Received: May 7, 2019

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

An Efficient Distributed Frequent Itemset Mining Algorithm Based on Spark for Big Data

Yassir Rochd[1]* **Imad Hafidi[1]**

[1]*Laboratory of Process Engineering and Optimization of Industrial Systems, National School of Applied Science, Moulay Sultan Slimane University, Khouribga, Morocco*
* Corresponding author's Email: y.rochd@gmail.com

Abstract: Frequent item exploration is a fundamental element in many data mining problems aimed at finding interesting models in the data. Recently, the PrePost algorithm, a new algorithm for extraction frequent element sets based on the idea of N-lists, which in most cases surpasses other current state-of-the-art algorithms, has been introduced. The PrePost algorithm's performance deteriorates when it comes to handling big data. Nevertheless, the current existing PrePost algorithms in place implemented with the MapReduce model are not sufficiently powerful for iterative computation. To reduce IO overhead and take advantage of cluster memory, this article offers an enhanced version of PrePost, the Distributed PrePost (DisPrePost), a parallel algorithm built on the Spark framework, which incorporates the concept of resilient distributed datasets and performs in-memory processing to optimize the execution time of operation, that also utilises a HashMap to further refine the N-list creation process. Experience has shown that the DisPrePost algorithm is more efficient and scalable than the two advanced state-of-the-art methods HPrePostPlus and the well-known algorithm HFIM.

Keywords: Frequent itemset mining, PrePost, Spark, Big data.

1. Introduction

In the late years, the great evolution of technology and science has strongly affected the growth of data. In addition to its large size, these enormous datasets are also of an unstructured and semi-structured natures, including the challenges of capture, storage, sharing, research, analysis, etc.; they have been named "Big Data"[1]. The Data Mining, being the process used to evaluate great amounts of data stored in businesses, helps to facilitate making decisions. In addition, it helps to find hidden information in the database that demands specific capabilities. Classification, association mining rule, clustering, sequential pattern discovery, etc., are several required tasks in Data Mining [2].

By association mining rule, we mean a rule-based learning technique, discovering significant relationships between data object in dataset. Thus, only frequent itemsets are considered to build the association rules for the purpose of reducing the number of potential itemsets, and in order to extract frequent itemsets from data, various algorithms along with numerous improvements were proposed.

Agrawal, first, proposed mining customer transaction database itemsets problem [3], now FIM (frequent itemsets mining) has become an essential part of data mining. Most of the current algorithms can be grouped into two categories: Apriori-like algorithm and FP-growth algorithm. Repeatedly, the Apriori scans the database to find frequent itemsets with generating a large set of candidates [4]. FP-Growth algorithm scans database twice to mine frequent itemsets without generating candidates [5].

The FP-growth uses FP tree data structure to store database and employs a divide-and-conquer strategy to find frequent itemsets, which is much more efficient than Apriori method.

In recent years, the PrePost [6] and PrePost+ [7] algorithms based on the N-list data structure have been proposed to shorten mining time and memory utilization with frequent itemsets. These algorithms, working on single computers, have demonstrated high performance in processing small data. However, conventional methods encounter important challenges when computing power and memory space are constrained in big data era. There have been certain practices and attempts to mine frequent itemsets from massive data using parallel computing technologies [8, 9].

Parallel programming frameworks are split into two groups: memory sharing and distributed architectures (share nothing). Despite the fact that it is more convenient to make algorithms implemented in parallel on the memory sharing framework, their scalability is not sufficient. The Message Transmission Interface (MPI) [10], which is a common framework for scientific distributed computing, benefits from the locality of the memory. Due to some advantages of the MPI in iterative calculation, some researches apply it to mine frequent itemset [11]. Yet its drawbacks are its heavy workload of communication due to data interchange between various computer nodes and the absence of fault tolerance.

MapReduce [12], a framework embedded in Apache Hadoop for parallel processing of high volumes of distributed data, has been designed to work with distributed computing in a cloud computing model, proving to be a powerful and reliable platform for the parallel data mining of large datasets. However, MapReduce is not suitable for iterative algorithms due to high network and I/O overhead in writing intermediate output to disk and read data back from disk [13]. Moreover, the MapReduce framework follows a strict pre-defined execution order of mapper and reducer stage, which limits the flexibility of the algorithm.

To surmount the above problem, the Spark platform [14], a memory-based distributed framework, has been used as solution architecture in this paper. Spark is a better alternative framework, which is more efficient in batch and interactive processing, and assures high performance over MapReduce. Spark programming interface is based on a data structure called Resilient Distributed Dataset (RDD), a read only collection of data objects distributed across nodes of cluster. Many advanced features of Spark, i.e., in-memory processing, job caching, improved fault tolerance mechanism, etc., optimize the job execution.

A number of distributed frequent itemset mining methods [15] have been proposed, which are relatively simple extensions of a sequential method using distributed data processing frameworks.

Although the existing distributed methods solve the limit on scalability partially, most of them still have the following problems in terms of scalability. First, they do not have good scalability due to workload skewness. The existing parallel methods usually split the search space for patterns to be investigated into multiple blocks and allocate them to a single machine (or processor). Each sub-tree of the enumeration tree has a distinct workload size. In particular, the distributed methods based on Eclat and FP-Growth have this problem noticeably.

The other point is that the multiple database scans necessitate multiple MapReduce jobs: Each MapReduce job also needs to read sequence databases or projected databases. When the output of each MapReduce job and databases are stored in the HDFS, it leads to a high I/O overload. Hence, minimizing input-output overhead expenses becomes a vital aspect of algorithmic design.

Consequently, the existing parallel methods are not particularly scalable in terms of to the number of machines. In the same way, their effectiveness does not expand in proportion to the rise in the number of machines or processors.

In addition to that, they do not have good scalability due to high network communication overhead. The existing methods usually perform frequent itemsets mining by redistributing intermediate data via network. This approach could largely degrade the performance and scalability as the amount of data transferred among machines increases.

In this paper, we introduce an enhanced version of PrePost, the Distributed PrePost (DisPrePost), a parallel algorithm based on the RDD Spark framework. DisPrePost solves the above problems, and so can find frequent patterns on much larger datasets compared with the existing distributed methods.

Contrary to FP tree-based approaches, DisPrePost does not construct any extra tree at each iteration; it extracts frequent itemsets directly using the N-list concept. The efficiency of DisPrePost is achieved because: (i) Since the N lists are considerably more compact than the earlier suggested vertical structures, (ii) the support of a candidate frequent itemset can be determined through N-list intersection. Determining the intersection of the TID lists is more cost-effective than in the case of the TID lists since it has unnecessary comparisons.

To reduce IO overhead and take advantage of cluster memory, the first MapReduce job loads the sequence database from the HDFS into the Spark RDDs, and subsequent MapReduce jobs read the database from the RDDs and store intermediate results back into the RDDs.

For solving the problem of network communication overhead, DisPrePost broadcasts only frequent itemsets F_k, whose size is much smaller than that of intermediate data via network, we use a special feature of the Spark framework called the broadcast variable. As a result, DisPrePost shows much higher performance than the state-of-the-art approaches.

The main contributions of this paper are the following:

i. We propose an algorithm has been implemented over Apache Spark, a fast and general engine for large scale data processing, which could provide a solution for big data analytics.
ii. We propose DisPrePost, a scalable Spark-based method for frequent itemset mining, that has low IO overhead by adopting in-memory computation technique with the help of RDD storage.
iii. We propose an algorithm which has small network communication because we only broadcast frequent itemsets F_k, whose size is much smaller than that of intermediate data via network by using a special feature of Spark framework called broadcast variable.
iv. We use HashMap to traverse efficiently the PPC tree and to speed up the process of creating the N-lists associated with frequent 1-itemsets.

Experiments show that DisPrePost outperforms the state-of-the-art MapReduce-based methods in terms of speed and scalability.

The rest of the paper is organized as follows: Section 2 outlines survey of related works. Section 3 presents the basic concepts. Section 4 gives proposed approach. Then, section 5 gives results and discussion. Finally, section 6 provides the conclusion.

2. Related work

In the context of frequent pattern mining and association rules, numerous studies have been carried out. As a result, a broad spectrum of knowledge discovery techniques has been debated and are now being explored for big data.

Lin et al [16] developed three Apriori methods that were distributed on MapReduce: SPC, FPC and DPC. The SPC performs iteratively the steps of generating and checking applicants as a MapReduce cycle. In the k-th iteration, each mapper reads a partitioned database, generates the candidate itemsets and calculates support counts of them for the partitioned database. Afterwards, the reduce phase aggregates and tests the support counts of the same candidate itemset with regard to minsup. The outcome of the reduce step is broadcasted for use in the next iteration. FPC reduces the number of MapReduce rounds through the use of the map function that handles candidate k-itemsets, (k+1)-itemsets, (k+2)-itemsets together in a single MapReduce round. DPC automatically collects candidate itemsets to be handled by the mappers in a single MapReduce round depending on the number of candidates itemsets. By comparing these Apriori-based methods, DisPrePost performs support counting much faster from the intersection of N-lists, avoiding needless comparisons. The PApriori algorithm proposed by Li et al [17], is very similar to the SPC algorithm. The map function performs the procedure of counting each occurrence of all the candidates in a parallel way and then the reduce function sums up the occurrence. The PFP [18] and its variations [19] are the distributed methods based on the FP-Growth approach. Initially, they project an input database and construct separate FP trees, which are basically conditional databases, through the use of the projected databases. Then, they perform frequent itemset mining on each FP Tree independently in each machine. In summary, the main principle of PFP is to group the items and then distribute the conditional databases to the mappers, which is not efficient in memory or speed.

Moens et al [20] proposed BigFIM, a hybrid approach between Apriori and Eclat. It starts by finding frequent itemsets of short lengths using the distributed algorithm of the Apriori approach and generates conditional databases. Then, he executes the sequential Eclat algorithm [21, 22] on each conditional database separately. Its support counting is fast by using an efficient sequential Eclat algorithm.

Nevertheless, since the sizes of conditional databases are quite different with each other, i.e., there is workload skewness, mining task tends to fail due to lack of memory in a certain machine, or takes too long time due to the machine having the largest workload. Moreover, it generates a large amount of intermediate data and incurs large network communication surcharge when generating conditional databases. Therefore, BigFIM tends to show bad scalability as the number of machines increases.

Likewise with BigFIM, PFP and its variations have a number of drawbacks, in particular workload asymmetry, large intermediate data size and large network communication overhead. Thus, they tend to fail due to lack of memory, and show bad scalability. Compared to BigFIM and PFP, DisPrePost shows much better scalability with an increasing number of machines, since it does not intermediate data, and small network overhead.

MRPrePost [23] is a parallel program based on the Hadoop platform, which improves PrePost by inserting a prefix pattern, as well as on this basis in parallel design ideas, so that the MRPrePost program can be adapted to mining large data's association rules. Comparing to the precedent versions of PrePost based on Hadoop [24, 25], general tree method is utilized to traverse the tree PPC tree. The general tree method utilized linked list which is an implementation of the List interface.

It provides sequential access and effective for inserting and deleting items in the list. But, it became less efficient while accessing items in the list. In DisPrePost algorithm, general tree method is implemented with HashMap which is an implementation of the Map interface. It provides an efficient and fast for locating value based on the key. It does not save the item in the order and it provides an easy way to access and delete items on the basis of key value pairs. The DisPrePost algorithm uses also a HashMap to improve the process of creating the N-lists associated with 1-itemsets and combines the features of Hadoop in order to process large data.

In 2018, we presented, HPrePostPlus algorithm [26], a better version of PrePost, based on Hadoop, which uses a HashMap to traverse efficiently through the PPC tree and enhance the N-list creation process. The HPrePostPlus algorithm is very powerful and surpasses the state-of-the-art algorithms, such as PrePost [6], MRPrePost [23], PFP [18], and negFIN [27]. Although N-list are effective structures for mining frequent itemsets, hey need to contain pre-order and post-order number, which is memory-consuming and inconvenient to mine frequent itemsets.

Even though, Hadoop is not suitable for iterative algorithms as it saves intermediate data to HDFS and reads it back, which takes high I/O cost. Spark is a cluster computing framework, which deals with iterative algorithms in an efficient way by using its RDD architecture. RDDs store the results in the main memory at the end of iteration while making them available for the next iteration to faster the execution. To solve this issue, more efficient approaches are proposed by implementing Apriori in Spark platform. In 2014, Qiu [28] had already reported accelerations of more than 18 times on average for different benchmarks for the YAFIM algorithm (yet another frequent item exploration set) for the RDD Spark framework. Their results on real medical data are much faster than on the MapReduce framework. 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.

HFIM algorithm [29] is another Spark-based implementation of the Apriori algorithm for various data sets, which uses the vertical layout of the data set to solve the problem of scanning the dataset in each iteration. It is implemented on the Spark framework, integrating the concept of resilient distributed datasets and in-memory processing to optimize the processing time of the operation. These results motivated us to come with innovative approaches for distributed association rule mining algorithms. Afterwards, Zhang [30] proposed an association rule mining algorithm called DFIMA (Distributed Frequent Itemset Mining Algorithm) that is being implemented on Spark. A matrix size technique is used in the DFIMA algorithm to reduce the size of candidates. The author claims that it outperforms PFP algorithm when both are implemented on Spark.

A new efficient algorithm named R-Apriori [31] is proposed by Rathee, to solve the second phase when the number of candidate pairs is much. Rathee continued improving his idea. In 2018, he developed a new algorithm named Adaptive-Miner [32] which is one of the best state-of-the-art frequent itemset mining algorithms. It uses an adaptive method for extracting frequent itemsets with higher accuracy and efficiency. Based on the nature of dataset, Adaptive-Miner dynamically adapts execution for every iteration. It reduces time and space by selecting the best plan. R-Apriori and Adaptive-Miner use bloom filters, which are faster than hash trees that make these algorithms more efficient with respect to time and space [31, 32].

Although Rathee's algorithms are efficient, they have to visit the dataset in every iteration to filter the frequent items. They apply intersection every transaction with frequent itemsets that stored in the bloom filter, so this process takes extra time and space.

The existing methods in Spark are popular parallel recommendation methods, but getting the best performance only when the memory of machines can accommodate all immediate Resilient Distributed DataSets (RDDs). However, memory of many practice data centers, is still not large enough for large data sets. Therefore, in this paper, a caching-based DisPrePost algorithm is proposed which consists of an RDD-caching strategy to improve the efficiency.

3. Preliminaries

3.1 Frequent itemset mining

Suppose that I = {I_1, I_2, . . . , I_m} is an itemset composed of m items. A database D consists of a series of transactions. Each transaction is a subset of I and has a unique label denoted by TID. A set of items is referred to as an itemset. An itemset that contains k items is a k-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. Given an itemset X, the support number of X is the number of transactions in D that contain X. If the support number of X is greater than or equal to the specified minimum support threshold, then the itemset X is labelled as a frequent itemset. The purpose of frequent itemset mining is to find all frequent itemset in a given database.

3.2 Spark

Spark is defined as a distributed computing framework developed at the Berkeley AMPLab [33] which offers a certain quantity of features in order to make big data processing fast. The main key feature is its in-memory parallel execution model which memory contains all loaded data. This first key principally benefits the iterative computations. The second feature considers the fact that Spark may offer very scalable DAG-based (directed acyclic graph) data flow in deferring from the nominated two-stage data flow model in MapReduce. Both of these features may determinately rush the computation for those iterative algorithms just like the Apriori algorithm and other few machine learning algorithms. That way, Spark achieves 1-2 orders of magnitude in a way faster speed than MapReduce.

Spark's programming model is working with a new distributed memory abstraction applied on large cluster for in-memory computations named resilient distributed datasets (RDDs). An RDD which is an immutable collection of data records has the ability to offer a variety of built-in operations in order to change one RDD into another RDD. It's on the worker nodes where Spark caches the contents of the RDDs, and that's what makes data reuse way faster. The fault-tolerance that is based on lineage information may be achieved by RDDs rather than replication. If a node fails, Spark tracks enough information in order to reconstruct RDDs. Fig. 1 illustrates the working model of Spark framework.

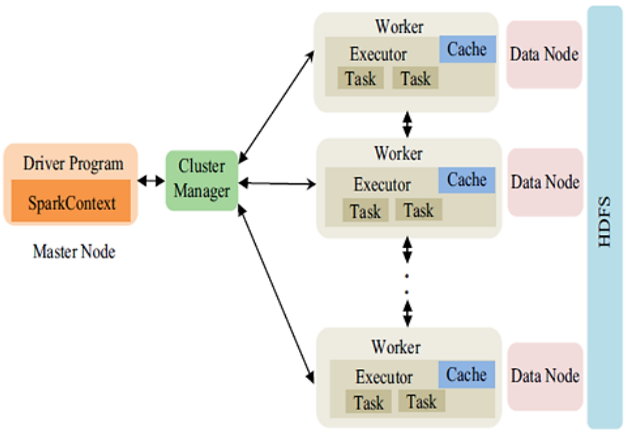


Figure 1. Working method of Spark framework

3.3 PrePost algorithm

PrePost algorithm [8, 9] presents a data structure named N-list, which is a modification of the vertical database, storing the association rule mining all the information needed. PrePost also need to scan the database twice to construct a PPC-Tree, and make use of PPC-Tree to generate the N-list of frequent 1-itemsets (FIM1). In the mining process, the database does not require rescanning, only need to intersect the merger N-list, and the complexity of the algorithm is O(m+n), m and n are the length of two N-list. Each element of N-list composed by PrePost Code, which is called after the sequence encoding the preamble, the composition in the form of «pre-order, post-order: count», PrePost Code is based on the PPC-Tree respectively from the previous order traversal and post order traversal. Fig. 1 shows the PPC-Tree, which is similar to FP-Tree, and the construction process is the same with the FP-Tree but not the same as the composition of the node, PPC Tree node consists of five components:
1. Item-name: represent node name
2. Count: represent node count
3. Children-list: represent a children collection of the node
4. Pre-order: represent order of node when pre-order
5. Post-order: represent order of node when post-order.

Each k-frequent itemsets F_k corresponds to a N-list, which in ascending order according to the pre-order, at the same time must also be ascending according to post-order. PPC-Tree's main purpose is to construct N-list liking shown by Fig. 2, then find all the frequent itemsets based on N-list. We can then delete the PPC-Tree to reduce memory overhead. The main steps of the PrePost algorithm:

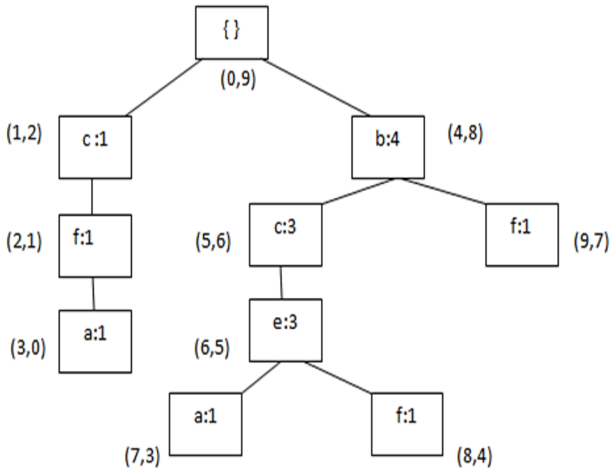


Figure. 1 PPC-Tree corresponding with Table 1

Table 1. Transaction database

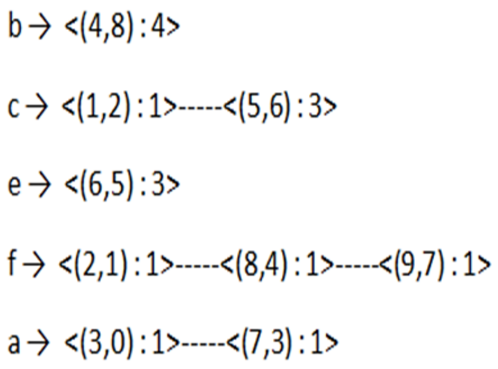
ID	Items	Ordered frequent items
1	a, c, g, f	c ,f ,a
2	e, a, c, b	b ,c ,e ,a
3	e, c, b, i	b ,c ,e
4	b ,f, h	b ,f
5	b, f, e, c,d	b ,c ,e ,f

1. Scan transaction database named D, output the FIM 1, and in descending order according to the number of its support to generate F_1.
2. Scan D again, select the frequent items in each record and arrange them in the order of F_1, assuming list of items in each record is [p|P], p is the first item in the list, P is the rest of the items. Call the function insert tree ([p|P], T_i).
3. Tree formed on the second step, respectively pre-order traversal and post-order traversal, set pre-order and post-order of each node and establish N-list of I-frequent itemsets.
4. Mining frequent itemsets based on N-list using the method liking Apriori Algorithm.
5. Table 1 shows a transaction database, corresponding to Fig. 1 for PPC-Tree, assuming the minimum support is 3.

4. DisPrePost algorithm

4.1 DisPrePost design

DisPrePost algorithm is a data mining algorithm for frequent itemsets which uses N-list data structure to represent the itemsets. All the required information of the itemsets is to be stored by N-list.


b→ <(4,8):4>

c→ <(1,2):1>-----<(5,6):3>

e→ <(6,5):3>

f→ <(2,1):1>-----<(8,4):1>-----<(9,7):1>

a→ <(3,0):1>-----<(7,3):1>

Figure. 2 N-list of frequent 1-itemsets

Efficiency of the DisPrePost algorithm is achieved by using the method of generating frequent itemsets without generation of candidate itemsets.

The DisPrePost algorithm is implemented with Spark to improve its performance. We store the big transactional data in Hadoop distributed file system (HDFS) of Hadoop framework, and multiple partitions of data are distributed across cluster nodes. Job execution on data partitions takes place in parallel by Spark engine.

In the DisPrePost algorithm, general tree method is used to traverse the PPC tree. The general tree method used the same technique that is used by linked based binary tree that uses linked list which is an implementation of the List interface. It provides sequential access and more efficient for inserting and deleting items in the list. But, it became less efficient while accessing items in the list. In DisPrePost, general tree method is implemented with HashMap which is an implementation of the Map interface to speed up the process of creating the N-lists associated with frequent 1-itemsets from the PPC tree. It provides an efficient for locating value based on the key. It does not store the item in the order and it provides an easy way to access and delete items on the basis of key value pairs.

Algorithm of parallel statistical 1-frequent itemsets and sort them

Input: D: Transactional Dataset, minsup= Minimum Support Threshold, I = item
Output: FL_1: RDD of the set of frequent l -itemsets by descending order

```
1. for each transaction t in D
2. flatMap(Id,t)
3. for each item I in t
4. Output (I,1)
5. End foreach
6. End flaMap
7. End foreach
8. reduceByKey(I,count)
9. Sum=0
```

```
10. While(Item I in partition)
11. Sum+=count
12. End While
13. If (sum>=minsup)
14. Output(Fim1)
15. End if
16. Sort(Fim1)
17. Output (FL_1)
```

Figure. 4 Pseudo code of parallel statistical 1-frequent itemsets and sort them

Phase1**:** As we are dealing with big data, dataset may have a large number of transactions. We store the big transactional data in Hadoop distributed file system (HDFS) of Hadoop framework, and multiple partitions of data are distributed across cluster nodes. Job execution on data partitions takes place in parallel by Spark engine. A set of RDDs is created and processed to produce the set of frequent l-iternsets by descending order: FL1. The transactional data are loaded into RDD, which makes better use of cluster memory and improves fault tolerance. All the items from dataset are generated by using flatMap () function. Then, map() function is applied on each item to produce a (key, value) pair, where key is the item and value is one. The phase 1 generates only singleton items. Pruning step is applied on itemsets to filter out the non-frequent items. The complete RDD of paired itemsets is grouped using Reducebykey() function and pruned using the filter() function .As a result, only frequent items ranked in descending order of support are generated at the end of Phase 1. The Pseudo-code for the complete process of phase 1 is presented in Fig. 4.

Phase 2: All the non-frequent items are removed from the original input data, which reduces the data size. The transactional data are stored in form of RDD and distributed over cluster nodes. The resulting FL1 of the first phase is shared with all the executors of cluster using a special feature of Spark framework called broadcast variable. Initially, the set of frequent 1-itemset FL1 from phase 1 is assigned to broadcast variable, which is shared among all the nodes. Therefore, scanning the FL1 reduces the cost of I/O and required disk space. Then we filter the Transactions RDD based on FL1 by applying a flatMap () function. For each transaction, sort frequent item based on sequence of FL1 and output the result as value. By applying a flatMap () function. Finally, the reduceByKey () function constructs the compressed tree similarly constructing FP-Tree. Post-order traversal the tree to determine post-order and pre-order the tree to determine pre-order, and then use a HasMap to speed up the process of creating the N-lists associated with 1-frequent items. The Pseudo-code is shown in Figs. 5 and 6.

Algorithm of constructing PPC-Tree and corresponding HashMap

```
Input: shard of the transactional data and FL_1: the set of frequent l -itemsets by descending order
Output: PPC-Tree, H_1 the HashMap of FL_1
1. Create H1 the hash table of FL1
2. for each transaction t in D
3. flatMap(Id,t)
4. Select the frequent item in T and sort out them according to the order of FL1, Let the sorted frequent item list in T be a path[p|P] as the value to output<Id, [p|P]> where is the first element and P is the remaining list
5. End for
6. reducerByKey(Id, [p|P])
7. Create the root of a PPC-tree, R, and label it as null
8. for each [p|P])
9. Call insert_tree([p|P],T)
10. End for
11. Scan PPC tree to generate the postorder of each node
12. Return H1
Function insert_tree([p|P],T)
1.if T has a child N such that N.item-name = p.item-name
2.then increase N's count by 1
3.else create a new node N, with its count initialized to 1, and add it to T's children-list
4.if P is nonempty then call insert tree(P,N) recursively.
5.end if
6.end if
```

Figure. 5 Pseudo code of constructing PPC-Tree

Phase 3: The N-lists of 1-frequents itemsets NL1 are stored in form of RDD and distributed over cluster nodes as a group of lists for loading balance on the cluster. For example, from PPC-tree of Fig. 2:
NLG1 = {b → {< (4,8): 4 >}, f → {< (2,1): 1 >, < (8,4): 1 >, < (9,7): 1 >}}
NLG2 = {c → {< (1,2): 1 >, < (5,6): 3 >}, a → {< (3,0): 1 >, < (7,3): 1 >}}
NLG3 = {e → {< (6,5): 3 >}.

We save thus the N-list of 1-frequent itemsets in a distributed cache using the broadcast variable of spark, which is shared among all the nodes. Each node independently depth-first traversals every frequent item in the group assigned, until all frequent item sets with the current prefixes sub-tree are located far. For b in group 1, the current prefix is b, when c and e are added to the prefix sub-tree to generate 2-frequent itemsets {bc, be} (bf and ba are not frequent itemsets). To bc, be prefixed to continue the operation, eventually get all the frequent item sets on b.{b,bc,be,bce} In the prefix

Received: May 7, 2019

subtree merge process, normally when b and c are combined, the original algorithm generates PPCode <(b.pre-order, b.post-order): c.count> when the condition is b.pre-order <c. pre-order && b.post-order> c.post-order. But, this paper will generate PPCode as "c.pre-order, c.post-order): c.count> in the same condition.

As a result of the depth-first and prefix subtree policy, we must promise the new added element and the current prefix subtree on the same path, necessary and sufficient condition is the new element added and the last element of the current prefix subtree are on the same path.This's the reason why we generate PPCode as <(b.pre-order, b.post-order): c.count>. Finally, reduce combines output. The Pseudo-code of phase 3 is presented in Fig. 7.

Algorithm of generating N-List of 1-frequent itemsets from the HasMap

Input: PPC-tree and FL_1 the set of frequent 1-itemsets, H_1 the HashMap of FL_1
Output: NL_1: the set of the N-lists of frequent 1-itemsets.

```
1. Procedure N-lists construction (R, $H_1$)
2. Let C=(R.preorder,R.postorder,R.count)
3. Add C to $H_1$ [R,name] count by C.count
4. Increase H1 [R.name].count by C.count
5. For each child in R.children do
6. N-lists construction(child)
7. End for
```

Figure. 6 Pseudo code of generating N-List of 1-frequent itemsets

Algorithm of mining frequent itemsets

Input: shared the NL_1: the set of the N-lists of frequent 1-itemsets to be saved in distributed cache and NLG [i]: group i of lists NL_1.
Output: frequent k-itemsets F

```
1. for each mapper do
2. for each NL_l of NLG [i] do
3. call mining_fim_k(NL_l, $FL_k$,$NL_1$, minsup)
4. End for
5. End for
6. Function mining_fim_k(NL_k, $NL_1$,minsup)
7. For i = 0 to $NL_1$ do
8. if (NL_k.count >= |DBI|* minsup)
9. F=F U $L_k$
10. if (NL_kcount >= $NL_1$[i].count)
11. Assume $L_k= x_1x_2….x_k$ , L[i].item = $x_{k+1}$ , supp($x_k$) > supp($x_{k+1}$)
12. $FL_{k+1} = FL_k+FL_1[i]$ // $FL_{k+1} = x_1x_2….x_kx_{k+1}$
13. $FL_k= FL_{k+1}$
14. compare N-list of NL_k with N-list of $NL_1$[i]
15. if ( NL_k.preorder < $NL_1$ [i].postorder && NL_k . postorder > $NL_1$[i].preorder)
16. NL_k+l.N-list.add ( $NL_1$[i].PrePost , $NL_1$[i].postorder.count ):$NL_1$[i].count)
17. End if
18. End If.
19. End if
20. End for
```

Figure. 7 Pseudo code of mining frequent itemsets

5. Performance evaluation

In this section, the DisPrePost algorithm has been compared to two advanced algorithms, HPrePostPlus [26] and the well-known HFIM [29]. DisPrePost is the first implementation of the PrePost algorithm in the Spark framework, HPrePostPlus is a recent implementation of the Hadoop-based PrePost parallel algorithm [26] with good results, and HFIM is a typical implementation of the Spark-based Apriori parallel algorithm [29] with good performance. We evaluated speed performance by analyzing runtime and scalability.

To implement the DisPrePost in distributed environment, we set up a Spark cluster of 3 nodes where each node contains Intel® Core ™ i5- 3230M CPU@2.60GHz processing units and 12.00GB RAM. Each node is installed with Hadoop version 2.6.0, Spark version 1.6.0 and Scala version 2.11.8. Source of DisPrePost were written in Scala language. HDFS was used for storage of input dataset and output frequent itemsets. The datasets T10I4D100K and T40l10D100K are used for experiments. These two real datasets which have been commonly used for many frequent itemset mining algorithms. Table 2 shows the information of these datasets, and all of these can be found in [34].

The running time with different support degree for dataset T10l4D100K and T40l10D100K is shown in Figs. 8 and 9 separately. The x-axis denotes the support degree and y-axis represents the running time. The support degree grows from 0.1% to 0.5%.

The purpose of the first comparisons is to estimate the speed performance by analyzing the operating time of DisPrePost, HPrePostPlus and HFIM.

Table 3 The properties of datasets used in experiment

Dataset	Size	Transactions	Items	Average length
T10I4D 100K	3.8MB	100.000	870	10
T40l10D 100K	14 MB	100.000	1000	40

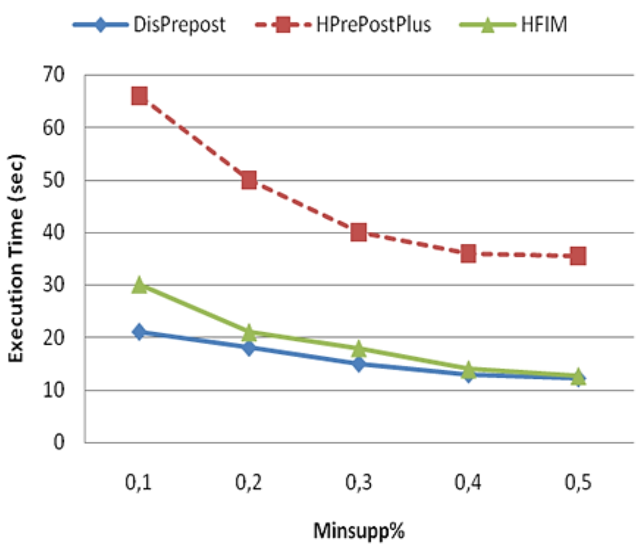


Figure. 8 The running running time of T10I4D100K

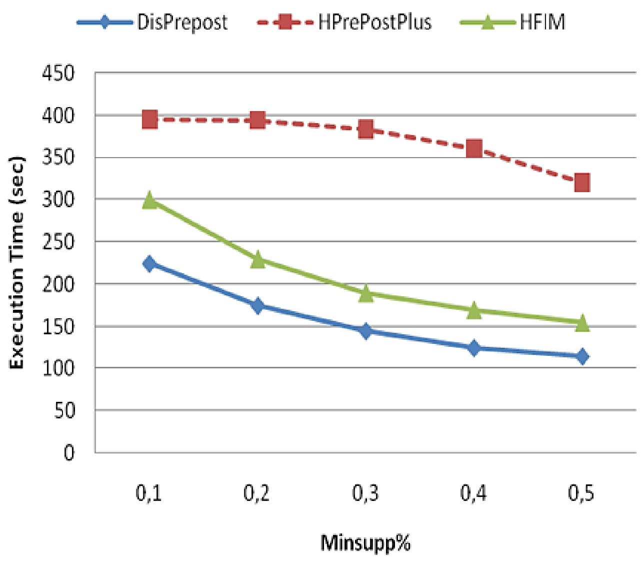


Figure. 9 The running time of T10I4D100K

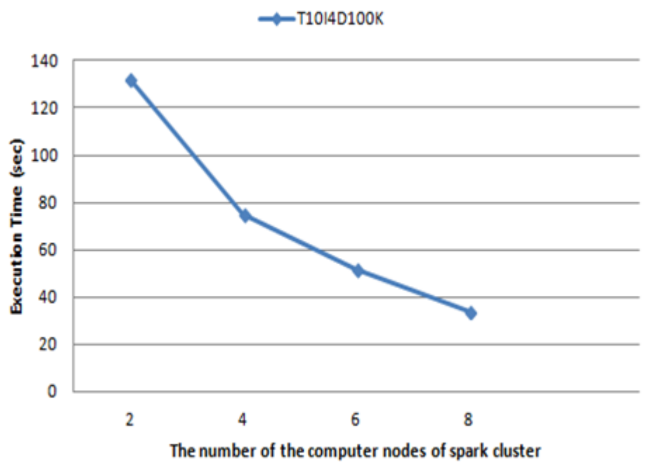


Figure. 10 The running time with different computer nodes

We can see that the execution time of DisPrePost is much lower than that of HPrePostPlus for both datasets. The both algorithms tend to be more efficient when the degree of support is set to a higher level. The execution time of HPrePostPlus is nearly three times that of DisPrePost. This is because the HPrePostPlus algorithms needs to read data from the HDFS in every MapReduce job, thus causing huge IO overhead. In contrast, DisPrePost load the input dataset from the HDFS into the RDDs and then just read the data and intermediate results from the RDDs later, thus reducing IO overhead.

We also notice that the execution time of DisPrePost is apparently lower than that of HFIM for the two datasets.

In Fig. 8, we observe that the value of the execution time HFIM is always greater than the same value for DisPrePost. Fig. 9 shows that the superiority of DisPrePost in terms of remaining time metrics becomes clearer when you use larger data sets. It can be seen that the difference between the two algorithms is greater than that observed in Fig. 8.

The results also reflect, whether on a large or small dataset, runtime of DisPrePost is shorter than HFIM, because of sharing large data with all the executors of cluster using a special feature of Spark framework called broadcast variable, when DisPrePost conducts a depth-first strategy, which reduces the communication. However, using a HasMap to speed up the process of creating the N-lists associated with frequent items from PPC tree is very effective.

For a given size of input large data, dividing the data into different numbers of partitions is crucial to the performance. The execution time of DisPrePost is also decreased through caching essential RDDs appropriately. The reason lies in the fact that essential intermediate RDDs are cached to avoid recomputing them when the memory is not large enough to hold all intermediate RDDs.

The following experiment in Fig. 10, evaluates the scalability of DisPrePost, which is also measured by the running time. The dataset T10I4D100K is used here. The experiment is performed on condition that the number of cluster computer nodes ranges from 2 to 8 while the support degree remains to be 0.5%.

In Fig. 10, the x-axis indicates the number of computer nodes in the Spark cluster, and the y-axis represents the runtime of the DisPrePost algorithm. Fig. 10 illustrates the execution time with different numbers of computer nodes. With more computer nodes, DisPrePost requires less running time, and the curve of DisPrePost has a nearly linear decline.

DisPrePost shows a characteristic of near-linear scalability.

6. Conclusion

This paper has suggested the DisPrePost algorithm as an effective algorithm for mining frequent itemsets using the N-list. First, we proposed several ameliorations on the previously published PrePost algorithm: (i) the use of a HashMap to improve the process of creating the N-lists associated with the frequent 1-itemsets from PPC tree and (ii) the implementation of a scalable Spark-based method for frequent itemset mining that has no intermediate data and small network communication (iii) the implementation an algorithm which has small network communication because we only broadcast frequent itemsets F_k by using a special feature of Spark framework called broadcast variable.

Through experiments, we compare the performance of DisPrePost, HPrePostPlus and HFIM. DisPrePost algorithm is the fastest of all algorithms. The experimental results indicate that the proposed algorithm shows better efficiency and scalability.

For future work we will focus on applying our approach for mining frequent over data streams, mining erasable itemsets, mining frequent closed itemsets and maximal itemsets.

References

[1] C.L. Philip and C.Y. Zhang, " Data-intensive applications, challenges, techniques and technologies: a survey on big data," *Information Sciences*, Vol.275, pp. 314–347, 2014.

[2] C. Bhat and C.K. Bhendadia, "Mining Big Data Using Modified Induction Tree Approach", *International Journal of Intelligent Engineering and Systems*, Vol.9, No.2, pp.14-20, 2016.

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules", In: *Proc. of the 20th Very Large Data Bases Conference*, pp. 487–499, 1994.

[4] C.-H. Chee, J. Jaafar, I.A. Aziz, M.H. Hasan, and W. Yeoh, "Algorithms for frequent itemset mining: a literature review", *Artificial Intelligence Review*, pp. 1-19, 2018.

[5] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation", In: *Proc. of the International Conference on Management of Data*, Vol.29, No.2, pp. 1-12, 2000.

[6] Z.H. Deng, Z.H. Wang, and J.I. Jiang, "A new algorithm for fast mining frequent itemsets using N-lists", *Science China Information Sciences*, Vol.55, No.9, pp. 2008-2030, 2012.

[7] Z H. Deng, and S.L. Lv, "PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning", *Expert Systems with Applications*, Vol.42, No.13, pp. 5424-5432, 2015.

[8] R. Myung, H. Yu, and D. Lee, "Optimizing Parallelism of Big Data Analytics at Distributed Computing System", *International Journal on Advanced Science, Engineering and Information Technology*, Vol.7, No.5, pp.1716-1721, 2017.

[9] R. Kessl, "Probabilistic static load-balancing of parallel mining of frequent sequences", *IEEE Transactions on Knowledge and Data Engineering*, Vol.28,No.5 pp. 1299–1311, 2016.

[10] S. Li, T. Hoefler, and C. Hu, "Improved MPI collectives for MPI processes in shared address spaces", *Cluster Computing*, Vol. 17, No.4, pp. 1139– 1155, 2014.

[11] M.G. Kaosar, Z. Xu, and X. Yi, "Distributed Association rule mining with minimum communication overhead", In: *Proc. of the Eighth Australasian Data Mining Conference*, Vol. 101, pp. 17–23, 2009.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", In: *Proc. of the 6th conference on Symposium on Operating Systems Design & Implementation*, Vol. 6, pp. 10-10, 2004.

[13] Y. Rochd, I. Hafidi, and B. Ouartassi, "A Review of Scalable Algorithms for Frequent Itemset Mining for Big Data Using Hadoop and Spark", In: *Proc. of the 2nd International Conference on Real-Time Intelligent Systems*, Vol 2, pp .91-101, 2017.

[14] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets", In: *Proc.* of the 2Nd USENIX conference on hot topics in cloud computing, HotCloud'10, pp. 10, 2010.

[15] D. Apilatti, E. Baralis, T. Cerquitelli, P. Garza, Pulverenti, and L. Venturini, "Frequent itemset mining for big data: A Comparative analysis", *Big Data research*, Vol.9, pp.67-83, 2017.

[16] M. Lin, P. Lee, and S. Hsueh, "Apriori-based Frequent Itemset Mining Algorithms on MapReduce", In: *Proc. of the 16th International Conference on Ubiquitous Information Management and Communication*, 2012.

[17] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel Implementation of Apriori Algorithm Based on MapReduce", In: *Proc. of the 13th ACIS International Conference Software Engineering, Artificial Intelligence Networking and Parallel/Distributed Computing*, pp.236–241, 2012.

[18] H. Li, Y. Wang, and D. Zhang, "Pfp: parallel fp-growth for query recommendation", In: *Proc. of the Conference on Recommender Systems*, pp. 107-114, 2008.

[19] Y. Xun, J. Zhang, X. Qin, and X. Zhao, "Fidoop-dp: data partitioning in frequent itemsetmining on hadoop clusters", *IEEE Transactions on Parallel and Distributed Systems Journal*, Vol. 28, No.1, pp.101–114, 2017.

[20] S. Moens, E. Aksehirli, and B. Goethals, "Frequent Itemset Mining for Big Data," in *IEEE International Conference Big Data*, pp.111–118, 2013.

[21] M.J. Zaki, S. Parthasarathy, and M. Ogihara, "Parallel algorithms for discovery of association rules", *Data Mining and Knowledge Discovery*, Vol.1, No.4, pp.343-373, 1997.

[22] C. Zhang, P. Tian, X. Zhang, Q. Liao, Z.L. Jiang and X. Wang, "HashEclat: an efficient frequent itemset algorithm", *International Journal of Machine Learning and Cybernetics*, Vol. 1, pp 1-14, 2019.

[23] J. Liao, Y. Zhao, and S. Long, "MRPrePost: A parallel algorithm adapted for mining big data", In: *Proc. of Electronics, Computer and Applications IEEE Workshop*, pp. 564 – 568, 2014.

[24] J. Liao, Y. Zhao, and S. Long, "MRPrePost-A parallel algorithm adapted for mining big data", In: *Proc. of Electronics, Computer and Applications, IEEE Workshop*, pp. 564-568, 2014.

[25] S. Thakare, S. Rathi S, and R.R. Sedamkar, "An Improved PrePost Algorithm for Frequent Pattern Mining with Hadoop on Cloud", *Procedia Computer Science*, Vol. 79, No. 9, pp. 207-214, 2016.

[26] Y. Rochd and I. Hafidi, "Performance Improvement of PrePost Algorithm Based on Hadoop for Big Data", *International Journal of Intelligent Engineering and Systems*, Vol.11, No.5, pp.226-235, 2018.

[27] N. Arybarzan, B. Bidgoli, and M. Reshnehlab, "negFIN: An efficient algorithm for fast mining frequent itemsets", *Expert Systems with Applications*, Vol.105, pp.129-143, 2018.

[28] H. Qiu, R. Gu, C. Yuan, and Y. Huang, "YAFIM: A parallel frequent itemset mining algorithm with Spark", In: *Proc. of International Parallel Distribution Process of Symposium*, pp.1664–1671, 2014.

[29] K.K. Sethi and D. Ramesh, "HFIM: a Spark-based hybrid frequent itemset mining algorithm for big data processing", *The Journal of Supercomputing*, Vol.73, No.8, pp.3652–3668, 2017.

[30] F. Zhang, M. Liu, F. Gui, W. Shen, A. Shami, and Y. A. Ma, "distributed frequent itemset mining algorithm using spark for big data analytics", *Cluster Computing*, Vol.18, No.4, pp.1493–501, 2015.

[31] S. Rathee, M. Kaul, and A. Kashyap, "R-Apriori: an efficient apriori based algorithm on spark", In: *Proc. of the 8th Workshop on Ph.D. Workshop in Information and Knowledge Management*, pp.27–34, 2015.

[32] S. Rathee and A. Kashyap, "Adaptive–Miner: an efficient distributed association rule mining algorithm on Spark", *Journal of Big Data*, Vol 5, No.1, pp.1–17, 2018.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Shenker, S. Franklin, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", In: *Proc. NSDI, 2012 of the 9th USENIX Conference on Networked Systems Design and Implementation*, p. 2-2, 2012.

[34] Frequent Itemset Mining Dataset Repository, Available at: http://fimi.ua.ac.be/data. Accessed 18 Dec 2018.