The Journal of Supercomputing (2021) 77:133–151
https://doi.org/10.1007/s11227-020-03253-7

A Spark-based Apriori algorithm with reduced shuffle overhead

Shashi Raj[1] · Dharavath Ramesh[2] · Krishan Kumar Sethi[2]

Published online: 27 March 2020

Abstract
Mining frequent itemset is considered as a core activity to find association rules from transactional datasets. Among the different well-known approaches to find frequent itemsets, the Apriori algorithm is the earliest proposed. Many attempts have been made to adopt the Apriori algorithm for large-scale datasets. But the bottlenecks associated with Apriori like/such as repeated scans of the input dataset, generation of all the candidate itemsets prior to counting their support value, etc., reduce the effectiveness of Apriori for large-size datasets. When the data size is large, even distributed and parallel implementations of Apriori using the MapReduce framework does not perform well. This is due to the iterative nature of the algorithm that incurs high disk overhead. In each iteration, the input dataset is scanned that resides on disk, causing the high disk I/O. Apache Spark implementations of Apriori show better performance due to in-memory processing capabilities. It makes iterative scanning of datasets faster by keeping it in a memory abstraction called resilient distributed dataset (RDD). An RDD keeps datasets in the form of key-value pairs spread across the cluster nodes. RDD operations require these key-value pairs to be redistributed among cluster nodes in the course of processing. This redistribution or shuffle operation incurs communication and synchronization overhead. In this manuscript, we propose a novel approach, namely the Spark-based Apriori algorithm with reduced shuffle overhead (SARSO). It utilizes the benefits of Spark's parallel and distributed computing environment, and it is in-memory processing capabilities. It improves the efficiency further by reducing the shuffle overhead caused by RDD operations at each iteration. In other words, it restricts the movement of key-value pairs across the cluster nodes by using a partitioning method and hence reduces the necessary communication and synchronization overhead incurred by the Spark shuffle operation. Extensive experiments have been conducted to measure the performance of the SARSO on benchmark datasets and compared with an existing algorithm. Experimental results show that the SARSO has better performance in terms of running time and scalability.

Keywords Apache Spark · Apriori algorithm · Large-scale datasets · Shuffle overhead

Extended author information available on the last page of the article

1 Introduction

With the rapid evolution of the internet, a huge volume of data is being produced in our day-to-day life. These large volumes of data are collected to filter out some potential values that can further generate useful information and knowledge. Frequent itemset mining algorithms aim to extract certain association or relation among the items from transactional databases by discovering frequent patterns of itemsets [1–3]. It further acts as a basis to derive strong association rules. Apriori algorithm is a widely used classical approach to mine frequent itemsets. It shows good performance when dealing with small datasets. But in the Big Data era, data are too large in size and unstructured in nature, which makes it difficult to be processed by traditional data processing methods [4–7]. On the other hand, it faces significant challenges, such as limited computing power and memory resources. Many single machine variations of the Apriori have been proposed to address its limitations [8–11]. One of the key concerns in the majority of the algorithms was its requirement to have multiple passes over the input data set that resides on disk. Repeated scanning of the input dataset incurs a huge I/O overhead. It becomes considerable when it comes to processing large-scale datasets. The partition algorithm [12] tries to reduce this I/O overhead based on a novel approach. It makes non-overlapping partitions of the input dataset so that each partition fits well into the main memory. However, it may suffer from higher time complexity because of the generation of a larger set of candidate itemsets by different partitions.

Single machine variants of Apriori work well with relatively small datasets, but when it comes to the processing of large datasets, it demands a parallel and distributed computing environment. Hadoop-based MapReduce model fulfills many of the requirements for processing large datasets in a parallel manner by forming a cluster of computing nodes. Many MapReduce implementations are also proposed to get better efficiency [13–17]. However, because of the iterative nature of the Apriori, the MapReduce framework does not perform well for large dataset processing. All the intermediate results are stored back to the Hadoop Distributed File System (HDFS) after each iteration. Moreover, the input dataset that is being scanned in each iteration to count the frequency of a candidate itemset also resides on HDFS. The data access incurs a huge memory and I/O overhead. To overcome the above shortcoming, Apache Spark introduces in-memory processing capabilities to make better use of cluster memory [18–20]. Instead of writing all the intermediate results to the HDFS, it writes it to a distributed memory abstraction called resilient distributed datasets (RDDs). It has been observed in the past research works that implementing frequent itemset mining algorithms on Spark shows better performance than MapReduce-based implementations [21–23].

Many Spark-based implementations of the Apriori algorithm have been developed to improve the efficiency further in terms of run time and scalability. YAFIM is an adaptation of the Apriori algorithm in the Spark framework [21]. It utilizes the in-memory processing abstraction called RDD to make iterative computations faster. R-Apriori eliminates the candidate generation step altogether for the second iteration [22]. Also, it uses bloom filters in place of hash trees to avoid costly comparisons. In both the algorithms, it has been observed that the shuffle operation of Spark imposes a huge overhead in rearranging *key-value* pairs across the cluster nodes during RDD transformations. Key-value pairs act as a basic element of Spark's paired RDD. Many of the operations on RDDs require these key-value pairs to be redistributed or shuffled among the cluster nodes. While processing large-scale datasets, the number of key-value pairs generated would be large. So the shuffling of these huge pairs would incur communication and synchronization overheads. To handle this, in this paper, we propose a novel approach named SARSO that shows significant improvement in terms of efficiency by incorporating a solution for the above limitation. A partitioning technique is used where the whole input dataset is divided into n non-overlapping partitions.

SARSO works in two phases. In the first phase, the input dataset is scanned to find all the frequent local itemsets of all possible lengths for each partition using a local relative threshold. The local relative threshold for a partition is obtained by dropping the value of the user-supplied global threshold value in the proportion of its size. Then all these frequent local itemsets produced by each partition are collected at the master node. All the duplicate frequent itemsets from two different partitions are dropped. These revised locally found frequent itemsets act as global candidate itemsets with respect to the whole input dataset. In the second phase, the input dataset is scanned again to generate global frequent itemsets by extracting only those global candidate itemsets that have support value more than or equal to the global threshold. Here, we use the concept that a frequent local itemset may or may not be the frequency with respect to the entire dataset. However, any item set that is globally frequent must occur as a frequent itemset in at least one of the partitions.

By finding all the k-length local frequent itemsets first, we restrict the shuffling of key-value pairs in each iteration as required by the Apriori algorithm. Shuffling of key-value pairs occurs only when global frequent itemsets are being calculated. Some additional communication overhead is imposed on collecting frequent local itemsets from different partitions to get global candidate itemset and again broadcasting it to all the computing nodes. The performance of SARSO is evaluated in terms of efficiency and scalability for different well-known datasets. The experimental results show that SARSO exhibits better performance in comparison with YAFIM for lower values of minimum support.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 gives a brief introduction about the Apriori algorithm and shuffle operation of the Apache Spark framework. In Sect. 4, the proposed methodology is discussed in detail. Experimental results are analyzed to measure the performance of the proposed scheme in comparison with YAFIM in Sect. 5. Section 6 discusses the conclusion and future scope related to the computation.

2 Related literature

Many versions of the Apriori algorithm have been proposed to process large-scale datasets with improved efficiency. This was achieved by parallelizing the computation over a cluster of nodes to make algorithms more suitable for large-scale datasets. Lin et al. [21] proposed three different versions of MapReduce-based Apriori algorithm, namely SPC, FPC, and DPC. SPC is an adaptation of the Apriori algorithm on the MapReduce framework. FPC counts the frequency of kth, $(k+1)$th and $(k+2)$th itemsets altogether in a single map-reduce phase and hence reduces the number of MapReduce phases to improve the efficiency. DPC tries to make a balance between SPC and FPC. It finds a trade-off between reductions in the number of map-reduce phases and increments in the number of pruned candidate itemsets. Moens et al. [22] introduced two different approaches, namely *Dist-Eclat* and *BigFIM*. *Dist-Eclat* is a MapReduce implementation of the Eclat algorithm, which uses a simple load balancing scheme for speedup. It requires a specific encoding of the input dataset. On the other hand, *BigFIM* focuses on mining very large-scale datasets. It is a hybrid approach that uses an Apriori variant with Eclat algorithm of the MapReduce framework.

MRApriori developed by Hammoud et al. [23] iteratively switches between vertical and horizontal database layouts to mine all frequent itemsets. At each iteration, the database is partitioned and distributed across mappers for support counting. Hammoud et al. [24] introduced a parallel MapReduce-based cluster enabled algorithm that uses a hybrid learning approach to transform the intermediate data communicated among cluster nodes to find the frequent itemsets quickly. Yu et al. [25] developed a distributed parallel Apriori algorithm (DPA) that stores metadata in the form of transaction identifiers (TIDs), such that only a single scan to the database is needed. DPA balances workload among processors by taking the factor of itemset counts into consideration, hence reducing the processor idle time. Aouad et al. [26] presented a performance study of a novel distributed Apriori-like algorithm. The proposed approach was intended to limit synchronization and communication overheads. It showed that the intermediate communications among the cluster nodes are computationally inefficient, and the resultant global pruning step does not constitute enough useful information. Chen et al. [27] proposed BE-Apriori that uses pruning optimization and transaction reduction strategy to improve the performance compared to Apriori.

All the algorithms mentioned above are of iterative nature, i.e., they fully scan the input dataset in each iteration. This repeated scanning incurs a huge I/O overhead in case of large-scale datasets and consequently shows performance bottleneck. Because of the iterative nature of the Apriori-based algorithm, even MapReduce framework does not fit well. Every MapReduce job needs to read the input dataset and again write the intermediate results back to the HDFS that incurs a huge I/O overhead. Zhang et al. [28] introduced a distributed algorithm for frequent itemset mining (DFIMA) that uses a matrix-based pruning approach to reduce the number of candidate itemsets generated in each iteration. To make iterative with I/O devices efficient, DFIMA was implemented using the Spark framework.

Qiu et al. [29] proposed Yet Another Frequent Itemset Mining (YAFIM) algorithm that is an adaptation of the Apriori algorithm in the Spark environment. It utilizes the in-memory processing abstraction called RDD to make iterative computations faster. Comparative performance analysis of YAFIM and MapReduce version of the Apriori is done in each iteration on different benchmark datasets. YAFIM shows a speedup of about $18\times$than its MapReduce implementation. Rathee et al.

[30] proposed R-Apriori that eliminates the candidate generation step altogether in the second iteration. In general, the number of candidate itemsets generated in the second round are huge. Removing the traditional candidate generation scheme improves efficiency by reducing the total number of candidate itemsets generated for the second iteration. It uses bloom filters in place of hash trees to avoid costly comparisons. R-Apriori shows speed up in the performance of the second iteration, specifically in comparison with YAFIM. Sethi et al. [31] introduced HFIM that utilizes both vertical and horizontal layouts of the input dataset to deal with memory and computation overheads. A horizontal layout is used to find the candidate itemsets, and a vertical layout is used to count the support value for the candidate itemsets.

In most of the Spark-based algorithms, candidate itemsets at each iteration are generated by master node using frequent itemsets derived in the previous iteration. Then it is broadcasted to all the slave nodes. MapReduce architecture is used to get the support of the candidate itemsets and filter out the frequent itemsets. MapReduce is mainly used for parallel processing of large datasets stored in Hadoop cluster. It provides parallelism, data distribution, and fault tolerance. Additionally, none of the above-mentioned frequent pattern mining algorithms paid attention toward repeated scanning of input datasets at each iteration that incurs very high disk I/O. The in-memory processing capabilities of Spark reduce disk overhead significantly by keeping the input dataset in the form of RDD. However, the total number of input dataset scans required still remains the same. Moreover, very little attention has been paid toward communication overhead incurred due to the movement of key-value pairs to get the support of the candidate itemsets at each iteration. We propose a Spark-based algorithm called SARSO that improves the efficiency by reducing shuffle overheads caused by RDD transformations. It is parallel in nature and incurs minimal communication and synchronization overhead among the processing nodes. Also, it generates all the frequent itemsets in two scans of the input dataset.

3 Preliminaries

In this section, first, we discuss briefly on the Apriori algorithm and the impact of shuffle operation on Apache Spark's efficiency. Then the proposed scheme SARSO is explained in detail.

3.1 Apriori algorithm

The Apriori algorithm proposed by Agarwal et al. [8] is an iterative two-step process consisting of *join* and *prune* operations. At each iteration, the *join* step generates a set of candidate itemsets from the frequent itemsets found at the previous iteration. Then the *prune* step is performed to filter out only the potential candidate itemsets from the candidate itemsets generated in *join* step using apriori property. Finally, the input dataset is scanned to count the *support* of these candidate itemsets. If the support of a candidate itemset is more than the user-defined threshold value, that candidate itemset is called a frequent itemset. Here, support value represents the occurrence frequency of an itemset, i.e., the number of transactions that contain that itemset. Apriori property says that an itemset is frequent only if all its non-empty subsets are also frequent. As the name Apriori suggests, it uses prior knowledge. It finds the k-frequent itemsets based on previously found (k-1) itemsets where k represents the iteration number and also the length of the itemset.

Algorithm$_1$ depicts the pseudo-code of the Apriori algorithm. In the first iteration, the input dataset is scanned to find the support of each item. 1-frequent itemsets are determined by filtering only those items whose support count is equal or more than a user-specified threshold value called minimum support (line 1). Then k-frequent itemsets (for $k > 1$) were found using (k-1) frequent itemsets (line 2). The *join* step produces all the possible k-candidate itemsets by joining (k-1)-frequent itemsets with itself (line 3). The *prune* step uses *apriori property* to reduce the number of candidate itemsets (line 4–5) by filtering out the promising candidates only. Then the input dataset is scanned again to find the frequency of these promising candidate itemsets to filter out the k-frequent itemsets (line 6–10). The set of all the k-frequent itemsets ($k \geq 1$) are then returned as the final output (line 12).

Algorithm$_1$: ***Apriori algorithm***

```
Input:    D: Input Dataset
          minSup: minimum support threshold
Output:   All 2 to k-frequent itemsets

 1.  $L_1$= {1-frequent itemset} // found separately
 2.  for ($k = 2$; $L_{k-1} \neq \varphi$; $k$++)
 3.    $C_k$ = apriori_gen($L_{k-1}$) // finds k-candidate itemsets by
         joining and pruning $L_{k-1}$ with itself
 4.      for each transaction t in D
 5.          $C_t$ = subset ($C_k$, t) // finds candidate itemsets in t
 6.          for each c in $C_t$
 7.              c.count++
 8.          end for each
 9.      end for each
10.      $L_k = \{c \in C_k \mid c.\text{count} \geq minSup\}$
11.  end for
12.  Return $\cup_k L_k$
```

3.2 Apache Spark

Apache Spark [18] is an open-source project developed in the AMPLab at UC Berkeley. It is a framework to process large-scale datasets with parallel and distributed computing environment and supports in-memory processing capabilities.

It is written in Scala and provides a variety of programming interfaces such as Java, Scala, Python, and R for lightning-fast cluster computing. It is an extension to Hadoop's MapReduce methodology designed for batch processing. Spark can process both batch and real-time (streaming) applications efficiently. Apache Spark is faster than MapReduce and offers low latency due to reduced I/O operations. It maintains the intermediate results in memory rather than writing it to disk every time. The main feature of Spark is its in-memory cluster computation using the abstraction called RDDs. Datasets in the form of RDDs remain logically partitioned across the cluster nodes and preferably accommodated in the primary memories of the cluster nodes. As the word resilient suggests RDDs are fault-tolerant, it is an immutable (read-only) collection of data by which it provides fast and efficient iterative operations. We can create a new RDD either by loading an input dataset in the spark environment or by applying transformation actions on some existing RDDs.

To understand shuffle operation, imagine that there are five branches of a bank in a city. All these branches are recording their daily transactions made by the customers. Suppose the controlling authority of the bank wants to calculate the total number of transactions recorded by its branches on different dates. The authority will access the records at different branches and would set the *date* as a search key. Then, for every record it would emit a pair $<date, 1>$, where *date* represents the day on which that transaction was recorded. This pair acts as a key-value pair in Spark. Now key-value pairs with the same key are summed up to get the result. Since all the key-value pairs are on the different nodes, key-value pairs with the same key move to the same node to get the summed up value. This redistribution of data is known as shuffling operation in Spark. Certain operations within Spark needs this redistribution of data across the partitions of RDD that resides on different nodes so that it can be grouped differently. This involves the movement of data from one node to another, i.e., shuffling of key-value pairs. Shuffle operation incurs extra overhead and is limited by network bandwidth [32]. Operations that can cause shuffle to include *repartition*, *ByKey,* and *join* operations. The shuffle operation of Spark is a complex and expensive operation due to the involvement of disk and network I/O.

4 Proposed methodology: SARSO

In this section, we describe the proposed approach SARSO in detail. SARSO is an improved version of the existing YAFIM algorithm. Spark provides a distributed and parallel environment with in-memory processing capabilities to enhance performance. YAFIM is an iterative algorithm where kth iteration is responsible for finding k-frequent itemsets. It iteratively generates k-frequent itemsets using (k-1)-frequent itemsets for all $k \geq 1$ where $k_0 = \varphi$. YAFIM uses traditional *apriori_gen* function to calculate the k-candidate itemsets based on (k-1)-frequent itemsets. Then, the Spark's parallel MapReduce framework is used to count the support of the k-candidate itemsets and consequently filter out only the frequent itemsets that have the support value more than the user-specified threshold value.

On the other hand, SARSO executes in two phases. In the first phase, it logically partitions the input dataset in n non-overlapping partitions of approximately equal size residing on different computing nodes. All the partitions are processed independently and on individual nodes in parallel. First, each processing node generates all the frequent itemsets of all possible lengths for their local data partition. Since the size of local data at each node is approximately equal, all the computing nodes will finish almost simultaneously. Also, at each node, only local data are being processed to find all the k-length local frequent itemsets. So, there is no communication overhead imposed on the cluster. The frequent itemsets found local to each node is then sent to the master node. Master node merges all the frequent itemsets from different slave nodes and deletes the duplicates. This union set works as the global candidate itemsets for the second step. Now, the master node broadcasts this union set to all the slave nodes. In the second phase, frequent itemsets that are globally qualified are discovered using the global candidate itemsets broadcasted in the first phase. SARSO also does not use the traditional *apriori_gen*() function. YAFIM uses this function to determine the candidate itemsets before the original dataset is scanned to count their frequency. SARSO rather generates candidates 'on-the-fly' when the transaction is being read from the database. In other words, the generation of candidate itemsets and the counting of their support values go simultaneously. The most interesting feature of the SARSO is that it reduces the shuffle overhead caused by RDD transformations that rearranges the key-value pairs across the cluster nodes using the partitioning method.

4.1 Design

SARSO is an enhanced version of the Spark-based Apriori algorithm. It has three segments to discuss the workflow. **Algorithm$_2$** is the main module that finds all the possible k-frequent itemsets, where k represents the length of the itemset. **Algorithm$_2$** uses **Algorithm$_3$** to find all the local frequent itemsets of all possible lengths at each of the processing nodes in a parallel and independent manner. **Algorithm$_2$** then merges all the locally found frequent itemsets to generate global candidate itemsets. **Algorithm$_4$** helps Algorithm$_2$ to compute the support values for the global candidate itemsets and then filter out the global frequent itemsets.

In **Algorithm$_2$**, input transactional dataset D is loaded from HDFS to a spark RDD named *rdd* (line 1). The variable *count* represents the total number of elements/transactions in *rdd* (line 2). Then the absolute global minimum support *globalMinSup* is calculated using the *count* and user-specified relative minimum support *minSup* (line 3). Variable *globalCI* is an accumulator of ListBuffer type initialized to null (line 4). It collects all the locally found k-frequent itemsets received from slave nodes and treats them as global candidate itemset. The *maxLengthFI* is an integer-type accumulator that records the maximum-length frequent itemset found by any of the partitions (line 5). We use mapPartitions() function with *findFreqItemsetsPP*() function and a Boolean *true* value as parameters. **Algorithm$_3$**, i.e., *findFreqItemsetsPP*(), works on each partition independently. It finds all the frequent local itemsets of all possible lengths iteratively and appends it to the accumulator variable *globalCI* at the master node (line 6). At the end of mapPartitions() function, *globalCI* would be $localFI_1 \cup localFI_2 \cup localFI_3 \cup \ldots\ldots \cup localFI_n$. Then, *globalCI* and *maxLengthFI* are broadcasted to all the slave nodes. *globalCI* is going to be used as global candidate itemsets, and *maxLengthFI* is the length of the maximum-length frequent itemsets found at any node (line 7).

```
Algorithm2: SARSO
Input:    D: Input Dataset
          minSup: relative minimum support threshold
          partition: number of non-overlapping partitions
Output:   RDD of all k-frequent itemsets

     1.   rdd = RDD(D, partition)
     2.   count = rdd.count()
     3.   globalMinSup= minSup*count //absolute global minimum support
     4.   globalCI= null // accumulator of Listbuffer type
     5.   maxLengthFI = 0 //accumulator of integer type
     6.   rdd .mapPartitions( findFreqItemsetsPP ( minSup), true)
     7.   broadcast(globalCI and maxLengthFI)
     8.   globalFI= rdd.mapPartitions(countFrequency(), true)
                    .reduceByKey().filter(_._2 >=globalMinSup).map(_._1)
```

The original input dataset is scanned again, and mapPartitions() function is invoked with *countFrequency*() method and a boolean literal *true* as parameters. Method *countFrequency*(), i.e., **Algorithm$_4$** returns an RDD with <*global_candidate_itemset*, 1> as key-value pair. Partitioning of the resulting RDD produced by mapPartitions() remains intact as the partitions of source RDD. This is because of the *true* value passed to it. In other words, elements within a partition continue to reside on the same partition after applying RDD transformation. Now, reduceByKey() method is called the shuffle locally found key-value pairs and reduces them to merge *values* for the same *keys*. The filter() method is used to filter out only those frequent itemsets that have support count more than *globalMinSup*. Finally, map() function is applied to discard the support value and get an RDD *globalFI* with all the global k-frequent itemsets for $k \geq 1$(line 8).

Algorithm$_3$ generates all the local k-frequent itemsets '*on-the-fly,*' i.e., generation of candidate itemset and a count of their support values go simultaneously. It works on all the partitions in parallel. The *findFreqItemsetsPP()* is applied to each element of a partition one by one. First, we find the size of the partition, i.e., the number of transactions in that partition (line 1). Variable k represents the current iteration number and also the length of the frequent itemset (line 2). Variables *converged* and *finalFI and previousFI* (lines 3–5) are initialized only once ($k=1$). Variable *converged* is *false* until all the k-frequent itemsets are found local to a partition. Variable *previousFI* indicates frequent itemsets generated in the previous iteration, and *finalFI* represents set of all k-frequent itemsets local to a partition. In kth iteration, if the algorithm has not converged in the previous iteration, we initialize the *candidate* by a null (line 6–7). It accumulates all the candidate itemsets for a particular iteration. The flatMap() function is applied to get a particular transaction of that partition (lines

8–9). Then *combinationGenerator*() function is called to generate all *k*-length combinations (itemsets) from the items of each transaction (line 10). These *k*-length combinations may be one among candidate itemsets for *k*th iteration. Then *apriori property* is used to prune unpromising candidates such that a *k*-itemset can be discarded if any of (*k*-1) length subset is not frequent. After the pruning stage, remaining *k*-itemsets are found as potential candidates (lines 11–21). Each candidate itemset is associated with 1, i.e., <*candidate_itemset, 1*> to form a key-value pair. After all the key-value pairs have been generated for all the transactions in a partition, the variable *candidate* contains <*candidate_itemset, 1*> pair for all *k*-candidate itemsets.

Algorithm3: ***findFreqItemsetsPP***

```
Input:   minSup: relative minimum support
Output:  RDD of local k-frequent itemsets and its respective frequency for each partition

 1.  partitionSize= PartitionN.size
 2.  k=1
 3.  Converged=false
 4.  finalFI=null   //ListBuffer
 5.  previousFI=null  //ListBuffer
 6.  while (!Converged)
 7.      candidate=null
 8.      for each transaction t in PartitionN
 9.          flatMap (line offset, t)
10.          Ct = combinationGenerator (k, t)  //generates combinations of k items
11.          for each c in Ct
12.              validCandidate = true
13.              if (k>1)
14.                  H= findSubset (k-1, c)
15.                  for each h in H
16.                      if (h !є previousFI)
17.                          validCandidate= false
18.                          break
19.                  end for each
20.              if (validCandidate)
21.                  candidate=candidate + (c, 1)
22.          end for each
23.          end flatMap
24.      end for each
25.      localFI=candidate.groupBy(_._1)
26.          .mapValues(_.map(_._2).reduce(_+_))
27.              .filter(_._2 >=minSup*partitionSize)
28.      if (localFI.count==0)
29.          Converged = true
30.      else
31.          finalFI += localFI
32.          previousFI = localFI
33.      k++
34.  end while
35.  globalCI +=finalFI
36.  if (maxLengthFI < k-1)
37.      maxLengthFI=k-1
```

The absolute local minimum support is calculated as *partitionSize* × *minSup*. All the *key-value* pairs of *the candidate* RDD are now grouped based on their *keys* using groupBy() function. This produces a RDD of < *candidate_itemset*, iterable < *1,1,1.....*>>. Then mapValues() is used to get the sum of support values for the same *key* to produce an RDD of < *candidate_itemset*, total_support> . This grouping and counting of support values happen within the partition. Therefore, shuffling of key-value pairs across the partitions does not occur (lines 25–26). Then filter() method is invoked to discover the candidate itemsets whose support is equal or more than minimum local support (= *minSup* × partition_*size*). All the discovered candidate itemsets are *k*-frequent itemsets and represeted as *localFI* (line 27). If *localFI* for any *k*th iteration is empty for all the partitions, it means **Algorithm$_3$** has found all the frequent itemsets of all possible lengths. Otherwise, we go for the next iteration by appending *k*-frequent itemsets to *finalFI* that acts as a repository for frequent itemsets of all lengths for a particular partition (lines 28–33). At the convergence of the **Algorithm$_3$**, all locally found frequent itemsets *finalFI* are appended to the accumulator variable *globalCI,* and it also updates *maxLengthFI* conditionally (lines 35–37). The conditional statement ensures *maxLngthFI* to be maximum among all the partitions.

```
Algorithm4: countFrequency
Input:    D: Input Dataset
          maxLengthFI: length of maximal sized frequent itemset
Output:   RDD of <k-global_candidate_itemsets, 1>

1.   for each transaction t in PartitionN
2.       flatMap (line offset, t)
3.       for (j=1, j<=maxLengthFI , j++)
4.           C_t = combinationGenerator (j, t)
5.           for each c in C_t
6.               if (c ϵ globalCI)
7.                   out(c, 1)
8.           end for each
9.       end for
10.      end flatMap
11.  end for each
```

Algorithm$_4$ uses the same methodology as used by **Algorithm$_3$**. It also works partitionwise and iteratively finds the support value of the global candidate itemsets at each partition. It takes every transaction of a partition one by one and generates combinations of different lengths varying from 1 to *maxLengthFI* (line 1–4). If a combination found also belongs to *globalCI* it generates < *candidate_itemset*, 1 > as key-value pair (line 5–7), then **Algorithm$_2$** uses reducebykey(), filter() and map() methods to filter out global frequent itemsets. Figure 1 depicts the workflow of the proposed methodology.

4.2 Discussion

In this section, we analyze the impact of partition and shuffle operation on the execution time of the SARSO in comparison with YAFIM. SARSO uses a partitioning technique to mine frequent itemsets. Figure 1 illustrates the workflow graph for SARSO. It operates in two phases. In Phase-I, it divides the input dataset into *n* distinct partitions. For each partition, all the local candidate itemsets are generated with their respective support count values. Then, the frequent local itemsets (i.e., itemsets frequent within the partition) are searched along with their support counts. If the

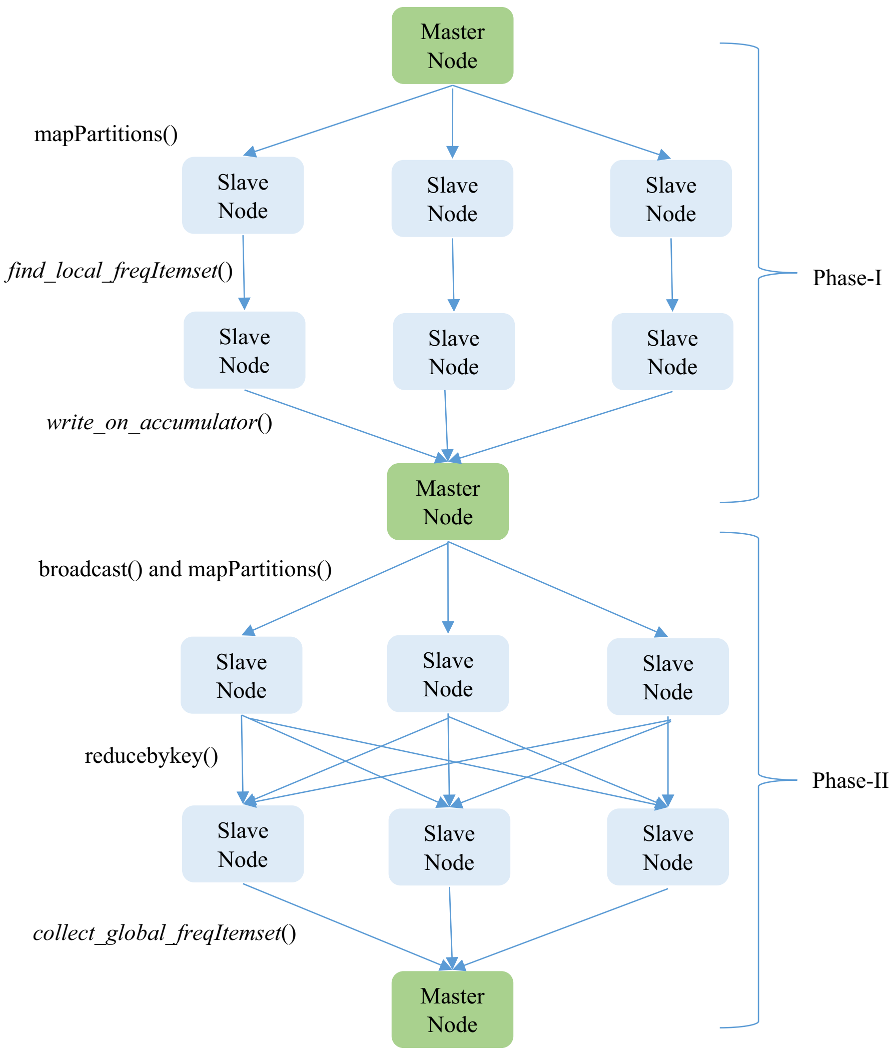

Fig. 1 Workflow of SARSO

minimum relative support is *minSup,* then the absolute minimum support for a partition is *minSup*$\times$total number of transactions in that partition. Local minimum support is set lower than the global minimum support depending upon the number of transactions falling into that partition. Then, we use the concept that frequent local itemset may or may not be globally frequent. However, any itemset that is probable to be globally frequent must occur as a frequent itemset in at least one of the partitions. Therefore, all locally found frequent itemsets from different partitions act as global candidate itemsets for the input dataset as a whole. In Phase-II, all the global candidate itemsets are shuffled across partitions to merge the support count values for the same candidate. Then frequent global itemsets are found by discarding those candidates whose support count value is less than global absolute minimum support. It can be easily observed that SARSO generates surplus candidate itemsets for each iteration in comparison with YAFIM. The time required to filter out the frequent itemsets would be more. This overhead would be overcome by the gain achieved in terms of reduced shuffle overhead.

Instead of using the map(*func*) as in YAFIM, SARSO uses mapPartitions(*func*). The map(*func*) applies the user-specified function (*func)* to every record and returns one processed record as output. Since the records are being processed in parallel, there are multiple instances of the applied function in the system working simultaneously. However, mapPartitions(*func*) returns a new RDD by applying a function (*func)* to each partition of the RDD. We get iterator as an argument for mapPartitions(), through which we can iterate all elements in a partition one by one. Here, user-specified function *func* is initialized on per partition basis rather than per element basis. Therefore, tasks with high per record overhead perform better with mapPartitions() than map() due to the high cost of setting up a new task. Like map(), mapPartitions() have exactly the same partitions as the parent RDD. Also, it can be noticed that SARSO generates the candidates 'on-the-fly,' i.e., for each partition, candidate generation and the count of its support values go simultaneously when the input dataset is being scanned.

What should be the partition size and the optimal number of partitions? RDDs are fault-tolerant datasets that are huge in size and partitioned across the cluster nodes. Spark partitions RDDs automatically and distributes the partitions across different nodes. By default, one partition is created for each block of the file in HDFS. A partition in spark is a logical unit of data. Every partition is stored on a specific node in the cluster. RDD partitions help Spark to achieve parallelism. Every node in a Spark cluster contains one or more partitions, and a single partition cannot span over multiple nodes. The number of partitions in spark is configurable, and having few or too many partitions is not good. If an RDD has too many partitions, then task scheduling may take more time than the actual execution time. In contrast, having very few partitions is also not beneficial as some of the worker nodes may remain idle, resulting in less concurrency. This could lead to improper resource utilization and data skewing, i.e., data might be skewed on a single partition and a specific worker node might be doing more than other worker nodes. Thus, there is always a trade-off when it comes to deciding on the number of partitions. In the following section, we analyze the effect of partition number on the run time of the SARSO algorithm.

5 Performance evaluation

In this section, the performance of SARSO is evaluated in comparison with YAFIM. YAFIM is the earliest proposed adaptation of the Apriori algorithm in the Apache Spark environment. To analyze both the algorithms over Spark, we set up a cluster of four nodes. All the four nodes were having Xeon(R) CPU E3-1225 v5 clocked at 3.30 GHz with four computing cores each. Every node of the cluster was deployed with the same memory configurations, i.e., 16 GB of RAM and 2 TB of the hard disk. The computing nodes are installed with Ubuntu 16.04, Hadoop 2.6.0, Spark 1.6.0, JDK 1.8.0, and Scala 2.11.8. Input datasets and frequent output patterns were stored on the HDFS.

5.1 Datasets

Some benchmark datasets are used in our experiments to evaluate the performance of SARSO and YAFIM. All the experiments were performed three times, and the average is recorded as the final result. Experiments were conducted with six benchmark datasets with different characteristics. The datasets used for the analysis include mushroom dataset, chess dataset, retail dataset, and synthetic datasets T1014D100k, T1012D100k, and T2014D100k generated by IBM's random transactional data generator. Detailed information about the above-mentioned datasets can be found on the machine learning repository [32–34]. Some of their characteristics are listed here for convenience in Table 1.

5.2 Execution time analysis

The performance of SARSO and YAFIM was evaluated by conducting exhaustive experiments with the above-mentioned datasets. We measure gains in terms of speedup and scalability of SARSO in comparison with YAFIM. The number of nodes in the cluster is kept constant for all the experiments. Run time of both the algorithms with different datasets for decreasing values of minimum support is calculated to evaluate the speed-up measures. The speed-up performance of both the algorithms is illustrated in Fig. 2. The number of partitions of the input dataset

Table 1 Characteristics of the datasets used for experimental analysis

Datasets	No. of transactions	No. of different Items
Mushroom	8124	119
Chess	3196	75
T1014D100K	100,000	870
Retail	87,988	16,470
T1012D100K	100,000	1000
T2014D100K	100,000	1000

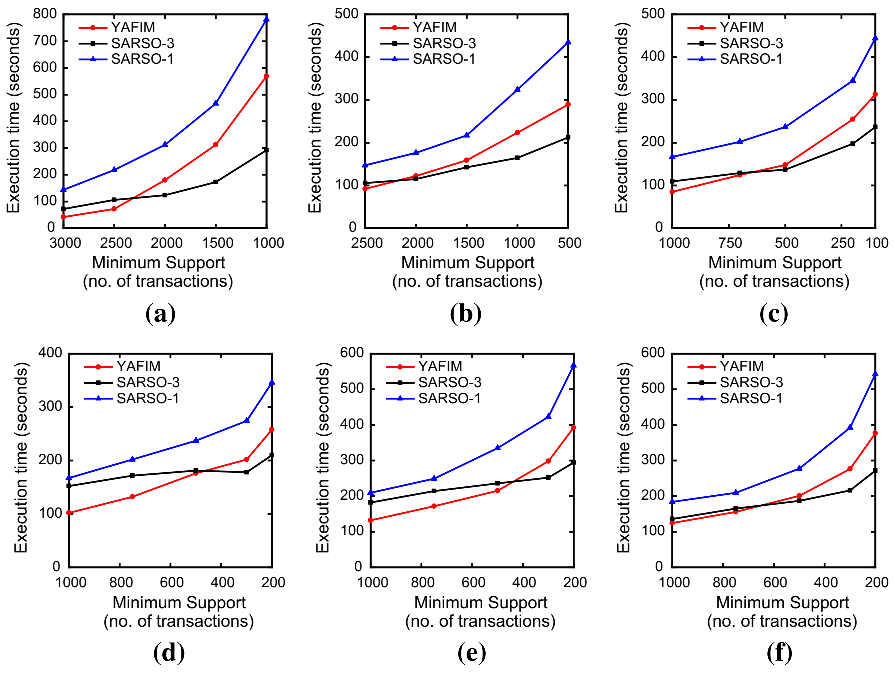

Fig. 2 Execution time taken by YAFIM and SARSO with 1 partition and 3 partitions. *X*-axis shows minimum support values, and *Y*-axis indicates execution time in seconds. **a** Chess, **b** mushroom, **c** retail, **d** T10I4D100K, **e** T10I2D100K, **f** T20I4D100K

is taken as 1 for SARSO-1 and 3 for SARSO-3 in all the experiments. As we can notice that the execution time increases for all the algorithms as the minimum support is reduced. This is because of the increase in the total number of frequent and candidate itemsets generated. We can observe that both YAFIM and SARSO-3 perform better than SARSO-1 in all cases. The reason is that SARSO-1 reduces the communication overhead to the greatest extent, but the parallel processing benefits are lost. As expected SARSO-3 performs better than YAFIM for low minimum support values. For higher values of minimum support, YAFIM exhibits better performance over SARSO-3. The reason is the overhead incurred in setting up the accumulator variables and merging the accumulated values to drop duplicates. All the locally found frequent itemsets are collected and merged on the master node that is going to work as global candidate itemsets. Also, a large number of itemsets were found locally frequent, but very few of them emerged out as globally frequent. So the partitioning method does not benefit much due to the setting up of new data structures and generation of surplus candidate itemsets. As an instance, we can observe that SARSO-3 performed inadequate than YAFIM for the T10I4D100K dataset at a higher support value of 1000 transactions (102 s for YAFIM and 152 s for SARSO-3).

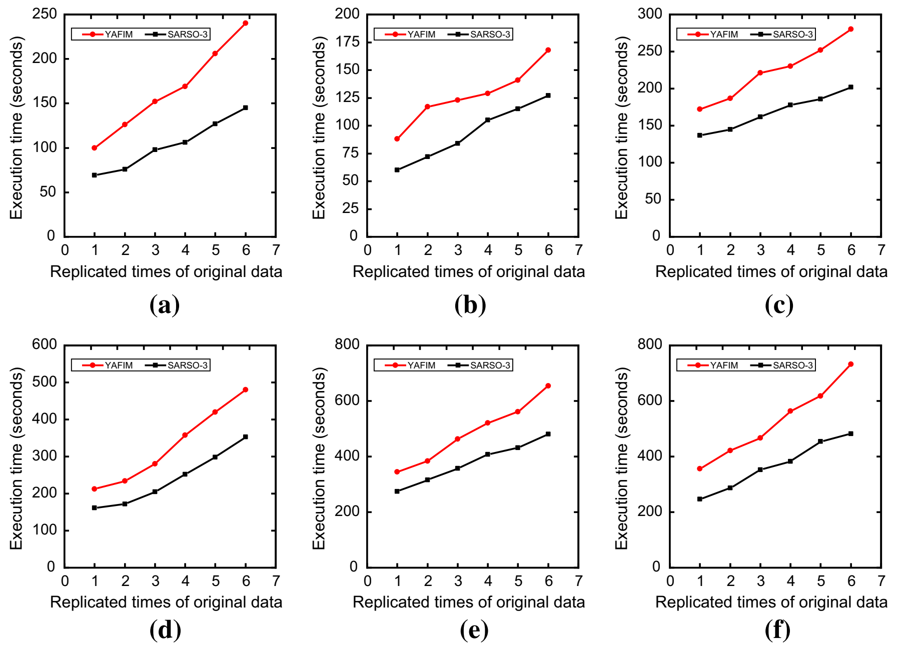

Fig. 3 Scalability performance evaluation for the experimental datasets with 3 partitions. *X*-axis represents replicated times of the original dataset, and *Y*-axis represents execution time in seconds. **a** Chess: Min. Sup. = 90%, **b** mushroom: Min. Sup. = 30% **c** Retail: Min. Sup. = 0.18%, **d** T1014D100K: Min. Sup. = 0.20%, **e** T1012D100K: Min. Sup. = 0.20%, **f** T2014D100K: Min. Sup. = 0.25%

At the lower minimum support values, the least improvement recorded was for the T1014D100K dataset (210 s for SARSO-3 and 258 s for YAFIM at a minimum support of 200), and highest improvement recorded was for the chess dataset (292 s for SARSO-3 and 568 s for YAFIM at a minimum support of 1000). The main reason for this improvement for low support values is due to the restriction of shuffling of key-value pairs among the cluster nodes, hence reducing the communication and synchronization overhead to a great extent.

5.3 Scalability performance analysis

The scalability performance of SARSO is measured by replicating the dataset to enlarge its size. We replicate the datasets to 2, 3, 4, 5, and 6 times to magnify the data size and measure run time performance. Since SARSO-1 does not perform better than YAFIM and SARSO-3 in any of the cases, it is not included in this experiment. As shown in Fig. 3, both the algorithms show nearly the same nature, i.e., when the size of the datasets increases, the execution time grows slowly. Since the shuffling overhead of key-value pairs among the cluster nodes has been reduced to a great extent, we can observe that SARSO-3 grows flatter than YAFIM.

6 Conclusion and future work

Apriori algorithm is one of the popular algorithms to discover frequent itemsets from transactional datasets. It further acts as a primary step to find association rules. It suffers from several drawbacks like repeated scanning of the input dataset, generation of a huge number of candidates in each iteration, etc. It does not suit well to large-scale datasets because of the additional requirement of high computation power to process large datasets efficiently. Large primary memory is also desirable to keep large input datasets ready for faster execution. Many single machine versions of the Apriori algorithm have been proposed to improve efficiency. But if the input dataset is large, it demands a parallel and distributed computing environment. A variety of algorithms based on apriori are also proposed to improve efficiency further using the MapReduce framework. However, MapReduce-based implementations involve iterative computation and impose high disk usage that makes them less significant for large datasets. Spark versions of Apriori show performance enhancement due to its in-memory processing capabilities. But, very little attention has been paid toward the overhead caused due to Spark's RDD operations. Shuffling of key-value pairs across the cluster nodes during RDD operations consume the cluster bandwidth and subsequently makes the algorithm slower. In this paper, we proposed an efficient two-phase partition-based approach named SARSO to mine frequent itemsets from large-scale datasets. It utilizes the Spark environment efficiently to reduce shuffle overhead, keeping Apriori as the base algorithm. It improves the efficiency further by reducing the shuffle overhead caused by RDD operations. Here, the Partition method is used to reduce the necessary shuffle overhead incurred by the Spark framework. The performance of SARSO is evaluated in terms of efficiency and scalability for different well-known datasets. The experimental results show that it exhibits better performance in comparison with YAFIM for lower minimum support values. This method can be further extended to generate association rules and to find high utility itemsets for large-scale transactional datasets.

Acknowledgments This work is partially funded by IIT(ISM), Govt. of India, Dhanbad. The authors would like to express their gratitude and heartiest thanks to the Department of Computer Science & Engineering, Indian Institute of Technology (ISM), Dhanbad, India, for providing their research support.

References

1. Aggarwal CC (2015) Data mining: the textbook. Springer, Berlin
2. Aggarwal CC, Bhuiyan MA, Al Hasan M (2014) Frequent pattern mining algorithms: a survey. In: Frequent pattern mining. Springer, Cham, pp. 19–64
3. Han J, Pei J, Kamber M (2011) Data mining: concepts and techniques. Elsevier, Amsterdam
4. Wu X, Zhu X, Wu GQ, Ding W (2013) Data mining with big data. IEEE Trans Knowl Data Eng 26(1):97–107
5. Fan W, Bifet A (2013) Mining big data: current status, and forecast to the future. ACM sIGKDD Explor Newsl 14(2):1–5

6. Che D, Safran M, Peng Z (2013, April) From big data to big data mining: challenges, issues, and opportunities. In: International Conference on Database Systems for Advanced Applications. Springer, Berlin, pp 1–15
7. Sagiroglu S, Sinanc D (2013, May) Big data: a review. In: 2013 International Conference on Collaboration Technologies and Systems (CTS). IEEE, pp 42–47
8. Agrawal R, Srikant R (1994, September) Fast algorithms for mining association rules. In: Proceedings of the 20th International Conference Very Large Data Bases, VLDB, Vol 1215. pp 487–499
9. Goswami DN, Anshu C, Raghuvanshi CS (2010) An algorithm for frequent pattern mining based on apriori. Int J Comput Sci Eng 2(04):942–947
10. Borgelt C (2003, November) Efficient implementations of apriori and eclat. In: FIMI'03: proceedings of the IEEE ICDM workshop on frequent itemset mining implementations
11. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. ACM sigmod record 29(2):1–12
12. Savasere A, Omiecinski ER, Navathe SB (1995) An efficient algorithm for mining association rules in large databases. Georgia Institute of Technology, Atlanta
13. Lin MY, Lee PY, Hsueh SC (2012, February) Apriori-based frequent itemset mining algorithms on MapReduce. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication. pp 1–8
14. Li N, Zeng L, He Q, Shi Z (2012, August) Parallel implementation of apriori algorithm based on mapreduce. In: 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. IEEE, pp 236–241
15. Yang XY, Liu Z, Fu Y (2010, June) MapReduce as a programming model for association rules algorithm on Hadoop. In: The 3rd International Conference on Information Sciences and Interaction Sciences. IEEE, pp 99–102
16. Lin X (2014, June) Mr-apriori: Association rules algorithm based on mapreduce. In: 2014 IEEE 5th international conference on software engineering and service science. IEEE, pp 141–144
17. Yahya O, Hegazy O, Ezat E (2012) An efficient implementation of Apriori algorithm based on Hadoop-Mapreduce model. Int J Rev Comput 12
18. Apache hadoop (2013). https://hadoop.apache.org/. Accessed Mar 2019
19. Apache Spark: Lightning-fast cluster computing. (2016) The Apache Software Foundation. Spark1.6.0. https://spark.apache.org/. Accessed Mar 2019
20. Karau H, Konwinski A, Wendell P, Zaharia M (2015) Learning spark: lightning-fast big data analysis. O'Reilly Media Inc., Champaign
21. Lin MY, Lee PY, Hsueh SC (2012, February) Apriori-based frequent itemset mining algorithms on MapReduce. In: Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication. pp 1–8
22. Moens S, Aksehirli E, Goethals B (2013, October) Frequent itemset mining for big data. In: 2013 IEEE International Conference on Big Data. IEEE, pp 111–118
23. Hammoud S (2011) MapReduce network enabled algorithms for classification based on association rules (Doctoral dissertation, Brunel University School of Engineering and Design PhD Theses)
24. Thabtah F, Hammoud S (2013) Mr-arm: a map-reduce association rule mining framework. Parallel process lett 23(03):1350012
25. Yu KM, Zhou J, Hong TP, Zhou JL (2010) A load-balanced distributed parallel mining algorithm. Expert Syst Appl 37(3):2459–2464
26. Aouad LM, Le-Khac NA, Kechadi TM (2010) Performance study of distributed apriori-like frequent itemsets mining. Knowl Inf Syst 23(1):55–72
27. Chen Z, Cai S, Song Q, Zhu C (2011, August) An improved Apriori algorithm based on pruning optimization and transaction reduction. In: 2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC). IEEE, pp 1908–1911
28. Zhang F, Liu M, Gui F, Shen W, Shami A, Ma Y (2015) A distributed frequent itemset mining algorithm using Spark for Big Data analytics. Clust Comput 18(4):1493–1501
29. Qiu H, Gu R, Yuan C, Huang Y (2014, May) Yafim: a parallel frequent itemset mining algorithm with spark. In: 2014 IEEE international parallel & distributed processing symposium workshops. IEEE, pp 1664–1671
30. Rathee S, Kaul M, Kashyap A (2015, October). R-Apriori: an efficient apriori based algorithm on spark. In: Proceedings of the 8th workshop on Ph. D. Workshop in information and knowledge management. pp 27–34

31. Sethi KK, Ramesh D (2017) HFIM: a Spark-based hybrid frequent itemset mining algorithm for big data processing. J Supercomput 73(8):3652–3668
32. RDD Programming Guide (2019, January). https://spark.apache.org/docs/latest/rdd-programming-guide.html
33. IBM's synthetic datasets generated by IBM's Quest dataset generator (2019, January). https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php
34. Datasets for chess, mushroom and retail (2019, January). https://fimi.ua.ac.be/data/

Affiliations

Shashi Raj[1] · **Dharavath Ramesh**[2] · **Krishan Kumar Sethi**[2]

✉ Dharavath Ramesh
ramesh.d.in@ieee.org

Shashi Raj
shashirajmnnit@gmail.com

Krishan Kumar Sethi
kksethi@ieee.org

[1] Department of Computer Science and Engineering, Bakhtiyarpur College of Engineering, Patliputra, Patna, Bihar 800013, India

[2] Department of Computer Science and Engineering, Indian Institute of Technology (ISM), Dhanbad, Jharkhand 826004, India