IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 24, 2020, accepted February 1, 2020, date of publication February 5, 2020, date of current version February 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2971834

D-GENE: Deferring the GENEration of Power Sets for Discovering Frequent Itemsets 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: Muhammad Asif Habib (drasif@ntu.edu.pk)

ABSTRACT Sparseness is the distinctive aspect of big data generated by numerous applications at present. Furthermore, several similar records exist in real-world sparse datasets. Based on Iterative Trimmed Transaction Lattice (ITTL), the recently proposed TRICE algorithm learns frequent itemsets efficiently from sparse datasets. TRICE stores alike transactions once, and eliminates the infrequent part of each distinct transaction afterward. However, removing the infrequent part of two or more distinct transactions may result in similar trimmed transactions. TRICE repeatedly generates ITTLs of similar trimmed transactions that induce redundant computations and eventually, affects the runtime efficiency. This paper presents D-GENE, a technique that optimizes TRICE by introducing a deferred ITTL generation mechanism. D-GENE suspends the process of ITTL generation till the completion of transaction pruning phase. The deferral strategy enables D-GENE to generate ITTLs of similar trimmed transactions once. Experimental results show that by avoiding the redundant computations, D-GENE gets better runtime efficiency. D-GENE beats TRICE, FP-growth, and optimized versions of SaM and RElim algorithms comprehensively, especially when the difference between distinct transactions and distinct trimmed transactions is high.

INDEX TERMS Big data applications, pattern recognition, association rules, frequent item set mining, IoT.

I. INTRODUCTION

In the realm of data science, association analysis has emerged as an unavoidable technique that explores strong relationships in voluminous databases. Association analysis does the exploration in such a way that the transactional presence of some items assures the presence of other items. Unprecedented ability to mine and furnish profound data insights has made association analysis an inevitable tool at present. Since its inception, association analysis is adopted by giant retail companies to formulate ripened marketing strategies [1]. Association analysis is increasingly being deployed in numerous areas such as recommendation systems [2], study of market basket data [3], smart systems [4]–[7], IoT [8]–[10], fog and mobile edge computing [11], mining of data streams and mobile data streams[12], [13], natural catastrophes forecasting [14], medical [15]–[19], predicting weather[20], and securing networks [21]–[24].

Association analysis commences with identifying frequent itemsets that are used later to infer association rules [1]. Itemsets become frequent if the frequency of their transactional occurrence named as, support, is larger than the already defined threshold of minimum support. Identification of frequent itemsets entails numerous computing resources; therefore, it exerts a decisive burden on the efficiency of mining techniques. However, subsequent exploration of association rules can be done trivially without affecting the efficiency of the underlying technique. Therefore, efficient identification of frequent itemsets is still a vigorous research problem, even though numerous techniques have been proposed so far. It explores collections of items placed jointly in a transactional database [1]. In a database containing transactions,

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

the transactions represent baskets of various things shoppers purchase [25]. Numerous large retail businesses such as Netflix, Amazon, YouTube, and e-bay identify frequent itemsets to recommend exciting products to the users further. Besides generating association rules, frequent itemset mining also generates episodes, and correlations [26].

Sparseness is the distinctive aspect of large real-world data generated by numerous sources, including pervasive computing, behavioral data, transactional data, and IoT applications, especially fog and mobile edge computing (MEC). Mobile big data analytics is one of the biggest scenarios benefitting from fog and MEC. Various monitoring and surveillance systems can use fog nodes to find the interesting frequent patterns in the local and regional data without delays caused by the traditional cloud computing. Moreover, big transactional data is generated by the shopping made through mobile phones. Therefore, voluminous data is transmitted to cloud for exploring frequent items that lacks efficiency. However, computing in proximity to IoT devices can eficiently explore frequent items.

The applications using mobile fog nodes include SWAMP [27], protecting the Android devices with the GPU-aided antivirus [28], observation of traffic by drones [29], managing power usage [30], delivery by drones [31], managing congestion of traffic [32], driving autonomously [33], cognitively assisting by wearable [34], smart street lamp [35], and live broadcast of videos [36]. Several fog applications use mobile fog nodes such as sensors, laptops, and drones having limited processing capacity.

A database of transactions made by a hypermart depicts a real-world sparse dataset. A large number of products are available in a hypermart. However, a small subset of the products is purchased by a shopper, shown by his pertinent transaction. Hence, a transaction is merely a tiny subset of N, where N corresponds to a set of all distinct products a hypermarket contains. Sparsity is also shown by imbalanced behavioral big data [37].

Furthermore, the readings of mobile edge computing devices such as sensors, are several thousand in numbers, but the number of activity occurrences is negligible concerning the readings. Therefore, big sparse data is being generated today at rapid pace [38]–[49]. The computational and storage cost for mining large and sparse datasets is very high in edge-devices [50]. Therefore, there is an immense need to introduce the mining techniques requiring least computing resources for fog and mobile edge computing in IoT.

Regardless of the lack of comprehensiveness, the rarity of information generated by sparse data makes it a valuable asset for large businesses. This rare information is imperative for learning the varied activities of consumers; thus, it leads towards better predictive analytics. Empirical demonstration reveals that predictive models based on sparse data have massively improved the predictive performance [51].

Techniques based on Iterative Transaction Lattice (ITL) are recently proposed to learn frequent itemsets from large real datasets that are sparse too [52], [53]. Sparse real-world transactional datasets contain numerous similar transactions. Based on the Iterative Trimmed Transaction Lattice (ITTL), the TRICE algorithm efficiently explores frequent itemsets from sparse real datasets [53]. For large datasets, trimmed versions of two or more distinct transactions become similar. Due to its intrinsic feature of making ITTLs on the fly, TRICE has to generate ITTLs of similar trimmed transactions repeatedly. This redundant computation may affect the running time of TRICE.

For instance, let transaction1 = {1, 2, 3, 4} and transaction2 = {1, 2, 3, 4, 5} in a dataset. TRICE stores both transactions in a dictionary ADT named as *Dict*. *L-set* is a *set* ADT containing frequent 1-itemsets. Assume that the contents of *L-set* are {1, 2, 3}, because itemsets {4}, and {5} are assumed to be infrequent, thus rejected. The *FrequentItems*( ) procedure of TRICE will iteratively take each key from *Dict* and intersect it with *L-set*, and then generate the ITTL of the intersection on the fly. The intersection corresponds to a trimmed transaction because infrequent 1-itemsets are rejected. Interestingly, the intersection of each keyof *Dict* with *L-set* results in a similar trimmed transaction having items,{1, 2, 3}. TRICE will generate ITTL of {1, 2, 3} twice, because, it generates ITTL of a transaction just after trimming it (by intersecting it with *L-set*).

The experimental study has revealed that numerous similar trimmed transactions exist in sparse datasets. Therefore, TRICE has to generate ITTLs of the same trimmed transaction several times. Eventually, the redundant computations of TRICE affect runtime efficiency. This work is an attempt to get rid of the unnecessary computations by optimizing the behavior of TRICE. A technique namely, Deferring the GENEration of Power sets for Mining Frequent Itemsets from Sparse Big data (D-GENE), is proposed in this paper. D-GENE optimizes TRICE by introducing a deferred ITTL mechanism. Instead of generating ITTLs just after making intersections, D-GENE delays the procedure until the completion of the intersection phase.

The efficiency comparison of D-GENE is made with TRICE, optimized SaM [54], optimized RElim [54], and FP-growth algorithms on six sparse real datasets. D-GENE outperforms the other techniques by showing better run time performance and least memory usage.

The organization of this paper is as follows. A review of related work is given in Section 2. Section 3 describes the problem and its essential definitions. D-GENE is explained in Section 4, followed by an example in section 5. Experimental results and detailed discussion is given in Section 6. Section 7 concludes this study and presents some potential research issues.

II. RELATED WORK

The simple brute-force technique [55] first represents an Itemset Lattice (IL) by generating a power set that contains all subset combinations. Figure 1 shows the IL made by the simple technique. All subsets in the IL are designated as candidate subsets initially. Subsequently, the algorithm

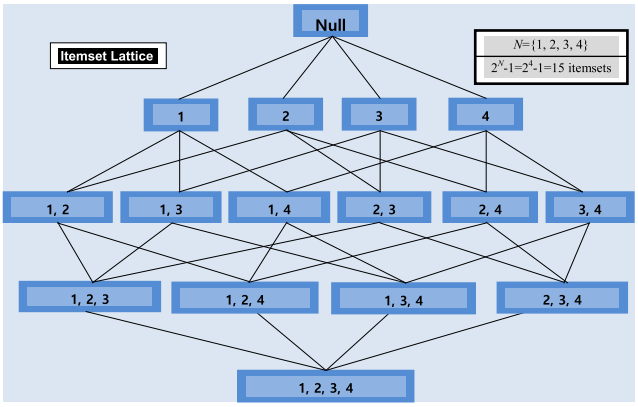

FIGURE 1. Itemset Lattice made by Naive Technique.

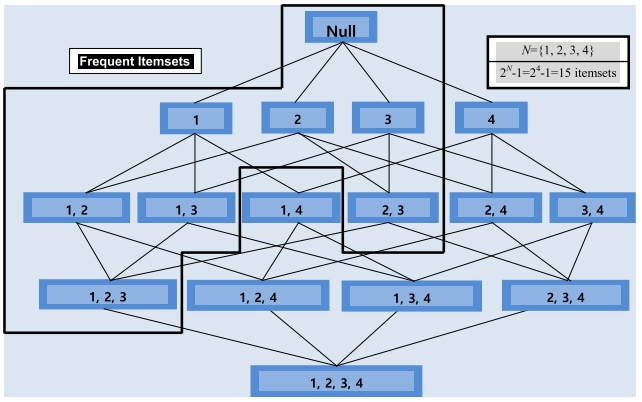

FIGURE 2. Hierarchical monotonicity in Apriori.

iteratively counts the transactional presence of each candidate subset by contrasting it against every transaction. A particular subset is kept as frequent if its presence count is not less than the threshold value of minimum support. Subsets having transactional presence count less than the threshold value of minimum support are regarded infrequent, thus rejected.

Though the algorithm follows a simple principle to explore all frequent subsets, yet it places a considerable burden on computing resources. The exponential increase of candidate subsets and their support counting afterward shows a phenomenal rise in the runtime. Furthermore, considering a large size of N, the algorithm has to produce 2^N candidate subsets that need gigantic memory to be stored. Both factors make the algorithm unusable in reality. At present, big retail companies present numerous items for sale resulting in a voluminous N.

Apriori candidate-set generation and test method exhibits a breadth-first search strategy to explore frequent itemsets [3]. Based on the hierarchical monotonicity principle, the technique generates numerous candidates. Hierarchical monotonicity states that all candidate subsets of a frequent itemset are frequent as well. Likewise, a superset of an infrequent candidate subset is also infrequent. Figure 2 shows a depiction of the principle, where a pass over the database reveals that the transactional occurrence of candidate {4} is less than the minimum support threshold. Thus, the hierarchical monotonicity principle declares the supersets of candidate itemset {4}, infrequent. Following this way, the Apriori algorithm optimizes the brute-force method by pruning the IL. Though the candidates are lesser in numbers than the candidates produced by brute-force technique, yet found abundant. Therefore, massive memory is required to store them. Moreover, numerous passes over the large database for counting candidates' transactional presence give a huge rise in the runtime. Many Apriori-based methods are proposed, but they also have to produce abundant candidates and count their transactional occurrences later [56]–[62].

Later, Apriori-based hashing and pruning techniques are proposed that improve the generation of candidate 2-itemsets. Adoption of database pruning and better hashing give efficient candidate contrasting and identification of frequent itemsets [63], [64]. Apriori-based clustering methods make multiple clusters according to the length of transactions [65], [66]. Afterward, the transactions are stored in pertinent clusters. Though candidates are generated in Apriori-like fashion, yet clustering methods contrast each candidate in its relevant cluster rather than the whole database. Therefore, clustering methods are more efficient than Apriori. Vertical representation- based recursive schemes show each itemset in a vertical diff-set or Tid-set manner [67]-[70]. The intersection of sets is used to count the support of itemsets. These techniques explore frequent itemsets by using the depth-first search strategy. Avoiding the repetitive passes over the database, efficient counting of support is done by Tid-set. However, large cardinality of Tid-set or a larger dataset slows down the algorithmic efficiency. Memory consumption becomes large when Tid-sets are longer, and abundant candidates are generated.

Sampling [71] and the dynamic count of itemsets (DIC) [72] perform more efficiently because they soften the stringent division between the support check and candidate generation. DIC initiates the production of additional candidate itemsets, whenever the support of a candidate and threshold of minimum support equalize. A prefix tree is utilized to explore frequent itemsets efficiently,

RElim (Recursive Elimination) and SaM (Split and Merge) algorithms are horizontal representation-based recursive algorithms [73], [74]. RElim recursively eliminates items. It commences by opting for the transactions that have a frequent item in the smallest amount. Later, the item is eliminated; thus, the resulting dataset is smaller. RElim recursively processes the enduring itemsets in the dataset. It retains information of the explored items during recursion and finds frequent itemsets pertinent to the eliminated item that is the cause of invoking the recursion. RElim performs this after identifying all itemsets in the reduced dataset.

The procedure is repeated by choosing the subsequent least frequent itemset and so forth. SaM algorithm simplifies RElim and works in two steps. In split-step, a copy of every array that commences with the first item of the foremost transaction is made. The copy is placed in a new array, and that opening item is eliminated. The procedure

is repeatedly performed recursively to explore each frequent itemset. Subsequently, a merging step is initiated along with the pruned dataset to gain the conditional pattern base. Later on, optimizations have been done that enabled both RElim and SaM to efficiently process sparse datasets.

In pattern growth-based methods, the FP-Growth algorithm is the maiden one [75]. It introduces an extended structure of prefix-tree. The database is kept in a trie data structure, and a linked list is held by every itemset that passes over all transactions that have the itemset. FP-tree (Frequent Pattern tree) is a condensed tree that holds this structure.The nodes keep a counter for tracking the number of transactions sharing the branches entirely through them. A pointer is used that points the subsequent occurrence of an itemset in the tree. Eventually, every presence of the itemset is connected and represented by the FP-tree. It maintains a table containing headers is maintained that stores each discrete itemset with its support accompanied by a pointer for showing its earliest occurrence. A compact representation enables FP-growth to use lesser memory as compared to Apriori.The algorithm becomes less efficient when patterns become longer or the threshold of minimum support is lower because conditional trees are produced in large quantities [76].

Consequently, the PPC tree (Pre-order Post-order Code tree) is introduced to hold the information of frequent itemsets [77]. It is more efficient than FP-tree because one tree pass is required for the detection of an N-list containing frequent 1-itemsets. Conversely, FP-tree based methods are needed repeated tree traversals. Based on PPC-tree, PrePost algorithm commences by constructing a PPC-tree [78]. It utilizes an algorithm to generate the tree. Subsequently, it creates N-lists to represent 1-itemsets. N-list depicts a compressed transaction ID list (TID list) to represent the attributes of an itemset. Later, a divide-and-conquer method is utilized to explore frequent itemsets. It is not obligatory to make added trees in the succeeding iterations, hence its efficiency is better than that of FP-Tree. The limitation of PrePost is its use of the Apriori-like technique to explore frequent itemsets; however, it uses N-list single-path property for reducing the search space.

An itemset is represented by PPC tree-based Nodeset. Post-order or preorder code is used to encode a node within Nodeset. Based on Nodeset, the FIN algorithm is proposed whose performance is comparable with that of PrePost; however, FIN requires less memory [79]. PrePost+ brings an optimization to PrePost by utilizing N-list to explore frequent itemsets [80]. Children-Parent Equivalence pruning is utilized to lessen the search space and to elude monotonous search.

Furthermore, the *subsume index* is introduced for added improvement inefficiency [81]. A *subsume* index is accompanied by a frequent 1-itemset that depicts a list of frequent 1-itemsets discovered alongside it. NSFI algorithm is introduced on the basis of *subsume* index [82] that performs its mingling with N-List for efficient working and least consumption of memory. Hash tables are used to construct N-lists that make NSFI efficient. N-list intersection method is also got better.

HARPP algorithm iteratively generatesthe power set of each transaction and represents it by a Transaction Lattice (TL) [52]. It takes every transaction as a *set* ADT, makes its TL, and checks the existence of every subset in F, a *set* that is used to store frequent itemsets. The existence of a subset in F manifests that it is frequent previously. For that reason, the subset is regarded as frequent and removed immediately. Otherwise, the subset with a count of its support is placed into a *dictionary* as a key and value pair. Soon after placing in a *dictionary*, the support of the subset and minimum support threshold are compared. An equalizer makes the subset frequent, discards it from *dictionary,* and stores in F. HARPP does not keep the dataset in memory and explores the frequent items in a single dataset scan. Though HARPP works efficiently, yet its efficiency deteriorates when the datasets have longer transactions.

TRICE algorithm optimizes HARPP by generating iterative TLs too; however, TRICE prunes the database first by discarding the infrequent itemsets from every distinct transaction [53]. Therefore, it represents Iterative Trimmed Transaction Lattices (ITTLs) by generating power sets of pruned transactions, where each ITTL is a tiny subset of IL. TRICE is found to be extremely efficient on sparse datasets. The probability of the existence of similar transactions is high in sparse real datasets. TRICE conserves memory by keeping similar transactions once.

TRICE immediately makes ITTL of a trimmed version of a transaction for further processing. For sparse datasets, trimmed versions of two or more dissimilar transactions become similar often. Since TRICE makes ITTL of each trimmed transaction on the fly, it generates ITTL of similar trimmed transactions repeatedly. This redundant computation may affect the run time efficiency of TRICE.

To get rid of the redundant computations of TRICE, another ITTL-based technique named as, D-GENE is proposed in this paper that introduces a deferred ITTL generation strategy. Deferring the step of ITTL generation enables D-GENE to make power set of identical trimmed transactions only once that further improves the running time.

III. BASIC CONCEPTS

This section initiates by presenting the concepts pertinent to D-GENE. Notations and descriptions are given in Table 1.

Table 2 presents a database, *DB,* to be used for illustrating purpose.Support of an *I-set* denoted by *Sup*(*I-set*), is the total transactions in numbers containing the *I-set*. An *I-set* is made frequent if *Sup*(*I-set*) $\geq$ (*minsup* X $|DB|$). A frequent *I-set* that has k items is denoted as a frequent k-*I-set*. Frequent itemset mining explores all *I-set*s whose support $\geq$ (*minsup* X $|DB|$). An ITTL is represented by making a power set of a trimmed T.

A power set contains all subsets of a set excluding the empty subset.

TABLE 1. Notations and descriptions.

Notation	Description
DB	A transactional database
N	Total number of items in a transactional database
T	A *set* ADT representation of *D*
I-set	An itemset
Sup(*I-set*)	Support of an *I-set*
|*DB*|	Total number of transactions in *DB*
minsup	A threshold
Dict1	A *dictionary* to keep *T* and support of *T* as a key and a value, respectively. It stores identical transactions only once; thus, the size of *Dict1* represents the total number of distinct transactions in a dataset.
K1	Represents a key in *Dict1*
Dict2	A *dictionary* to store *I-set* and its support as a key and a value, respectively.
Dict3	A *dictionary* to store pairs of key-value filtered from *Dict2* that represent frequent 1-*I-set*s.
L-set	A *set* that stores the keys that are taken out from *Dict3*
Z	A *set* to store the intersection of *L-set* with *K1*. Intersection results in a trimmed transaction.
Dict4	A dictionary that stores *Z* and its support as a key and a value respectively. It stores identical Zs only once; thus, the size of *Dict4* represents the total number of distinct trimmed transactions.
K2	Represents a key in *Dict4*
P-set	A *list* that stores the power set of *K2*
S	A subset of *P-set* that could be a frequent *I-set*
Dict5	A *dictionary* that stores *S* and support of *S* as a key and value respectively.
F-set	A *set* to contain frequent itemsets

IV. D-GENE: THE PROPOSED METHOD

N corresponds to a large itemset having all items, and each transaction is composed of some items of *N*. Thus, a transaction represents a subset of *N*. Consequently, an ITTL is a tiny subset of IL. D-GENE iteratively generates ITTL of each distinct trimmed transaction once to explore frequent *I-set*.

The pseudocode of D-GENE is shown in Figure 3.

D-GENE is composed of two procedures. It commences by invoking *FindFrequent1*( ), which makes a pass over the database and performs the following actions.

1. The following actions are performed in an iterative manner for all transactions.
   - 1.1. *FindFrequent1* ( ) reads a *T* in Step (1)-(6), makes *T* and its support, a key, and value respectively. Afterward, the key and value are stored in *Dict1*. The value is set to 1 if *T* occurs for the first time in the database; otherwise, the value is incremented on each subsequent arrival of the same *T*. Higher value for a key shows the multiple occurrences of a transaction in a database. However, similar transactions are kept once in *Dict1*.
   - 2.2. Each *I-set* of *T* is obtained in Step (7)-(14), placed in *Dict2* as a key, and its value is set to 1. The repeated occurrence of the same *I-set* causes an increment in its value.
2. After making a pass over the entire database, in Step (15), all keys whose supports are not less than *minsup* are obtained and placed in *Dict3*. Hence, *Dict3* contains keys and values to represent frequent 1-*I-set*s and their supports respectively.
3. Keys from *Dict3* are obtained and kept in *L-set* in Step (17). As a result, all frequent 1-*I-sets* are contained in *L-set*.

The output of the procedure consists of *Dict1* that contains all distinct *T* with their supports and *L-set*.

The flowchart of *FindFrequent1*( ) is shown in Figure 4.

Subsequently,*FindFrequentAll*( ) is called and provided with *L-set*, *Dict1,* |*DB*|, and *minsup*. Then for every key, *K1* pointing to *T* in *Dict1*, the next two actions are done.

1. *K1* is obtained in Step (1) - (2) and intersected with *L-set*. The intersection discarded the infrequent items of *K1* and kept in *Z*. Thus, *Z* represents a pruned version of the presently obtained transaction.
2. In Step (3) - (4), the existence of *Z* is checked in *Dict4*. If *Z* is there previously, the value of *Z* in *Dict4* is increased by the amount equal to that of *K1*. Otherwise, *Z* is stored in *Dict4* as a new key and assigned the value of *K1*.

After these two steps, *Dict4* contains all distinct trimmed transactions stored as keys with corresponding values. In *Dict4*, identical transactions are stored only once. The subsequent tasks are done for every Key, *K2* (trimmed *T*) kept in *Dict4*.

3. A *K2* is read in Step (9) - (10), and a power set *P-set* is generated that represents ITTL of *K2*. In Step (11), for each subset, *S* of *P-set*, the next sub-tasks are executed.
   - 4.1. In Step (12), an *S* is read, and its containment is verified in *F-set*. If *S* is present there previously, it is considered frequent, so rejected and the procedure reads next *S*. Or else, *S* is kept in *Dict5* as a key in Step (13) - (14) whose value is equivalent to that of *K2* in *Dict4* if *S* arrives for the first time. If *S* is there already, then the previous value of *S* is increased by the amount of *K2* in *Dict4* in Step (15) - (16).

```
Algorithm: D-GENE Algorithm
Input: A database of transactions DB, minimum support (minsup),
and |DB| be the total number of transactions in DB.
Output: F-set
Call FindFrequent1 (DB, minsup, |DB|)
Call FindFrequentAll (Dict1, L-set, |DB|, minsup)
FindFrequent1 (DB, minsup, |DB|)
(1) For each T in DB, do
(2)    If T not already exists in Dict1, then do
(3)       Store T as a key in Dict1 and set its value to 1
(4)    Else
(5)        Increment the value of T
(6)    End If
(7)    For each I-set in T, do
(8)       If I-set not already exists in Dict2, do
(9)           Store I-set as a key in Dict2 and set its value to
              1
(10)     Else
(11)        Increment the value of I-set
(12)     End If
(13)   End For
(14) End For
(15) Take out all key-value pairs from Dict2 having
     value ≥ (minsup X |DB|)and Store in Dict3
(16) Delete Dict2
(17) Store all keys of Dict3 into L-set
(18) Delete Dict3
(19) Return Dict1, L-set
FindFrequentAll (Dict1, L-set, minsup)
(1) For each K1 in Dict1, do
(2)    Store intersection of K1 and L-set in Z
(3)    If Z not already exists in Dict4, then do
(4)       Store Z in Dict4 as a key and assign to it the value of
          K1
(5)    Else, do
(6)       Increase the value of Z by the value of K1
(7)    End If
(8) End For
(9) For each K2 in Dict4, do
(10)  Make Power set of K2 to represent its ITTL and store in P-
      set
(11)  For each S in P-set, do
(12)    If S not already exists in F-set, then do
(13)     If S not already exists in Dict5, then do
(14)       Store S in Dict5 as a key and assign to it the value
           of K2
(15)     Else, do
(16)        Increase the value of S by the value of K2
(17)     End If
(18)     If value of S ≥ minsup X |DB|, then do
(19)        Take out S from Dict5 and store into F-set
(20)     End If
(21)    End If
(22)   End For
(23) End For
```

FIGURE 3. Pseudocode of D-GENE.

This happens as the support possessed by S is equal to the support owned by T, where Z represents the intersection of T and *L-set*.

4.2. When S is placed in *Dict4*, its value and *minsup* are contrasted in Step (18). If both equalize,S is considered frequent.

4.3. Then *I-set* is taken out from *Dict4* and stored in *F-set* shown by Step (19).

5. *F-set* is printed in Step (24).

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

V. D-GENE EXAMPLE

D-GENE example is given based on the database given in Table 2. In this example, *minsup* is set to 66%, which means that to be frequent, an *I-set* must occur in more than three transactions. D-GENE commences by invoking the *FindFrequent1*( ) procedure. *FindFrequent1*( ) obtains the first T

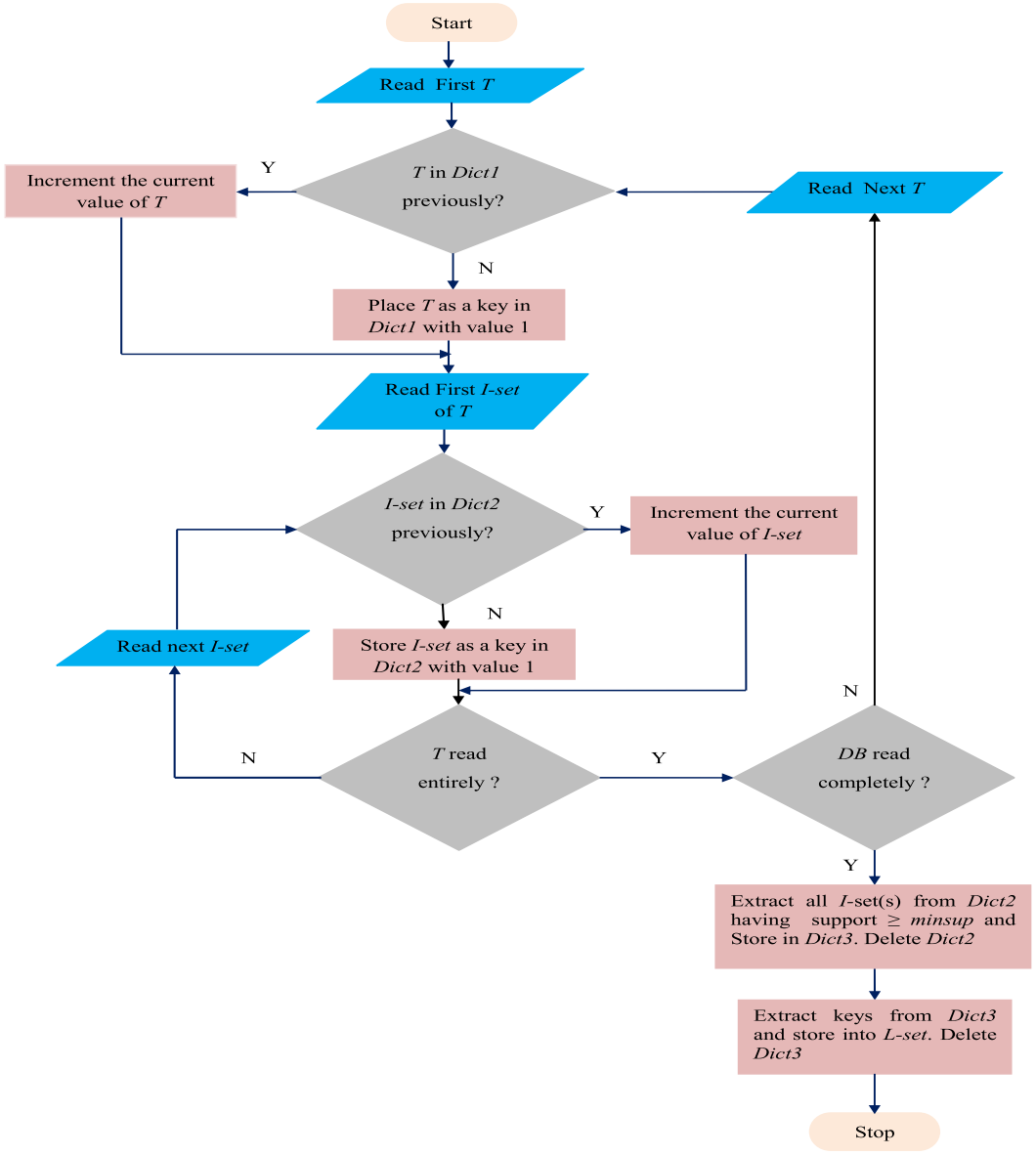

FIGURE 4. Flowchart of *FindFrequent1*( ) Procedure.

in Figure 6 and places it in *Dict1* by making it a key. The value assigned to the key is one, due to the initial appearance of the key. Shortly, *FindFrequent1*( ) obtains each *I-set* of *T* and places into *Dict2* by making it a key. The value assigned to each key is one because this is the first-ever arrival of all four keys to *Dict2*.

Figure 7 shows the arrival of the 2nd transaction that is placed in *Dict1* as a new key. The value of the key is set to one due to its maiden arrival. Subsequently, all *I-set*s are obtained and placed as keys in *Dict2*. *I-set*s {A}, {B}, and {C} are previously present; thus, their values are incremented.

In Figure 8, 3rd transaction is obtained and kept as a new key with value one in *Dict1*, due to its earliest arrival. *I-set*s {A}, {B}, and {D} are already present; thus, their values are incremented.

TABLE 2. A transactional dataset, *DB*.

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

The arrival of the 4th transaction is shown in Figure 9.

The arrival of the 5th transaction is shown in Figure 10. The last transaction shown in Figure 11 is similar to the 1st transaction; thus, it is not placed in *Dict1*. However, the value of the rejected transaction is incremented in *Dict1*. Moreover,

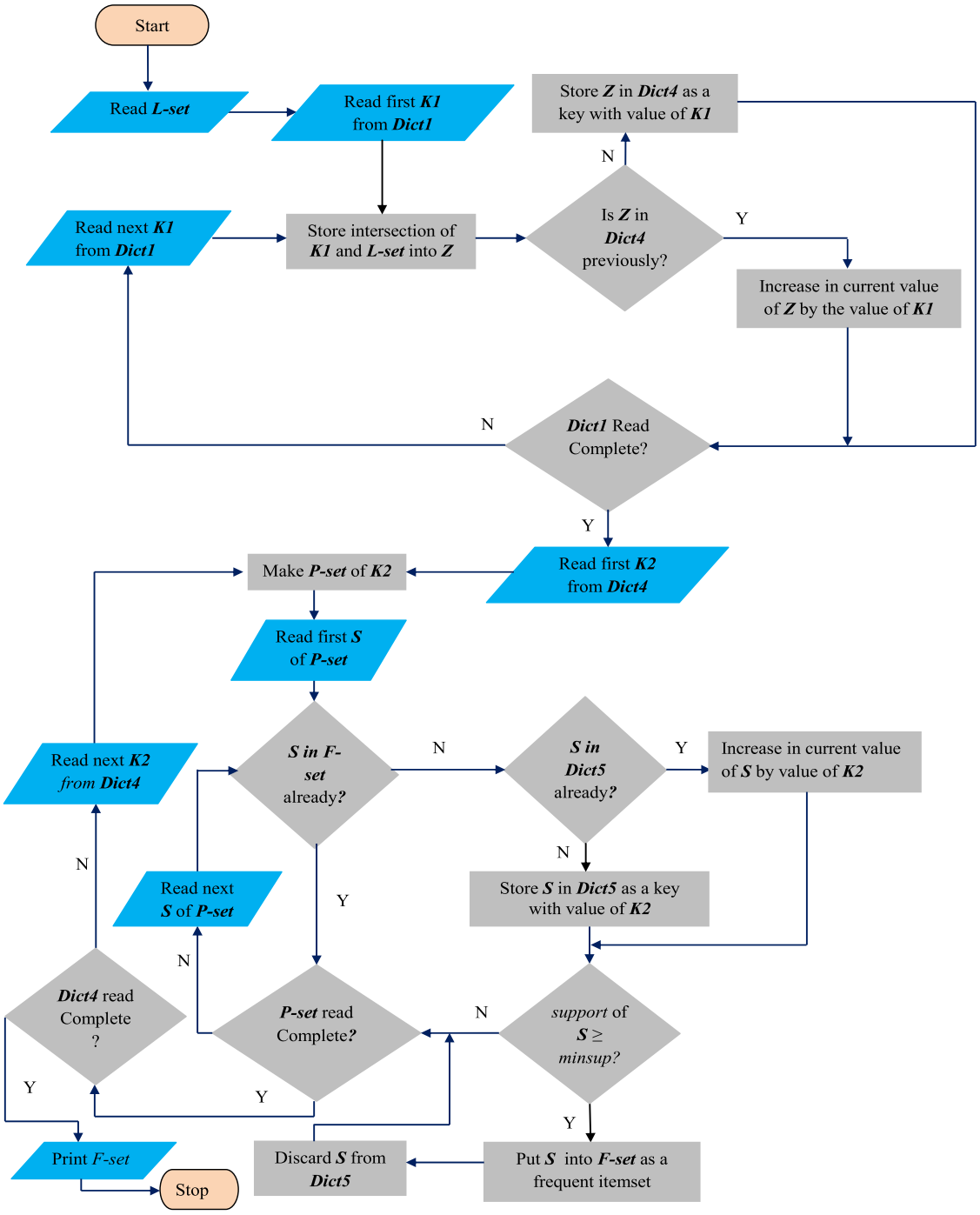

FIGURE 5. Flowchart of *FindFrequentAll*( ) Procedure.

the values of its *I-set*s are also incremented in *Dict2*. *I-sets*{D} and {E} have values less than *minsup*; therefore, both are eliminated.

I-sets {A}, {B}, and {C} are frequent 1-itemsets, obtained and kept in *Dict3* as keys with relevant supports. Finally, the keys are placed in *L-set.*

FindFrequentAll( ) procedure is invoked afterward. *K1* holds the 1st key in Figure 12.

K1 is intersected with *L-set,* and the result is placed in *Z*. *Z* holds a pruned transaction. *Z* is stored in *Dict4* as key and its value is two that is the value of *K1*. In Figure 13, *K1* points to the 2nd key.

Z again keeps the intersection that is already present in *Dict4*, showing similar pruned transaction. Therefore, the value of *Z* is increased by 1, which is the value of *K1* at present. In Figure 14, *K1* refers to the 3rd key. The intersection

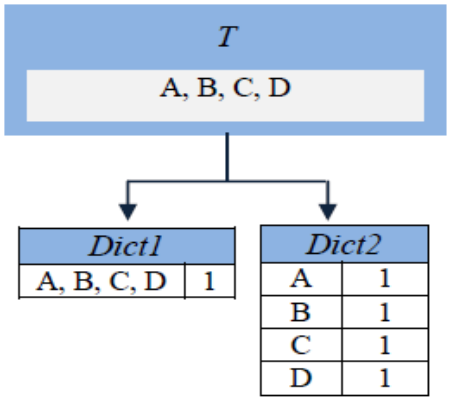

FIGURE 6. Placement of 1^{st} T.

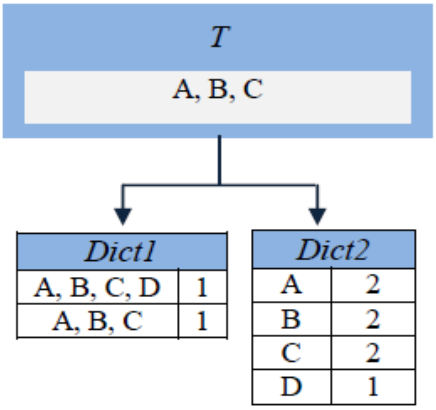

FIGURE 7. Placement of 2^{nd} T.

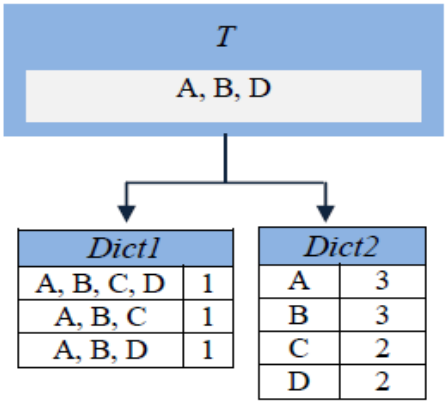

FIGURE 8. Placement of 3^{rd} T.

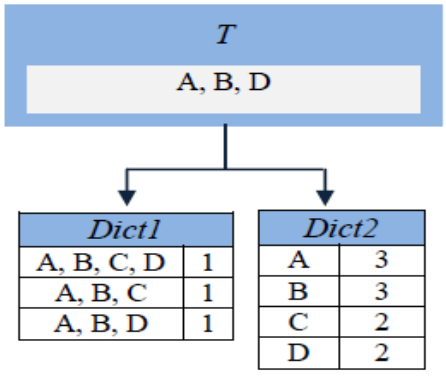

FIGURE 9. Placement of 4^{th} T.

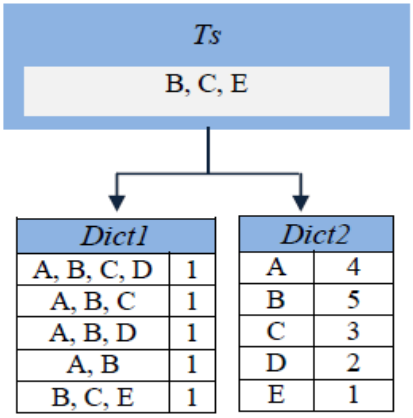

FIGURE 10. Placement of 5^{th} T.

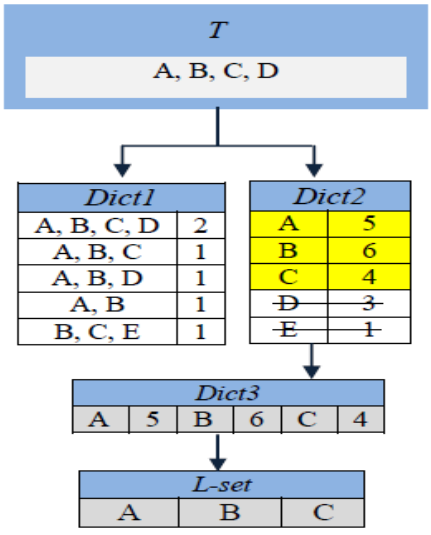

FIGURE 11. *Placement of* 6^{th} T.

results in a pruned transaction that is not already present in *Dict4*. Therefore, *Z* is kept in *Dict4* as a new key, and its value is 1, which is the value of *K1* at present.

In Figure 15, *K1* refers to the 4th key in *Dict1*. The intersection is already present in *Dict4*, showing similar trimmed transaction. Therefore, the value of *Z* is increased by 1 in *Dict4*, which is the value of the current *K1*.

In Figure 16, *K1* refers to the 5th key.The intersection is new for *Dict4*; thus, it is placed as a new key in *Dict4,* and its value is set to1, that is the value of *K1* presently.

The transaction pruning phase is completed after reading the last record from *Dict1*. The state of *Dict4* depicts that similar trimmed transactions are stored only once.

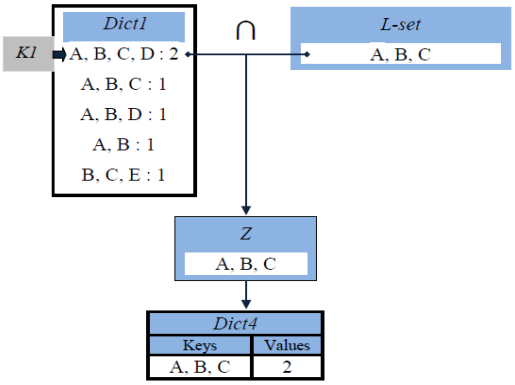

FIGURE 12. *Z* and *Dict4* after reading 1st key.

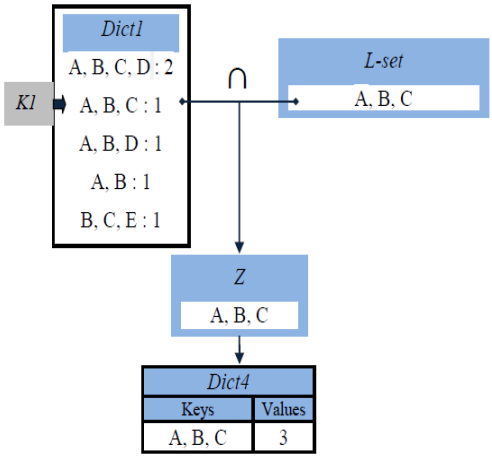

FIGURE 13. *Z* and *Dict4 after reading*2nd key.

Consequently, D-GENE generates ITTL of similar trimmed transactions once regardless of their transactional occurrence in *Dict4*. Eventually, it helps in getting performance gain.

Figures 17 - 19 show the ITTL phase, in which D-GENE constructs a power set of each distinct trimmed transaction present in *Dict4*. In Figure 17, *K2* refers to the 1st key in *Dict4*. *F-set* is empty at this stage. D-GENE generates ITTL of *K2* and stores each subset, *S,* into *P-set*. Iteratively, the presence of each *S* is verified in *F-set* first, which is empty at this stage. *S* is infrequent here; thus, it is put into *Dict5* as a new key with the value 3, which is the value of current *K2*. Just after storing in *Dict5*, the value of *S* is compared with *minsup*. The value of each *S* is less than *minsup*; thus, no *S* is taken out from *Dict5*. Subsequently, *F-set* remains empty in this iteration.

Figure 18 shows the next iteration in which *K2* points to the next record {A, B} in *Dict4* and the *P-set* of *K2* is generated afterward. Because *F-set* is still empty, each *S* is stored into *Dict5* with the values of current *K2*. All of the subsets are already present in *Dict5*; therefore, no new key is generated. The value of {A} is increased to 5 which is higher than *minsup*. Eventually, {A} becomes a frequent *I-set*, taken out from *Dict5,* and stored into *F-set*. Subsequently, {B}, and {A, B} also become frequent, picked from *Dict5*, and stored into *F-set,* respectively.

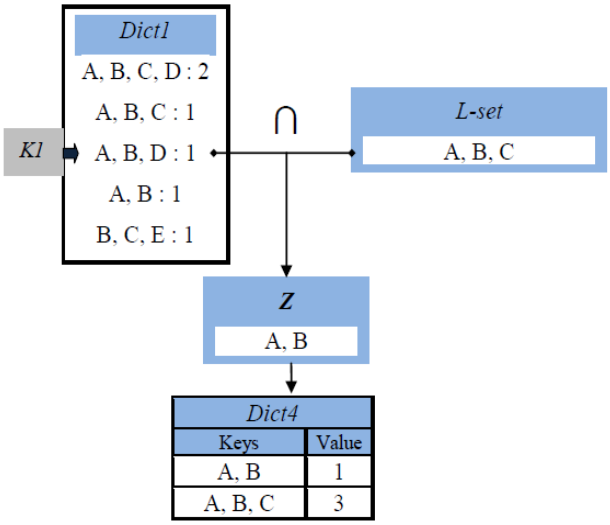

FIGURE 14. *Z* and *Dict4 after reading* 3rd key.

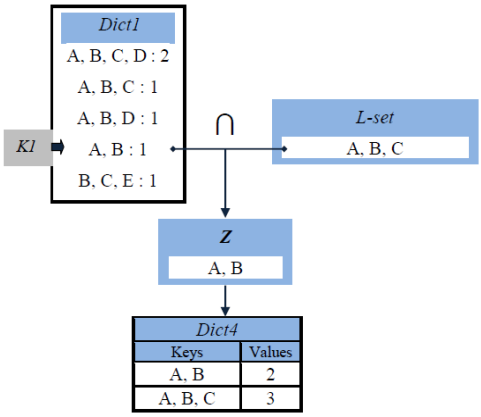

FIGURE 15. *Z* and *Dict4* after reading 4th key.

Dict5 now contains {C}, {A, C}, {B, C}, and {A, B, C}. Figure 19 shows the next iteration in which *K2* points to the last record in *Dict4*.

The value of {C} and {B, C} becomes 4, which is greater than *minsup*. Therefore, they become frequent *I-set*, picked from *Dict5,* and stored into *F-set*. D-GENE terminates after this step. From Figures 17- 19, it is evident that redundant processing is avoided by storing identical trimmed transactions in *Dict4* before generating ITTLs once.

VI. EXPERIMENTAL EVALUATION OF D-GENE

Performance evaluation relevant to runtime and memory utilization isreported here. Each algorithm explores similar frequent itemsets; thus, the results are verified.

A. EMPIRICAL SETTINGS

The comparison of D-GENE is made with TRICE, optimized SaM, optimized RElim, and FP-growth on six sparse real datasets. Dataset properties are given in Table 3. *Online Retail*, *Kddcup99*, *PowerC*, *Food Mart*, and *Record Link*

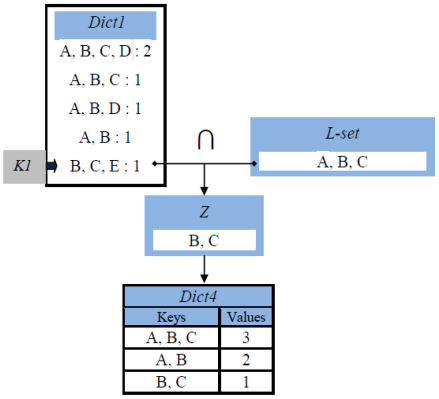

FIGURE 16. *Z* and *Dict4* after reading 5th key.

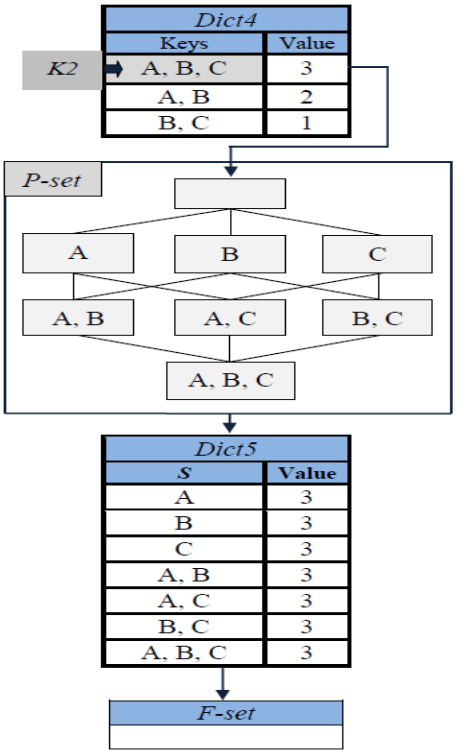

FIGURE 17. *P-set* showing ITTL, *Dict5*, and *F-set* when 1st*K2* is read.

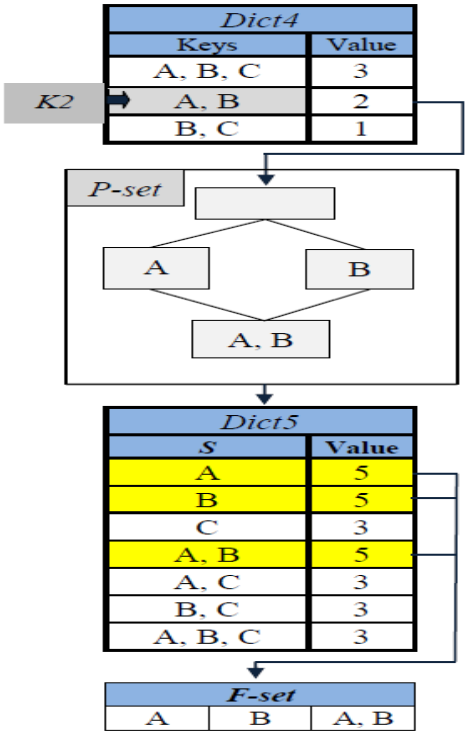

FIGURE 18. *P-set* showing ITTL, *Dict5*, and *F-set* when 2nd*K2* is read.

are attained from [83]. *The Extended Bakery* dataset is got from [84]. The reason why FP-Growth is chosen is its comparable performance with the successors for sparse datasets [78], [80], [82]. Optimized RElim and SaM also perform well on sparse datasets [54]. D-Gene is implemented in Python and the implementation of TRICE is taken from the authors' previous work [53]. RElim, SaM, and FP-Growth are obtained from [85]. A computer conducts experiments with Core i7-3667U Intel, 8G memory, 2.0 GHz processor, and Windows 8 Pro 64 Edition. It is worth-mentioning that the runtime of D-GENE is not directly comapred with those in published reports. Instead, implementations of all the algorithms are executed on the same machine and compared in the same running environment.

TABLE 3. Dataset features.

No.	Dataset	Distinct Items (*N*)	No. of Transactions	Sparseness Value
1	*PowerC*	140	1,040,000	0.05
2	*Kddcup99*	135	1,000,000	0.11
3	*Online Retail*	2,603	541,909	0.001
4	*Record Link*	29	574,913	0.34
5	*Extended Bakery*	50	75,000	< 1
6	*Food Mart*	1,559	4,141	0.002

B. D-GENE RUNTIME EVALUATION

For *Kddcup99*, at 70% *minsup*, D-GENE is a bit faster than TRICE in Figure 20. Moreover, both D-GENE and TRICE are quicker than RElim, SaM, and FP-Growth by above 3, 3, and above 4 times. SaM performs better than RElim though RElim is reported as more efficient on sparse datasets formerly [54]. As *minsup* gets lower, the performance of D-GENE starts improving further. It outperforms others comprehensively. At lower *minsup*, such as 30%, D-GENE is quicker than FP-Growth approximately by a factor of 4 and both RElim and SaM by a factor of 2.5. Moreover, D-GENE is quicker than TRICE by a factor of 1.5.

Runtime analysis for the *PowerC* dataset is given in Figure 21. The performance of D-GENE is better than all others. It beats FP-Growth and RElim nearly by a factor of 4 and SaM by a factor of 3 on each *minsup* value. Both D-GENE and TRICE have shown comparable performance

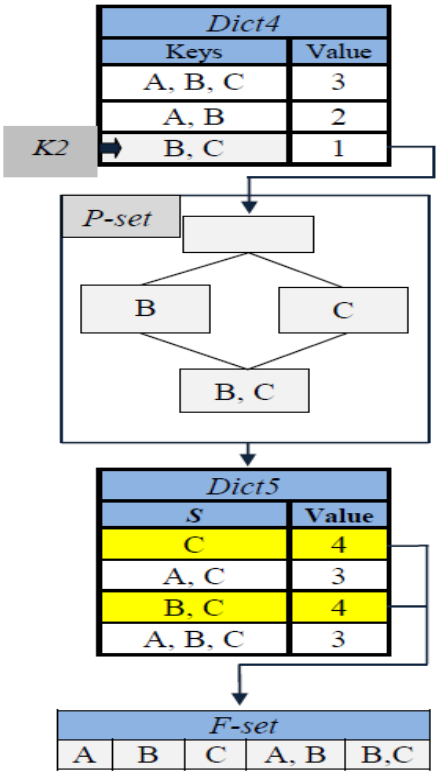

FIGURE 19. *P-set, Dict5,* and *F-set* when 3rd *K2* is read.

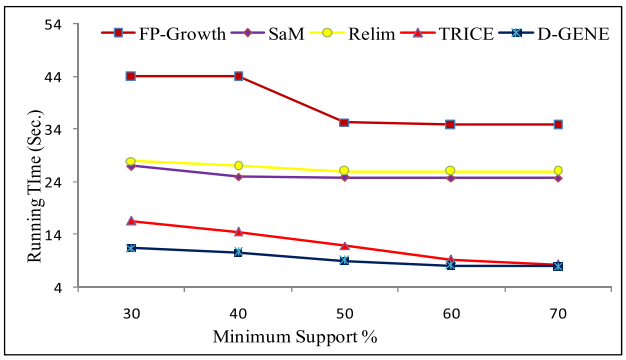

FIGURE 20. Runtime Analysis for *Kddcup99 Dataset*.

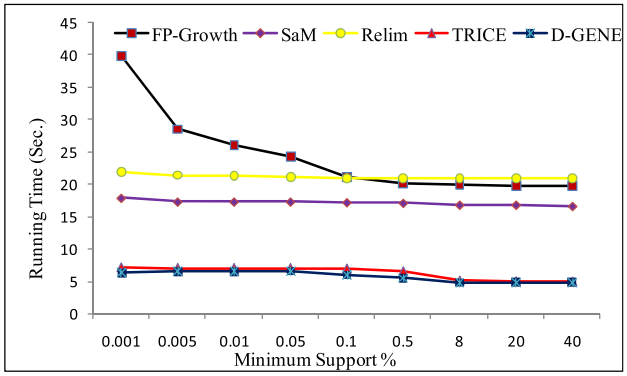

FIGURE 21. Runtime Analysis for *PowerC* Dataset.

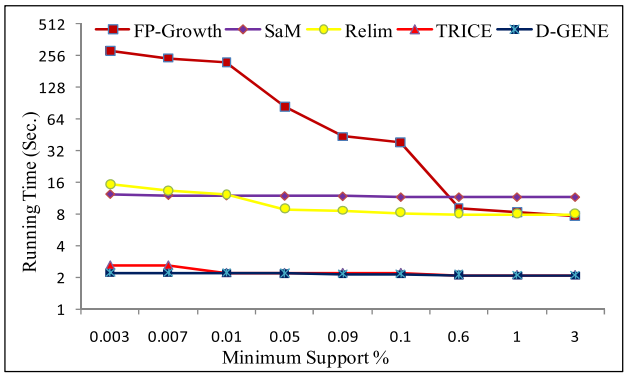

FIGURE 22. Runtime Analysis for *Online Retail* Dataset.

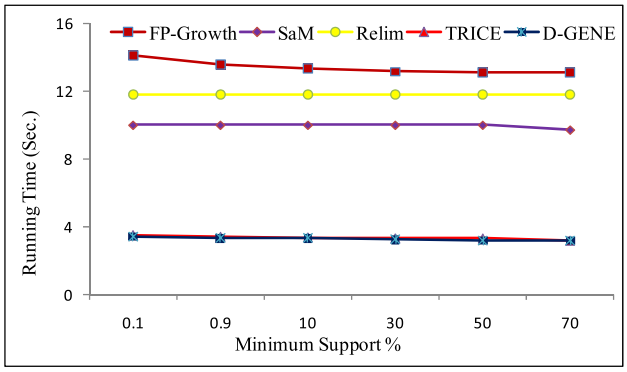

FIGURE 23. Runtime Analysis for *Record Link* dataset.

at a 40% value of *minsup*. D-GENE is slightly faster than TRICE. However, as *minsup* tends to become lower, D-GENE starts performing better. At *minsup* 0.001%, D-GENE widens the performance gap. It beats RElim, FP-Growth, and SaM almost by the factors of 4, 6, and 3 respectively. Moreover, D-GENE is faster than TRICE by the factor of 1.16.

Figure 22 depicts the comparison for the *Online Retail* dataset logarithmically. D-GENE beats all four algorithms, especially on lower minimum support thresholds. At *minsup* 3%, D-GENE is faster than RElim and FP-growth by a factor of 4, and beats SaM by a factor of 5. When *minsup* is 0.01%, D-GENE becomes faster than RElim and SaM by a factor of 5 and beats FP-Growth by more than two orders of magnitude. D-GENE and TRICE perform equally well on higher *minsup* values. However, D-GENE performs better than TRICE when *minsup* tends to decrease.

Figure 23 shows the analysis for the *Record Link* dataset. SaM outperforms RElim; however, D-GENE again performs well. When *minsup* is 70%, D-GENE beats RElim, SaM, and FP-growth by a factor of 3.5, 3, and 4 respectively. This significance continues throughout all *minsup* values. The performance gap of TRICE and D-GENE is almost negligible; however, D-GENE is slightly faster than TRICE on all *minsup* values.

Figure 24 presents the performance comparison for the *Extended Bakery* dataset. The FP-Growth is outperformed

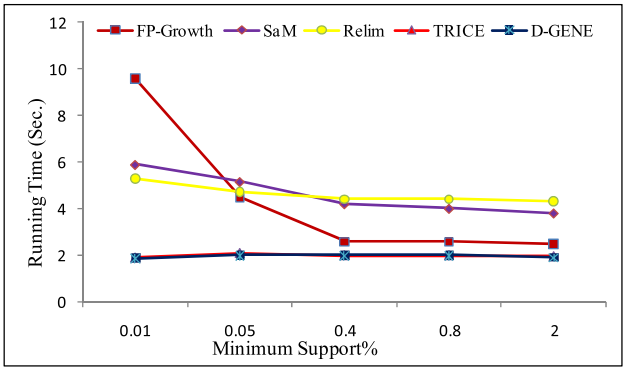

FIGURE 24. Runtime Analysis for *Extended Bakery* dataset.

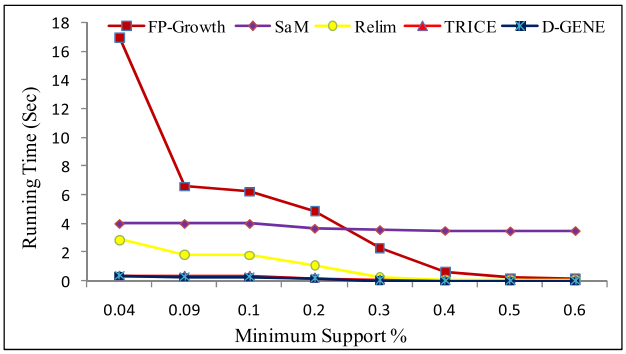

FIGURE 25. Runtime Analysis for *Food Mart* dataset.

by D-GENE approximately by a factor of 1.5. D-GENE beats RElim and SaM by more than a factor of 2. D-GENE performs consistently and outperforms RElim and SaM by a factor of 3 and FP-Growth by a factor of 5 at *minsup*0.01%. TRICE and D-GENE perform consistently well.

Figure 25 reports the analysis for the *Food Mart* dataset. When *minsup* is 0.5%, D-GENE is faster than FP-Growth, RElim, and TRICE by the factors of 8, 4, and 1.5, respectively. Moreover, it is faster than SaM by more than two orders of magnitude. As *minsup* is decreased, the performance gap widens. At *minsup* 0.04%, D-GENE outperforms RElim, SaM, and FP-growth by the factors of 8, 10, and 45 respectively. However, D-GENE beats TRICE by a tiny margin.

The analysis of runtime shows that D-GENE always performs better due to the following reasons.

1) The performance of FP-Growth deteriorates when datasets are sparse; thus, long repetitive patterns do not occur. FP-tree turns bigger; therefore, larger time is taken by the algorithm while making and traversing the conditional trees. In contrast, following its predecessor [53], D-GENE does not construct conditional trees and pattern bases; thus, performing better. The process of merging in the SaM algorithm limits its performance for sparse datasets [74]. It is expected that the two lists of transactions differ in length considerably, resulting in quadratic runtime of the *merge sort*. Hence, an optimized SaM with an improved merging strategy is used in this paper [54]. In the same way, optimized RElim eradicates duplicates present in the transaction lists and used a heuristically sorts the lists 54]. Despite all this, D-GENE outperforms the optimized versions of both algorithms comprehensively.
2) *FindFrequent1*( ) procedure of D-GENE stores similar transactions once in *Dict1*, thereby compressing the dataset. The value of a key tells its transactional occurrence in a dataset. The subsequent arrival of the same transaction causes its rejection, but an increment is done in its previous value in *Dict1*. Hence, regardless of the multiple occurrences, a transaction is intersected with *L-set* once. The efficiency is greatly improved.
3) The intersection shows a pruned transaction containing only frequent 1-*I-set*s. Therefore, ITTL is generated in less time, in contrast to other power set-based methods [52].
4) The ITTL of similar trimmed transactions is generated once in D-GENE as opposed to TRICE that makes the ITTL every time a similar trimmed transaction comes as a result of intersection. Thus, D-GENE gets efficiency by avoiding redundant computations.

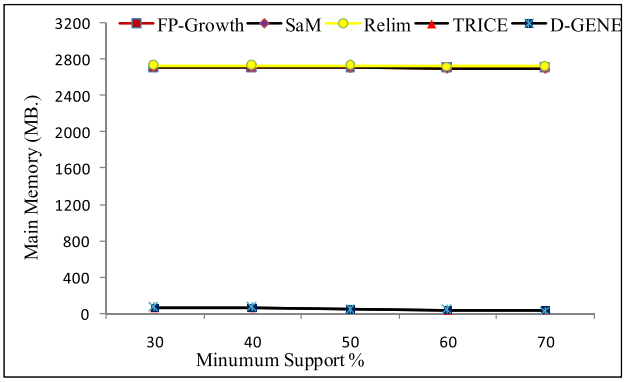

FIGURE 26. Memory Usage Analysis for *Kddcup99* dataset.

C. D-GENE MEMORY USE COMPARISON

D-GENE consumes the lowest amount of memory for the *Kddcup99* dataset in Figure 26. D-GENE consumes a little less memory than TRICE on all *minsup* costs as well. RElim consumes the most prodigious memory on all *minsup* costs. The memory consumed by RElim, SaM, and FP-growth are higher than consumed by D-GENE by a factor of 40 when *minsup* is 70%. The factor becomes more significant than 70 when *minsup* is 30%.

Memory usage analysis for the *PowerC* dataset is given in Figure 27. RElim takes massive memory. There is no considerable difference in the memory consumption of D-GENE and TRICE. Both algorithms take 42 times lower memory when *minsup* is 40% and 37 times lower memory when *minsup* is 1% than that of others.

Figure 28 depicts the comparison for the *Online Retail* dataset. RElim again requires the most massive memory.

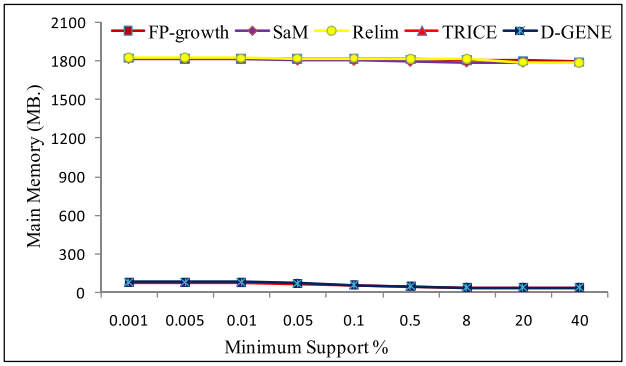

FIGURE 27. Memory Usage Analysis for *PowerC* dataset.

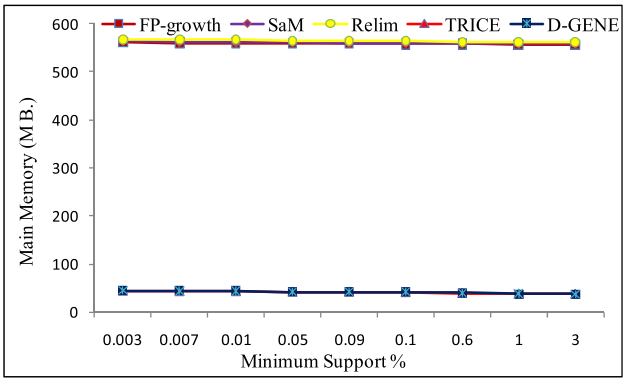

FIGURE 28. Memory Usage Analysis for *Online Retail* dataset.

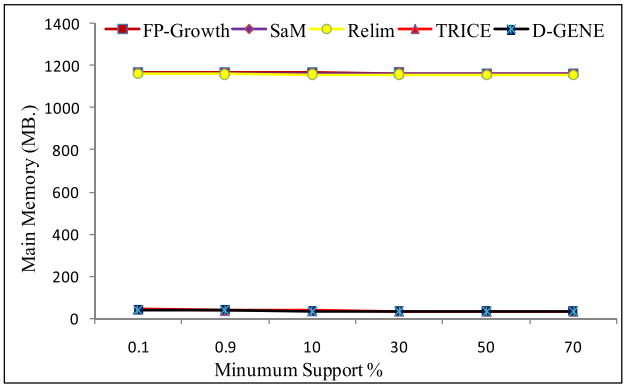

FIGURE 29. Memory Usage Analysis for *Record Link* dataset.

D-GENE is as efficient as TRICE. SaM, FP-growth and RElim use 15, 15, and 16 times larger memory than D-GENE when *minsup* is 3%. The same trend endures for each *minsup* value.

Figure 29 shows the comparison for the *Record Link* dataset.

FP-Growth consumes the most extensive memory. No significant performance gap exists between D-GENE and TRICE. D-Gene memory consumption isapproximately 33 times lower than RElim, SaM, and FP-growth when *minsup* is 70%, and this margin slightly reduces to 27 times when *minsup* is 0.1 %.

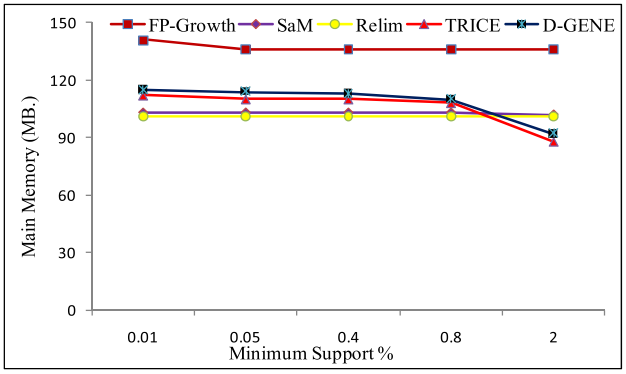

FIGURE 30. Memory Usage Analysis for *Extended Bakery* dataset.

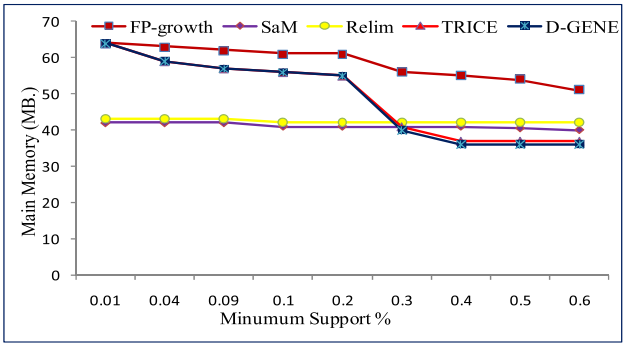

FIGURE 31. Memory Usage Analysis for *Food Mart* dataset.

Figure 30 gives the analysis for the *Extended Bakery* dataset, where FP-Growth needs the biggest memory. D-GENE consumes slightly more memory than TRICE on all minsup values. TRICE consumption is the lowest when *minsup* is 2%. RElim and SaM take the least memory but the distinction is marginal.

Figure 31 shows the usage analysis for the *Food Mart* dataset, where FP-growth requires the largest memory.

When *minsup* is high, D-GENE uses the least memory, whereas TRICE consumes a bit larger. At lower *minsup*, SaM needs the most moderate amount of memory. D-GENE and TRICE show comparable performance when *minsup* is lower.

The memory requirement of D-GENE is least at higher values of *minsup* and gradually raised as the value of *minsup* goes down. Less number of frequent 1-*I-set*s exist at higher *minsup*. Accordingly, the trimmed versions of transactions are shorter in length; thus, the resultant ITTLs are also smaller. But the number of frequent *I-set*s increases at lower values of *minsup,* causing larger intersections and corresponding ITTLs. The reasons why D-GENE consumes minimal memory are explained below.

1) Memory is conserved as repeated transactions are stored once.
2) Memory requirement increases if ITTL of the entire transaction is to be stored because several infrequent 1-itemsets are also present in it. The intersection performed by D-GENE results in smaller trimmed

TABLE 4. Attractive characteristic of datasets depicting superiority of D-GENE.

minsup	Dataset	No. of Transactions	Number of distinct transactions (*Dict1*) (independent of *minsup*)	Number of distinct trimmed transactions (*Dict4*) (dependent on *minsup*)	A difference of distinct transactions and distinct trimmed transactions	D-GENE is faster than TRICE by
0.1%	*RecordLink*	574913	257	255	2	0.1 sec.
0.003%	*OnlineRetail*	541909	4183	3964	219	0.4 sec.
30.0%	*Kddcup99*	1000000	1446	325	1121	5.0 sec.
0.005%	*PowerC*	1040000	14,623	14,592	31	1.0 sec.

transactions. Eventually, the size of the corresponding ITTLs is also reduced.

3) FP-growth has to build several conditional trees. Additionally, a few numbers of commonly shared prefixes occur in sparse datasets that result in larger conditional trees. Eventually, more significant memory is required.
4) Splitting and merging of RElim and SaM require recursive processing that uses massive memory. When the dataset is large, the stack and the data structure turn bigger.

The successors of FP-growth require even higher memory, due to the keeping of PPC-tree and representation of 2-itemset consisting of nodes of PPC-tree [80]. PPC-tree is bigger than related FP-tree.

D. INSIGHTFUL ANALYSIS OF RESULTS

As *minsup* tends to decrease, D-GENE starts outperforming TRICE on large datasets. The distinction between their running times becomes significant due to the deferral of ITTL generation in D-GENE. Table 4 shows an exciting dataset characteristic that plays a pivotal role in deciding when D-GENE defeats TRICE.

The number of distinct transactions in the *Record Link* dataset is 259. *Dict1* contains 259 key-value pairs only, which means that a vast amount of similar transactions exist in the dataset. Moreover, the number of distinct trimmed transactions is 255, which is slightly less than the number of distinct transactions. Therefore TRICE and D-GENE generate 257 and 255 ITTLs, respectively; though, D-GENE takes some time to put the trimmed transactions in *Dict4*.

However, this time is short because insertion, deletion, and presence check in a *dictionary* ADT takes constant time. Nevertheless, no significant distinction exists in the running times of both algorithms because they have to generate an almost equal number of power sets corresponding to ITTLs.

In contrast, for the *OnlineRetail* dataset in Figure 22, D-GENE significantly outperforms TRICE at lower *minsup* values, such as 0.007% and 0.003%. Though D-GENE beats TRICE by only 0.4 sec., yet it is significant because D-GENE and TRICE complete the task in 2.2 and 2.6 seconds, respectively. The decisive factor is the difference between the number of distinct transactions and distinct trimmed transactions. Now the difference is 219 that manifests that TRICE has to generate 219 more power sets as compared to D-GENE. This additional time will be higher than the time taken by D-GENE to defer the power set generation.

Similarly, the difference between distinct transactions kept in *Dict1* and distinct trimmed transactions kept in *Dict4* is 1121 for the *Kddcup99* dataset, which is quite huge. Therefore, TRICE has to generate 1121 more power sets as compared to D-GENE. The mechanism of deferring ITTL generation works excellently in favor of D-GENE, enabling it to discover frequent itemsets 5 seconds before. For *PowerC* dataset, the difference is 1 second, because the difference between distinct transactions and distinct trimmed transactions is just 31.

The above results are evidence that the deferred ITTL generation works well for D-GENE because it gets rid of doing redundant computations done by TRICE. Moreover, D-GENE needs the least memory because it stores similar transactions and similar trimmed transactions once and also prunes every distinctive transaction by eliminating its infrequent part. Consequently, the resulting ITTLs are not bigger. D-GENE shows well all-around runtime performance.

VII. D-GENE DEPLOYMENT ENVISAGE

D-GENE has shown premium run time efficiency and least memory consumption for real sparse datasets. In fog and mobile edge computing, where the computation is done in proximity to end-devices, frequent items can be explored efficiently. Frequent items can be explored at local IoT devices as well as at mobile fog nodes [86]. Regardless of the approach suggested in [86], IoT devices as well as numerous mobile fog nodes are found to be resource-poor.

Moreover, large and sparse datasets increase computing and storage cost in edge-devices. Therefore, D-GENE with its distinguising feature of least memory consumption and superior running time for sparse big data, is envisioned as a promising candidate for implementation in fog and mobile edge computing. D-GENE can trivially be implemented on an analytical engine within a fog node or an edge server to reveal the interesting frequent patterns, useful in smart home and numerous other IoT applications.

VIII. CONCLUSION

Predictive performance can be made better by insights taken from sparse real data. Thus, the sparse data is believed to

be a valuable asset for companies. In data science, exploring frequent itemsets is the identification of items jointly present in a database. This paper presents D-GENE, a novel technique to explore frequent itemsets from sparse real datasets.

Like the TRICE algorithm, D-GENE is also based on Iterative Trimmed Transaction Lattice (ITTL) to learn frequent itemsets. However, it further optimizes TRICE by introducing a deferred ITTL generation mechanism. TRICE makes the ITTL just after trimming each distinct transaction. In large sparse datasets, trimmed versions of two or more distinct transactions often become similar. Therefore, it has to generate power sets of similar trimmed transactions repeatedly. Redundant computations affect the performance of TRICE. To get rid of the redundant work done by TRICE, D-GENE first completes the transaction trimming phase and stores alike trimmed transactions once. Afterward, it generates ITTLs to explore frequent itemsets. This deferring mechanism helps D-GENE to gain even better runtimes.

D-Gene is validated on six sparse real datasets and contrasted with TRICE, FP-Growth, and optimized versions of RElim, and SaM algorithms. D-GENE has performed better than the other algorithms on all minimum support thresholds. Furthermore, it consumes minimal memory on all datasets, that makes D-GENE a perfect choice for big IOT data analytics. D-GENE can further be modified to solve added mining tasks such as exploring high utility itemsets, maximal frequent itemsets, weighted frequent itemsets, closed frequent itemsets, top-rank-k itemsets, and data streams.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.
[2] 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.
[3] 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.
[4] V. Jakkula and D. J. Cook, "Temporal pattern discoveryforanomalydetectionin asmarthome," in *Proc. Int. Conf. Intell. Environments*, 2007, pp. 339–345.
[5] 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.
[6] 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, Mar. 2007.
[7] 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*, 2017, pp. 1–18.
[8] S. K. Kim, J. H. Lee, K. H. Ryu, and U. Kim, "A framework of spatial colocation pattern mining for ubiquitous GIS," *Multimed Tools Appl.*, vol. 71, no. 1, pp. 199–218, Jul. 2014.
[9] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *Proc. Int. Symp. Spatial Databases*, 1995, pp. 47–66.
[10] J. Soung 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.
[11] A. Yassine, S. Singh, M. S. Hossain, and G. Muhammad, "IoT big data analytics for smart homes with fog and cloud computing," *Future Gener. Comput. Syst.*, vol. 91, pp. 563–573, Feb. 2019.
[12] M. R. Karim, M. Cochez, 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.
[13] M. H. U. 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.
[14] 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.
[15] T. Asha, S. Natarajan, and K. N. B. Murthy, "Associative classification in the prediction of tuberculosis," in *Proc. Int. Conf. Workshop Emerging Trends Technol. (ICWET)*, vol. 11, 2011, p. 1327.
[16] C. Y. Chin, M. Y. Weng, 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, no. 4, Apr. 2015, Art. no. e0122508.
[17] T. Exarchos, C. Papaloukas, D. Fotiadis, and L. 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.
[18] 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, Apr. 2015.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] K. Yoshida, Y. Shomura, and Y. Watanabe, "Visualizing network status," in *Proc. 6th Int. Conf. Mach. Learn. (ICMLC)*, vol. 4, 2007, pp. 2094–2099.
[24] 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)*, Jan. 2008, pp. 10–16.
[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] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Min. Knowl. Disc.*, vol. 15, no. 1, pp. 55–86, Jul. 2007.
[27] SWAMP Project. *Smart Water Management Platform*. Accessed: 2018. [Online]. Available: http://swamp-project.org/communication/WRNP2018_lamina_SWAMP_EN.pdf
[28] D. Deyannis, R. Tsirbas, G. Vasiliadis, R. Montella, S. Kosta, and S. Ioannidis, "Enabling GPU-assisted antivirus protection on Android devices through edge offloading," in *Proc. EdgeSys*, 2018, pp. 13–18.
[29] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann, "Dynamic urban surveillance video stream processingusing fog computing," in *Proc. BigMM*, 2016, pp. 105–112.
[30] Y.-D. Chen, M. Z. Azhari, and J.-S. Leu, "Design and implementation of a power consumption management system for smart home over fog-cloud computing," in *Proc. 3rd Int. Conf. Intell. Green Building Smart Grid (IGBSG)*, Apr. 2018, pp. 1–5.
[31] *Out of the Fog: Use Case Scenarios (High-Scale Drone Package Delivery)*. Accessed: 2018. [Online]. Available: http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip
[32] *Transportation Scenario: Smart Cars and Traffic Control (3.1)*. Accessed: 2017. [Online]. Available: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf
[33] *Autonomous Driving*. Accessed: 2018. [Online]. Available: http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip
[34] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. MobiSys*, 2014, pp. 68–81.

[35] G. Jia, G. Han, A. Li, and J. Du, "SSL: Smart street lamp based on fog computing for smarter cities," *IEEE Trans. Ind. Informat.*, vol. 14, no. 11, pp. 4995–5004, Nov. 2018.

[36] OpenFog Consortium. *Out of the Fog: Use Case Scenarios (Live Video Broadcasting)*. Accessed: 2018. [Online]. Available: http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip

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

[38] M. Kabeer, F. Riaz, S. Jabbar, M. Aloqaily, and S. Abid, "Real world modeling and design of novel simulator for affective computing inspired autonomous vehicle," in *Proc. 15th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Jun. 2019, pp. 1923–1928.

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

[40] S. Choudhury, Q. Ye, M. Dong, and Q. Zhang, "IoT big data analytics," *Wireless Commun. Mobile Comput.*, vol. 2019, May 2019, Art. no. 9245392.

[41] 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. (TOIT)*, vol. 18, no. 1, p. 8, 2017.

[42] 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.

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

[44] S. Khan, D. Paul, P. Momtahan, and M. Aloqaily, "Artificial intelligence framework for smart city microgrids: State of the art, challenges, and opportunities," in *Proc. 3rd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Apr. 2018, pp. 283–288.

[45] I. Al Ridhawi, Y. Kotb, and Y. Al Ridhawi, "Workflow-Net based service composition using mobile edge nodes," *IEEE Access*, vol. 5, pp. 23719–23735, 2017.

[46] X. Sun and N. Ansari, "EdgeIoT: Mobile edge computing for the Internet of Things," *IEEE Commun. Mag.*, vol. 54, no. 12, pp. 22–29, Dec. 2016.

[47] S. Khadim, F. Riaz, S. Jabbar, S. Khalid, and M. Aloqaily, "A non-cooperative rear-end collision avoidance scheme for non-connected and heterogeneous environment," *Comput. Commun.*, vol. 150, pp. 828–840, Jan. 2019, doi: 10.1016/j.comcom.2019.11.002.

[48] X. Masip-Bruin, E. Marín-Tordera, G. Tashakor, A. Jukan, and G.-J. Ren, "Foggy clouds and cloudy fogs: A real need for coordinated management of fog-to-cloud computing systems," *IEEE Wireless Commun.*, vol. 23, no. 5, pp. 120–128, Oct. 2016.

[49] S. Oueida, Y. Kotb, M. Aloqaily, Y. Jararweh, and T. Baker, "An edge computing based smart healthcare framework for resource management," *Sensors*, vol. 18, no. 12, p. 4307, Dec. 2018.

[50] K. A. Alam, R. Ahmad, and K. Ko, "Enabling far-edge analytics: Performance profiling of frequent pattern mining algorithms," *IEEE Access*, vol. 5, pp. 8236–8249, 2017.

[51] E. Junqué 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, Dec. 2013.

[52] 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, Sep. 2019.

[53] M. Yasir, M. A. Habib, M. Ashraf, S. Sarwar, M. U. Chaudhry, H. Shahwani, M. Ahmad, and C. M. N. Faisal, "TRICE: Mining frequent itemsets by iterative TRimmed transaction LattICE in sparse big data," *IEEE Access*, vol. 7, pp. 181688–181705, 2019.

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

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

[56] B. Lent, A. Swami, and J. Widom, "Clustering association rules," in *Proc. 13th Int. Conf. Data Eng.*, Nov. 2002, pp. 220–231.

[57] 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.

[58] R. Srikant, Q. Vu, and R. Agrawal, "Mining association rules with item constraints," *Kdd*, vol. 97, pp. 67–73, Aug. 1997.

[59] 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.

[60] 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, 1994, pp. 401–407.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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, Apr. 2005.

[66] 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.

[67] 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.

[68] 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 Intl. Conf. Manage. Data*, 2000, pp. 22–33.

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

[70] 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.

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

[72] 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.

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

[74] 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.

[75] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, Jun. 2000.

[76] 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, pp. 1–9.

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

[78] 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, Sep. 2012.

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

[80] 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, Aug. 2015.

[81] 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.

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

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

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

[85] B. Dagenais. (2015). *A Few Data Mining Algorithms in Pure Python*. [Online]. Available: https://github.com/bartdag/pymining
[86] P. Brarun, "An innovative framework for supporting frequent pattern mining problems in IoT environments," in *Proc. ICCSA*, 2018, pp. 642–657.

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

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, as well as 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 area of research is the Internet of things (IoT), machine-to-machine (M2M) communication, optical burst switched (OBS) networks, network-on-chip (NoC), and high-speed data centers. He has made significant contributions to research. He has already published over 40 articles in well reputed journals and conferences. His research has over 200 citations. He has already supervised over ten M.Phil. theses. He has graduated one Ph.D. and is currently supervising three Ph.D. students. Furthermore, as a Principal Investigator (PI) and Co-PI, he has secured research funding amounting around USD 1.0 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. 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 the Network Manager of 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, big data, and healthcare. He is currently an Associate Editor of 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 the National Textile University Faisalabad, Pakistan. His research interests include cognitive science, human performance, and evaluation of user interfaces for commercial applications.