IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 9, 2021, accepted December 20, 2021, date of publication December 23, 2021,
date of current version January 4, 2022.

Digital Object Identifier 10.1109/ACCESS.2021.3137789

Towards Enhancing the Performance of Parallel FP-Growth on Spark

AMR ESSAM, MANAL A. ABDEL-FATTAH, AND LAILA ABDELHAMID
Department of Information Systems, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo 12317, Egypt
Corresponding author: Amr Essam (amressam70@gmail.com)

ABSTRACT Frequent itemset mining (FIM) is a crucial tool for identifying hidden patterns in information. FP-Growth is an FIM algorithm used to find associations. When the data size increases, the execution of FIM algorithms on a single machine suffers from computational problems, such as memory and time consumption. For these reasons, parallel and distributed processing on platforms such as Spark is essential. The parallel frequent pattern (PFP) is the implementation of FP-Growth in Spark. The main problem with PFP is that it does not consider the load balancing between cluster units. This research proposes an enhanced balanced parallel frequent pattern "EBPFP" algorithm to enhance and balance the PFP. The proposed algorithm (EBPFP) proposes two ideas. First, a strategy for load balancing between groups is proposed to ensure that the items are evenly divided between the nodes, and the cluster resources are used more effectively. Second, the improved conditional pattern base (ICPB) method aims to remove infrequent items from the conditional pattern base before constructing local FP-Trees. The experimental results show that the proposed EBPFP algorithm outperforms PFP, and the difference in running time between EBPFP and PFP was 21.56% and 39.72%, respectively.

INDEX TERMS Big data, data mining, association rule analysis, frequent pattern growth algorithm, spark, load balancing.

I. INTRODUCTION

Association rule mining (ARM) is a popular method for determining relationships that can be hidden or implicit in large datasets. The first step in the association rule mining process and the essential stage is frequent itemset mining (FIM), which aims to discover frequent itemsets [1]. One of the most popular itemset mining algorithms is the Apriori algorithm [2]. Apriori is an iterative algorithm that starts the first iteration by generating frequent items that satisfy the minimum support requirement. In the second iteration, it generates candidate frequent 2-itemsets from the 1-itemsets generated from the previous level. Apriori continues joining k-frequent itemsets to generate $k+1$ itemsets and $k+1$ itemsets to generate $k+2$ itemsets till it generates all frequent itemsets. Apriori suffers from two problems: Each iteration generates a large number of candidate frequent item sets and requires a scan of the database, which is inefficient when the data size increases.

To overcome these shortcomings, frequent pattern growth (FP) growth was proposed [3]. Unlike the Apriori algorithm, FP-Growth does not generate candidate frequent item sets, and it only scans the database twice. The FP-Growth algorithm is more efficient than the Apriori algorithm [3]. FP-Growth aims to compress the dataset using tree data called FP-Tree, a structure that represents data in a compact form. First, it scans the database to discover frequent items and sorts them in a list called Frequent List (F-List). Second, it performs a second scanning of the database to construct an FP-Tree using F-List. Next, the FP-Growth algorithm uses FP-Tree to build a conditional pattern base and a conditional FP-tree for each frequent items to generate frequent itemsets. When it comes to processing large-scale data, such as most FIM algorithms, the FP-Growth algorithm encounters the problem of high memory consumption and is time-consuming owing to the limitation of hardware resources on a single machine [3]. This makes it necessary to use big data processing frameworks that are designed to process large-scale datasets in a parallel and distributed manner, such as the MapReduce and Spark frameworks.

MapReduce is an efficient distributed execution-processing framework. It decomposes the complete process into two major tasks: a map task and a reduction task [4]. It uses the Hadoop Distributed File System (HDFS) to store a large

The associate editor coordinating the review of this manuscript and approving it for publication was Aasia Khanum.

dataset and the results of the map and reduce operations. However, MapReduce is not the best choice for algorithms that require intensive iterative processing owing to heavy disk input and output (I/O) overhead [5]. Spark is a parallel processing framework for big-data solutions [6]. This is based on an in-memory parallel processing model to overcome the disk I/O problem. Therefore, Spark is 100× faster than MapReduce and is more suitable for iterative algorithms [7].

The current parallel implementation of FP-Growth in the Spark framework is parallel FP-growth (PFP) [8]. PFP can achieve good parallelization by eliminating the computational dependencies between parallel tasks. However, it does not consider load balancing between the working nodes [9]. The node load is determined by the work that the node needs to complete its task. Load imbalance in PFP means that some nodes have more work assigned to them than others, requiring more time, which increases the overall time of algorithm execution.

In this paper, we propose an enhanced balanced parallel frequent pattern (EBPFP), which uses two methods. First, a load-balancing strategy is used to evenly group the items over the groups to achieve efficient utilization of the cluster nodes. Second, an improved conditional pattern base (ICPB) is a method for removing infrequent items while building the conditional pattern base before constructing conditional FP-Trees. To validate the efficiency of EBPFP, it is compared with two established parallel FP-Growth algorithms, the PFP algorithm [8] and the balanced parallel FP-growth (BPFP) algorithm [9]. The results showed that EBPFP outperformed both algorithms in terms of time consumption and scalability.

The rest of the paper is organized as follows. An overview of the related work is presented in Section II. Section III presents the proposed EBPFP algorithm. Subsequently, The experiments and results are presented in Section IV. Finally, Section V provides concluding remarks.

II. RELATED WORK

This section presents attempts by researchers to parallelize the FP-Growth algorithm and overcome the shortcomings of the parallelization process and the improvements of the parallel FP-Growth algorithm over the years.

[10] introduced a multiple local frequent pattern tree (MLFPT) that executes FP-Growth on symmetric multiprocessors that share the same memory. The algorithm constructs multiple local FP-trees for each processor. Hence, local FP-Trees are accessed from all processors at the growth step. However, MLFPT encounters scalability issues related to large-scale datasets that require large amounts of memory. Meanwhile, [11] proposed parallel FP-growth for a cluster. It overcame the problem of memory size bottlenecks from that MLFPT suffered from. However, it suffers from a high communication cost between the cluster nodes.

[12] implemented FP-Growth under the new architecture features of modern processors, which introduced the concept of cache prefetching, where FP-Tree is implemented using a cache-conscious structure that improves the performance of growth operations. The algorithm uses a multithreading approach, in which the threads can reuse the cache. On the other hand, [13] presented a parallel FP-growth algorithm for a cluster that uses the master–slave approach. It uses a compressed data structure to avoid the drawbacks of [11] by achieving lower communication costs and improving the cache memory performance and I/O utilization [12]. In this algorithm, each node can extend its structure beyond memory limitations by supporting 64-bit architecture leveraging. It scales well over hundreds of nodes. However, it lacks fault-tolerance capability.

[8] implemented a parallel frequent pattern (PFP), which is a parallel implementation of FP-Growth based on the MapReduce framework, to achieve more effective parallelization and higher scalability to thousands of computers. It groups items over a set of groups by calculating the size of each group by dividing the total number of items by the number of groups. Next, each group's share was consecutively taken from the list of items. For each group, group-dependent transactions were grouped to generate a group-dependent dataset. Unlike in [13], PFP supports the fault recovery feature to lower the probability of a node crash during the execution of the task. Unfortunately, the PFP grouping strategy does not create groups with the same load weight, which is inefficient for the overall runtime of PFP.Moreover, [9] introduced the balanced parallel frequent pattern algorithm (BPFP), which adds a load-balancing feature to the PFP algorithm. The BPFP algorithm groups the items over the groups using a queue, which is better than the grouping strategy of PFP, but it still makes a large difference between the first and last groups.

[14] proposed an improved parallel frequent pattern algorithm FPM. When the dataset is composed of a large number of small files, HDFS consumes a large amount of memory. The IPFP strategy integrates small files to lower heavy I/O overhead as a pre-step for the PFP algorithm.In addition, [15] introduced an improved balanced parallel frequent pattern algorithm (IBPFP), which provides a load-balancing strategy for PFPs. They also proposed a cutting method that aims to reduce the size of local FP-trees by merging their paths. Therefore, the IBPFP improved the performance of the PFP algorithm.

[16] implemented the (FiDoop-DP: Data Partitioning in Frequent Itemset Mining on Hadoop Clusters) algorithm that aims to balance parallel FP growth by using the Voronoi-based model in partitioning data across reducers and preventing the duplication of transactions. However, it suffers from a preprocessing overhead. Furthermore, [17] proposed a parallel improved FP-growth algorithm called the IFPS. The IFPS aims to compress large-scale datasets using a matrix to reduce traffic between nodes and optimize the performance. The IFPS algorithm outperforms the PFP algorithm. IFPS focused on improving the memory performance of the FP-Growth algorithm on Spark but did not consider loading balancing between the groups. [18] The proposed HBPFP algorithm introduces a heuristic-based load-balancing strategy for the PFP algorithm. The grouping strategy of the

HBPFP was better than those of the PFP and BPFP algorithms. However, it does not consider improving the FP growth mining process.

Moreover, when moving to the second phase of the FP-Growth algorithm, which is association rule mining for the generated frequent patterns. [19] proposed a multitask association rule miner (MTARM) algorithm that tends to overcome the problem of standard ARM algorithms in discovering the rules using a multitask approach by ignoring the interrelationships among multiple tasks. MTARM has two phases. First, highly frequent rules were discovered for each task. Second, it combines the local rules from each task using a majority-voting mechanism. [20] proposed a distributed frequent pattern mining (DFP) method that aims to improve the execution time of FP-Growth by reducing the data transmission cost between the nodes in the parallel and distributed processing frameworks, high memory cost, and redundant execution time owing to unadaptable nodes. DFP provides a set of algorithms for providing data and workloads in a fast and scalable manner, as well as a data structure for storing items with their counts to reduce network data transmission.

In summary, some of the aforementioned algorithms have attempted to improve the performance of the FP-Growth algorithm. Some have attempted to overcome the shortcomings related to the parallelization of the FP-Growth algorithm, such as load imbalance between the working nodes. Thus, this study proposes an enhanced balanced parallel frequent pattern algorithm (EBPFP) to improve the FP-Growth mining step and introduces a load-balancing strategy to overcome the parallelization shortcoming.

III. PROPOSED EBPFP ALGORITHM

The proposed Enhanced Balanced Parallel Frequent Pattern algorithm (EBPFP) puts forward two methods. The first is the EBPFP load balancing strategy. Second, an improved conditional pattern base is used. To validate the efficiency of EBPFP, it is compared with two established parallel FP-growth algorithms: the PFP algorithm [8] and balanced parallel FP-growth (BPFP) [9]. This section is divided into three parts. Section III.A proposes a detailed load-balancing strategy for the EBPFP algorithm. In Section III. B, the ICPB method is presented. The overall outline of the EBPFP algorithm is presented in Section III.C.

A. EBPFP LOAD BALANCING STRATEGY

This section demonstrates the grouping strategies of the PFP, BPFP, and EBPFP algorithms, and outlines the differences between them. These three algorithms are the first steps of the FP-growth algorithm. The first step of FP-Growth is scanning the dataset to find frequent items whose support is equal to or greater than the minimum support. They are then ordered in a list called the Frequent List (F-List) in descending order. For example, the transactional dataset in Table 1 contains ten transactions, and the minimum support is three. After performing the first step, the F-List was generated, as shown in Table 2.

TABLE 1. Example transaction dataset.

Transaction ID	Transaction Items
1	44,52,57,45,49,61,56
2	52,64,61,50,44,59
3	60,58,56,46,48,59
4	63,51,54,50,46,45
5	64,62,51,49,57
6	46,49,52,56,59,63,45
7	45,48,61,59,62,63
8	48,55,60,63,50,53,44,49
9	48,44,46,51,59,60,63
10	50,54,57,60,47,64

TABLE 2. Frequent list - Frequent items load weights.

Item	Frequency	Load weight
59	5	0.0
63	5	0.30
44	4	0.47
45	4	0.60
46	4	0.70
48	4	0.78
49	4	0.85
50	4	0.90
60	4	0.95
51	3	1.00
52	3	1.04
56	3	1.07
57	3	1.11
61	3	1.15
64	3	1.18

The overall load required to mine each item is determined by the number of recursions executed by building the conditional FP-tree for the item. Therefore, the position of the item in F-List determines the depth of the corresponding conditional FP tree [9]. Accordingly, the load estimation equation is as follows:.

$$\boldsymbol{L_i = \mathbf{Log}(P_i)} \tag{1}$$

where L_i is the load of item i and P_i is the position of the item in F-List. According to Equation 1, The load weights of the items according to Equation 1 are listed in Table 2.

1) GROUPING STRATEGY OF PARALLEL FP-GROWTH (PFP)

PFP groups the items in a straightforward way as follows:

Step1: It specifies the number of groups, which is usually set manually by the user or based on the cluster's total number of nodes.

Step2: It specifies The number of items in each group was specified by dividing the number of F-List items (marked as K) by the total number of groups (marked as G).

Step3: It takes the first n items from F-List and places them in the first group. Next, we take the second n items, put them in the second group, and repeat this step until all F-List items are grouped.

where $n = K/G$ if the result of division is an integer or the next largest integer if the result is not an integer.

TABLE 3. PFP Groups items with loading weights.

Group ID	Items	Items load weights	The total group load weight
$\mathbf{G_1}$	59, 63, 44, 45, 46	0, 0.3, 0.47, 0.6, 0.7	**2.07**
$\mathbf{G_2}$	48, 49, 50, 60, 51	0.78, 0.85, 0.90, 0.95, 1.00	**4.48**
$\mathbf{G_3}$	52, 56, 57, 61, 64	1.04, 1.07, 1.11, 1.15, 1.18	**5.55**

TABLE 4. BPFP Groups items with loading weights.

Group ID	Items	Items load weights	The total group load weight
$\mathbf{G_1}$	59, 45, 49, 51, 57	0, 0.6, 0.85, 1.0, 1.11	**3.56**
$\mathbf{G_2}$	63, 46, 50, 52, 61	0.3, 0.7, 0.90, 1.04, 1.15	**4.09**
$\mathbf{G_3}$	44, 48, 60, 56, 64	0.47, 0.78, 0.95, 1.07, 1.18	**4.45**

For example, in Table 2, there are 15 frequent items, and the group number is three. Each group consisted of five items. Table 3 presents the three groups with associated items and item loads from Table 2. The load of the group equals the sum of the loads on the items.

Table 3 shows that there is a significant difference between the load of the group and how the PFP grouping strategy is unbalanced.

2) GROUPING STRATEGY OF BALANCED PARALLEL FP-GROWTH (BPFP)

To improve load balancing, the BPFP groups the items over the groups using the priority queue strategy, as follows:

Step1: From F-List, one item was placed in each group. Next, the load of each group that has been initialized with the load weight of the first item is calculated.

Step2: Add the following item from F-List to the group with the minimum load:

Step3: Recalculate the group's load where the new item has been added, and then repeat steps 2 and 3 until all F-List items are grouped.

From Table 4, it can be observed that the BPFP strategy is much more effective than the PFP strategy. However, there was a recognizable difference in the loads between the first and last groups.

3) GROUPING STRATEGY OF ENHANCED BALANCED PARALLEL FP-GROWTH (EBPFP)

For comparison, the number of groups is three groups, as in the previous algorithms. The EBPFP grouping strategy is explained in the following steps.

Step1: Take the first G items from F-List and add them to the G groups, one item per group. where, G denotes the number of groups.

Step 2: Calculate The load for each group was calculated and sorted in descending order.

TABLE 5. EBPFP Groups items after iteration 1.

Group ID	Items	Total Group Load
G_1	59	0
G_2	63	0.3
G_3	44	0.47

TABLE 6. EBPFP Groups items after iteration 2.

Group ID	Items	Total Group Load
G_1	59, 48	0.78
G_2	63, 46	1.0
G_3	44, 45	1.07

TABLE 7. EBPFP Groups items after iteration 3.

Group ID	Items	Total Group Load
G_1	59, 48, 60	1.73
G_2	63, 46, 50	1.9
G_3	44, 45, 49	1.92

Step3: Take the following G items from the F-List and place them into the sorted groups correspondingly. This means that the first item of the current G items whose load weight is minimum will be added to the group with the maximum load.

Step4: Repeat step 2 and step 3 till all items are grouped.

The following example explains the EBPFP load balancing strategy iteration by iteration. Tables 5–9 outline the items of the group with the corresponding total load for each iteration.

Iteration 1: The first three items of F-List were placed into three groups.

Iteration 2: The order of groups based on descending sorting is G_3, G_2, and G_1. Therefore, the next three items were placed in the groups in this order.

Iteration 3: The order of groups remains the same. Therefore, the next three items are placed in the same order.

TABLE 8. EBPFP Groups items after iteration 4.

Group ID	Items	Total Group Load
G_1	59, 48, 60, 56	2.8
G_2	63, 46, 50, 52	2.94
G_3	44, 45, 49, 51	2.92

TABLE 9. EBPFP Groups items after iteration 5.

Group ID	Items	Total Group Load
G_1	59, 48, 60, 56, 64	3.98
G_2	63, 46, 50, 52, 57	4.05
G_3	44, 45, 49, 51, 61	4.07

TABLE 10. Comparison of the load of the groups between EBPFP, BPFP, and PFP.

Group ID	PFP	BPFP	EBPFP
G_1	**2.07**	**3.56**	**3.98**
G_2	**4.48**	**4.09**	**4.05**
G_3	**5.55**	**4.45**	**4.07**

Iteration 4: The order of the loads is the same. Thus, the items were grouped in the same order.

Iteration 5: The order of the groups is different during this iteration. G_2 exhibited the maximum load. Since the first of the next three items will be put in G_2, the second item will be put in G_3, and the last item will be placed in G_1.

The loads in the PFP, BPFP, and EBPFP groups are listed in Table 10.

From Table 10, the EBPFP load-balancing strategy outperforms PFP and BPFP and is more effective in terms of runtime performance and cluster resource utilization. The pseudocode for the EBPFP load-balancing strategy is presented in Algorithm 1. The time complexity of Algorithm 1 is $O(n \times \log(g))$, where n is the number of items and g is the number of groups.

Algorithm 1 EBPFP Load Balancing Strategy

```
Input 1: Hash map that maps items to their load weights (itemsLoadsMap)
Input 2: List of group Ids (gList)
Output: Hash map that maps group id to member items (groupItemsMap)
1: var groupsTotalLoadWeights = new LinkedHashMap<String,
Double>();
2: var groupItemsMap = new LinkedHashMap<String,List<String≫();
3: var i = 1;
4: while (i <= gList.size())
5:   begin
6:     groupsTotalLoadWeights.put(gList.get(i), 0.0);
7:     groupItemsMap.put(gList.get(i), new LinkedList<>());
8:     i++;
9:   end
10: var Items = new LinkedList<String>(itemsLoadsMap.keySet());
11: var ItemsWeights = new
 LinkedList<Double>(itemsLoadsMap.values());
12: i = 1;
13: while (i <= Items.size())
14: begin
15: for (Map.Entry<String, Double> entry:
 groupsTotalLoadWeights.entrySet())
16: begin
17:   if (i > Items.size())
18:    beign
19:     break;
20:   end
21:   entry.setValue(entry.getValue() + ItemsWeights.get(i));
22:   List<String> groupItems = groupItemsMap.get(entry.getKey());
23:   groupItems.add(Items.get(i));
24:   i++;
25:  end
26:  var groupsList = new LinkedList<Map.Entry<String,
 Double≫(groupsTotalLoadWeights.entrySet());
27: Collections.sort(groupsList, new Comparator<Map.Entry<String,
 Double≫()
28:     begin
29:      public int compare(Map.Entry<String, Double> g1,
 Map.Entry<String, Double> g2)
30:        begin
31:         return g2.getValuecompareTo(g1.getValue());
32:        end
33:     end
34: groupsTotalLoadWeights=groupList;
35: end
```

B. IMPROVED CONDITIONAL PATTERN BASE METHOD (ICPB)

The ICPB method is an improvement over the original FP-growth algorithm. Therefore, this section provides an example of an FP-Growth algorithm. Next, the proposed ICPB method was demonstrated.

1) FP-GROWTH ALGORITHM

To explain the ICPB method, it is necessary to understand the overflow of the FP-Growth algorithm and how it constructs conditional FP-trees. The FP-Growth algorithm process has two main phases: construction of FP-Tree and mining of FP-Tree for FPs. To construct the FP-Tree, database scanning is required to build the F-List, which contains frequent items in a decreasing order. Next, the transactions in the dataset are reordered according to the order of the items in F-List, and infrequent items are pruned. The algorithm then transforms each sorted transaction into a tree path by sequentially inserting its items into path nodes. The order of the items is crucial in building the FP-Tree because if the transactions share the same prefix, because the existing item node count is incremented if the transactions share the same prefix. Otherwise, a new node is created for item. While building the FP-Tree, the header table is created to have the frequent items in F-List with their node links that connect the nodes of the same item in the FP-Tree. This section provides an example of FP-Growth using the dataset in Table 1, and the frequencies of the items are presented in Table 2. The minimum support for our example was four.

Using the minimum support, the frequent items in Table 2, whose frequencies meet the minimum support requirement, are sorted and presented in Table 11, and the infrequent items are pruned. Based on Table 11, the transactions are reordered, and infrequent items are removed, as shown in Table 12.

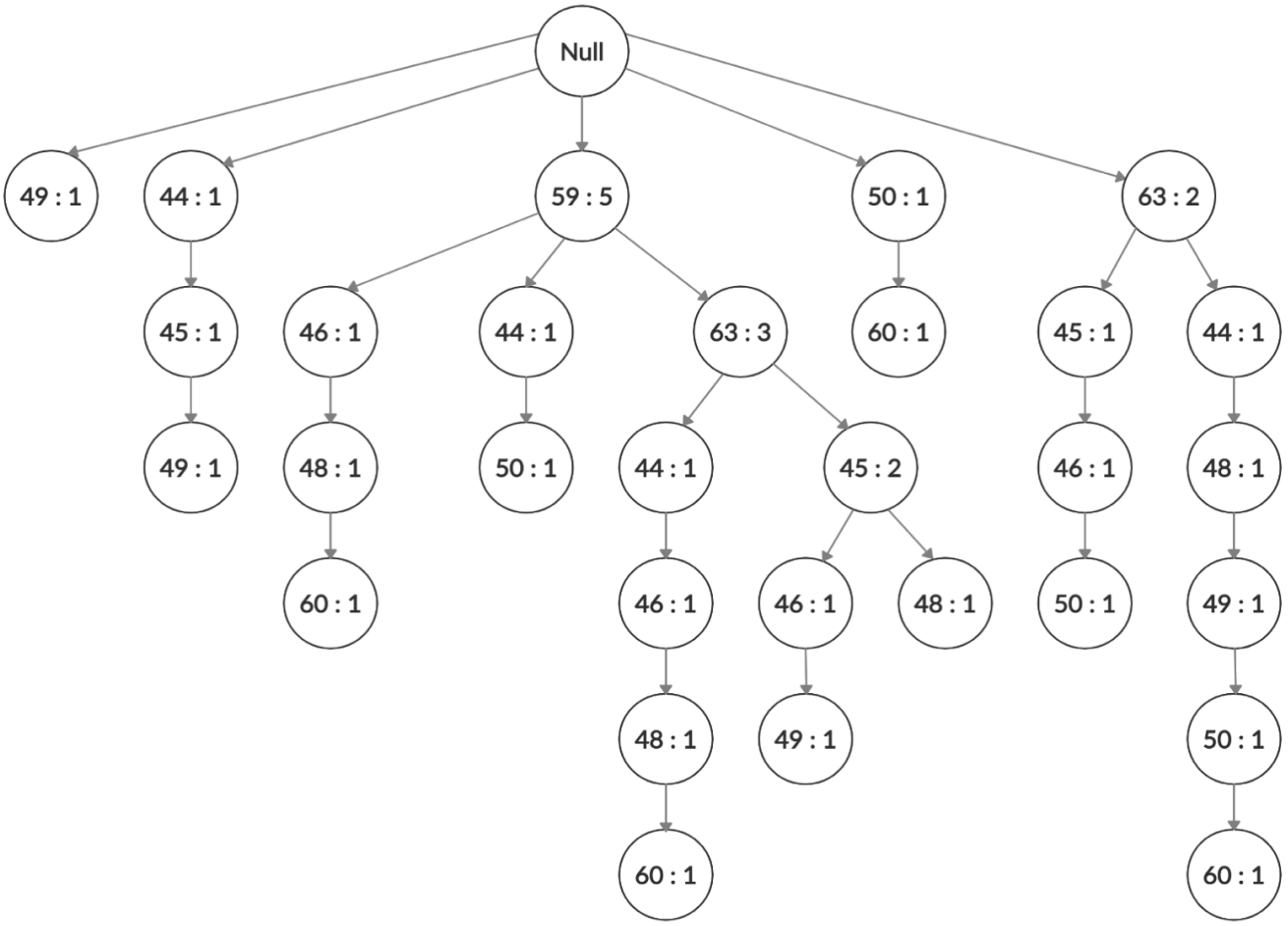

FIGURE 1. FP-Tree for the transactions in Table 12.

TABLE 11. Ordered frequent items (F-list).

Item	Frequency
59	5
63	5
44	4
45	4
46	4
48	4
49	4
50	4
60	4

Ordered transactions are used to build an FP-tree. The FP-tree is shown in Fig.1.

FP-Tree and minimum support are the inputs for the second phase of the algorithm. To discover the frequent patterns, FP-Growth starts to pass over the items in the header table upward from the bottom to the top, and for each item, it traverses its nodes in the FP-Tree using the node links in the header table. For each node, the algorithm stores the prefix path of the node containing the items in the path from the item node to the root node of the tree. The prefix paths of a specific item are called its conditional pattern bases. Next, FP-Growth uses the conditional pattern base to build the conditional FP-Tree using the same approach used in building the FP-Tree. From the conditional FP-tree,.

algorithm generates frequent patterns that co-occur with a current item. The same process is repeated for all items in the header table until all frequent patterns are generated. Table 13 presents the conditional pattern base for frequent items.

As '59' is the most frequent item, it has no prefix path in all transactions. Therefore, there is no conditional pattern base for item '59'. All frequent patterns for item '59'' were generated in the previous items.

2) PROPOSED ICPB METHOD

The conditional pattern base for any item in FP-Growth is a set of patterns that co-occurs with this item. These patterns may contain some infrequent items. These items are ignored after transforming the conditional pattern base into a conditional FP-tree. The improved conditional pattern base (ICPB) method aims to improve the construction of a conditional pattern base by removing infrequent items from the patterns before building the conditional FP-Tree, which reduces the

TABLE 12. Ordered transactions based on frequent items in Table 11.

Transaction ID	Transaction Items
1	44, 45, 49
2	59, 44, 50
3	59, 46, 48, 60
4	63, 45, 46, 50
5	49
6	59, 63, 45, 46, 49
7	59, 63, 45, 48
8	63, 44, 48, 49, 50, 60
9	59, 63, 44, 46, 48, 60
10	50, 60

TABLE 13. Conditional pattern base for the FP-Tree in Fig.1.

Item	Conditional Pattern Base
60	[59, 46, 48: 1]; [59, 63, 44, 46, 48: 1]; [63, 44, 48, 49, 50: 1]; [50: 1]
50	[59, 44: 1]; [63, 45, 46: 1]; [63, 44, 48, 49: 1]
49	[44, 45: 1]; [59, 63, 45, 46: 1]; [63, 44, 48: 1]
48	[59, 46: 1]; [59, 63, 44, 46: 1]; [59, 63, 45: 1]; [63, 44: 1]
46	[59: 1]; [59, 63, 44: 1]; [59, 63, 45: 1]; [63, 45: 1]
45	[44: 1]; [59, 63: 2]; [63: 1]
44	[59: 1]; [59, 63: 1]; [63: 1]
63	[59: 3]
59	-

size of the conditional FP-Tree and the number of recursions required to mine the tree.

To remove infrequent items from the conditional pattern base, the ICPB method uses a table called a pattern item table that tracks the items in the paths and their count. For each item node, the algorithm traverses the path from the item node to the root node to compose its prefix path. While visiting the node, the algorithm checks the pattern item table and increments the count of the item in the table by one, and if the item is found. Otherwise, it inserts the item into the table with a count of 1. After building the conditional pattern base for the item, the items in the pattern item table that do not meet the minimum support, considered infrequent items, are removed. Next, the patterns in the conditional pattern base are filtered using a pattern item table. This process was repeated for each item.

To observe the difference between the original method and the ICPB method, the conditional pattern base for item '60, which is the last item in the F-List, will be discussed. Fig.2 illustrates the conditional FP-Tree for item '60' in the original FP-Growth method, and Table 14 presents the pattern item table for the item. While composing the conditional pattern base for the item, we found that the item that co-occurs with it is '48'' three times. Considering a minimum support of 4, there were no items that were considered to be frequent with item '60'. Therefore, all items were removed from the pattern item table and filtered from the conditional pattern base. Accordingly, there is no need to construct a conditional

TABLE 14. Patterns items for item '60' before and after filtration.

Item	Patterns Items before filtration	Patterns Items after filtration
D	48:3, 59:2, 63:2, 44:2, 46:2, 50:2, 49:1	{}

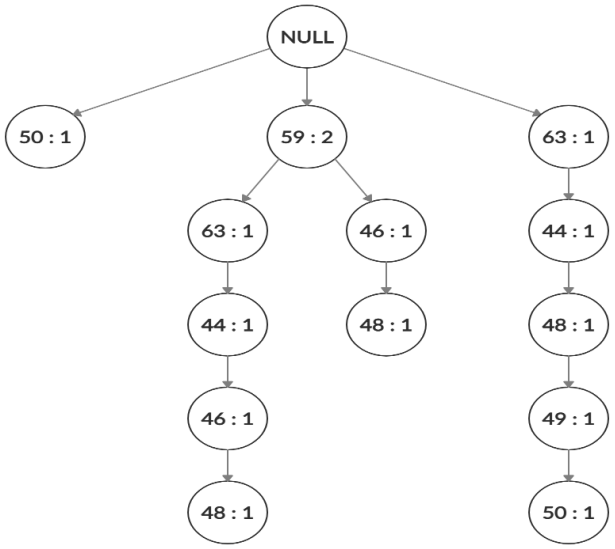

FIGURE 2. Conditional FP-Tree for item '60' using FP-Growth method.

FP-Tree for the item because in the ICPB method, the conditional FP-Tree is constructed only for frequent items in the conditional pattern base.

From Table 15 and Fig.2, it is evident that removing infrequent items from the conditional pattern base reduces the size of the conditional FP-tree, thereby reducing the number of recursions required to mine it. Algorithm 2 presents the ICPB method. The time complexity of Algorithm 2 is O(n*m), where n is the size of the conditional pattern base (the number of transactions containing the examined item) and m is the length of the longest transaction in the conditional pattern base.

C. EBPFP OUTLINE ON SPARK FRAMEWORK

The EBPFP is implemented in the Spark framework in the following steps. Fig.3 presents an overview of the workflow of the algorithm.

Step 1 (Sharding): Spark is based on resilient distributed datasets (RDDs) to execute parallel tasks. Therefore, the original database was divided into partitions and stored in different nodes of the cluster. These partitions are known as shards.

Step 2 (Parallel counting): In this step, the algorithm counts the frequencies of items in the dataset. It generates a list of key-value pairs, where the key is the item and the value is the item count. Next, it sorts the items in descending order and removes infrequent ones. This step results in an F-List.

Step 3 (EBPFP grouping): The items in F-List are divided into N groups using the grouping strategy of the proposed

TABLE 15. Comparison of FP-Growth original method and ICPB method.

	FP-Growth method	ICPB method
Conditional pattern base	[59, 46, 48: 1]; [59, 63, 44, 46, 48: 1]; [63, 44, 48, 49, 50: 1]; [50: 1]	[]

TABLE 16. Characteristics of the datasets.

Dataset	Number of transactions	Number of different items
Kosarak	990002	41270
Accidents	340183	486
Webdocs	1692082	5267656

algorithm. (N is determined by cluster cores). The list of groups is called the G-List.

Step 4 (Group-oriented datasets): According to G-List, each transaction is mapped into a set of groups containing the items in this transaction. The result of this step was a set of group-dependent transaction datasets.

Step 5 (Parallel-enhanced FP-Growth): For each group-dependent transaction dataset, the enhanced version of the FP-Growth algorithm is executed, and local frequent patterns are generated in parallel. Enhanced FP-Growth is the original FP-Growth algorithm with an improved conditional pattern base (ICPB) method.

Step 6 (Aggregating): The final step is to aggregate the local FPs generated from the previous step to generate the final global FPs.

Finally, this section proposes the EBPFP algorithm in detail using these two methods. The experiments and results of the algorithm are explained and discussed in Section IV.

IV. EXPERIMENTS AND RESULTS

The performance of EBPFP was evaluated by comparing it with the Spark PFP algorithm in the machine learning library (MLlib) [22] and balanced PFP (BPFP) [9] in terms of runtime performance and scalability. MLlib [21] is a spark machine learning library that includes PFP algorithm implementation [8]. For runtime performance, the three algorithms were executed on the Kosarak dataset [24] and Accidents dataset [25]. For scalability, the algorithms were executed on a WebDocs dataset [23]. Section IV.A illustrates the characteristics of the datasets used in the experiments. Section IV.B presents the results and discussion of the runtime performance, and Section IV.C presents the results and discussion of the scalability.

A. DATASETS

The characteristics of the datasets are presented in Table 16.

B. RUNTIME PERFORMANCE

The algorithms were executed and run on the Amazon Web Services (AWS) elastic map reduce (EMR) cluster. The cluster consisted of five nodes of type m5.xlarge. This type of system is suitable for general-purpose applications. One node was considered the master node, and the other four were slaves. Each node had four CPU cores and 16 GB of RAM. The experiments were performed with the default EMR SPARK cluster driver (master) and executer memory settings. MPFP, BPFP, and EBPFP were executed on the Kosarak and accident datasets for different minimum support values.

Algorithm 2 ICPB Method

```
Input 1: Item conditional patterns list (cpList)
Input 2: Minimum support (minSup)
Output: Item Improved conditional patterns list (icpList)
var i = 0;
var icpList = new List<List<String≫();
var cpListSize=cpList.size();
var patternItemsArray= new List<String>();
var newPatternItemsArray=new List<String>();
var patternsItemsTable = new HashMap<String, integer>;
while (i< cpListSize )
 begin
  patternItemsArray = cpList.get(i);
  Foreach (item in patternItemsArray)
   begin
    if (patternsItemsTable.contains(item))
    begin
     patternItemsTable.put(item,patternsItemsTable.getValue(item) + 1);
     end
    else
    begin
      patternItemsTable.put(item, 1);
    end
   end
 i++;
end
i=0;
while (i< cpList.size())
 begin
  patternItemsArray = cpList.get(i);
  Foreach (item in patternItemsArray)
   begin
    if (patternsItemsTable.getValue(item)>=minSup)
     begin
      newPatternItemsArray.push(item);
     end
    end
  icpList.push( newPatternItemsArray );
  newPatternItemsArray.clear();
 end
```

1) RUNTIME PERFORMANCE FOR KOSARAK DATASET

The algorithms were run for a range of minimum support values from 0.2% to 1.1%. Table 17 lists the number of generated frequent patterns for each minimum support value. Fig.4 shows the average running times in seconds for the algorithms using the thresholds. Each run was executed ten times to obtain a valid result for the running times, and the average was calculated.

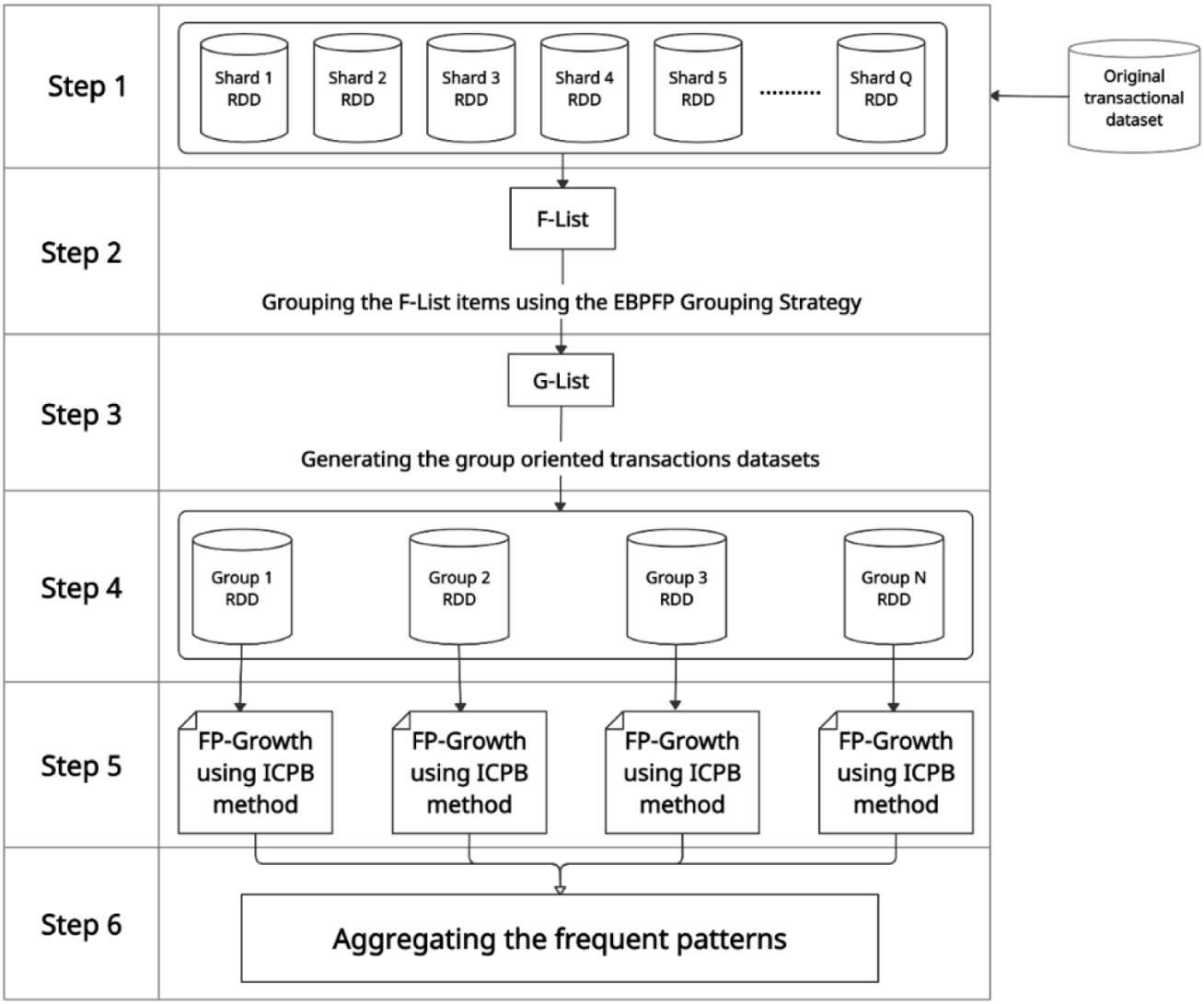

FIGURE 3. Architecture of EBPFP.

TABLE 17. Frequent patterns for each minimum support threshold for the kosarak dataset.

Minimum support threshold (%)	Generated frequent patterns
0.2	39464
0.3	5011
0.4	2522
0.5	1618
0.6	1135
0.7	789
0.8	583
0.9	465
1	383
1.1	312

As shown in Fig.4, for most thresholds, EBPFP performed better than BPFP and MPFP. On average, EBPFP was 21.56% faster than MPFP, and 5.21% faster than BPFP.

2) RUNTIME PERFORMANCE FOR ACCIDENTS DATASET

From Table 18, it is obvious that the number of generated frequent patterns in the accident dataset is much larger than that in the Kosarak dataset. Therefore, the algorithms were run on different ranges of minimum support thresholds from 1% to 10%. Fig.5 shows the average running time in minutes, not in seconds, as in the previous dataset.

On average, EBPFP was 27.53% faster than MPFP, and 7.84% faster than BPFP. In the accident dataset, more time was required to mine conditional FP-trees. Therefore, the improvement percentage for EBPFP compared to BPFP and MPFP is more significant.

C. SCALABILITY

The algorithms were executed and run on an AWS EMR clusters. Cluster nodes are of type m5.xlarge, the driver memory was 12 GB, and the executers' memory was 10 GB

TABLE 18. Frequent patterns for each minimum support threshold for accidents dataset.

Minimum support threshold (%)	Generated frequent patterns
1	4094267605
2	849626361
3	316307813
4	151219267
5	83153238
6	50023500
7	32028549
8	21484794
9	14933930
10	10691549

TABLE 19. Average running times of MPFP, BPFP and EBPFP algorithms in minutes on different numbers of cluster nodes in the Webdocs dataset.

Cluster Nodes	MPFP	BPFP	EBPFP
5	68.10	54.87	51.10
6	58.73	46.36	43.11
7	48.93	38.89	35.95
8	44.31	35.09	32.02
9	36.89	29.34	27.31
10	35.08	27.29	25.35
11	32.20	24.42	22.12
12	27.13	20.73	18.31
13	26.00	19.50	16.88
14	25.50	18.38	15.73
15	24.52	17.54	14.78

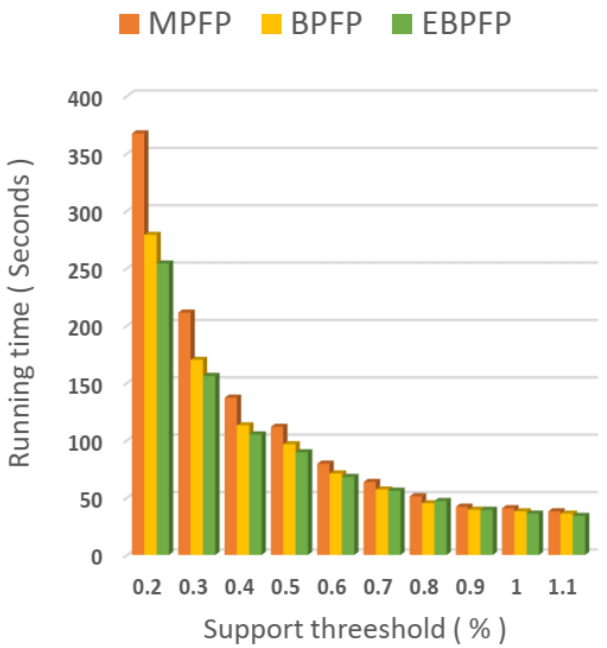

FIGURE 4. The running times of MPFP, BPFP and EBPFP algorithms in the kosarak dataset.

for all experiments. Fig.6 shows the average running times of the algorithms for different numbers of nodes ranging

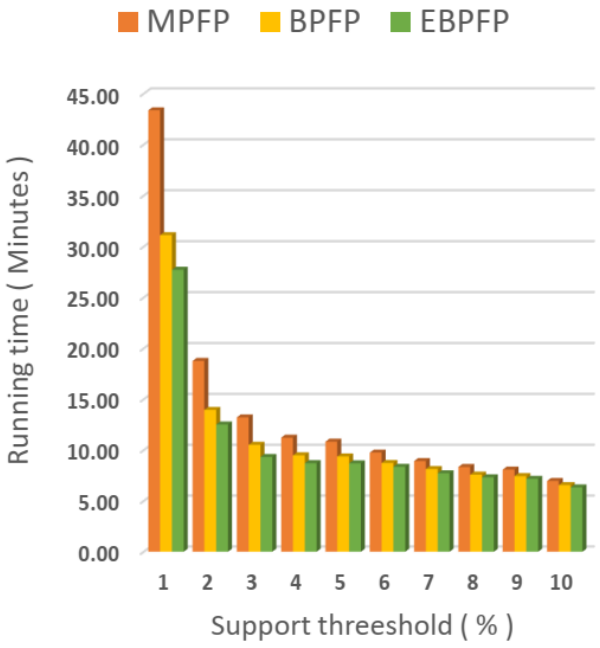

FIGURE 5. The running times of MPFP, BPFP and EBPFP algorithms in accidents dataset.

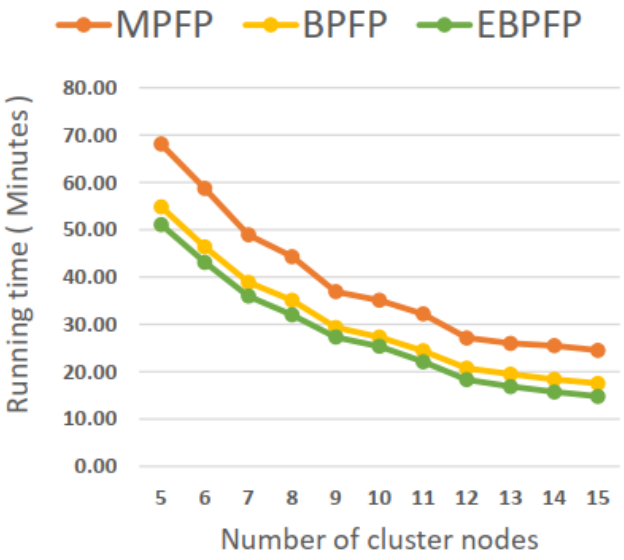

FIGURE 6. The running times of MPFP, BPFP and EBPFP algorithms in the webdocs dataset on different number of cluster nodes.

from 5 to 15, and Table 19 specifies the values for Fig.6. The experiments were run with a minimum support of 15%, which was the most suitable number for running all experiments without throwing memory exceptions. The number of frequent patterns generated for this minimum support was 10388. The results show that when the number of nodesincreases, the number of groups increases, which is determined by the number of cluster cores, and EBPFP shows better results. On five nodes, EBPFP was 24.96% faster than MPFP and 6.87% faster than BPFP, but on 15 nodes, EBPFP was 39.72% faster than MPFP and 15.74% faster than BPFP. On average, EBPFP was 30.6% faster than MPFP, and 9.95% faster than BPFP.

V. CONCLUSION

In a parallel and distributed processing environment, especially when the data size increases, balancing the load among the worker nodes becomes crucial and significantly affects the overall process performance. In this paper, an enhanced balanced parallel FP-growth algorithm is introduced, which proposes a load-balancing strategy that aims to divide items evenly across groups. Second, an enhanced method improves the construction of the conditional pattern base by removing infrequent items from the conditional patterns before building conditional FP-trees. Our results show that the EBPFP balancing strategy balances the load among groups better than the PFP and BPFP balancing strategies. The EBPFP outperformed the MPFP and BPFP.

Moreover, FP-Growth in distributed processing environments, similar to most mining algorithms, requires a large amount of data to be transmitted over the network. Therefore, bandwidth limitation is one of the main problems for FP-Growth, particularly in this era of big data. In the future, we can further improve the EBPFP by minimizing data transmission among the nodes, which will enhance the efficiency of the algorithm and reduce the execution time.

REFERENCES

[1] S. Kotsiantis and D. Kanellopoulos, "Association rules mining: A recent overview," *Science*, vol. 32, no. 1, pp. 71–82, 2006.
[2] Z. Wang and L. Xue, "A fast algorithm for mining association rules in image," in *Proc. IEEE 5th Int. Conf. Softw. Eng. Service Sci.*, Jun. 2014, pp. 513–516, doi: 10.1109/ICSESS.2014.6933618.
[3] J. Han, J. Pei, and Y. Yin, "1 Introduction," *Construction*, vol. 29, pp. 1–12, 2000.
[4] J. Dean and S. Ghemawa, "MapReduce: Simplified data processing on large clusters," in *Proc. 83rd Annu. Meeting Expand. Geophys. Frontiers Soc. Explor. Geophys. Int. Expo.*, 2004, pp. 2140–2144, doi: 10.1190/segam2013-1277.1.
[5] K. Grolinger, M. Hayes, W. A. Higashino, A. L'Heureux, D. S. Allison, and M. A. M. Capretz, "Challenges for mapreduce in big data," in *Proc. IEEE world Congr. Services*, Jun. 2014, pp. 182–189, doi: 10.1109/services.2014.41.
[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, and J. Ma, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implement.*, 2012, pp. 15–28.
[7] E. E. Drakonaki and G. M. Allen, "Magnetic resonance imaging, ultrasound and real-time ultrasound elastography of the thigh muscles in congenital muscle dystrophy," *Skeletal Radiol.*, vol. 39, no. 4, pp. 391–396, Apr. 2010, doi: 10.1007/s00256-009-0861-0.
[8] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-growth for query recommendation," in *Proc. ACM Conf. Recommender Syst.*, 2008, pp. 107–114, doi: 10.1145/1454008.1454027.
[9] J. Li, Z. Zhong, J. Z. Huang, and S. Feng, "Balanced parallel FP-growth with mapreduce institute of advanced computing and digital engineering," in *Proc. IEEE Youth Conf. Inf., Comput. Telecommun.*, 2010, pp. 243–246, doi: 10.1109/YCICT.2010.5713090.
[10] O. R. Zaiane, M. El-Hajj, and P. Lu, "Fast parallel association rule mining without candidacy generation," in *Proc. IEEE Int. Conf. Data Mining*, 2001, pp. 665–668, doi: 10.1109/icdm.2001.989600.
[11] I. Pramudiono and M. Kitsuregawa, "Parallel FP-growth on PC cluster," *Lect. Notes Artif. Intell.*, vol. 2637, pp. 467–473, Oct. 2003, doi: 10.1007/3-540-36175-8_47.
[12] A. Ghoting, "Cache-conscious frequent pattern mining on a modern processor," in *Proc. 31st Int. Conf. Very Large Data Bases*, vol. 2, 2005, pp. 577–588.
[13] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz, "Toward terabyte pattern mining: An architecture-conscious solution," in *Proc. Symp. Princ. Pract. Parallel Program.*, vol. 2, 2007, pp. 1–12, doi: 10.1145/1229428.1229432.
[14] Z. Z. Dawen Xia, Y. Zhou, and Z. Rong, "IPFP: An improved parallel FP-growth algorithm for frequent itemsets mining," in *Proc. 59th ISI World Stat. Congr.*, Aug. 2013, pp. 4034–4040.
[15] Q. Yang, F.-Y. Du, X. Zhu, and C.-G. Jiang, "Improved balanced parallel FP-growth with mapreduce," in *Proc. DEStech Trans. Comput. Sci. Eng.*, 2016, pp. 1–5, doi: 10.12783/dtcse/aice-ncs2016/5681.
[16] Y. Xun, J. Zhang, X. Qin, and X. Zhao, "FiDoop-DP: Data partitioning in frequent itemset mining on Hadoop clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 101–114, Jan. 2017, doi: 10.1109/TPDS.2016.2560176.
[17] Y. Miao, J. Lin, and N. Xu, "An improved parallel FP-growth algorithm based on spark and its application," in *Proc. Chin. Control Conf. (CCC)*, Jul. 2019, pp. 3793–3797, doi: 10.23919/ChiCC.2019.8866373.
[18] S. Bagui, K. Devulapalli, and J. Coffey, "A heuristic approach for load balancing the FP-growth algorithm on MapReduce," *Array*, vol. 7, Sep. 2020, Art. no. 100035, doi: 10.1016/j.array.2020.100035.
[19] P. Y. Taser, K. U. Birant, and D. Birant, "Multitask-based association rule mining," *Turkish J. Elect. Eng. Comput. Sci.*, vol. 28, no. 2, pp. 933–955, 2020, doi: 10.3906/elk-1905-88.
[20] P.-Y. Huang, W.-S. Cheng, J.-C. Chen, W.-Y. Chung, Y.-L. Chen, and K. W. Lin, "A distributed method for fast mining frequent patterns from big data," *IEEE Access*, vol. 9, pp. 135144–135159, 2021, doi: 10.1109/ACCESS.2021.3115514.
[21] *MLlib/Apache Spark*. Accessed: Aug. 3, 2021. [Online]. Available: https://spark.apache.org/mllib/
[22] *Frequent Pattern Mining—RDD-based API—Spark 3.1.2 Documentation*. Accessed: Aug. 4, 2021. [Online]. Available: https://spark.apache.org/docs/latest/mllib-frequent-pattern-mining.html
[23] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "WebDocs: A real-life huge transactional dataset," in *Proc. Workshop Freq. Itemset Min. Implementations*, 2004, p. 15.
[24] *Kosarak*. Accessed: Jul. 21, 2021. [Online]. Available: http://fimi.uantwerpen.be/data/kosarak.dat
[25] *Accidents*. Accessed: Jul. 20, 2021. [Online]. Available: http://fimi.uantwerpen.be/data/accidents.dat

AMR ESSAM was born in Cairo, Egypt, in 1994. He received the B.S. degree in information systems from Helwan University, in 2015, where he is currently pursuing the M.Sc. degree in data mining. He is also a Teaching Assistant at the Faculty of Computers and Artificial Intelligence, Helwan University.

MANAL A. ABDEL-FATTAH received the Ph.D. degree in information systems from the Faculty of Computers and Information, Cairo University. She worked as a Business Development Consultant at the Management National Institute and a Project Manager at the Ministry of State and Administrative Development. She is currently an Associate Professor with the Faculty of Computers and Artificial Intelligence, Helwan University. She has supervised many master's and Ph.D. theses. Her research interests include big-data analytics, data mining, and evaluation methodologies. She is a reviewer of many information systems journals.

LAILA ABDELHAMID received the bachelor's, M.Sc., and Ph.D. degrees in information systems from Helwan University, in 2005, 2011, and 2018, respectively. She is currently a Lecturer at the Faculty of Computers and Artificial Intelligence, Helwan University. Her research interests include data streaming, data mining, sentiment analysis, and software engineering.