of transactions in dataset.

Space complexity to store candidate set equals

$$\frac{bf(f-1)}{2} \approx O(bf^2)$$

Space complexity to create a hash tree equals

$$\frac{bf(f-1)}{2} + \frac{cf(f-1)}{2} \approx O((b+c)f^2)$$

The overall space complexity for YAFIM mapper equals

$$\frac{f(f-1)}{2}(2b+c) \approx O((b+c)f^2)$$

For R-Apriori, the space complexity of the $2^{nd}$ iteration is the sum of:

(i) Space complexity to store singleton frequent items in Bloom filter $= cf$

Therefore, the space complexity of R-Apriori is much lower in comparison to conventional Apriori on Spark. Since Spark is an in-memory computing model so reduction in space complexity makes R-Apriori more efficient in handling large datasets.

**Observation 3: R-Apriori scales up better as a function of number of cores.**

The Reducer receives the same input and thus produces the same output for R-Apriori as in the case of conventional Apriori. However, the mapper performs fewer computations in the case of R-Apriori.

For Apriori, the total time complexity of the $2^{nd}$ iteration is the sum of:

(i) Time to generate candidate set ($T_g$)

(ii) Time to store candidate set in HashTree ($T_h$)

(iii) Time taken by mapper ($T_m$)

In our proposed R-Apriori, time to generate candidate set ($T_g$) becomes zero and time to store candidate set in Hash Tree ($T_h$) is the time to store singleton frequent items in bloom filter ($T_b$) which is negligible. Further the time taken by mapper of R-Apriori is also lesser as number of comparisons are reduced. Therefore, in case of R-Apriori, total time taken by mappers will drop down quickly if number of mappers(or cores) are increased.

Figure 2.b shows lineage graph for $2^{nd}$ phase of R-Apriori algorithm. Here a flatMap() function is used to read every transaction from the dataset. Then, another flatMap function is used to get the intersection set of singleton-frequent itemsets and each transaction to get frequent items in each transaction. So we can say that each transaction is pruned to have only items, which were frequent in phase 1. Now, a map() function is used to create pairs. Then a map() function produces(Item, 1) for each Item. Graphical diagram (Figure 2.c) shows how mapper and reducer takes input and produce results. At last reduceByKey() function calculates the frequency of each item and prunes items whose frequency is less than the minimum support count. Remaining items are stored in 2-frequent itemsets.
3.1.3 Phase III

Here we use K-frequent itemsets to generate (K+1)-frequent itemsets. Initially K-frequent itemset L_k are loaded into RDDs and candidate set C_k+1 is generated and stored in hash tree to increase searching speed of K+1 itemset.

Now a flatMap() function is used to get each transaction and check possible K+1 frequent sets for that transaction and a map() function yield (Item, 1) for each Item. At last reduceByKey() function calculate the frequency of each item and prune items for which frequency is less than minimum support count. Remaining items are stored in K-frequent itemset. Lineage graph (Figure 3.b) shows flow of data and control through various stages of phase III. Graphical diagram (Figure 3.c) shows how mapper and reducer takes input and produce results.

4. PERFORMANCE EVALUATION

In this section, we evaluated the performance of R-Apriori and compared it to standard Apriori on Hadoop(MRApriori[20]) and Apriori on Spark(YAFIM [5]). R-Apriori is implemented using Spark, an in-memory computing framework. There are many other MapReduce implementations of Apriori on Hadoop but all of them are very similar to MRApriori in performance. Each of them reads data from HDFS and after each iteration writes it back to HDFS. Due to which performance of every MapReduce implementation of Apriori is nearly the same. In our experiments, many benchmark datasets were used. All experiments were executed four times and average results were taken as the final result. R-Apriori and YAFIM were implemented on Spark-1.0.1 and MRApriori was implemented on Hadoop-2.0. All datasets are available on the same HDFS cluster. All experiments were conducted on a cluster of 2 nodes each having 24 cores and 180GB RAM. The computing cores were all running on Ubuntu 12.04 LTS and java-1.8.0.

4.1 Datasets

Experiments were done with five large datasets having different characteristics. The first dataset was T10I4D100K (artificial datasets generated by IBM’s data generator) [4] which contains 10^7 transactions with 870 items in it. Retail dataset[4] was used for market-basket model, it contains various transactions carried out by customers in a shopping mall. Kosarak dataset[4] was donated by Ferenc Bodon and contains the click-stream data of a Hungarian on-line news portal. BMSWebView2[3] is a dataset used for the KDD cup 2000 competition and has average length 4.62 with 6.07 standard deviation. T25110D10K[3] is a synthetic dataset generated by a random transaction database generator. Properties of these datasets are as follow:
### Table 1. Datasets characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Items</th>
<th>Number of Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I4D100K</td>
<td>870</td>
<td>100,000</td>
</tr>
<tr>
<td>Retail</td>
<td>16470</td>
<td>88,163</td>
</tr>
<tr>
<td>Kosarak</td>
<td>41,270</td>
<td>9,90,002</td>
</tr>
<tr>
<td>BMSWebView2</td>
<td>3340</td>
<td>77,512</td>
</tr>
<tr>
<td>T25I10D10K</td>
<td>990</td>
<td>4,900</td>
</tr>
</tbody>
</table>

### 4.2 Speed Performance Analysis

Performance for all algorithms with different datasets were evaluated. Results except 2nd iteration are same because except 2nd iterations, R-Apriori follows same procedure as standard Apriori on Spark. For all five datasets, comparison of R-Apriori and standard Apriori on Spark is shown. For T10I4D100K dataset, R-Apriori outperforms standard Apriori on Spark by 3 times(Figure 4.1). For Retail and BMSWebView-2 dataset, R-Apriori is nearly 9 times faster than standard Apriori on Spark (Figure 4.2 and Figure 4.3).
R-Apriori is more than 2 times faster for Kosarak dataset with minimum support 0.6% (Figure 4.4) and more than 4 times faster for T2S110D10K dataset with 1% minimum support (Figure 4.5). For every dataset, R-Apriori outperforms standard Apriori on Spark.

5. CONCLUSION
A new faster and efficient association rule mining algorithm based on Apriori is implemented to mine frequent patterns from large datasets having various attributes. Conventional Apriori algorithm consumes too much time and space for the 2nd iteration of frequent itemset generation. For instance, given 10^4 singleton frequent items, candidate set for 2nd iteration is nearly 10^8. Checking occurrences of such a huge candidate set for every transaction is infeasible. We reduce the total number of computations for the 2nd iteration and improve the performance of Apriori many times for a large dataset. R-Apriori gives improved performance as the size of the dataset and the number of items increases. We implemented R-Apriori on Spark, which provides a highly parallel computational environment as compared to other platforms. Theoretical and empirical comparison of R-Apriori with already existing Apriori implementation on Spark platform (YAFIM) are conducted to give insight into the superiority of our approach. Additionally, R-Apriori outperforms classical Apriori on Spark for different standard datasets.

6. REFERENCES
[5]. Hongjian Qiu, Rong Gu, Chunfeng Yuan and Yihua Huang. YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark. In 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, 2014.
[8]. J. Han, H. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. In Proc. Conf. on the Management of Data (SIGMOD’00, Dallas, TX), ACM Press, New York, NY, USA 2000.