IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

SPECIAL SECTION ON SCALABLE DEEP LEARNING FOR BIG DATA

Received November 28, 2019, accepted December 11, 2019, date of publication December 16, 2019, date of current version December 26, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2959878

TRICE: Mining Frequent Itemsets by Iterative TRimmed Transaction LattICE in Sparse Big Data

MUHAMMAD YASIR[1], MUHAMMAD ASIF HABIB[1], MUHAMMAD ASHRAF[2], SHAHZAD SARWAR[3], MUHAMMAD UMAR CHAUDHRY[4], HAMAYOUN SHAHWANI[5], MUDASSAR AHMAD[1], AND CH. MUHAMMAD NADEEM FAISAL[1]

[1]Department of Computer Science, National Textile University, Faisalabad 37610, Pakistan
[2]Department of Computer Engineering, Balochistan University of Information Technology, Engineering, and Management Sciences, Quetta 87100, Pakistan
[3]Punjab University College of Information Technology, University of the Punjab, Lahore 54000, Pakistan
[4]Department of Computer Science, National College of Business Administration and Economics, Multan 60000, Pakistan
[5]Department of Telecommunications, Balochistan University of Information Technology, Engineering, and Management Sciences, Quetta 87100, Pakistan

Corresponding author: Mudassar Ahmad (mudassar@ntu.edu.pk)

ABSTRACT Sparseness is often witnessed in big data emanating from a variety of sources, including IoT, pervasive computing, and behavioral data. Frequent itemset mining is the first and foremost step of association rule mining, which is a distinguished unsupervised machine learning problem. However, techniques for frequent itemset mining are least explored for sparse real-world data, showing somewhat comparable performance. On the contrary, the methods are adequately validated for dense data and stand apart from each other in terms of performance. Hence, there arises an immense need for evaluating these techniques as well as proposing new ones for large sparse real-world datasets. In this study, a novel method: Mining Frequent Itemsets by Iterative TRimmed Transaction lattICE (TRICE) is proposed. TRICE iteratively generates combinations of varying-sized trimmed subsets of I, where I denote the set of distinct items in a database. Extensive experiments are conducted to assess TRICE against HARPP, FP-Growth, optimized SaM, and optimized RElim algorithms. The experimental results show that TRICE outperforms all these algorithms both in terms of running time and memory consumption. TRICE maintains a substantial performance gap for all sparse real-world datasets on all minimum support thresholds. Moreover, assessment of memory use of optimized SaM and RElim algorithms has been performed for the first time.

INDEX TERMS Association rules, big data applications, data mining, frequent itemset mining, pattern recognition, pervasive computing.

I. INTRODUCTION

The mining of association rules is regarded as one of the leading problems in data mining. It is used to uncover obscured, interesting relationships in large databases. The task is to be aware of associations occurring among different things in such a way that the presence of some elements in a transaction is implied by the presence of other elements. The maidan use of association rule mining in retail industry has achieved fruitful results. It has provided assistance to companies in making wiser business decisions such as offering items on sale, bundling items together, and formulating marketing strategies [1]. At present, the exceptional power of mining and furnishing deep insights of data has made association rule mining a necessary tool. It is currently used in a number of fields such as analyzing market basket data [2], IoT services and infrastructure [3]–[5], smart home [6]–[8], smart retail [9], mining of data streams [10], mining of mobile data stream (mdsm) [11], recommender systems [12], medical [13]–[17], predicting natural catastrophes [18], safeguarding Internet and web [19]–[22], and predicting weather [23].

First and foremost step of association rule mining discovers frequent itemsets [1]. Itemsets are considered frequent if found present in a higher number of transactions than the minimum support, a threshold defined in advance. This step is computationally expensive. Association rules are generated

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

in the next step with the help of frequent itemsets. In contrast to the former one, this step is trivial and computationally inexpensive. Hence, the overall performance of an association rule mining technique is strictly dependent on the first step. Frequent itemset mining holds a distinguished stature in data science to generate association rules, episodes, and correlations [24]. It finds collections of items placed collectively in a database of transactions [1]. A transactional database is composed of a sequence of transactions where the transactions are mapped to baskets of different items customer buy [25]. Several large retailers such as e-bay, YouTube, Netflix, and Amazon mine frequent itemsets to offer further recommendations to the users about interesting items/products.

Current frequent itemset mining algorithms are comprehensively validated for dense datasets, and a clear dissimilarity exists among their performance. On the other hand, they are not adequately applied to sparse real-world datasets lacking significant validation. Furthermore, a comparable performance has been exhibited by them for sparse datasets. Real-world sparse datasets have the following features.

1) There are no significant repetitions as well as transaction overlapping.
2) They generally have several distinct items.
3) They contain transactions of varying lengths.
4) Frequent itemsets discovered are hundreds or thousands in numbers.

A transactional database of a hypermart is an example of a real-world sparse dataset. A hypermart usually contains a multiplicity of items available. However, a small portion of these items is contained within each transaction. Therefore, a transaction represents a minimal subset of I, where I is the set of distinct elements present in a hypermart. Sparse datasets are also generated by the sensors containing several thousand readings and a few occurrences of activity leading towards big data [26]–[30].

Moreover, imbalanced behavioral big data is also sparse [31]. Though the information represented by sparse data is less comprehensive, yet it is reasonably rare, and that is why more useful and precious for companies. This information is essential for understanding the behaviors of a variety of clientele; hence, better predictive analytics can be made. It has been empirically demonstrated that sparse data-based predictive models have brought a massive improvement in predictive performance [32].

In this paper, a novel method, Mining frequent itemsets by Iterative TRimmed Transaction lattICE (TRICE), is proposed to dig out frequent itemsets from sparse real-world transactional datasets efficiently. TRICE has optimized the HARPP (HARnessing the Power of Powersets for Mining Frequent Itemsets) algorithm [33] by getting rid of its memory exhaustiveness for the datasets having longer average transaction length. HARPP is based on Iterative Transaction Lattice (ITL). Though HARPP's novel feature of making a single pass over the database is advantageous, it limits the performance as well. In a transactional database, a transaction contains both frequent and infrequent items. Since HARPP reads a transaction only once, it lacks the mechanism to eliminate the infrequent 1-itemsets from it. It affects the performance of HARPP when applied to the datasets having longer average transaction length. Limitations of HARPP can be eradicated by introducing a mechanism that cutoffs the infrequent 1-itemsets from each transaction. Thus, each trimmed transaction contains only frequent 1-itemsets before making its power set. This approach can effectively reduce the memory requirement because the resultant power sets will be smaller in size.

Itemset Lattice (IL) of I items contains 2^I itemsets, where I is the set of all distinct elements in a transactional database. Since I tends to be very large, generating IL of I spawns exponential possible itemsets. These abundant possible itemsets need a massive memory to be stored, which is not viable in veracity. Moreover, extracting frequent itemsets from 2^I possible itemsets is trivial but extremely inefficient. In contrast, a transaction is a smaller subset of I because it contains fewer items of I. Therefore, a Transaction Lattice (TL) is composed of minimal possible itemsets, and frequent itemsets can be generated efficiently. TRICE introduces the concept of Iterative Trimmed Transaction Lattice (ITTL).

Following HARPP, TRICE also generates power sets of transactions. However, TRICE generates power sets of varying-sized trimmed subsets of I, where each subset represents a transaction in a database. In fact, TRICE iteratively makes power set of every trimmed transaction described by its respective ITTL. Furthermore, several identical transactions are contained in large real-world sparse datasets. TRICE stores and trims identical transactions only once, therefore the corresponding power set is also constructed once. This mechanism avoids redundant processing later. It also helps in getting efficiency and conserving memory. Trimming before making power sets helps in getting rid of infrequent 1-itemsets from the transactions. Thus, in contrast to Itemset Lattice (IL), which is a huge power set containing 2^I subsets, an ITTL represents a small subset of IL. These ITTLs are then used to find frequent itemsets.

TRICE achieves efficiency and conserves memory due to its elegant transaction trimming mechanism as well as treating identical transactions. It is compared with HARPP, FP-Growth, and optimized SaM (Split and Merge) and RElim (Recursive Elimination) algorithms on six real-world sparse datasets. Optimized versions of SaM and RElim have been claimed as efficient performers for sparse datasets. TRICE has shown its superiority and outperformed all these algorithms both in terms of running time and memory consumption. Moreover in this study, the memory consumption of optimized versions of both SaM and RElim algorithms [34] has been evaluated for the first time.

The paper is organized as follows. Related work is presented in Section 2. Section 3 presents the detailed depiction of the problem and its essential definitions. TRICE algorithm is presented in Section 4. A detailed example is given in section 5. Section 6 presents experimental results in detail and performance study. Section 7

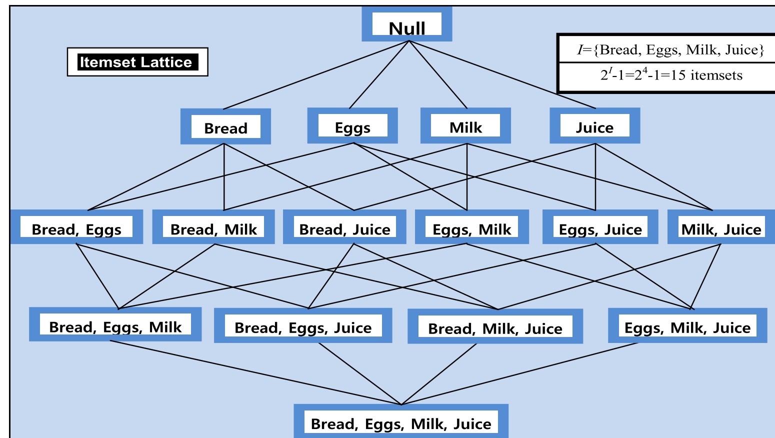

FIGURE 1. Itemset Lattice generated by brute-force algorithm.

concludes this study and presents some potential research issues.

II. RELATED WORK

Frequent itemset mining is a well-researched area. Table 1 describes the characteristics of existing techniques and their validation adequacy for real-world sparse datasets.

At the very start, the basic brute-force algorithm [35] builds all possible itemsets shown by IL in Figure 1. Following this, it begins calculating the support or frequency of presence of every itemset (except empty set) by comparing it against each transaction in a transactional dataset. If the support of an itemset remains less than the user-defined minimum support threshold, it becomes infrequent, thus rejected. This algorithm works in a simple manner but the exponential increase of itemsets makes it is exceptionally inefficient and useless. If I denotes a large set of different items in a transactional dataset, then the generated itemsets are 2^I. Consequently, massive memory is required to store these itemsets, which makes this algorithm unreasonable. Abundant items are kept by large retailers; therefore, I tends to be very large.

Based on the method of *candidate-set generation* and *test*, Apriori is believed as a standard algorithm [2]. Apriori introduces the notion of hierarchical monotonicity to bring improvement in the basic brute-force algorithm. According to hierarchical monotonicity, if a subset of a frequent itemset is present, it should be realized as frequent too. Likewise, an infrequent itemset makes its superset infrequent too. Figure 2 depicts the principle, in which after making first pass over the database, the support of {Juice} is found to be less than minimum support threshold.

Thus, according to the principle of hierarchical monotonicity, the supersets of candidate itemset {Juice} will also be infrequent. In this way, Apriori algorithm brings improvement in the brute-force algorithm by pruning the itemset lattice. Adopting breadth-first approach, Apriori constructs candidate itemsets in iterations having length (k+1) by using the collection of frequent itemsets of length k (for k≥1) and calculates their support in the database.

Several later studies [2], [36]–[42] are based on Apriori. However, they have to deal with the same challenge of generating gigantic candidate itemsets and calculating their support afterward.

Several database scans are required that limit the performance of these techniques [43]. Many vertical methods are proposed to bypass this tedious scanning [44]–[47]. These techniques vertically represent each itemset, such as Tid-set or diff-set. Support of itemsets is calculated by employing set intersection. Tid-set counts support of itemsets efficiently because it does not scan the database repetitively. However, if Tid-set cardinality or the dataset itself is large, efficiency of vertical techniques deteriorates.

Furthermore, according to the direct hashing pruning (DHP) algorithm [48], creation of frequent 2-itemsets is the key contributor to the growth of running time. Therefore, improvement in the preliminary construction of candidate itemsets can improve the performance of the algorithm on the whole. Moreover, scanning of the voluminous data also degrades the performance. DHP lowers the volume of the database and efficiently generates frequent itemsets. Perfect hashing and pruning (PHP) [49] makes hash tables for $C\kappa+1$ and efficiently escapes from hash table collisions that happen regularly in DHP. Accordingly, $C\kappa+1$ keeps the real count of the $C\kappa+1$ itemsets. Thus, the desire to count the frequency of $C\kappa+1$ itemsets once more in D is lowered. Sampling [50] and counting itemsets dynamically (DIC) [51] further lifted the efficiency by softening the harsh division between counting the support and building candidates. Every time the support of a candidate itemset and minimum support threshold become equal, DIC initiates the generation of additional candidate itemsets. A prefix tree is employed that extracts frequent itemsets quickly.

The Cluster-based association rule (CBAR) algorithm follows Apriori and makes use of a clustering technique [52]. In the beginning, CBAR carries out a dataset scan and constructs cluster tables. During this process, CBAR keeps a transaction record of k length to k-th cluster table. It makes candidate-2 itemsets in accordance with the procedure used

TABLE 1. Characteristics of existing techniques.

Technique	Search Strategy	Memory Used	No. of Passes over the Database	Runtime	Validation Adequacy for Sparse Data
Brute-force [35]	Breadth-first	Impractical due to generation of exponential candidates	Equal to the number of transactions.	Impractical in veracity	Inadequate
Apriori-based [2], [36]-[42]	Breadth-first	Large due to the generation of numerous candidates	Equal to the number of transactions.	Highest due to the generation of abundant candidates and repeated passes over the database	Inadequate
Apriori-based Hashing and Pruning [48]-[49]	Breadth-first	Large but lower than simple Apriori-based techniques due to database pruning.	Equal to the number of transactions.	Lower than simple Apriori-based techniques due to the reduced number of candidate 2-itemsets	Inadequate
Apriori based-clustering [52]-[53]	Breadth-first	Large due to the generation of numerous candidates	Less than number of transactions due to database pruning	Lower than simple Apriori-based techniques due to scanning the partial cluster tables rather than the entire database	Inadequate
Vertical representation-based recursive [44]-[47]	Depth-first	Large because Tid-sets may be longer, and numerous candidates are generated.	One	Higher if Tid-set cardinality or dataset is large	Inadequate
Pattern growth-based [43], [55]-[60]	Depth-first	Lower than Apriori-based due to compact representation	Two	Lower than Apriori-based techniques due to fewer database scans and absence of candidates	Inadequate
Horizontal representation-based recursive [61]-[62]	Depth-first	Large due to recursive processing	Two	Higher due to worst-case quadratic behavior of merge sort in merging step.	Inadequate
Sampling [50]	Breadth-first	Large but lower than Apriori-based due to the reduced number of candidates	One or two in worst case	Lower than Apriori-based techniques due to single pass over the database in most of the cases	Inadequate
Dynamic Itemset Counting [51]	Breadth-first	Large but lower than Apriori-based due to reduced number of candidates	Fewer	Lower than Apriori-based due to randomization, fewer candidate itemsets, and fewer database scans	Inadequate
Iterative Transaction Lattice- based [33]	Breadth-first	Very low as dataset is not loaded into main memory	One	Lowest due to single pass over the database and on the fly support counting of itemsets	Fair to some extent

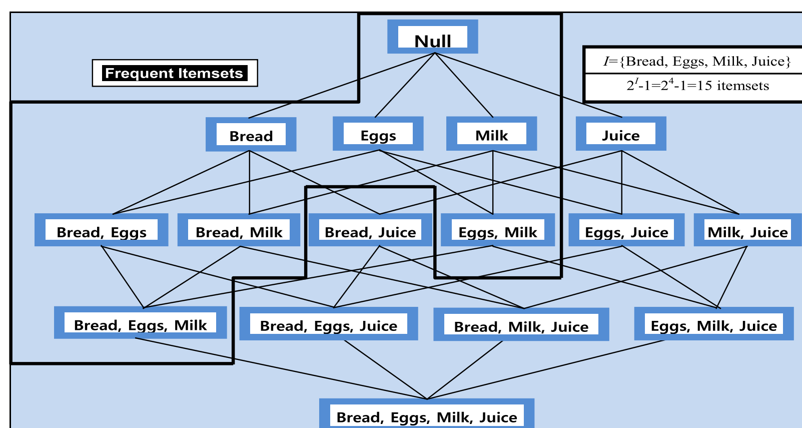

FIGURE 2. Hierarchical monotonicity in apriori.

by Apriori algorithm. Following this, CBAR contrasts candidate 2-itemsets with in the 2nd cluster. An itemset is believed to be frequent if its support becomes equal to the minimum support threshold, and additional checking in larger clusters is stopped. Likewise, candidate 3- itemsets are examined in the 3rd cluster and so on. CBAR performs in a fast manner because it prefers contrasting partial cluster tables over the entire database.

A modified cluster-based method performs two optimizations in the CBAR algorithm [53]. At the start, it performs

database trimming by ignoring all infrequent items before constructing clusters. The database is scanned two times. The first scan of the database helps to find out frequent 1-itemsets. In the later scan, infrequent itemsets are removed from each transaction before clustering. It further optimizes CBAR by employing a counter for every transaction. Every occurrence of a transaction causes an increment in the counter. An identical transaction is discarded. The transaction is then placed in the cluster table. This adjustment significantly trims the volume of cluster tables.

FP-Growth algorithm employs extended structure of prefix-tree. A trie structure is used for holding the database. A link list is kept by each itemset that performs traversal of all transactions containing the itemset. A condensed form is used to keep this structure and denoted as, FP-tree (Frequent-Pattern tree) [43]. Each node maintains a counter to track the number of transactions that share the branch all the way through the node. It holds a link that points to the next presence of the itemset in FP-tree. Finally, it connects every occurrence of an itemset and represents it by FP-tree. Also, it holds a header table to store every distinct itemset and its support along with a pointer in FP-tree that points to its first presence. Discovery of frequent itemsets is done by employing a divide-and-conquer method. FP-Growth lacks efficiency if the patterns are longer or minimum support becomes low because conditional FP-trees are spawned in abundance [54].

Subsequently, PPC-trees (Pre-order Post-order Code trees) are proposed for holding the information regarding frequent itemsets [55]. The efficiency of the PPC tree is better than FP-tree because the algorithm needs to traverse the tree once for identification of the N-list holding frequent 1-itemsets. On the contrary, FP-Tree based techniques need to traverse the tree several times. PrePost algorithm is proposed that employs PPC-tree [56]. In the beginning, PREPOST constructs a PPC-tree utilizing an algorithm for tree generation. Later, it creates N-lists related to 1-itemsets. N-list denotes a transaction ID list (TID list) in compressed form that represents the features of an itemset. Then a divide-and-conquer approach is incorporated to extract frequent itemsets afterwards. There is no requirement of constructing additional trees in the successive iterations, thus its performance is superior than that of FP-Tree. PrePost has a limitation due to its utilization of Apriori-like method to mine frequent itemsets, still single-path property of N-list is utilized for pruning the search space.

Nodeset represents an itemset that is based on the PPC tree. The encoding of a node in Nodeset is carried out by post-order or preorder code. FIN algorithm is proposed based on Nodeset [57]. FIN and PrePost perform equally well; however, FIN consumes less memory. PrePost+ optimizes PrePost as utilizes N-list to corresponding to frequent itemsets and performs mining in a straight manner [58]. Children-Parent Equivalence pruning is used for reduction of the search space as well as to evade monotonous search.

Moreover, the *subsume index* is proposed for additional enhancing mining performance [59]. A frequent 1-itemset keeps a *subsume* index to represent a list that holds frequent 1-itemsets occurring with it. NSFI is proposed that is based on *subsume* index [60]. NSFI mingles *subsume* index and N-Lists to gain efficiency and minimal memory. The use of a hash table to generate N-lists makes NSFI an efficient algorithm. Further, it has improved the N-list intersection process. Subsume index helps in finding frequent itemsets and eliminates the need to recognize associated N-lists.

RElim [61] is another algorithm that is based on FP-growth, but rather than using FP-trees, it uses recursion for eradicating items. It begins by selecting the transactions that contain the least frequent item. That item is then removed, resulting in a much smaller trimmed dataset. The remaining items in this dataset are processed recursively. It remembers the discovered items during the recursive process and computes all frequent itemsets related to the removed item for which the recursive procedure is called. RElim does this when all the items in the trimmed dataset are explored. RElim does this procedure again by selecting the next least frequent item and so on. RElim is simplified by introducing SaM (Split and Merge) [62].

SaM is based on a horizontal demonstration. It works in two steps. The first step is the split-step in which every array, which starts with the first item of the first transaction is copied into a new array, and that foremost item is then eliminated. This procedure is then repeated recursively to discover all frequent itemsets for the first item. Afterward, a merge step is performed with the trimmed dataset (the foremost item is eliminated) to get the conditional pattern base. Both RElim and SaM have been optimized later, and it is claimed that they have shown excellent runtime performance on sparse datasets.

HARPP algorithm is evaluated on three sparse real-world datasets and has shown its superiority [33]. It makes combinations of transactions represented by power sets in a transactional database. Treating each transaction as a *set* ADT, the containment of each of its subset is checked within a *set* ADT first that contains all frequent itemsets. Containment within this set signifies that the subset is frequent already. Thus the subset is labeled frequent and discarded immediately. Or else, the subset and its support count are stored in a *dictionary* ADT as a pair of key and value. Shortly after saving, the support of the subset is compared with the minimum support threshold. If its support becomes equal to minimum support threshold, it is regarded frequent, removed from the *dictionary*, and kept in the *set* containing frequent itemsets. Without storing the database in main memory, HARPP makes a single pass over the database and discovers frequent itemsets on the fly.

Most of the techniques have been usually evaluated on dense datasets, a significant distinction exists in their performance. However, there is no substantial dissimilarity among their performance for large real-world sparse datasets.

TABLE 2. Notations and descriptions.

Notation	Description
D	A transactional database
I	Total number of items in a transactional database
T	A transaction of *D*
Ts	A *set* ADT representation of *D*
Is	An itemset
Sup(*Is*)	Support of an *Is*
|*D*|	Total number of transactions in *D*
minsup	Given threshold
Dict1	A *dictionary* ADT that keeps *Ts* and support of *Ts* as a pair of key and value.
Dict2	A *dictionary* ADT that keeps *Is* and support of *Is* as a pair of key and value.
Dict3	A *dictionary* ADT to store extracted key-value pairs from *Dict2*, representing frequent 1-itemsets (*Is*(s))
Kc	Represents a key in *Dict1*
Ls	A *set* ADT to store the keys extracted from *Dict3*
Z	A *set* ADT that represents the intersection of *Kc* with *Ls*.
Ps	A list ADT to store power set of *Z*.
S	A subset within *Ps* that can be a frequent *Is*.
Dict4	A *dictionary* ADT that keeps *S* and support of *S* as a pair of key and value
Fs	A *set* ADT that stores all frequent itemsets

Furthermore, they are not often applied to real-world sparse datasets and have shown inferior performance [48]. For example, NSFI is applied to *Retail* dataset only, and PrePost performs marginally better than NSFI [60]. Comparable performance is shown by PrePost, FP-Growth*, and FP-Growth on sparse datasets [56]. Likewise, running time of PrePost+, FIN, and PrePost is alike on *Kosarak* dataset [58]. The running time of these algorithms grows swiftly when minimum support goes beneath 0.4% for *Kosarak* dataset [58].

Optimized SaM and RElim algorithms have been applied to only two sparse datasets, out of which only one is real. Moreover, the memory consumption of both SaM and RElim has not been investigated at all [34]. Therefore, their efficiency on sparse datasets needs to be further investigated. HARPP is efficient, yet its performance degrades for the datasets having longer transactions. Due to scanning the database once, HARPP generates combinations of a transaction on the fly. Thus, it lacks the capability of removing infrequent 1-itemsets before making combinations. Resultant power sets are larger; therefore, they exceed the memory limits.

To beat this issue, a novel method: Mining Frequent Itemsets by iterative TRimmed transaction lattICE(TRICE) is proposed. TRICE optimizes HARPP by getting rid of its memory exhaustiveness and efficiently finds frequent itemsets from several real-world sparse datasets. Moreover, the assessment of memory use of optimized SaM and RElim algorithms is done for the first time.

III. BASIC CONCEPTS

This section commences by introducing the concepts relevant to TRICE. Suppose $I = \{i_1, i_2, i_3, \ldots, i_m\}$ be the set containing all items. Let $D = \{T_1, T_2, T_3, \ldots, T_n\}$ be a dataset containing *n* transactions in a way that every transaction consists of various items belonging to I. *Is* indicates an itemset if *Is* is a set of items. A transaction *Ts* includes *Is* if and only if *Is* is a subset of *Ts*.

TABLE 3. A transactional dataset, *D*.

Tid	Transactions
T_1	A, B, C, D
T_2	A, B, C
T_3	A, B, D
T_4	A, B
T_5	B, C, E
T_6	A, B, C, D

Notations with their descriptions are presented in Table 2.

A dataset, *D* is shown in Table 3 for the purpose of illustration throughout the paper. *Sup*(*Is*) denotes the support of an *Is* in *D*, and it represents the number of transactions that contain all the items in the *Is*. An *Is* becomes frequent if $Sup(Is) \geq (minsup \text{ X } |D|)$. A frequent itemset that holds *k* elements is called a frequent *k*-itemset. Frequent itemset mining problem can be expressed as discovering the set that holds all itemsets having support $\geq (minsup \text{ X } |D|)$.

The power set of a *set* comprises all subsets of the set, including the set itself, but excluding the empty subset.

IV. TRICE: THE PROPOSED METHOD

A typical transaction in a dataset consists of some items contained in I, which is the set of all distinct items. Thus, a transaction represents a subset of I. In fact, TRICE iteratively generates Transaction Lattice (TL) by making power sets of each transaction in a transactional dataset, hence denoted as Iterative Trimmed Transaction Lattice (ITTL). The likelihood of the occurrence of identical transactions (containing similar items) is high in large real-world sparse datasets. TRICE

```
Algorithm: TRICE Algorithm
Input: A database of transactions D, minimum support
(minsup), and |D| be the total number of transactions in D.
Output: Fs

Call Frequent1 (D, minsup, |D|)
Call FrequentItems (Dict1, Ls, |D|, minsup)

Frequent1 (D, minsup, |D|)
(1) For each Ts in D, do
(2)     If Ts is not already present in Dict1, then do
(3)         Store Ts as a key in Dict1 and set its value to 1
(4)     Else
(5)         Increase the value of Ts in Dict1 by 1
(6)     End If
(7)     For each Is in Ts, do
(8)         If Is is not already present in Dict2, do
(9)             Store Is as a key in Dict2 and set its value
                to 1
(10)        Else
(11)            Increase the value of Is in Dict2 by 1
(12)        End If
(13)    End For
(14) End For
(15) Pop all key-value pairs from Dict2 having value ≥
     (minsup x |D|)and store into Dict3
(16) Delete Dict2
(17) Store all keys of Dict3 into Ls
(18) Delete Dict3
(19) Return Dict1, Ls

FrequentItems (Dict1, Ls, minsup)
(1) For each Kc in Dict1, do
(2)     Store intersection of Kc and Ls into Z
(3)     If Z is not already present in Fs, then do
(4)         Make power set of Z and store in Ps
(5)         For each S of Ps, do
(6)             If S is not already present in Fs, then do
(7)                 If S is not already present in Dict4, do
(8)                     Store S as a key in Dict4 and set its
                        value equal to the value of Kc in Dict1
(9)                 Else, do
(10)                    Increase value of S in Dict4 by the value
                        of Kc in Dict1
(11)                End If
(12)                If value of S ≥ minsup x |D|, then do
(13)                    Pop out S from Dict4 and store into Fs
(14)                End If
(15)            End If
(16)        End For
(17)    End If
(18) End For
(19) Print Fs
```

FIGURE 3. Pseudocode of TRICE.

stores and trims alike transactions only once, therefore the corresponding ITTL is also constructed once. This mechanism avoids redundant processing in later stage. It also helps in getting efficiency and conserving memory.

Trimming before making ITTL helps in getting rid of infrequent items from the transactions. Thus, in contrast to Itemset Lattice (IL), which is a vast power set containing 2^I subsets, an ITTL represents a small subset of IL. ITTL of TRICE is even smaller than the IL generated by HARPP because IL in HARPP is made up of the complete transaction without removing infrequent 1-itemsets. TRICE incorporates efficient *set* and *dictionary* data structures for storing, containment checking, and support counting of itemsets. These operations take constant running times, thereby achieving efficiency. The pseudocode of TRICE is shown in Figure 3.

TRICE algorithm is comprised of two steps. In the beginning, TRICE invokes *Frequent1*( ) procedure to do the subsequent tasks.

1) The procedure scans the dataset and iteratively does the following two sub-tasks for every transaction.

   a) In Step (1)-(6), a *Ts* is read. *Ts* and support of *Ts* are placed as a pair of key and value in *Dict1*. The value of *Ts* is one if *Ts* occurs for the first time, or else, the value is incremented. A value greater than one shows that identical transactions exist in a dataset. *Dict1* stores identical transactions only once.
   b) In Step (7)-(14), every single *Is* is extracted from *Ts* and stored into *Dict2* as a key with value one,

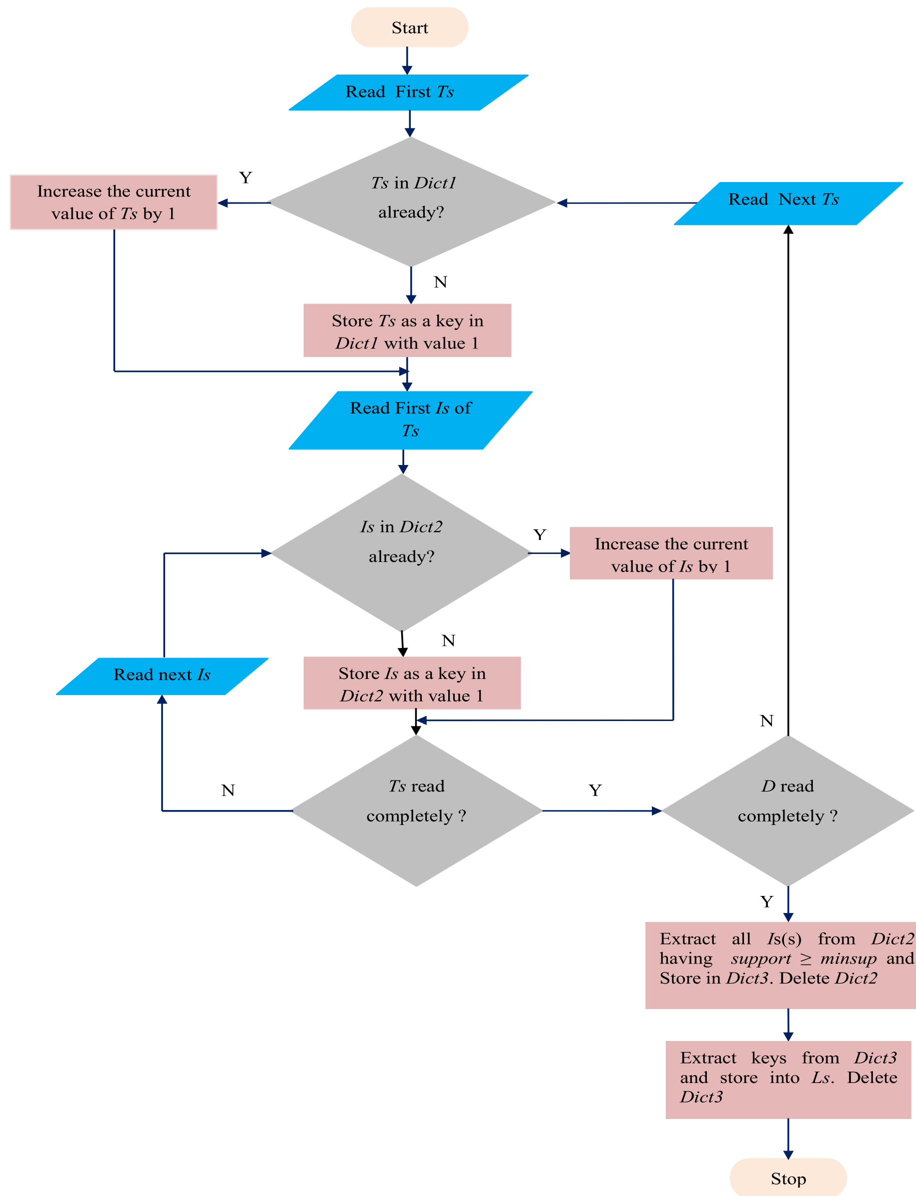

FIGURE 4. **Flowchart of *Frequent1* ( ) procedure.**

if it comes for the first time. Or else, the current value is incremented.

2) When the dataset is read thoroughly, in Step (15), all *Is* whose support is equal or greater than *minsup* are filtered out and stored into *Dict3*. Thus, *Dict3* consists of all frequent 1-*Is* and their values (support) as the pairs of keys and values.
3) Keys kept in *Dict3* are extracted and stored into *Ls* in Step (17). Thus, *Ls* contains frequent 1-itemsets.

The procedure ends with *Dict1* that contains all distinctive transactions (*Ts*) and their associated supports as pairs of keys and values, and *Ls*. The flowchart of the procedure *Frequent1*() is shown in Figure 4.

FrequentItems( ) procedure is invoked in the next step, which takes *minsup*, *Dict1*, *Ls*, and $|D|$ as parameters. The following tasks are done for every *Ts* placed in *Dict1* pointed to by *Kc*.

1) Step (1) - (2) gets a *Kc* and intersect it with *Ls* stored in *Z*. This intersection discards the infrequent items from *Kc*. Thus, *Z* is a trimmed version of *Kc* (currently read *Ts*) because it contains the frequent items of *Kc* only. After that it does the following actions.
   a) Step (3) - (4) verifies the containment of *Z* in *Fs*. If *Z* exists already, it is considered frequent, thus deleted. Following this, the procedure takes next *Kc* from *Dict1*. Or else, it generates the power set, *Ps* of *Z*, that represents the ITTL of *Z*.
   b) For every *S* within *Ps* in Step (5), the procedure does subsequent sub-tasks.
      i) Step (6) takes a *S* and verifies its containment in *Fs*. If *S* exists, it is regarded as frequent,

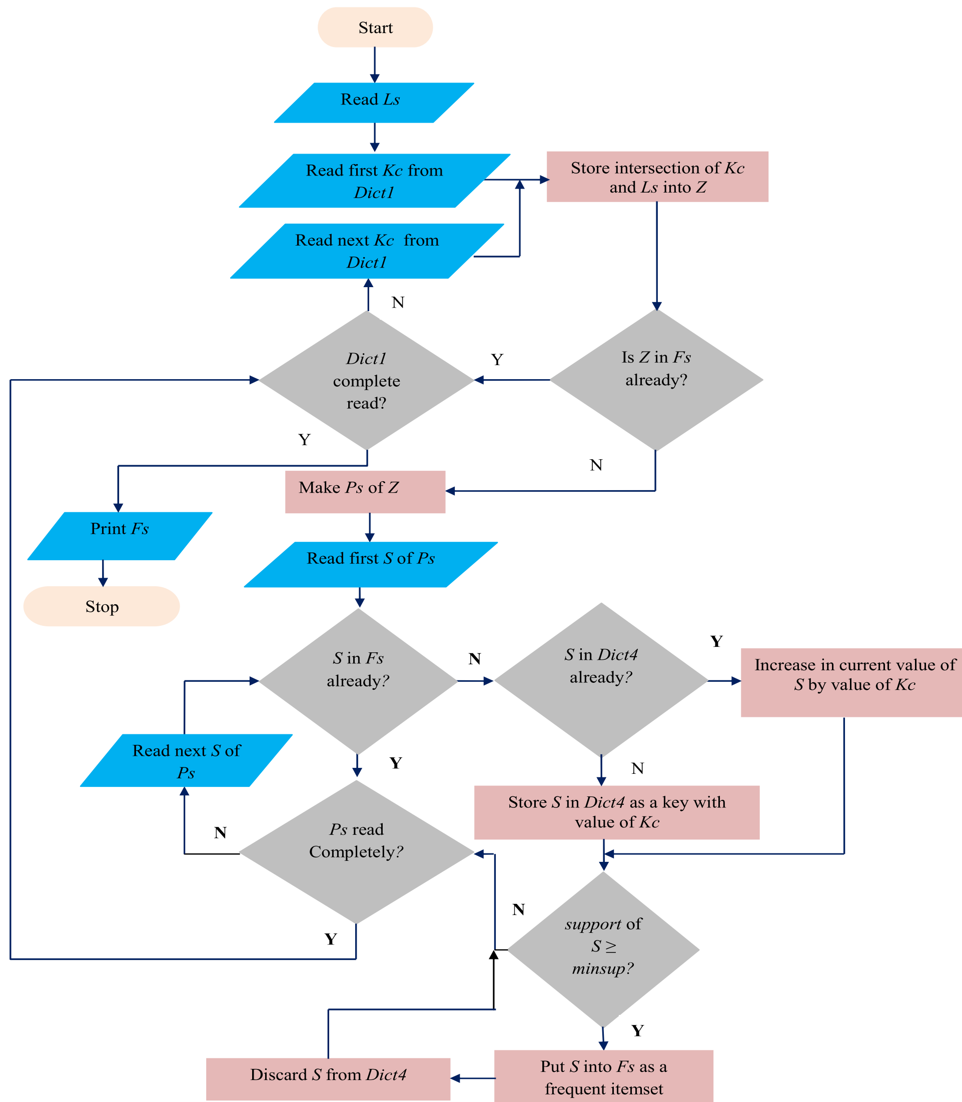

FIGURE 5. Flowchart of *FindFrequent* ( ) procedure.

thus removed and next S is taken from Ps. Or else, in Step (7)-(8) S is placed in *Dict4* as key, and the value of S is made equal to that of Kc. If S exists previously, then an increase is done in its current value by the value of Kc in Step (9)-(10). This process happens for the reason that the support of S is equivalent to support of Ts and Z shows intersection of this Ts with Ls.

ii) After placing S in *Dict4*, the value of S and *minsup* are compared in Step (11). If both are equal, S is regarded as a frequent Is.

iii) Then this Is is removed from *Dict4* and kept in Fs in Step (12).

The flow chart of *FrequentItems* ( ) is shown in Figure 5.

V. TRICE EXAMPLE

This representative example is based on the dataset given in Table 3. Let us assume *minsup* = 66%. According to the dataset, an Is will become frequent if it occurs in at least four transactions (66%). At first, TRICE invokes the procedure, *Frequent1* ( ), as shown by figures 6-11. 1st Ts is read by *Frequent1* ( ) in Figure 6 and kept in *Dict1* as key. The associated value of the Ts is one because this does not exist in *Dict1* already. Following this, each single itemset, Is of Ts is extracted and put into *Dict2* as a key, and the value of Is will be one. The values of all keys in *Dict2* are set to one, because it will be stored in *Dict2* for the first time.

In Figure 7, 2nd Ts is read and stored as a key in *Dict1* having value 1. Following this, the procedure extracts each Is of Ts and places it as a key in *Dict2*. Itemsets, A, B, and C exist there already; thus, an increment is done in their values. The values of A, B, and C are now equal to 2 in *Dict2*.

Figure 8 shows the states of *Dict1* and *Dict2* when 3rd Ts is read. This Ts is placed in *Dict1* as a key and the value of Ts is one because the Ts arrives for the first time. After extorting

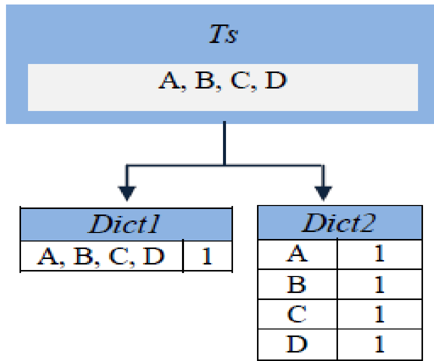

FIGURE 6. State of ***Dict1*** and ***Dict2*** after placing T_1.

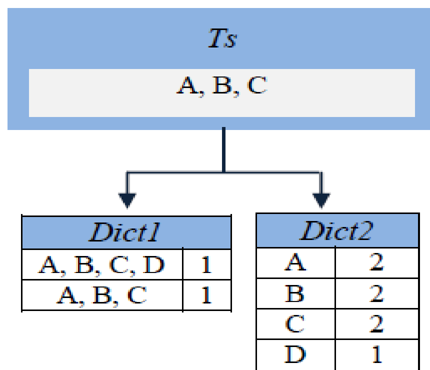

FIGURE 7. State of ***Dict1*** and ***Dict2*** after placing T_2.

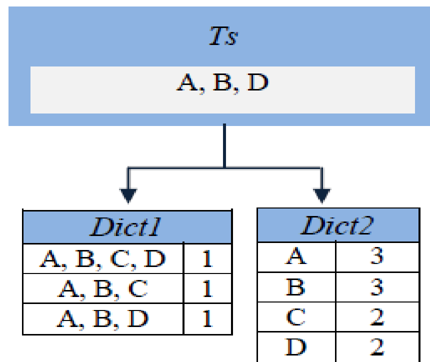

FIGURE 8. State of ***Dict1*** and ***Dict2*** after placing T_3.

individual *Is*s, an increment is done in the values of A, B, and D.

Figure 9 shows *Dict1* and *Dict2* when 4th *Ts* is read.

Figure 10 shows *Dict1* and *Dict2* when 5th *Ts* is read.

Figure 11 shows *Dict1* and *Dict2* when 6th *Ts* is read, which is a repeated transaction. This repeated *Ts* is not kept in *Dict1* again; instead its value is incremented. Values of *Is*s {D} and {E} in *Dict2* are smaller than *minsup*, thus ignored. Eventually, frequent 1-*Is*s are {A}, {B}, and {C} that are stored into *Dict3* with their supports. Finally, *Dict3* keys that

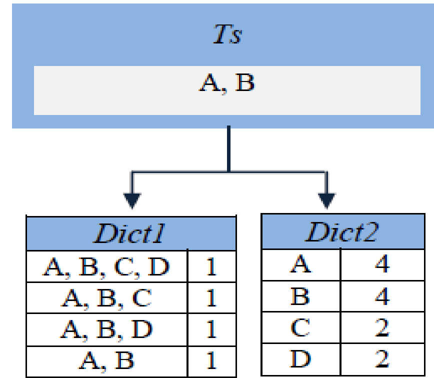

FIGURE 9. State of ***Dict1*** and ***Dict2*** after placing T_4.

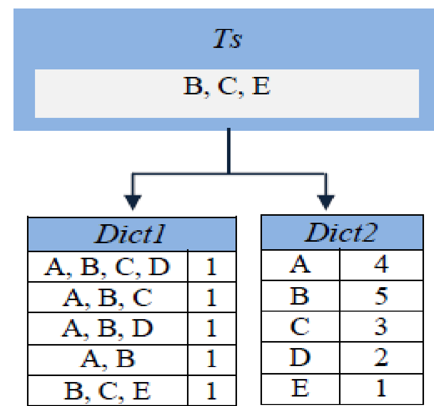

FIGURE 10. State of ***Dict1*** and ***Dict2*** after placing T_5.

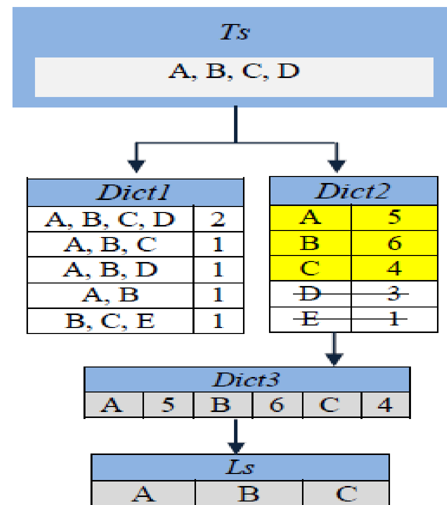

FIGURE 11. ***Dict1***, ***Dict2***, ***Dict3***, **and** ***Ls*** **after reading** T_6.

represent frequent 1-*Is* are kept in *Ls,* however the values of these keys are not required any longer.

Figures 12-16 depict the *FrequentItems*( ) procedure. In Figure 12, *Kc* refers to the 1st key in *Dict1*.

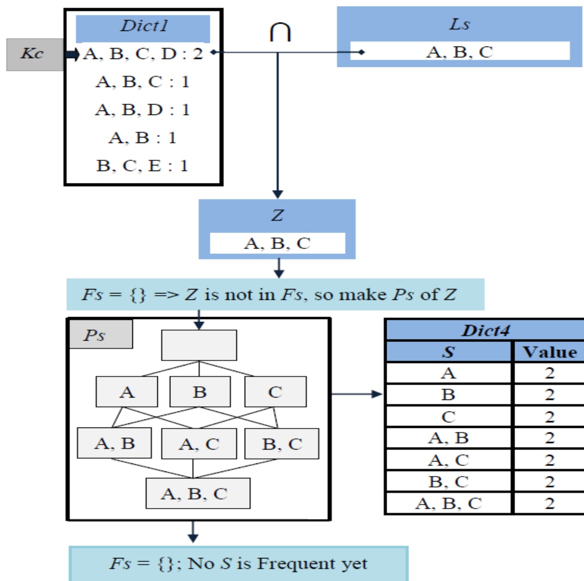

FIGURE 12. ***Ps*** **showing ITTL of 1st key,** ***Dict4*****, and** ***Fs*****.**

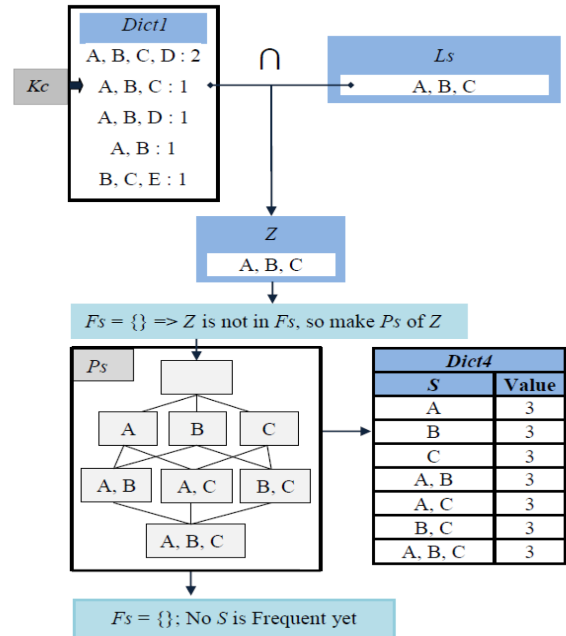

FIGURE 13. ***Ps*** **showing ITTL of 2nd key,** ***Dict4*****, and** ***Fs*****.**

The intersection of *Kc* and *Ls* is stored in *Z*. Itemset D is discarded because it is not present in *Ls*; therefore, *Z* contains a trimmed transaction. Containment check in *Fs* is done here. *Z* is infrequent yet; thus, *Fs* does not contain *Z*. The procedure advances and *Ps* of *Z* is created that shows its ITTL. Then iteratively, all subsets (*S*) of *Ps* are placed as keys into *Dict4* one by one with value two that is equal to that of recent *Kc* in *Dict1*. Immediately after storing, support of each *S* and *minsup* are contrasted. If the value of *S* equalizes *minsup* or becomes higher, it is regarded as frequent. The value of every *S* is less than *minsup*, so it is not frequent yet. *Fs* is still empty.

Figure 13 shows *Dict4* and *Fs* when *Kc* refers to the next key of *Dict1*. Because *Fs* is unfilled, the *Ps* of the present intersection is generated and stored in *Z*. Then each subset (*S*) is stored into *Dict4*. Because each *S* already exists in *Dict4*, the value of each *S* is incremented. Yet the support of every *S* is far behind *minsup*; therefore, *Fs* remains empty.

Figure 14 shows *Fs* and *Dict4* when 3rd key of *Dict1* is read. *Ps* is generated again because *Fs* does not possess current *Z* already.

After the creation of *Ps*, each *S* is kept in *Dict4* as a key with updated value. Values of subsets {A}, {B}, and {A, B} are now equivalent to *minsup*, hence they are frequent. Thus, they are deleted from *Dict4* and kept in *Fs*. Figure 15 depicts *Fs* and *Dict4* when 4th key of *Dict1* is read.

The intersection containing A, B is discarded because it already resides in *Fs*. Thus, the additional steps are not performed, and the procedure reads the next *Kc*. Figure 16 shows *Fs* and *Dict4* when 5th key of *Dict1* is read.

Z contains a subset {B, C}. Due to its absence in *Fs*, *Ps* of this subset is generated. Each *S* and support of *S* is kept in *Dict4* as a pair of key and value. The support of {C} and {B, C} and *minsup* are equal now. Thus, they are regarded as frequent, removed from *Dict4* and kept in *Fs*. The algorithm terminates because no further key remains in *Dict1*. *Fs* holds each frequent itemset.

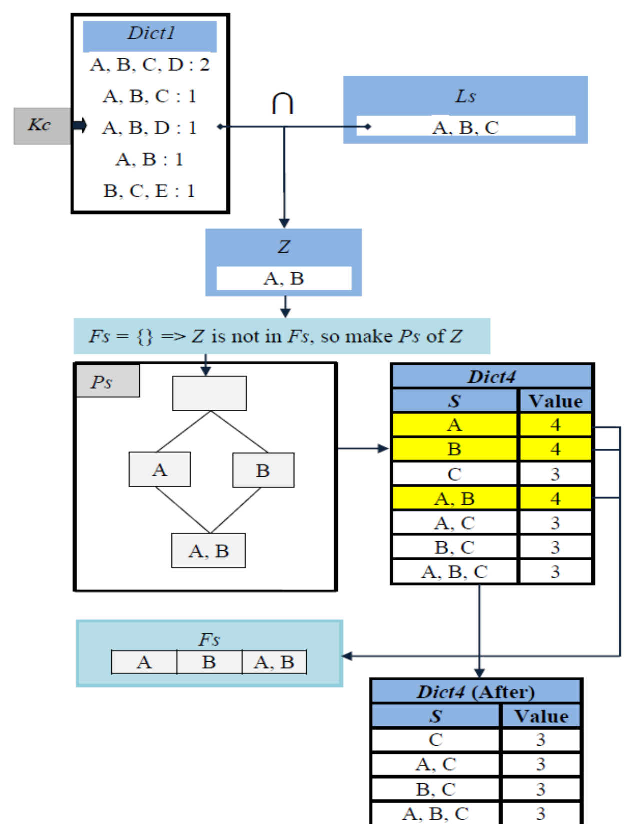

FIGURE 14. ***Ps*** **showing ITTL of 3rd key,** ***Dict4*****, and** ***Fs*****.**

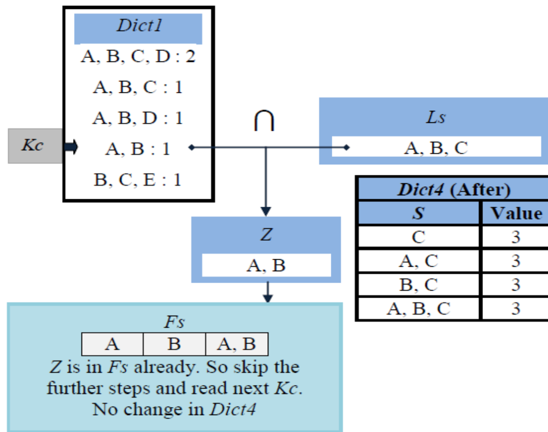

FIGURE 15. ***Ps*** **showing ITTL of 4th key,** ***Dict4*****, and** ***Fs*****.**

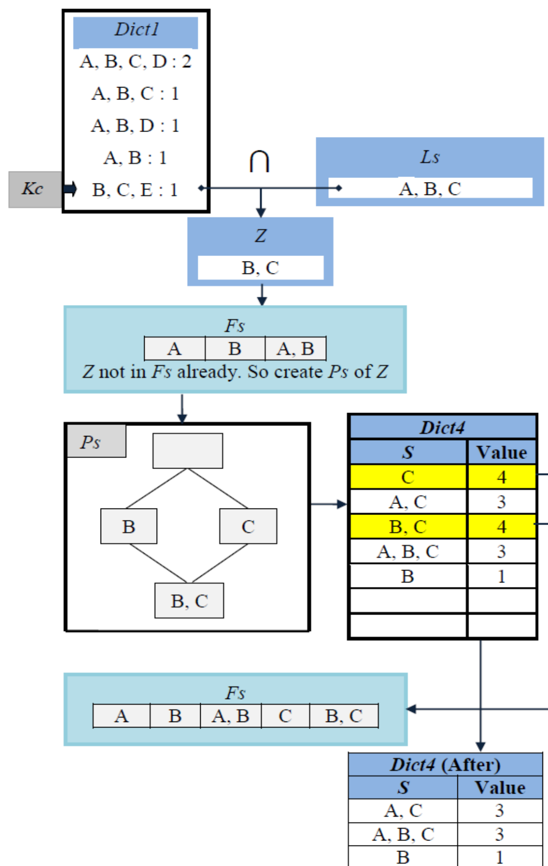

FIGURE 16. ***Ps*** **showing ITTL of 5th key,** ***Dict4*****, and** ***Fs*****.**

VI. EXPERIMENTAL EVALUATION OF TRICE

In this section, experimental results related to runtime and use of peak memory are reported. All algorithms discover identical frequent itemsets, thus verifying the accuracy of the results.

A. EXPERIMENTAL SETUP

TRICE is compared with HARPP, FP-Growth, and optimized versions of SaM and RElim algorithms on six real-world sparse datasets. Characteristics of the datasets are summarized in Table 4.

TABLE 4. **Features of the datasets.**

No.	Dataset	Distinct Items (*I*)	No. of Transactions
1	*PowerC*	140	1,040,000
2	*Kddcup99*	135	1,000,000
3	*Online Retail*	2,603	541,909
4	*Record Link*	29	574,913
5	*Extended Bakery*	50	75,000
6	*Food Mart*	1,559	4,141

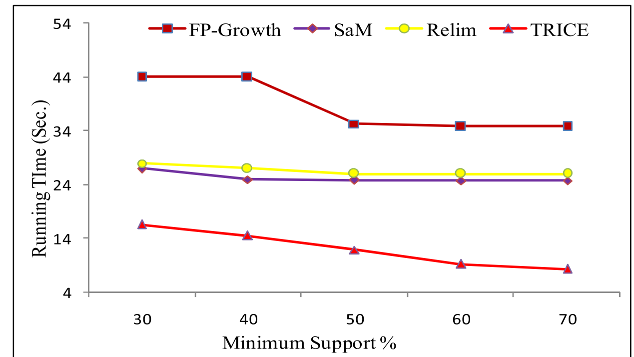

FIGURE 17. **Assessment of runtime for** ***Kddcup99*** **dataset.**

PowerC, *Kddcup99*, *Record Link*, *Food Mart*, and *Online Retail* are obtained from [63]. *Extended Bakery* dataset is derived from [64]. FP-Growth is preferred as one of the baseline algorithms, because its running time is reasonably equivalent to that of the other state-of-the-art algorithms for sparse datasets [56], [58], [60]. HARPP has also shown efficiency for sparse datasets. Optimized SaM and RElim algorithms have been demonstrated as efficient algorithms for sparse datasets [34]. Therefore, they have been chosen as other baseline algorithms. Python is used to implement TRICE and HARPP. FP-Growth, SaM, and RElim python implementations are taken from [65]. For experiments, a computer having Intel Core i7-3667U, 2.0 GHz processor, Windows 8 Pro ×64 Edition, and 8G memory has been used.

B. ASSESMENT OF RUNTIME OF TRICE

Running time of TRICE is compared with that of the others in this subsection. The assessment of runtime for the *Kddcup99* dataset is given in Figure 17. At lower *minsup*, such as 30%, FP-Growth, SaM, and RElim are beaten by TRICE almost by a factor of 3, 2, and 2 respectively. At higher *minsup*, such as 70%, TRICE is more than four times faster than FP-Growth, three times faster than SaM and more than three times faster than RElim. SaM is a bit better performer than RElim as opposed to the formerly reported analysis [34], where RElim has shown superiority on sparse datasets. The result of HARPP is not plotted because its running time exceeded 1000 seconds at *minsup* 30% and worsened when

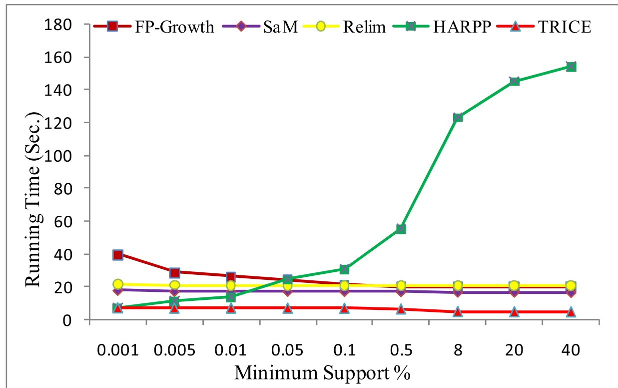

FIGURE 18. Assessment of runtime for *PowerC* dataset.

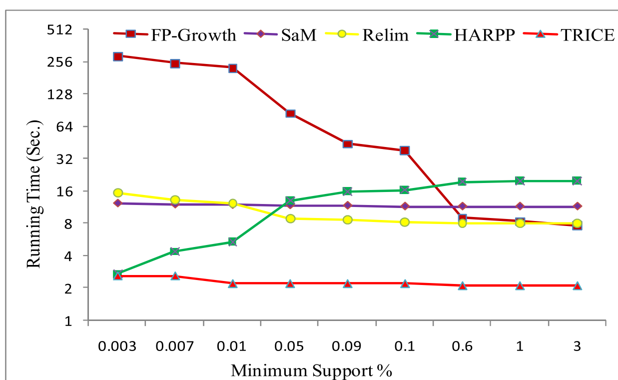

FIGURE 19. Assessment of runtime for *Online Retail* dataset.

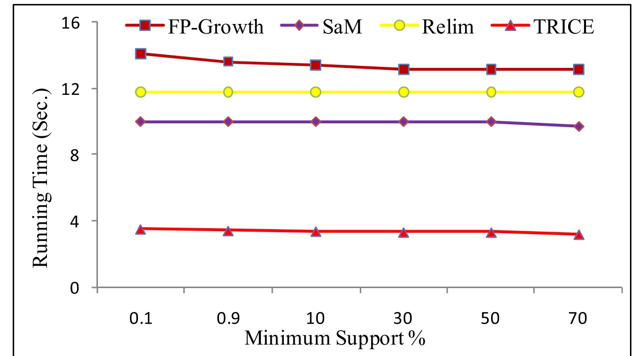

FIGURE 20. Assessment of runtime for *Record Link* dataset.

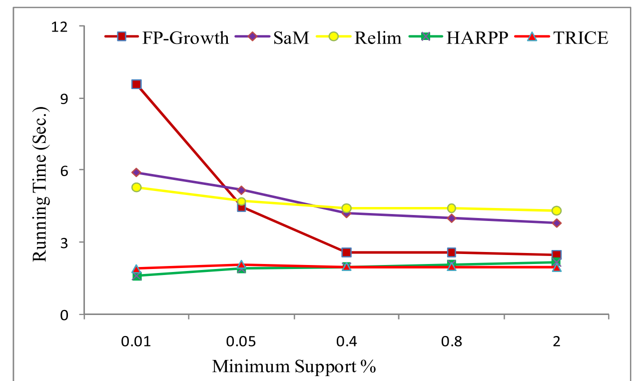

FIGURE 21. Assessment of runtime for *Extended Bakery* dataset.

minsup is increased further. However, the result shows that TRICE is leading by the widest of margins.

Figure 18 shows the assessment of runtime for the PowerC dataset. TRICE succeeds in maintaining the performance gap all the way. FP-Growth, RElim and SaM are beaten by TRICE almost by the factor of 4, 4, and 3 respectively on all *minsup* values. TRICE and HARPP perform equally well at *minsup* 0.001%, but runtime of TRICE improves with the increase in *minsup*. On the other hand, running time of HARPP starts rising with the rise in *minsup*. Eventually, HARPP is beaten by TRICE by the factor of 31 when *minsup* is 40%.

Figure 19 shows the assessment of runtime for the *Online Retail* dataset on a logarithmic scale. TRICE throughout outperforms all other algorithms and gets the same running time on each *minsup* threshold. At *minsup* 3%, TRICE beats FP-Growth, RElim, and SaM by a factor of 4, 4, and 5 respectively. At *minsup* 0.01%, TRICE becomes quicker than FP-Growth by two orders of magnitude and SaM and RElim by a factor of 5. HARPP and TRICE give a comparable performance at *minsup* 0.003%. But HARPP starts taking more time in discovering frequent itemsets as *minsup* tends to increase. However, performance of TRICE gets better with the increase in *minsup*. TRICE beats HARPP by a factor of 10 at *minsup* 3%.

Figure 20 demonstrates the evaluation for *Record Link* dataset. SaM again beats RElim though it is found to be slow performer on sparse datasets [34]. Nevertheless, TRICE again wins the performance battle.

When *minsup* is 70%, TRICE is quicker than FP-Growth, SaM and RElim by a factor of 4, 3, and 3.5 respectively. This significant performance gap persists throughout all *minsup* values. The result of HARPP is not plotted because its running time exceeded 600 seconds at *minsup* 0.1% and worsened when *minsup* is increased further.

Figure 21 shows the assessment for the *Extended Bakery* dataset.

TRICE beats FP-Growth almost by a factor of 1.5 and SaM and RElim by more than a factor of 2. TRICE performs consistently well and beats FP-Growth and the others by a factor of 5 and 3, respectively, when *minsup* is 0.01%. TRICE and HARPP give a comparable performance over all *minsup* values.

Figure 22 shows the assessment for the *Food Mart* dataset. At *minsup* 0.6%, TRICE is faster than HARPP by an order of magnitude and SaM by two orders of magnitude. Moreover, it beats RElim and FP-growth by the factors of 2 and 5 respectively. At *minsup* 0.04%, performance of HARRP and TRICE is comparable but TRICE is faster than FP-growth, SaM, and RElim by the factors of 47, 11, and 8, respectively.

The performance of TRICE becomes better with the increase in *minsup*. It is evident from the above performance analysis that TRICE always achieves better running times on

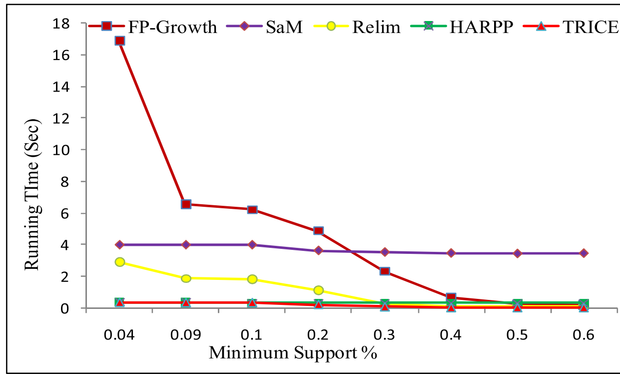

FIGURE 22. Assessment of runtime for ***Food Mart*** dataset.

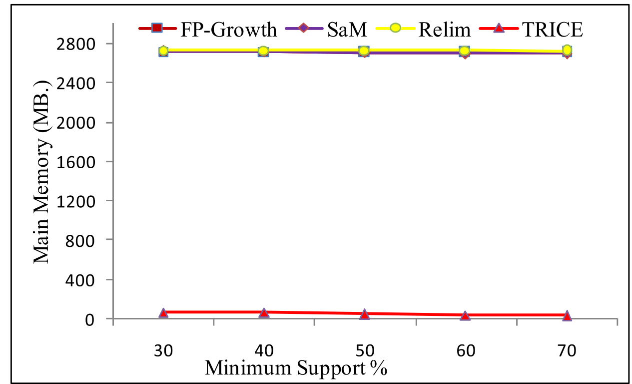

FIGURE 23. Assessment of memory use for ***Kddcup99*** dataset.

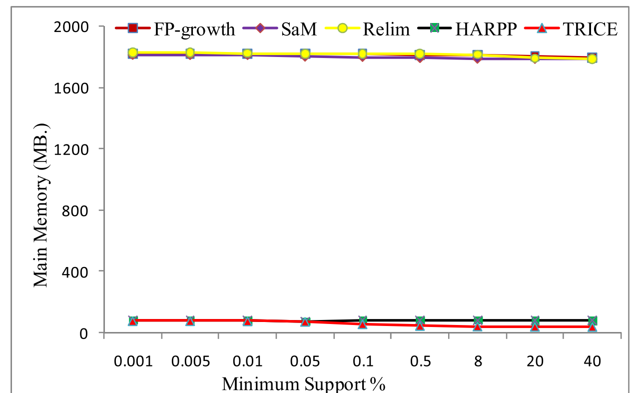

FIGURE 24. Assessment of memory use for ***PowerC*** dataset.

each dataset and on each *minsup* threshold. The reasons are explained below.

1) FP-Growth underperforms for sparse datasets because long repeated patterns seize to exist. FP-tree continues to grow; thus, additional time is required by the algorithm for construction and traversal of conditional FP-trees. On the contrary, TRICE does not need to generate numerous conditional pattern bases and conditional FP-trees; thus, less time is taken. The bottleneck in the original SaM algorithm is its merging step that deteriorates its performance when applied to sparse datasets [62]. Likely, the lengths of two lists of transactions in sparse datasets differ significantly, which compels the *merge sort* to show its worst-case quadratic behavior. Therefore, the optimized SaM has been used in this paper for comparison, which employs a modified merging scheme [34]. Similarly optimized RElim gets rid of duplicates in the transaction lists and used a heuristic approach to sort the lists [34]. However, TRICE has still managed to outperform the optimized SaM and RElim algorithms in a meaningful manner.
2) *Frequent1*( ) procedure of TRICE performs the dataset compression in a way that it places a transaction in P as a key only once. The value associated with the key refers to the total number of occurrences (support) of a transaction in a dataset. If a transaction comes again, it is rejected, but the corresponding value is incremented. This procedure is shown in steps (1) - (6). For that reason, steps (1) - (2) of *Frequent1*( ) show that no matter how much is the frequency of a transaction, it is intersected with Ls once only. It gives a considerable performance boost because duplicate unnecessary computations are avoided. It also prohibits additional duplicate processing, thus contributes to achieving efficiency.
3) In the step (2) of *FrequentItems*( ), Z shows a trimmed Ts from which, infrequent-1 items are removed. Therefore, lesser time is required to build power set of Z.
4) Twofold itemset containment check in Fs also helps to get efficiency. Containment check is done for the first time at Step (3) of *FrequentItems*( ), where containment of Z is verified in Fs. Because the probability of the existence of identical Z is high in large datasets, this containment checking stops from doing additional processing. It is useless to check the containment of Z again if it is already frequent (residing in Fs). The second containment is examined at step (6) of *FrequentItems*( ), where containment of every S of Ps is verified in Fs. Once become frequent, it is useless to process this S further.

C. ASSESEMENT OF MEMORY USAGE OF TRICE

Peak memory consumed by the algorithms is shown in Figures 23-28. Memory consumption of TRICE is minimum on majority of datasets for all minimum support thresholds. In Figure 23, memory utilized by TRICE is far less than that of the others on all *minsup* limits for the *kddcup99* dataset. No significant variation in the memory consumption of baseline algorithms is observed. The memory consumed by these algorithms is higher than that by TRICE almost by a factor of 75 at *minsup* 70%. The factor is 40 when *minsup* is decreased to 30%.

Figure 24 shows the peak memory consumption for the *PowerC* dataset. Baseline algorithms consume colossal memory. On the whole, RElim consumes the most massive memory. TRICE consumption of memory is almost 42 times less

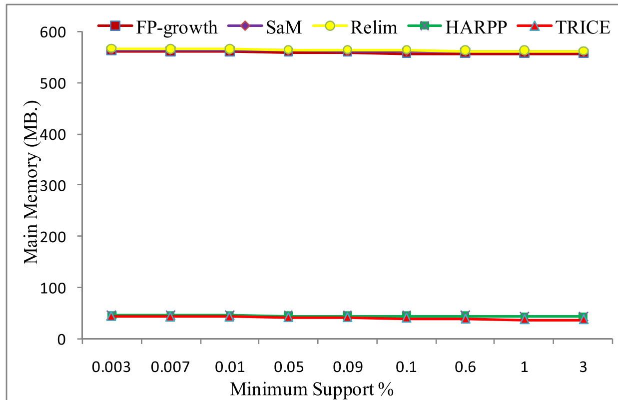

FIGURE 25. Assessment of memory use for *Online Retail* dataset.

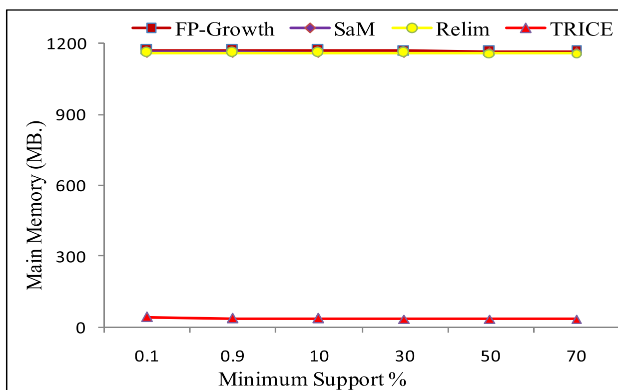

FIGURE 26. Assessment of memory use for *Record Link* dataset.

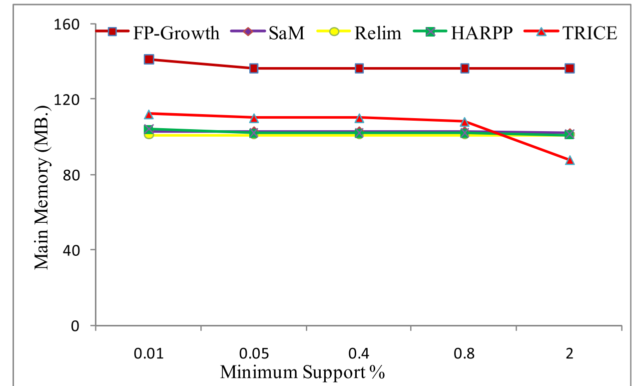

FIGURE 27. Assessment of memory use for *Extended Bakery* dataset.

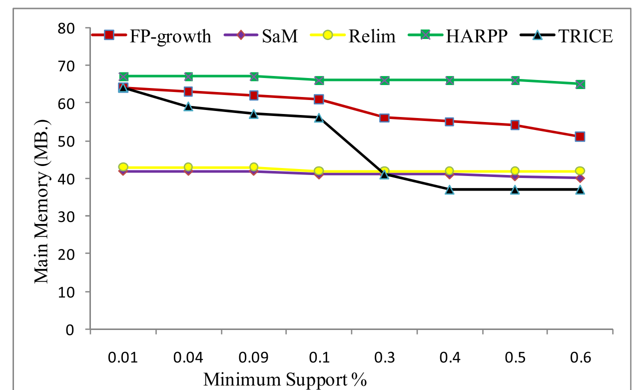

FIGURE 28. Assessment of memory use for *Food Mart* dataset.

at *minsup* 40% and 37 times less at *minsup* 1% than that of the others. Figure 25 shows the assessment of memory use for the *Online Retail* dataset. RElim takes the lead by consuming most massive memory. TRICE consumes 15 times less memory than FP-Growth and SaM and 16 times less memory than RElim at *minsup* 3%. The almost similar trend continues throughout all *minsup* thresholds.

Figure 26 shows the evaluation of memory use for *Record Link* dataset.

Now FP-Growth requires the most massive memory, but the difference among the baseline algorithms is not remarkable. TRICE consumes almost 33 times less memory than the other algorithms at *minsup* 70%, and this margin drops slightly to 27 times at *minsup* 0.1 %. Figure 27 shows the assessment on the *Extended Bakery* dataset. FP-Growth consumes the largest more memory among all algorithms. TRICE consumes the least memory at *minsup* 2%. On other *minsup* values, SaM and RElim consume less memory than TRICE, but the difference is not significant at all. This dataset is quite small in size and contains a fewer number of distinct items.

Figure 28 shows the comparison on the *Food Mart* dataset FP-growth consumes the most massive memory on the whole. At higher *minsup* values, TRICE consumes the least memory whereas, SaM consumes the least memory on lower *minsup* values.

Figures 23-28 show that memory consumption of TRICE is minimum when *minsup* is higher, but it tends to increase with the decrease in *minsup*. This behavior is apparent because, at higher *minsup*, fewer frequent 1-itemsets exist. Consequently, the intersections are not longer, resulting in smaller power sets. But as *minsup* decreases, a large number of itemsets become frequent 1-itemsets. As a result, the intersections and corresponding power sets tend to grow in size. However, TRICE consumes least memory on the majority of datasets due to the following reasons.

1) Transactions occurring, again and again, are kept once in *D1*. It helps in conserving memory because several duplicate transactions exist in large real-world datasets.
2) Large memory is required to store the power set of the whole transaction because a transaction contains both frequent and non-frequent items. Therefore, TRICE intersects each transaction to *Ls* at Step (2) in *FrequentItems* ( ) ahead of building its power set. It trims the transaction due to the removal of its non-frequent part. Thus, the resulting power set is smaller in size needing a lesser amount of memory to be stored.
3) Numerous conditional FP-trees are constructed by the FP-Growth algorithm. Furthermore, shared common

prefixes exist in a few numbers for real-world sparse datasets. Therefore, conditional FP-trees become larger, thus needing large memory for storage.

4) The split and merge steps of SaM and RElim involve recursive processing. One of the drawbacks of recursion is the extensive memory consumption, which cannot be left unattended when the problem size is too big. Eventually, the size of the underlying data structure, as well as the stack space, will grow substantially for large datasets.

Memory requirements of the descendants of FP-Growth are even higher because FIN, PrePost+, and PrePost keep PPC-tree, including 2-itemset representation, which consists of nodes existing in PPC-tree [58]. Additionally, size of PPC-tree is larger than that of associated FP-tree. According to the results, TRICE is quicker than others by almost up to two orders of magnitude. Additionally, TRICE consumes the least memory because it efficiently trims the dataset and handles duplicate transactions.

Above results manifest that TRICE consumes the least memory because it holds same transactions once, and prunes each distinct transaction by removing infrequent 1-itemsets from it. Therefore, the corresponding power sets are not bigger. Moreover, TRICE has shown exceptional runtime performance, because the intersection of identical transactions with *Ls* is done once. Besides, if an itemset is found within *Fs* during first containment check, TRICE skips the forthcoming processing for that itemset, and immediately starts checking the next itemset. This phenomenon also helps in getting efficiency.

VII. CONCLUSION

Voluminous sparse big data is being generated at a rapid pace from a variety of applications such as pervasive computing, IoT, and imbalanced behavior data. Sparse real-world data is found to be extremely useful for companies, and its inclusion leads to improved predictive analytics. Frequent itemset mining, a cornerstone of data science finds collections of items occurring together in a database. Current algorithms for mining frequent itemsets are not often validated for real-world sparse datasets, and the difference among their running times on such datasets is insignificant. This paper presents TRICE, a novel algorithm to mine frequent itemsets from real-world sparse datasets.

TRICE optimizes the HARPP algorithm by introducing the idea of Iterative Trimmed Transaction Lattice (ITTL). Instead of generating transaction lattice of entire transaction done by HARPP, TRICE cutoffs the infrequent part of a transaction first. Afterward, TRICE iteratively creates the lattice of trimmed transactions to find frequent itemsets. Furthermore, TRICE compresses the database by storing identical transactions once that prevents TRICE to do redundant processing in later steps. TRICE has been applied on six real-world sparse datasets and compared with HARPP, FP-Growth, optimized SaM and optimized RElim algorithms. TRICE has outperformed these algorithms on all datasets for all minimum support thresholds. Moreover, its memory consumption is considerably minimal on most datasets.

Modifications can be made in TRICE for solving other linked problems such as maximal frequent itemsets mining, closed frequent itemset mining, high utility itemset mining, frequent weighted itemset mining, and top-rank-k frequent pattern mining. TRICE can also be customized for mining streaming data.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, 1993.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th Int. Conf. Very Large Data Bases (VLDB)*, 1994, pp. 1–2.

[3] S. K. Kim, J. H. Lee, K. H. Ryu, and U. Kim, "A framework of spatial co-location pattern mining for ubiquitous GIS," *Multimedia Tools Appl.*, vol. 71, no. 1, pp. 199–218, 2014.

[4] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *Proc. Int. Symp. Spatial Databases*, 1995, pp. 47–66.

[5] J. S. Yoo and S. Shekhar, "A joinless approach for mining spatial colocation patterns," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 10, pp. 1323–1337, Oct. 2006.

[6] V. Jakkula and D. J. Cook, "Temporal pattern discovery for anomaly detection in a smart home," in *Proc. Int. Conf. Intell. Environ.*, 2007, pp. 339–345.

[7] K. J. Kang, B. Ka, and S. J. Kim, "A service scenario generation scheme based on association rule mining for elderly surveillance system in a smart home environment," *Eng. Appl. Artif. Intell.*, vol. 25, no. 7, pp. 1355–1364, Oct. 2012.

[8] S. Lühr, G. West, and S. Venkatesh, "Recognition of emergent human behaviour in a smart home: A data mining approach," *Pervas. Mobile Comput.*, vol. 3, no. 2, pp. 95–116, 2007.

[9] A. Jayaram, "Smart retail 4.0 IoT consumer retailer model for retail intelligence and strategic marketing of in-store products," in *Proc. 17th Int. Bus. Horizon-INBUSH ERA*, 2017.

[10] M. C. M. R. Karim, O. D. Beyan, C. F. Ahmed, and S. Decker, "Mining maximal frequent patterns in transactional databases and dynamic data streams: A spark-based approach," *Inf. Sci.*, vol. 432, pp. 278–300, Mar. 2018.

[11] M. H. ur Rehman, C. S. Liew, T. Y. Wah, and M. K. Khan, "Towards next-generation heterogeneous mobile data stream mining applications: Opportunities, challenges, and future research directions," *J. Netw. Comput. Appl.*, vol. 79, pp. 1–24, Feb. 2017.

[12] M. K. Najafabadi, M. N. Mahrin, S. Chuprat, and H. M. Sarkan, "Improving the accuracy of collaborative filtering recommendations using clustering and association rules mining on implicit data," *Comput. Hum. Behav.*, vol. 67, pp. 113–128, Feb. 2017.

[13] T. Asha, S. Natarajan, and K. N. B. Murthy, "Associative classification in the prediction of tuberculosis," in *Proc. Int. Conf. Workshop Emerg. Trends Technol. (ICWET)*, vol. 11, 2011, p. 1327.

[14] C. Y. Chin, Y. W. Meng, T. C. Lin, S. Y. Cheng, Y. H. K. Yang, and V. S. Tseng, "Mining disease risk patterns from nationwide clinical databases for the assessment of early rheumatoid arthritis risk," *PLoS ONE*, vol. 10, pp. 1–20, Apr. 2015.

[15] T. P. Exarchos, C. Papaloukas, D. I. Fotiadis, and L. K. Michalis, "An association rule mining-based methodology for automated detection of ischemic ECG beats," *IEEE Trans. Biomed. Eng.*, vol. 53, no. 8, pp. 1531–1540, Aug. 2006.

[16] Y. Ghafoor, Y. P. Huang, and S. I. Liu, "An intelligent approach to discovering common symptoms among depressed patients," *Soft Comput.*, vol. 19, no. 4, pp. 819–827, 2015.

[17] J. Nahar, T. Imam, K. S. Tickle, and Y. P. P. Chen, "Association rule mining to detect factors which contribute to heart disease in males and females," *Expert Syst. Appl.*, vol. 40, no. 4, pp. 1086–1093, Mar. 2013.

[18] M. Makhtar, N. A. Harun, and A. A. Aziz, "An association rule mining approach in predicting flood areas," in *Recent Advances on Soft Computing and Data Mining*, vol. 549. Cham, Switzerland: Springer, 2017.

[19] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly extraction in backbone networks using association rules," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1788–1799, Dec. 2012.

[20] S. C. Jeeva and E. B. Rajsingh, "Intelligent phishing url detection using association rule mining," *Hum.-Centric Comput. Inf. Sci.*, vol. 6, no. 10, pp. 1–19, 2016.

[21] K. Yoshida, Y. Shomura, and Y. Watanabe, "Visualizing network status," in *Proc. 6th Int. Conf. Mach. Learn. (ICMLC)*, vol. 4, 2007, pp. 2094–2099.

[22] H. Zhengbing, L. Zhitang, and W. Junqi, "A novel network intrusion detection system (NIDS) based on signatures search of data mining," in *Proc. 1st Int. Workshop Knowl. Discovery Data Mining (WKDD)*, 2008, pp. 10–16.

[23] H. Vathsala and S. G. Koolagudi, "Prediction model for peninsular Indian summer monsoon rainfall using data mining and statistical approaches," *Comput. Geosci.*, vol. 98, pp. 55–63, Jan. 2017.

[24] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Mining Knowl. Discovery*, vol. 15, no. 1, pp. 55–86, Aug. 2007.

[25] R. Anand and D. U. Jeffrey, *Mining of Massive Datasets*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2014, ch. 6, sec. 6.1.2, pp. 204–205.

[26] F. Albinali, N. Davies, and A. Friday, "Structural learning of activities from sparse datasets," in *Proc. 5th Annu. IEEE Int. Conf. Pervasive Comput. Commun.(PerCom)*, Mar. 2007, pp. 221–228.

[27] S. Choudhury, Q. Ye, M. Dong, and Q. Zhang, "IoT big data analytics," *Wireless Commun. Mobile Comput.*, vol. 2019, p. 8, Apr. 2019.

[28] M. M. Rathore, A. Paul, A. Ahmad, M. Anisetti, and G. Jeon, "Hadoop-based intelligent care system (HICS): Analytical approach for big data in IoT," *ACM Trans. Internet Technol.*, vol. 18, no. 1, p. 8, 2017.

[29] B. Twardowski and D. Ryzko, "IoT and context-aware mobile recommendations using multi-agent systems," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol. (WI-IAT)*, Dec. 2015, pp. 33–40.

[30] J. Zhou, L. Hu, F. Wang, H. Lu, and K. Zhao, "An efficient multidimensional fusion algorithm for IoT data based on partitioning," *Tsinghua Sci. Technol.*, vol. 18, no. 4, pp. 369–378, Aug. 2013.

[31] J. Vanhoeyveld and D. Martens, "Imbalanced classification in sparse and large behaviour datasets," *Data Mining Knowl. Discovery*, vol. 32, no. 1, pp. 25–82, 2018.

[32] E. J. de Fortuny, D. Martens, and F. Provost, "Predictive modeling with big data: Is bigger really better?" *Big Data*, vol. 1, no. 4, pp. 215–226, 2013.

[33] M. Yasir, M. A. Habib, S. Sarwar, C. M. N. Faisal, M. Ahmad, and S. Jabbar, "HARPP: HARnessing the power of power sets for mining frequent itemsets," *Inf. Technol. Control*, vol. 48, no. 3, pp. 415–431, 2019.

[34] C. Borgelt, "Simple algorithms for frequent item set mining," in *Advances in Machine Learning*, 2nd ed. Berlin, Germany: Springer, 2010, pp. 351–369.

[35] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. London, U.K.: Pearson, 2009, p. 1203.

[36] B. Lent, A. Swami, and J. Widom, "Clustering association rules," in *Proc. 13th Int. Conf. Data Eng.*, 1997, pp. 220–231.

[37] S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating association rule mining with relational database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, vol. 98, 1998, pp. 343–354.

[38] R. Srikant, Q. Vu, and R. Agrawal, "Mining association rules with item constraints," in *Proc. KDD*, vol. 97, 1997, pp. 67–73.

[39] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang, "Exploratory mining and pruning optimizations of constrained associations rules," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 13–24, 1998.

[40] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo, "Finding interesting rules from large sets of discovered association rules," in *Proc. 3rd Int. Conf. Inf. Knowl. Manage. (CIKM)*. New York, NY, USA: ACM Press, 1994, pp. 401–407.

[41] G. Grahne, L. V. S. Lakshmanan, and X. Wang, "Efficient mining of constrained correlated sets," in *Proc. 16th Int. Conf. Data Eng.*, 2000, pp. 512–521.

[42] A. Savasere, E. R. Omiecinski, and S. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proc. Int. Conf. Very Large Data Bases (VLDB)*, vol. 5, 1995, pp. 432–444.

[43] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2000, pp. 1–12.

[44] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A maximal frequent itemset algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 11, pp. 1490–1504, Nov. 2005.

[45] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging vertical mining of large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2000, pp. 22–33.

[46] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 372–390, May 2000.

[47] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2003, p. 326.

[48] J. Park, M. Chen, and P. Yu, "An effective hash-based algorithm for mining association rules," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, vol. 95, 1995, pp. 175–186.

[49] S. A. Ozel and H. A. Guvenir, "An algorithm for mining association rules using perfect hashing and database pruning," in *Proc. 10th Turkish Symp. Artif. Intell. Neural Netw.*, 2001, pp. 257–264.

[50] H. Toivonen, "Sampling large databases for association rules," in *Proc. 22th Int. Conf. Very Large Data Bases*, 1996, pp. 134–145.

[51] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *ACM SIGMOD Rec.*, vol. 26, no. 2, pp. 255–264, 1997.

[52] Y. J. Tsay and J. Y. Chiang, "CBAR: An efficient method for mining association rules," *Knowl.-Based Syst.*, vol. 18, nos. 2–3, pp. 99–105, 2005.

[53] R. Duwairi and H. Ammari, "An enhanced CBAR algorithm for improving recommendation systems accuracy," *Simul. Model. Pract. Theory*, vol. 60, pp. 54–68, Jan. 2016.

[54] R. P. Gopalan and Y. G. Sucahyo, "High performance frequent patterns extraction using compressed FP-tree," in *Proc. SIAM Int. Workshop High Perform. Distrib. Mining*, Orlando, FL, USA, 2004.

[55] Z. Deng and Z. Wang, "A new fast vertical method for mining frequent patterns," *Int. J. Comput. Intell. Syst.*, vol. 3, no. 6, pp. 733–744, 2010.

[56] Z. Deng, Z. Wang, and J. Jiang, "A new algorithm for fast mining frequent itemsets using N-lists," *Sci. China Inf. Sci.*, vol. 55, no. 9, pp. 2008–2030, 2012.

[57] Z.-H. Deng and S.-L. Lv, "Fast mining frequent itemsets using nodesets," *Expert Syst. Appl.*, vol. 41, no. 10, pp. 4505–4512, 2014.

[58] Z. H. Deng and S. L. Lv, "PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning," *Expert Syst. Appl.*, vol. 42, no. 13, pp. 5424–5432, 2015.

[59] W. Song, B. Yang, and Z. Xu, "Index-BitTableFI: An improved algorithm for mining frequent itemsets," *Knowl.-Based Syst.*, vol. 21, no. 6, pp. 507–513, Aug. 2008.

[60] B. Vo, T. Le, F. Coenen, and T. P. Hong, "Mining frequent itemsets using the N-list and subsume concepts," *Int. J. Mach. Learn.*, vol. 7, no. 2, pp. 253–265, 2016.

[61] C. Borgelt, "Keeping things simple: Finding frequent item sets by recursive elimination," in *Proc. 1st Int. Workshop Open Source Data Mining, Frequent Pattern Mining Implement.*, 2005, pp. 66–70.

[62] C. Borgelt and X. Wang, "SaM: A split and merge algorithm for fuzzy frequent item set mining," in *Proc. IFSA/EUSFLAT Conf.*, 2009, pp. 968–973.

[63] P. Fournier-Viger, J. C. W. Lin, A. Gomariz, T. Gueniche, A. Soltani, and Z. Deng. (2016). *The SPMF Open-Source Data Mining Library Version 2*. [Online]. Available: http://www.philippe-fournier-viger.com/spmf/

[64] A. Dekhtyar and J. Verburg. (2009). *ExtendedBakery–Datasets*. [Online]. Available: https://wiki.csc.calpoly.edu/datasets/wiki/ExtendedBakery

[65] B. Dagenais. (2015). *A Few Data Mining Algorithms in Pure Python*. [Online]. Available: https://github.com/bartdag/pymining

MUHAMMAD YASIR received the M.S. degree in computer science from SZABIST, Karachi, Pakistan. He is currently pursuing the Ph.D. degree in computer science with National Textile University, Faisalabad, Pakistan. His research interests include data science, machine learning, agent based modeling and simulation, network security and cryptography, and the IoT.

MUHAMMAD ASIF HABIB received the Ph.D. degree from JKU Linz Austria. He is currently working as an Associate Professor with the Department of Computer Science, National Textile University, Faisalabad, Pakistan. His research interests include information/network security, authorization/role based access control, the IoT, cloud/grid computing, association rule mining, recommender systems, wireless sensor networks, and vehicular networks. He is also serving as a Technical Reviewer of top journals and conferences.

MUHAMMAD ASHRAF received the B.S. degree in computer systems engineering from the Balochistan University of Engineering and Technology Khuzdar, the M.S. degree in computer systems engineering from the University of Engineering and Technology Taxila, Pakistan, and the Ph.D. degree in computer engineering from Sungkyunkwan University, South Korea, in 2019.

He is currently working as an Assistant Professor with the Faculty of Information and Communication Technology, Balochistan University of Information Technology, Engineering and Management Sciences, Pakistan. His research interests include wireless sensor networks, intelligent systems, and modeling and simulation.

SHAHZAD SARWAR received the B.Sc. degree in civil engineering from the University of Engineering and Technology (UET) Taxila, Pakistan, in 1998, the M.S. degree in computer science from the Lahore University of Management Sciences (LUMS), Pakistan, in 2004, and the Ph.D. degree in electrical engineering and information technology from the Vienna University of Technology, Austria, in 2008. He is currently an Assistant Professor with the Punjab University College of Information Technology (PUCIT), University of the Punjab, Lahore, Pakistan. His main areas of research are the Internet of things (IoT), machine-to-machine (M2M) communication, optical burst switched (OBS) networks, network-on-chip (NoC), and high-speed data centers.

Dr. Sarwar has made significant contributions to research. He has already published more than 40 articles in well reputed journals and conferences. His research has more than 200 citations. He has already supervised more than 10 M.Phil. theses. He has guided one Ph.D. and is currently supervising three Ph.D. students. Further, as a principal investigator (PI) and Co-PI, he has secured research funding amounting around USD1.0 million.

MUHAMMAD UMAR CHAUDHRY received the B.S. degree in computer engineering from Bahauddin Zakariya University, Multan, Pakistan, in 2009, and the Ph.D. degree in computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2019. His research interests include data science, recommender systems, feature selection, and machine learning.

HAMAYOUN SHAHWANI received the B.S. degree from the Balochistan University of Information Technology, Engineering, and Management Sciences, in 2010, and the Ph.D. degree from Sungkyunkwan University, South Korea, in 2019.

He is currently serving as an Assistant Professor with the Balochistan University of Information Technology, Engineering, and Management Sciences. His research interests include machine-to-machine communication, device-to-device communication, and vehicular adhoc networks.

MUDASSAR AHMAD has 17 years' experience as a Network Manager at Textile Industry. He is currently serving as an Assistant Professor with the Department of Computer Science, National Textile University, Pakistan. His research work is published in many conferences and journals. His research interests include the Internet of Things, bid data, and healthcare. He is an Associate Editor in the IEEE NEWSLETTERS.

CH. MUHAMMAD NADEEM FAISAL received the B.S. degree in information technology from Allama Iqbal Open University, Pakistan, in 2005, the M.S. degree in computer science from the Blekinge Institute of Technology, Karlskrona, Sweden, in 2009, and the Ph.D. degree in computer engineering from the University of Oviedo, Oviedo, Spain, in 2017.

He is currently an Assistant Professor with National Textile University, Faisalabad, Pakistan. His research interests include cognitive science, human performance, and the evaluation of user interfaces for commercial applications.