Research Article

An Efficient Parallel Algorithm for Frequent Itemsets Mining Using BitTable on Spark

Manwika Kittipron* and Chuleerat Jaruskulchai
Department of Computer Science, Faculty of Science, Kasetsart University, Bangkok, Thailand

* Corresponding author. E-mail: manwika.k@ku.th DOI: 10.14416/j.asep.2022.01.005
Received: 16 July 2021; Revised: 27 September 2021; Accepted: 29 October 2021; Published online: 24 January 2022

Abstract
A variety of techniques have been used to improve the performance of an algorithm in finding frequent item sets, which is one of the important processes to obtain frequent pattern mining. It was found that today's technology has resulted in an ever-increasing amount of information, which should be analyzed for various benefits. Therefore, efforts have been made to improve the aalgoruthm's efficiency to accommodate the nature of data stored through the working process of the main internal memory. Efforts have been made to prepare algorithms for the ever-increasing information. This research provided an appropriate data structure of BitTable to help improve the functionality of the algorithms. Moreover, the principle of parallel frequent itemset mining algorithm based on Map-Reduce design was used in this research to assess the performance of algorithms, named as Adaptive Hybrid Parallel Algorithm (AHP). Additionally, to investigate the performance of the AHP Algorithm Using Apache Spark Technology with the type of data that was accumulated during the process of the main internal memory.

Keywords: Frequent pattern mining, Frequent itemsets mining, Parallel algorithm, Distributed computing, Various

1 Introduction

Finding frequent itemsets from the database list is a part of the processes of Frequent Pattern Mining. As the amount of data increases drastically, algorithms have to deal with big data accordingly. In the Big Data era, It is important to improve the performance of algorithms in finding frequent itemsets with particular types of information.

The classical frequent itemsets mining algorithm is mainly divided into two categories: the first is the discovered frequent itemsets to generate the candidate itemsets, count the candidate itemsets to find new frequent itemsets, Typical algorithms include Apriori, etc. The second is directly generates frequent itemsets through recursive traversal on data structure, and Typical algorithms include FP-Growth, etc. The classical frequent itemsets mining algorithm is executed on a single machine with data centralization. In A computation-intensive task, the size of search space is 2^n–1 when there are n items in the dataset that the computing load is extremely heavy. In the mining process, frequent itemsets mining needs to store specific data structures or candidate datasets in memory [1]–[3]. Thereby, parallel frequent itemsets mining [4], which seamlessly integrates parallel computing, has been widely used in various applications.

In recent years, the Map-Reduce method was found to be capable of parallel work. Accordingly, the PFIMD algorithm [4] designed an optimization parallel frequent itemset mining algorithm based on a Map-Reducing programming model to find the association rule on the Apache Hadoop Technology. It is one of the frameworks that help to distribute big data processing on a larger scale of networking computers. There is another model called "Hadoop Distributed File System" (HDFS) [5], which helps store data in a way that can be quickly accessible. It reduces storage space, allows faster processing and automatic backup on disk drives. By this means, data is not stored on the memory drive at all. Under this setting, when any nodes on computers malfunction, other nodes can continue working on the information on the disk. This method has been utilized

to help improve the performance of algorithms in the calculation of frequent itemsets. At present, Apache Spark is a popular technology capable of both parallel and distributed computing processes. Accordingly, the SARSO algorithm [6] investigated the benefits of Spark's parallel and distributed computing environment to futher improve efficiency by reducing the shuffle overhead caused by RDD operations at each iteration. HBPFP-DC algorithm [7] on Spark platform to define node computation workload estimation model and to realize the balanced grouping of the calculation tasks among computing nodes, based on the problems presented above.

Because of of this, we present the design and implementation of the AHP Algorithm using the Map-Reduce Principle on Apache Spark technology. Moreover, data structure of BitTable was adjusted to boost mining efficiency, by helping to expand the working pattern and and to increase the efficiency of the Map-reduce technique by increasing the speed of the clusters on a computer's memory, which resulted in a faster processing time.

The contributions of this paper are mainly focused in three aspects. Firstly, discussion about the main ideas of AHP algorithm design, proposed paradigms of AHP algorithm, and the paradigms from the aspects of processing speed and characteristics of the dataset. Secondly, the implementation of paradigms using Map-Reduce Principle on Spark technology was explained and algorithm for direct implementation of paradigms named Adaptive Hybrid Parallel Algorithm (AHP) was proposed. Thirdly, the algorithm's performance such as processing speed and characteristics of the dataset were analyzed through experiments.

1.1 *Reduced-Apriori: R-Apriori*

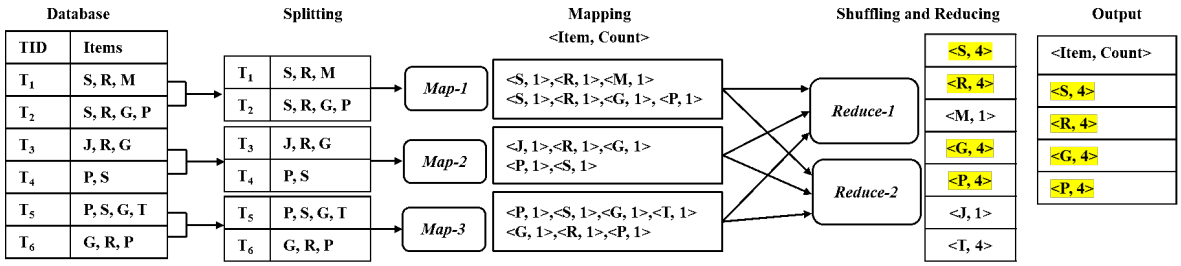
The R-Apriori algorithm utilizes an intersection principle [8] to reduce the redundancy features with a data frequency greater than or equal to the minimum support value. This reduction lowers the database needed for the creation of candidate datasets. The R-Apriori algorithm was developed from the YAFIM algorithm [9], further developed from the Parallel Apriori. It has two main operational processes. The first step involves creating a dataset with an item length at a value of one (1-frequent itemset), also known as Singleton Frequent Items. The Minimum Support value is used for the selection of frequent itemsets. The Map-Reduction method explains the working process of algorithms (a)–(c) as shown in Figure 1. In the second step, the Singleton Frequent data obtained in the first step, including data sets; S, R, G, P as shown in Figure 1(a), is intersected with the entries in the database. This process reduces the number of database entries needed to create the candidate dataset as shown in Figure 1(b) and (c). Based on the data transitions T1, items S, R, and M are included. When these items are intersected with the items in Singleton Frequent data (items, S, R, G, P), only items S, R, remain in T1 transitions. When this intersection system is applied with the Apache Spark technology, in which data is stored during the computing process within the main memory units, the algorithm's performance can be raised with a faster operation time.

1.2 *Distributed frequent itemset mining algorithm: DFIMA*

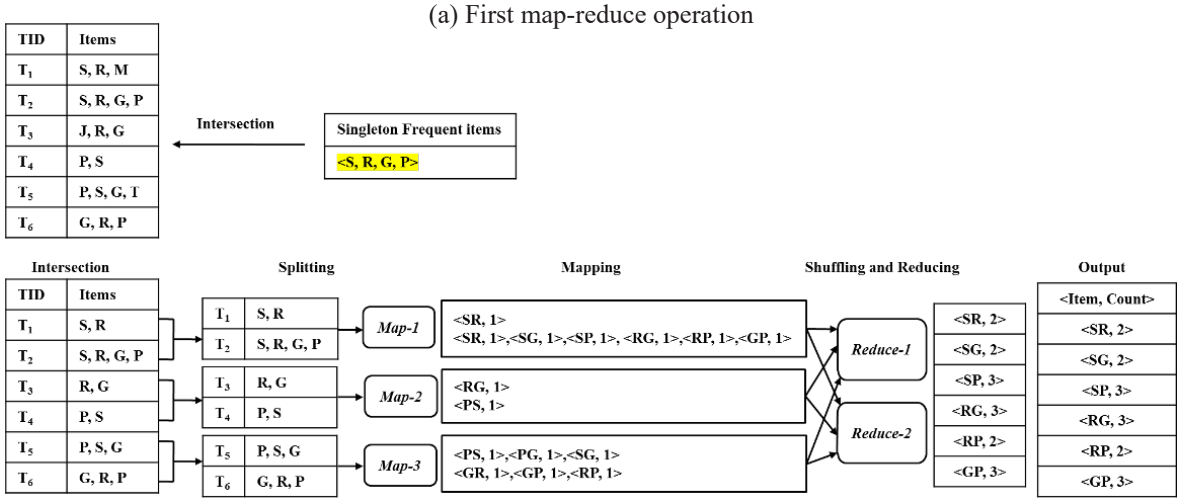
Another algorithm used for the mining of frequent itemset is the matrix-based pruning method. It is found to reduce the time required for calculating candidate datasets and the re-reading of information within the database. The distributed frequent itemset mining algorithm (DFIMA) [10] is used in the mining of the frequently accumulated datasets. The method is used for the improvement of the fundamental apriori algorithm. The implementation of the DFIMA algorithm begins by creating a Singleton frequent item (1-frequent itemset) based on the principle of the Map-Reduce process as shown in Figure 1(a). The frequent singleton itemsets obtained in this process are S, R, G, P. Figure 2 shows Singleton Frequent Itemsets to create vector Boolean from the database in each transaction. Particularly in this process, the entries in each transaction with the Singleton frequent itemset are represented by 1, and those without the Singleton frequent itemset are represented with 0. This is to reduces the amount of data before calculating the candidate dataset, as shown in Figure 2. The next step is to transform the data in Boolean vectors to a matrix (2-itemset matrix) to generate a candidate dataset.

1.3 *Map-reduce operation on apache spark technology*

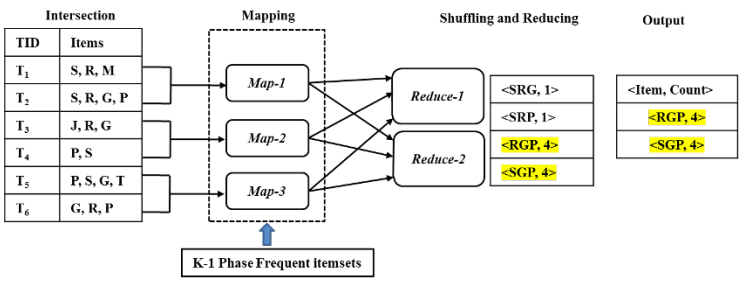
Apache Spark technology is an open-source technology capable of handling big data [11]. It has two operational



(a) First map-reduce operation



(b) Second map-reduce operation



(c) Third map-reduce operation

Figure 1: Map-Reduce system of the R-Apriori algorithm, in which; (a) shows a process for the making of Singleton frequent items at a length of 1-frequent itemsets, (b) shows a process for the making of Singleton frequent items at a length of 2-frequent itemsets using intersection technique, and (c) shows a process for the making of Singleton Frequent Items at a length of k (k-1frequent itemsets) in the creation of dataset (k + 1) frequent itemsets (L3) [8].

parts. The first one is called resilient distributed datasets (RDD), which store data in the main memory unit. The second is the Map-Reduce portion for data processing. The map-reduce in Apache Spark Technology has two functions: Flat Map Function and Map Function, as shown in Figure 3, which help to convert data into a form of <Key, Value>. The converted data is sent to the groupByKey function, which works similarly to the Reduce function, to combine data values with the same key together. The filter function is processed to obtain the desired results [5], [6]. These processes are operated on the main memory.

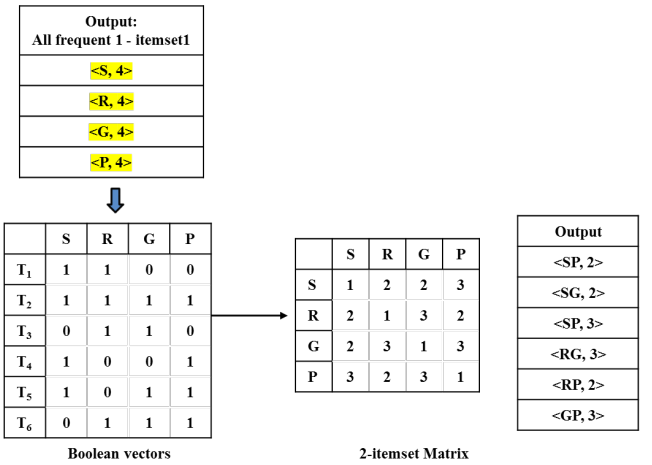


Figure 2: The functionality of the DFIMA algorithm and the use of the Singleton frequent itemset to create Boolean vectors for candidate datasets creation [10].

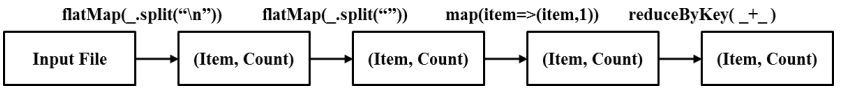


Figure 3: Main functioning of Map-reduce algorithm [11].

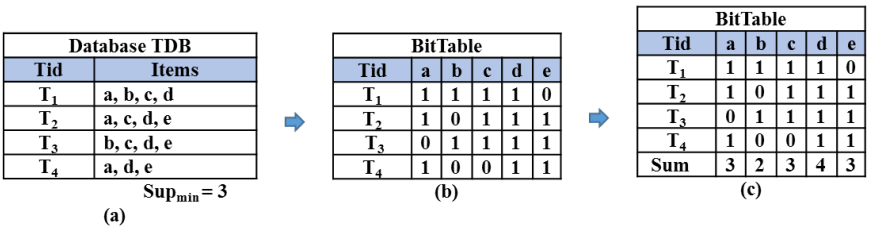


Figure 4: Illustrative example for a transactional dataset and the binary matrix representation.

1.4 *BitTable representation of data*

Data structures related to frequent itemsets are stored in sparse matrices, and vector multiplications are used to calculate the support of the potential k+1 itemsets. BitTable structure is used compresses the datasets horizontally and vertically for quick candidate itemsets generation and support count, which bitwise operations are used in place of the item position during the information gathering process. The key idea is to store the data related to a given itemset in a binary vector. The bitmaps of frequent itemsets are generated based on the binary vector's elementwise products corresponding to the building k-1 frequent itemsets [12]. The processing time was minimized when the mining of the frequent itemsets was implemented via the matrix method.

An illustrative example for D transactional database is shown in Figure 4(a). The transactional database can be transformed into a bitmaps matrix as shown in Figure 4(b) representation, where if an item i = 1,…, m appears in transaction Tj , j = 1,…,N , the bit i of the j-th row of the binary incidence matrix will be marked as one. As the support of an itemset is a percentage of the total number of transactions, the Summary of the columns of the B^0_{Nxn} matrix represents the support of the j = 1,…,n items is shown in Figure 4(c). Therefore, if b^0_j represent j-th column of B^0_{Nxn}, which is related to the occurrence of the i_j-th item, then the support of the i_j item can be calculated as Equation (1)

$$Sup(X = i_j) = (b^0_j)^T b^0_j / N \quad (1)$$

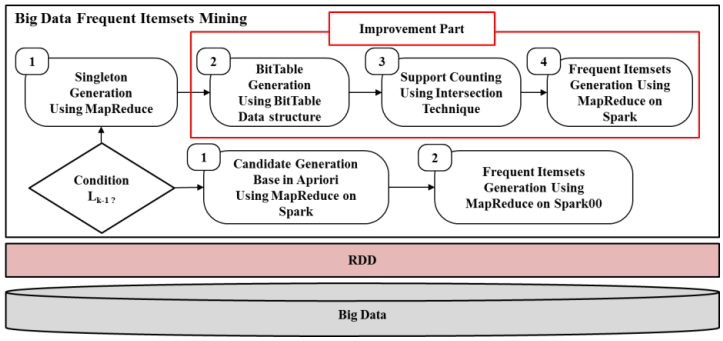


Figure 5: Adaptive hybrid parallel algorithm with the application of Map-Reduce method on the RDD architecture.

Similarly, the support of an $X_{i,j} = \{i_i, i_j\}$ itemset can be easily calculated by a simple vector product of the two related bit vectors, since when both i_i and i_j items appear in a given transaction, the product of the two related bits can represent the AND connection of the two items [Equation (2)]:

$$\text{Sup}(X_{i,j} = \{i_i, i_j\}) = (b^0_j)^T b^0_j / N \quad (2)$$

The matrix representation allows the effective calculation of all of the itemsets [Equation (3)]:

$$S^2 = (B^0)^T B^0 \quad (3)$$

Where the i,j-th element of S^2 matrix represents the support of the $X_{i,j} = \{i_i, i_j\}$ 2-itemset. The upper triangular element of this symmetrical matrix has to be checked, whether the $X_{i,j} = \{i_i, i_j\}$ 2-itemsets are frequent or not.

1.5 *Proposed method: Adaptive hybrid parallel algorithm*

This research study proposed an approach for improving the efficiency of algorithms based on frequency pattern mining. The techniques used for enhancing the algorithm performance included reducing the creation of the candidate datasets and the number of re-reading cycles. The data structure is presented in BitTable based on the processes in Figure 5.

1.6 *Design of adaptive hybrid parallel algorithm based on Map-Reduce method*

The adaptive hybrid parallel algorithm based on applying the Map-Reduce method on Apache Spark technology has two main functions. Part 1 (Phase 1): This phase, as exemplified in Algorithm 1 in Figure 6, involves finding Singleton frequent 1- Itemset. In this step, Map-Reduce processes one cycle of work and uses the minimum support value to select a Singleton frequent itemset. The results are stored in the RDD with the construction of MinHashingFI data to be used for the creation of a candidate dataset and the support information of the candidate dataset.

Part 2 (Phase 2): In this step, candidate dataset is created from a Frequent k–1 itemset, which is similar to algorithm Apriori that creates a candidate dataset with the item lengths of k and Ck from the frequent itemsets with the item lengths of k–1, and Lk–1. However, the method for creating and counting support values for the candidate datasets is different (Modified approach).

2 Materials and Methods

2.1 *Singleton frequent items algorithm (Frequent 1-itemsets) on spark*

The adaptive hybrid parallel algorithm searches for a frequent singleton dataset from the entries in the transaction lists from an extensive database using a Map-Reduce method. It was found in many search studies that the Map-Reduce method had been used effectively to search for singleton frequent items in large databases.

Transactions stored at HDFS are loaded into the Spark RDD as input for the singleton frequent item search, as shown in Algorithm 1 in Figure 6. The input data is broken down and distributed to every working node. The flatMap function is used for every

Algorithm 1: Phase I – Singleton Frequent items

Input: Load the transactional Dataset D from Input file into a cached RDD

OutPut: Singleton Frequent item L_1

```
1.  Procedure SINGLETON -GEN
2.    For each Transaction T ∈ D  do
3.     flatMap (line offset, T)
4.     For each item I ∈ T  do
5.          Yield (I, 1)
6.     End flatMap
7.     storeAtRDD1
8.     RDD2 = RDD1.reduceByKey
9.     For each tuple t ∈ RDD2  do
10.      flatMap (I, count)
11.      If (count < minSup) then
12.         Yield (I, 1)
13.      End flatMap
14.      StoreAtRDD3
```

Figure 6: Singleton frequent itemset generating algorithm.

transaction (Algorithm 1, Line 2). Each transaction is executed and put in a <key, value> dyad format where the value equals integer 1 (Algorithm 1, Lines 4–6). Data is stored at RDD in the working memory section. Then, the reduceByKey function combines values with the same keys and discards the impediment that does not pass the specified minimum support (Algorithm 1, Line 8–9). The results from this operation are more minor size data that is later stored in the RDD. Figure 6 shows the functioning of the program in finding the Singleton frequent itemset (Frequent 1- itemsets).

2.2 *The processes for searching of Singleton frequent items (Frequent 1-itemsets) on spark*

Figure 7 portrays a process of Singleton Frequent Itemset searching using the minimum support rate of 33% (with at least two cases of minimum support found in transactions in the database). Transactions stored in HDFS are loaded as inputs to SparkRDD before the flatMap divides and subsections them to all mappers nodes. As exemplified in Figure 7, MAP-1 receives transaction data T1 and T2 within each node. The flatMap function is used for every transaction, and each transaction is paired (RDD.split) in the <key, value> format, where the value is equal to integer 1. As shown in Figure 7, the T1 transaction consists of (S, R, M). These data are matched as <S, 1>, <R, 1> and <M, 1>, and stored in RDD in the main memory. All working processes are performed on the RDD.

The next working process involves the reduce byKey function in combining the values with the same key (RDD.reduceByKey) and discarding the results that fail to meet the required minimum support, resulting in smaller sizes of data to be stored it in the RDD.

2.3 *The algorithm for generated MinHashingFI table by Singleton frequent items*

The processes for creating the MinHashingFI table in Algorithm 2 (Figure 8) are implemented to help improve the operation of the algorithm (Modified approach). The process starts from assigning each of the singleton frequent items in each of the transactions in the database with a bit value of 1. Any transaction without the singleton frequent items is represented with a value of 0. For this instance, the MinHashingFI value is the numerical figure obtained from the representation of the item's number in bit value. Each MinHashingFI is then stored in SparkRDD in the main memory unit.

The working process of an adaptive hybrid parallel algorithm starts with importing the input that will be used to create a candidate dataset. The imported

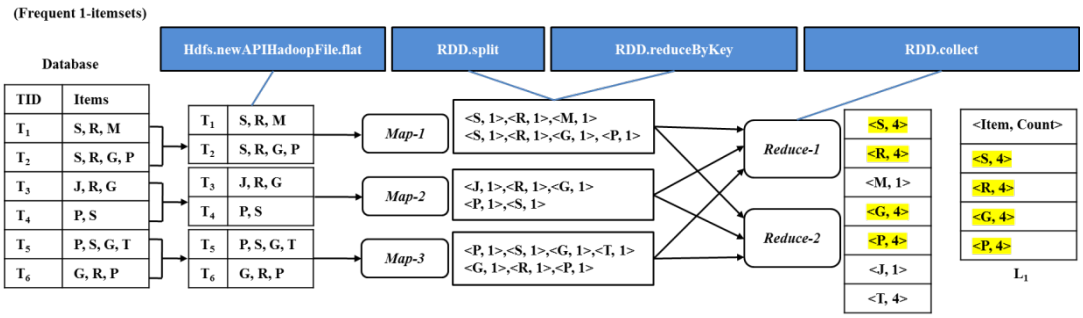


Figure 7: Processes for the searching of Singleton frequent itemsets.

Algorithm 2 : MinHashing FI

Input: k : MinHashing parameter minSup : min support e: tolerance error
Output: all the frequent L

```
1.  Freq_set = build_invert_list(dataset)
2.  δ = 1/√k
3.  Matrix = build_signature_matrix(fre_set,k)
4.  L = L U fre_set
5.  L = L U HashingAdapTive Hybrid(L, matrix)
6.  HashingAdapTive Hybrid(L)
7.      Lk = ∅
8.    For Xi ∈ L do
9.    For Xj ∈ L do
10.   |TIDset(R)|= Calculate(TIDset(Xi), TIDset(Xj))
11.     If |TIDset(R)| >=minSup do
12.       L = L U R, Lk = Lk U R
13.     End if
14.   End for
15.   End for
16.   If (Lk ≠ ∅) then
17.     L = L U HashingAdapTive Hybrid(Lk);
18.   Else return L
19.   End if
20. End HashingAdapTive Hybrid
```

Figure 8: BitTable algorithm and the creation of BitTable and a candidate dataset.

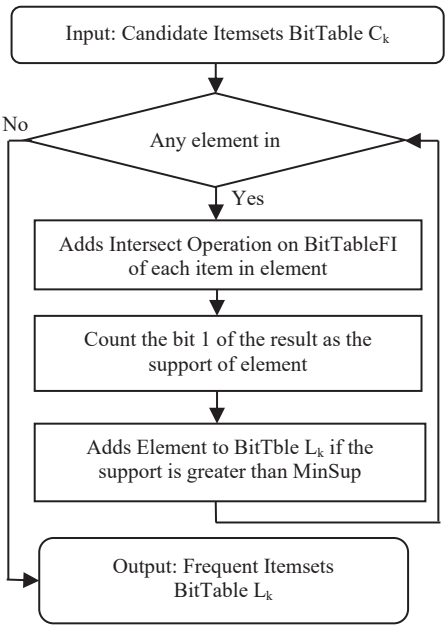


Figure 9: Procedures for the counting of support information of candidate data in MinHashingFI table.

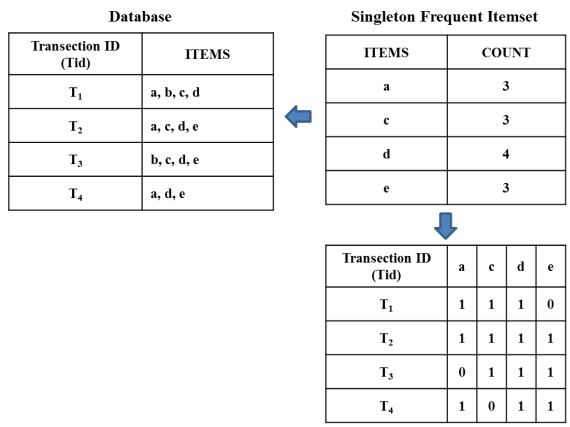


Figure 10: Creating MinHashingFI with singleton frequent itemset.

input is represented by bit values derived from the representation of the item position with the number in the MinHashingFI table. For example, L1 consists of 4 items {S, R, G, P}. Therefore 4 positions of bits are used to replace each of these four items. Item S is represented in the first-bit position with values of 1 and 3. Other remaining positions are equal 0 items. Next is R, the first bit's position is placed with 0, the second bit with 1, and the other 2 remaining positions with 0. This pattern of practice is applied with all of the remaining candidate datasets. The MinHashingFI table, used for collecting candidate data, will collect the following information: items information, support value (Count) and bitset position. Support value of the data set is counted to show which dataset is frequent itemsets. Item support counting can be done at the bit operation level using the data in the MinHashingFI table. More specifically, there is an intersection of the positions with a bit value of 1. The working process for counting support values in the MinHashingFI table is shown in Figure 9.

2.4 *The processes for MinHashingFI generation by Singleton frequent items*

In order to gain comprehension about the flow in the making of MinHashingFI, data, and singleton frequent itemset in Figure 10 are used to exemplify the making process of MinHashngFI. The sample set of data consists of <a, 3>, <c, 3>, <d, 4> and <e, 3>, the minimum support value is set at 75% (this rate must be found in at least 3 transactions in the database).

Step 1 begins with assigning each of the singleton frequent items containing <a, 3>, <c, 3>, <d, 4> and <e, 3> that appear in each transaction in the database

Transection ID (Tid)	a	c	d	e
T_1	1	1	1	0
T_2	1	1	1	1
T_3	0	1	1	1
T_4	1	0	1	1

	a	c	d	e
T_1	1	1	1	0
T_2	1	1	1	1
T_3	0	1	1	1
T_4	1	0	1	1
σ(X)	3	3	4	3

ITEMS	COUNT	BITSETS
a	3	1101
c	3	1110
d	4	1111
e	3	0111

Figure 11: Assigning bitsets position to items.

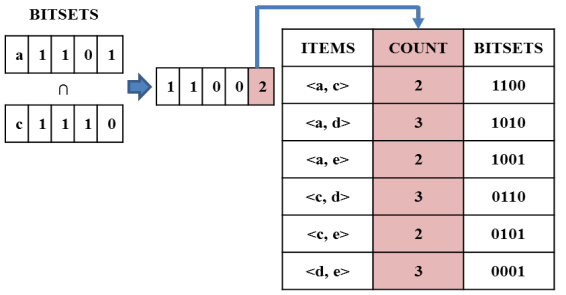


Figure 12: Example of the counting the support value of the candidate dataset in MinHashingFI table.

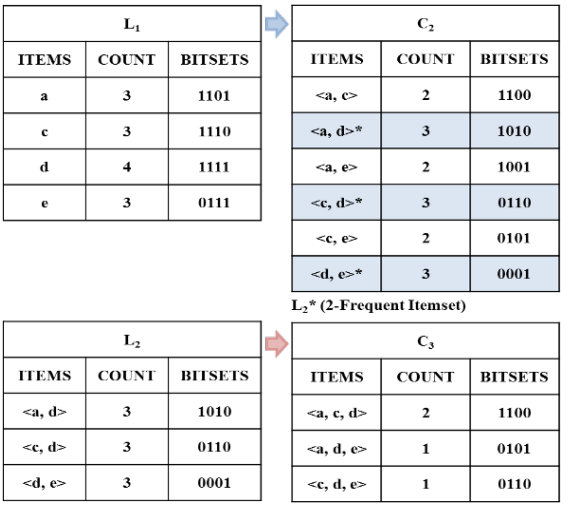
with a bit value of 1. On the other hand, any of the transactions in the database without a singleton frequent item will be represented as 0. For this instance, the MinHashingFI value is the numerical value obtained from the set of bits representing each item in the database. The MinHashingFI is later stored in Spark RDD for processing in the core memory.

Step 2 involves importing the singleton frequent items from RDD to be used as data to create of a candidate dataset, using a Map-Reduce method. Figure 10 shows the creation of a candidate dataset by means of frequent itemsets combination.

Step 3 is the counting of support of the candidate dataset to verify which candidate dataset is a frequent itemset information. A bitwise operation or the intersection of the bits with the value of 1 in the MinHashingFI table are performed. The counting processes of support value in the MinHashingFI table is represented in Figure 11, while Figure 12 shows an example of the counting the support value of the candidate dataset in the MinHashingFI table that the support counting in demo information.

2.5 *The frequent itemsets generation (Frequent k-itemsets)*

In Part 2 (Phase II) in Algorithm 3 (Figure 14), a candidate



L_1

ITEMS	COUNT	BITSETS
a	3	1101
c	3	1110
d	4	1111
e	3	0111

C_2

ITEMS	COUNT	BITSETS
<a, c>	2	1100
<a, d>*	3	1010
<a, e>	2	1001
<c, d>*	3	0110
<c, e>	2	0101
<d, e>*	3	0001

L_2* (2-Frequent Itemset)

L_2

ITEMS	COUNT	BITSETS
<a, d>	3	1010
<c, d>	3	0110
<d, e>	3	0001

C_3

ITEMS	COUNT	BITSETS
<a, c, d>	2	1100
<a, d, e>	1	0101
<c, d, e>	1	0110

Figure 13: Creating of candidate data for each round by combining the free itemsets.

dataset is created from the Frequent k–1 itemsets. The result of this creation is a candidate dataset with the lengths of Item k and Item Ck, which are the item lengths in Item k–1 (Lk–1). However, the counting of the supporting value to verify which dataset is a frequent itemset is not obtained from repeated reading from the database but the MinHashingFI table. This can be done by using bitwise operations. More specifically, the items with a bit-value of 1 are intersected. Figure 13 shows an example of the counting of support values of candidate datasets in the MinHashingFI table. An approach was proposed to improve the algorithm's functionality.

Algorithms are designed to choose to use an Apriori algorithm for an assessment, using either the data from the imported input or the size of the frequent item Lk–1. The Apriori algorithm is capable of processing small size data and fewer item sets. This type of algorithm stores information with a HashTree's system, which can increase work flexibility.

3 Results and Discussion

3.1 *Experimental setup*

The algorithm runs on the Apache Spark technology version 2.1.0 consists of 3 node clusters. It is installed in the Centos 7.0, with a memory size of 8 GB and a 1TB hard disk. Three groups of datasets from UCI

Algorithm 3: Phase II – Frequent k –itemset generation

Input: Load the transactional Dataset D from Input file into a cached RDD, Frequent k-1 itemset L_{k-1}
OutPut: Frequent k-itemsets L_k

```
1.  Procedure FREQUENT –GEN
2.    If (L(k-1).size is large) then
3.      L(k-1).storeInBitTable
4.      for each Frequent k-1 itemset L(k-1) ∈ T  do
5.        flatMap (line offset, T)
6.        B_T = Intersaction (C_k, L(k-1))
7.        Count_k = Pair (B_T)
8.        end flatMap
9.        storeAtRDD1
10.   Else if (L(k-1).size is less) then
11.     C_k = CANDIDATE –GEN (L(k-1))
12.     for each Transaction T ∈ D  do
13.       flatMap (line offset, T)
14.       C_T = subset (C_k, T)
15.     for each item c ∈ C_T  do
16.       Yield (c, 1)
17.       end flatMap
18.       storeAtRDD1
19.   RDD2 = RDD1.reduceByKey
20.   For each tuple t ∈ RDD2  do
21.     flatMap (c, count)
22.     If (count < minSup) then
23.       Yield (c, count)
24.     End flatMap
25.     StoreAtRDD3
```

Figure 14: Algorithm for the creation of the singleton frequent items.

and IBM were used. An appropriate minimum support value counting for each dataset [13] is as show in Table 1.

3.2 *Processing speed results*

The processing time of Three algorithms on six datasets with various minimum support is presented in Figures 15–17. The experimental results show in Figures 15(a), 16(a), and 17(a) that AHP Algorithm was found to have had the best processing time per each round of performance in the sparse dataset. This is because AHP Algorithm exploits the bit table for its data structure, at which Bit values are used in place of the item position during the information gathering process. Using intersect two-bit vector for counting the support and Bittable is used to store data in memory. On the other hand, DFIMA Algorithm [10] has to build large matrices using FP-tree for generating frequent itemsets. It then joins pairs of bit-vectors using AND operation and computes the support. At the same time, its processing unit is located separately in the main memory unit. This results in a significant difference in the data processing time. Even though the AHP algorithm has a relatively similar working process to that of the DFIMA Algorithm, the AHP algorithm uses a bittable where a bit value is used in place of the item location during the data collecting process, it was found that the application of bit value was able to reduce the number of data readings from several times of reading to only one time reading. It also helped reduce the amount of memory required for data entry during processing. Moreover, the intersection process helped reduce the amount of duplicated data in frequentitemsets findings. The processing time was minimized when the mining of the frequent itemsets was implemented via the matrix method.

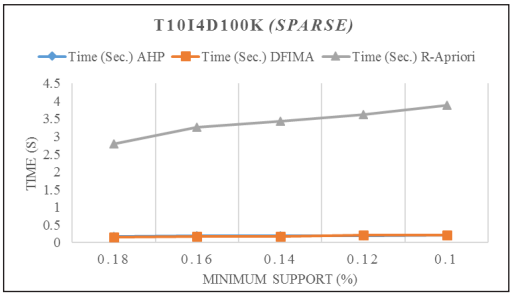
3.3 *Characteristics of the dataset results*

The characteristics of the dataset are diversified because of various factors such as the characteristics ranging from very sparse to very dense, the size and number of items, the average number of items, the number of transitions, the data density, and the similarity of information. For the sparse datasets, low-density, and

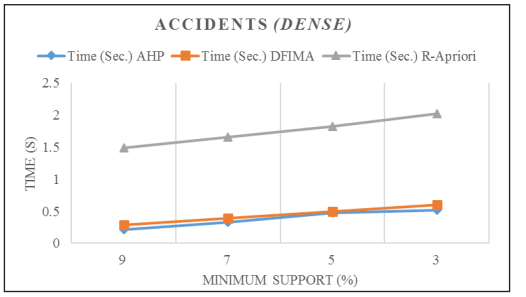
Table 1: Details of datasets used in the experiment

Dataset	Items	Items per Transection	Transection	Density (%)	Similarity
Small Dataset					
Chess	75	37	3196	49.33	0.3148
Food mart	1559	4.4	4141	0.28	0.292
Average Dataset					
Connect	129	43	67555	33.33	0.1626
Retail	16470	9.8	88163	0.06	0.0094
Large Dataset					
T10I4D100K	870	10	100000	1.15	0.0137
Accidents	468	33.8	340183	7.22	0.0248
* Density (%) = (Average Transaction Length / Number of Items) × 100					

The results of Comparing the performances of algorithms (Experiment summary)

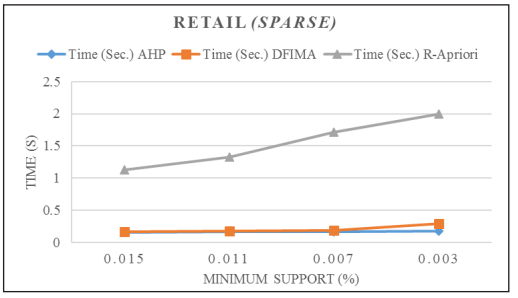


(a)

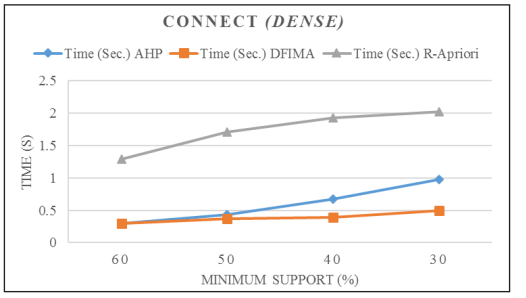


(b)

Figure 15: Performance of algorithm in LARGE DATASET group, in which; (a) shows T10I4D100k in sparse dataset. (b) shows ACCIDENTS in dense dataset.

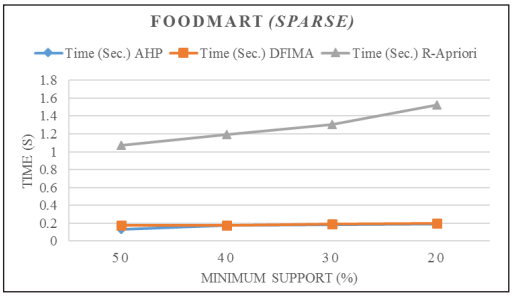


(a)

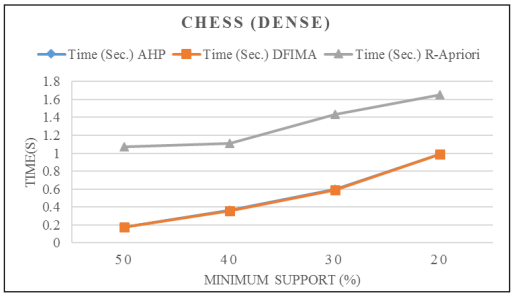


(b)

Figure 16: Performance of algorithm in Average Dataset group in which; (a) shows RETAIL in sparse dataset. (b) shows CONNECT in dense dataset.



(a)



(b)

Figure 17: Performance of algorithm in dataset Chess, Small Dataset group in which; (a) shows FOODMART in sparse dataset. (b) shows CHESS in dense dataset.

a large amount of data, the AHP algorithm is a better choice because the AHP algorithm approach does not require generating a large number of infrequent candidate itemset, as shown in Figures 15(a), 16(a) and 17(a). Similarly, under this context, for the dense datasets, the AHP algorithm is approximately processing time with DFIMA Algorithm [10], as shown results in Figures 15(b), 16(b), and 17(b). Even though that the AHP algorithm has a relatively similar working process to the DFIMA Algorithm, the AHP algorithm uses a bittable where a bit value is used in place of the item location during the data collecting process. It was found that the application of bit value was able to reduce the number of data readings from several times

of reading to only one time reading. It also helped reduce the amount of memory required for data entry during processing. Moreover, the intersection process helped reduce the amount of duplicated data in frequent itemsets findings. The algorithms take a different time to process for each cycle of the dataset with low density. This is because when the number of items is large, but the average of items per transaction is small, the possibility of having singleton frequent items is rare. Under this circumstance, however, it takes a longer time to intersect each dataset, resulting in a longer processing time for R-Apriori Algorithm [8].

4 Conclusions

This article investigates the performances of the three types of the R-Apriori Algorithm [8], DFIMA Algorithm [10], and AHP Algorithm on low-density databases with data similarity. The development of an algorithm was implemented using Map-reduce Principle on Apache Spark technology. It was observed that the AHP algorithm could work more efficiently than its counterparts on all types of information settings. This is because the AHP Algorithm does not re-read the data in the database, resulting from the singleton frequent items in each of the 1-Frequent itemsets being converted into bittable where the item position is replaced with a bit value during the data procession. The application of bitTable system has been found to be able to; reduce the number of data readings to just one time, reduce the memory space required during data processing to generate candidate data prior to using this candidate data to search for the next level of the dataset (k + 1 frequent itemsets).

DFIMA Algorithm was found to have had a similar speed to AHP Algorithm when working on a low-density database. R-Apriori Algorithm was found to have a low level of performance because it required multiple database readings for the creation of a candidate dataset, which is another step in finding the frequent items in each round. This research study was based on Apache Spark technology, which is now a prevailing technique for the development of an algorithm's performance. It is important for this kind of study to be conducted with a larger database context to validate the efficiency of the RDD architecture and find an add-up on the performance development of the DFIMA Algorithm. It is important to stay open to a new data structure for the enhancement of an algorithm capacity.

References

[1] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *2013 IEEE International Conference on Big Data*, 2013, pp. 111–118.

[2] D. C. Anastasiu, J. Iverson, S. Smith, and G. Karypis, "Big data frequent pattern mining," in *Frequent Pattern Mining*. Switzerland: Springer, 2014, pp. 225–259.

[3] W. Xiao and J. Hu. "Paradigm and performance analysis of distributed frequent itemset mining algorithms based on Mapreduce," *Microprocessors and Microsystems*, vol. 82, p. 103817, 2021.

[4] M. Yimin, G. Junhao, D. S. Mwakapesa, Y. A. Nanehkaran, Z. Chi, D. Xiaoheng, and C. Zhigang, "PFIMD: A parallel MapReduce-based algorithm for frequent itemset mining," *Multimedia Systems*, vol. 27, pp. 709–722, 2021.

[5] Apache Hadoop, "Open-source software for reliable, scalable, distributed computing," 2021. [Online]. Available: http://hadoop.apache.org/docs/

[6] S. Raj, D. Ramesh, and K. K. Sethi, "A Spark-based Apriori algorithm with reduced shuffle overhead," *The Journal of Supercomputing*, vol. 77, pp. 133–151, 2021.

[7] Y. Xun, J. Zhang, H. Yang, and X. Qin, "HBPFP-DC: A parallel frequent itemset mining using spark," *Parallel Computing*, vol. 101, p. 102738, 2021.

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

[9] H. Qiu, R. Gu, C. Yuan, and Y. Huang, "YAFIM: A parallel frequent itemset mining algorithm with spark," in *2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, 2014, Art. no. 13872289.

[10] F. Zhang, M. Liu, F. Giu, W. Shen, A. Shami, and Y. Ma, "A distributed frequent itemset mining algorithm using Spark for big data analytics," *Cluster Computing*, vol. 18, no. 4, pp. 1493–1501, 2015.

[11] T. S. and R. Nagarajan, "Spark based distributed frequent itemset mining technique for big data," *International Journal of Advanced Research in Engineering and Technology*, vol. 11, no. 10, pp. 1800–1814, 2020.
[12] J. Abonyi, "A novel bitmap-based algorithm for frequent itemsets mining," in *Computational Intelligence in Engineering*. Germany: Springer, 2010, pp. 171–180.
[13] FIMI, "Frequent itemset mining dataset repository," 2021. [Online]. Available: http://fimi.ua.ac.be/data