IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Efficiently mining frequent itemsets on massive data

XIXIAN HAN, XIANMIN LIU, JIAN CHEN, GUOJUN LAI, HONG GAO, JIANZHONG LI

School of Computer Science and Technology, Harbin Institute of Technology, China (e-mail: hanxx@hit.edu.cn, liuxianmin@hit.edu.cn, jianchen_1997@163.com, laigjwork@163.com, honggao@hit.edu.cn, lijzh@hit.edu.cn)

Corresponding author: Xixian Han (e-mail: hanxx@hit.edu.cn).

This work was supported in part by the National Natural Science Foundation of China under grant nos. 61872106, 61832003, 61632010, 61502121, National key research and development program of China under grant no. 2016YFB1000703, Weihai-HIT co-construction program under grant no. ZMZ001702.

ABSTRACT Frequent itemset mining is an important operation to return all itemsets in the transaction table, which occur as a subset of at least a specified fraction of the transactions. The existing algorithms cannot compute frequent itemsets on massive data efficiently, since they either require multiple-pass scans on the table, or construct complex data structures which normally exceed the available memory on massive data. This paper proposes a novel precomputation-based PFIM algorithm to compute the frequent itemsets quickly on massive data. PFIM treats the transaction table as two parts: the large old table storing historical data and the relatively small new table storing newly generated data. PFIM first pre-constructs the quasi-frequent itemsets on the old table whose supports are above the lower-bound of the practical support level. Given the specified support threshold, PFIM can quickly return the required frequent itemsets on the table by utilizing the quasi-frequent itemsets. Three pruning rules are presented to reduce the size of the involved candidates. An incremental update strategy is devised to efficiently re-construct the quasi-frequent itemsets when the tables are merged. The extensive experimental results, conducted on synthetic and real-life data sets, show that PFIM has a significant advantage over the existing algorithms and runs two orders of magnitude faster than the latest algorithm.

INDEX TERMS Frequent itemset mining, massive data, PFIM algorithm, pruning rule, incremental update

I. INTRODUCTION

FREQUENT itemset mining is an important operation that has been widely studied in many practical applications, such as data mining [1]–[3], software bug detection [4], spatiotemporal data analysis and biological analysis [5]. Given a transaction table, in which each transaction contains a set of items, frequent itemset mining returns all sets of items whose frequencies (also referred to as *support* of the set of items) in the table are above a given threshold.

Due to its practical importance, since firstly proposed in [6], frequent itemset mining has received extensive attentions and many algorithms are proposed [7]–[9]. The existing frequent itemset mining algorithms can be classified into two groups: candidate-generation-based algorithms [10]–[14] and pattern-growth-based algorithms [15]–[17]. The candidate-generation-based algorithms first generate candidate itemsets and these candidates are validated against the transaction table to identify frequent itemsets. The anti-monotone property is utilized in candidate-generation-based algorithms to prune search space. But the candidate-generation-based algorithms require multiple-pass table scans and this will incur a high I/O cost on massive data. The pattern-growth-based algorithms do not generate candidates explicitly. They construct the special tree-based data structures to keep the essential information about the frequent itemsets of the transaction table. By use of the constructed data structures, the frequent itemsets can be computed efficiently. However, pattern-growth-based algorithms have the problem that the constructed data structures are complex and usually exceed the available memory on massive data. To sum up, the existing algorithms cannot compute frequent itemsets on massive data efficiently.

In frequent itemset mining, the number of the frequent itemsets normally is sensitive to the value of the support threshold. If the support threshold is small, there will be a large number of frequent itemsets and it is difficult for the users to make efficient decisions. On the contrary, if the support threshold is large, it is possible that no frequent itemsets can be discovered or the interesting itemsets may

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

be missed. Therefore, a proper support threshold is crucial for the practical frequent itemset mining and the users often need to perform frequent itemset mining for several times before the satisfactory support threshold is determined. The process often is interactive. On massive data, the existing algorithms often need a long execution time to compute frequent itemsets and this will affect users' working efficiency seriously [18]. The focus of this paper is to find a new efficient algorithm to compute frequent itemsets on massive data quickly.

One useful trick, which is adopted to speed up the execution in the existing algorithms, is to reuse the work done in the counting operation of the shorter itemsets for that of the longer itemsets. In this paper, we want to utilize this reuse idea to a much larger degree. In typical massive data applications, with the increasing data volume and the disk I/O bottleneck, data usually is stored in read/append-only mode [19]. Therefore, the overall data set can be divided into two parts: the much larger old data set storing the historical data, and the relative small new data set storing the newly generated data. Based on the description above, this paper devises a new PFIM algorithm (*P*recomputation-based *F*requent *I*temset *M*ining algorithm) on massive data, which utilizes the pre-constructed frequent itemsets on the old data set to return the frequent itemsets quickly. Since the too small value of support threshold will generate too many frequent itemsets, we assume in this paper that there exists a lower-bound ω of the support threshold specified by the users in practical applications. Because of the real/append-only mode, given the old table T_O, PFIM first pre-constructs the frequent itemsets (refer to as *quasi-frequent itemsets* in this paper) whose supports are no less than ω. The new transactions are accumulated in the new table T_Δ. Taking advantage of the pre-constructed quasi-frequent itemsts, given the specified support threshold, PFIM can compute the frequent itemsets on $T_O \cup T_\Delta$ quickly. In the process of execution of PFIM, three pruning rules are devised in this paper to reduce the number of candidate frequent itemsets. An incremental update strategy is proposed in this paper to quickly update the quasi-frequent itemsets when T_O and T_Δ are merged. The extensive experiments are conducted on synthetic and real-life data sets. The experimental results show that, PFIM outperforms the existing algorithms significantly, it runs two orders of magnitude faster than the latest algorithm.

The contributions of this paper are listed as follows:

- This paper proposes a novel precomputation-based PFIM algorithm to compute frequent itemsets on massive data efficiently.
- Three pruning rules are proposed in this paper to reduce the number of the candidate frequent itemsets.
- An incremental update strategy is devised to reconstruct the quasi-frequent itemsets quickly.
- The experimental results show that PFIM has a significant advantage over the existing algorithms.

The rest of the paper is organized as follows. Section II surveys the related works. Preliminaries are described in Section III. PFIM algorithm is introduced in Section IV. The performance evaluation is provided in Section V. Section VI concludes the paper.

II. RELATED WORKS

The existing algorithms for frequent itemset mining can be divided into two groups mainly: candidate-generation-based algorithms and pattern-growth-based algorithms. This section will review the two kinds of algorithms respectively.

A. CANDIDATE-GENERATION-BASED ALGORITHMS

The candidate-generation-based algorithms firstly generate the candidates of the frequent itemsets, then the candidates are validated against the transaction table, and the frequent itemsets are discovered.

Apriori algorithm [11], [20] adopts a level-wise execution mode. It uses the downward closure property, i.e. any superset of an infrequent itemset must also be infrequent, to prune the search space. By a pass of scan on the transaction table, it first counts the item occurrences to find the frequent 1-itemsets F_1. Subsequently, the frequent k-itemsets in F_k are used to generate the candidates C_{k+1} of the frequent $(k+1)$-itemsets. Another pass of scan is needed to compute the supports of candidates in C_{k+1} to find the frequent $(k+1)$-itemsets F_{k+1}. This process iterates similarly until the F_{k+1} is empty. Apriori algorithm often needs multiple passes over table, it will incur a high I/O cost on massive data.

Savasere et al. [12] propose Partition algorithm to generate frequent itemsets by reading the transaction table at most two times. The execution of Partition consists of two stages. In the first stage, Partition algorithm divides the table into a number of non-overlapping partitions in terms of the allocated memory, and the local frequent itemsets for each partition are computed. All the local frequent itemsets are merged at the end of first stage to generate the candidates of frequent itemsets. In the second phase, another pass over table is performed to acquire the support of the candidates and the global frequent itemsets can be discovered. The useful property adopted in Partition is that, every global frequent itemsets must be appeared in local frequent itemsets of at least one partition. Partition algorithm utilizes vertical table representation of transaction table and the support counting is performed by recursive TID (transaction identifier) list intersection. In the first phase, Partition may generate many false positives, i.e. the itemsets are frequent locally but not frequent globally. Therefore, it needs another table scan to remove the false positives.

Zaki [13] proposes another vertical mining algorithm Eclat. Eclat decomposes the original search space by a lattice-theoretic approach into smaller sublattices, each of which is a group of itemsets with a common prefix (referred to as prefix-based equivalence class). Depending on the allocated memory size, Eclat can recursively partition large classes into smaller ones until each class can be maintained entirely in the memory. Then, each class is processed independently in the

breath-first fashion to compute the frequent itemsets. Eclat processes the sublattices sequentially one by one and does not need post-processing overhead as Partition algorithms. The main problem of Eclat is that when the intermediate results of vertical TID lists can become too large for memory, especially in dense database, the performance of Eclat starts to suffer. In order to solve the problem, Zaki et al. [14] devise a novel vertical data representation called diffset, which keeps differences in the TIDs of a candidate pattern from its generating frequent patterns. A variation (dEclat) of Eclat by diffset is presented in [14], which performs a depth-first search of the enumeration tree. By the incorporation of diffset, the memory requirement of dEclat is cut down drastically.

Deng et al. [21] propose PPV algorithm to integrate the advantages of vertical mining and FP-growth. PPV utilizes a coding prefix tree structure PPC-tree to store the table. Each node in PPC-tree is associated with pre-post code via the pre-order and post-order traversal on the PPC-tree. Each frequent item can be represented by a node-list, i.e. the list of Pre-Post code consisting pre-order code, post-order code and the count of nodes registering the frequent item. PPV fully uses candidate generation to discover frequent itemsets, i.e. the node-lists of the candidate itemsets of length $(k + 1)$ are generated by intersecting node-lists of frequent itemsets of length k, then the frequent itemsets can be reported. PPV can achieve a high execution efficiency since (1) the node-list is more compact than the vertical structure, (2) the support counting is transformed into the intersection of node-lists, (3) the ancestor-descendant relationship of two nodes can be verified efficiently by their pre-post codes. [22] proposes PrePost to improve PPV. The core difference between PrePost and PPV is that PrePost can directly find frequent itemsets without generating candidates in some cases by using the single path property of N-list. [23] points out that node-list and N-list need to encode each node of PPC-tree with both pre-order code and post-order code, thus they are memory-consuming. A more efficient data structure, NodeSet, is adopted in [23], which only requires the pre-order code (or post-order code) of each node. And based on NodeSet, an algorithm FIN is devised to compute frequent itemsets. The algorithm dFIN is presented in [24] to improve FIN further. The algorithm dFIN uses an enhanced NodeSet, DiffNodeset, which is combined by the idea of diffset [14]. Aryabarzan et al. [25] find that the calculation of the difference between DiffNodeset takes a long time on some tables. They propose a new data structure, NegNodeset, which also uses prefix tree. NegNodeset employs a set-bitmap-representation-based encoding model for nodes. By using NegNodeset data structure, negFIN is proposed in [25]. Three key advantages of negFIN are: (1) employing bitwise operator to generate new sets of nodes, (2) reducing the time complexity of discovering frequent itemsets to $O(n)$, (3) using a promotion method to prune the search space in set-enumeration tree. Because of these advantages, negFIN rapidly finds all frequent itemsets.

B. PATTERN-GROWTH-BASED ALGORITHMS

Pattern-growth-based algorithms do not generate candidate itemsets explicitly but compress the required information for frequent itemsets in specific data structure. The frequent itemsets can be acquired quickly with the notion of projected databases, a subset of the original transaction database relevant to the enumeration node.

Agarwal et al. [26] present DepthProject algorithm to mine long itemsets in databases. DepthProject examines the nodes of the lexicographic tree in depth-first order. The examination process of a node refers to the support counting of the candidate extension of the node. During the search, the projected transaction sets are maintained for some of the nodes on the path from the root to the node P currently being extended. Normally, the projected transaction sets only contain the relevant part of the transaction database for counting the support at the node P. In the process of depth-first search, the projected database can be reduced further at the children of P and DepthProject can reuse the counting work of its previous exploration. At the lower levels of the lexicographic tree, a specialized counting technique called bucketing is used to substantially improve the counting time.

Han et al. [16] propose a FP-tree-based FP-growth algorithm to mine the complete set of frequent patterns by pattern fragment growth. FP-tree (frequent-pattern tree) is a compact prefix-based trie structure to store the essential information about frequent patterns. In each transaction, only frequent length-1 items, which are sorted with the descending order of support, are used to construct the FP-tree. Then the FP-growth algorithm works on FP-tree rather than on the original database to mine frequent patterns. FP-growth algorithm starts with a frequent length-1 pattern (initial suffix pattern), and the set of frequent items co-occurring with the suffix pattern is extracted as conditional-pattern base, which is then constructed as conditional FP-tree. With the current suffix pattern and the conditional FP-tree, if the conditional FP-tree is not empty, FP-growth performs mining recursively. The frequent patterns are acquired by concatenating the new ones generated from the conditional FP-tree and the suffix pattern. FP-growth transforms the problem of finding long frequent patterns to looking for shorter ones and then concatenating the suffix. An additional optimization is proposed for FP-growth, i.e. if all the nodes of the FP-tree lie on a single path, the frequent patterns can be generated by enumeration of all the combinations of the sub-paths with the support being the minimum support of the itemsets contained in the sub-path.

Grahne et al. [15] find out that about 80 percent of the CPU time in frequent itemset mining is used for traversing FP-trees. A special data structure, FP-array, is devised. Given an itemset of m items, FP-array is a $(m-1) \times (m-1)$ matrix, where each element of the matrix corresponds to the counter of an ordered pair of items. By the special data structure, a new FPgrowth* is proposed, which can reduce the traversal time on FP-tree and speed up the FP-growth method significantly.

Pei et al. [17] devise a novel hyper-linked data structure

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

Transaction table T

TID	*Items*
1	7, 8, 9
2	1, 6, 7
3	9
4	0, 4, 9
5	3, 6, 9
6	0, 1, 4, 9
7	0, 6
8	0, 1, 4, 9
9	1, 2, 3, 6, 7, 8, 9
10	1, 2, 3, 8, 9
11	0, 9
12	2, 8, 9
13	4, 6, 9
14	7
15	2, 3, 4, 7, 8, 9

frequent itemsets given minsup = 0.2

{0}: 0.33, {1}: 0.33, {2}: 0.27, {3}: 0.27, {4}: 0.33, {6}: 0.33, {7}: 0.33, {8}: 0.33, {9}: 0.80, {0, 4}: 0.20, {0, 9}: 0.27, {1, 9}: 0.27, {2, 3}: 0.20, {2, 8}: 0.27, {2, 9}: 0.27, {3, 8}: 0.20, {3, 9}: 0.27, {4, 9}: 0.33, {6, 9}: 0.20, {7, 8}: 0.20, {7, 9}: 0.20, {8, 9}: 0.33, {0, 4, 9}: 0.20, {2, 3, 8}: 0.20, {2, 3, 9}: 0.20, {2, 8, 9}: 0.27, {3, 8, 9}: 0.20, {7, 8, 9}: 0.20, {2, 3, 8, 9}: 0.20

FIGURE 1. The illustration of transaction table in running example.

H-struct and a new mining algorithm, H-mine, to efficiently mine databases with different data characteristics. H-mine has very limited and precisely predictable space overhead, and can be scaled up to large database by partitioning. On dense data set, FP-trees can be constructed dynamically as part of the mining process.

As pointed in Liu et al. [27], it is hard to reduce the traversal cost and the construction cost of the conditional database in pattern-growth-based algorithms, [27] proposes the AFOPT algorithm which uses a compact data structure, ascending frequency ordered prefix-tree, to represent the conditional databases. The tree is traversed in top-down depth-first order. It is shown that the combination of the top-down traversal and the ascending frequency order is more efficient than FP-tree, which adopts the combination of the bottom-up traversal and descending frequency order. AFOPT is improved further by incorporating the opportunistic projection technique.

III. PRELIMINARIES

Given a transaction table *T* of *n* transactions, each of transactions is a subset of the universe of items $U = \{i_1, i_2, \ldots, i_d\}$. Here, the itemset is a subset of U and a *k*-itemset is an itemset with *k* items. A unique transaction identifier *TID* is associated with every transaction. Given an itemset *IS*, its *support* $sup(T, IS)$ is defined as the fraction of transactions in *T* which contain *IS* as a subset, i.e.

$$sup(T, IS) = \frac{|\{t|IS \subseteq t, t \in T\}|}{n}$$

Obviously, the *support* measures the correlation of the items. For an itemset *IS*, its greater *support* value means that the items of *IS* occur together more frequently in *T*.

Definition 3.1: **(Frequent itemset mining)** Given a transaction table *T* and a specified support threshold *minsup*, frequent itemset mining determines all itemsets whose supports are no less than *minsup*.

The frequent itemset mining is defined in Definition 3.1.

TABLE 1. Summary of symbols

Symbol	Meaning
T	the transaction table
d	the number of items
n	the number of transactions in T
T_O	the old table storing historical data
tn_o	the number of transactions in T_O
T_Δ	the new table storing newly generated data
tn_Δ	the number of transactions in T_Δ
$minsup$	the specified support threshold
ω	the lower-bound of practical support threshold
F_{qf}	the file storing the quasi-frequent itemsets
cnt_Δ	the array keeping the count of the items in T_Δ
mas_Δ	the maximum count for items in T_Δ

Example 3.1: In this paper, we use a running example, depicted in Figure 1, to illustrate the processing of frequent itemset mining. Given the support threshold *minsup* = 0.2, the discovered frequent itemsets in the running example are listed in Figure 1. For example, the itemset $\{3, 9\}$ is contained in 4 transactions in Figure 1, its support is $\frac{4}{15} = 0.26$ and it is frequent.

Theoretically, the number of all itemesets that need to be checked is $(2^d - 1)$, which is a prohibitively huge search space. Therefore, downward closure property is normally used to reduce the search space.

Definition 3.2: **(Downward closure property)** If *IS* is a frequent itemset, then all its subsets are also frequent. If *IS* is an infrequent itemset, then all its supersets are infrequent.

The downward closure property is provided in Definition 3.2. The rationale under the downward closure property is that, given an itemset *IS*, if *IS* is a subset of a transaction *P*, then all the subsets of *IS* are also contained in *P* as a subset. The frequently used symbols in this paper are shown in Table 1.

IV. PFIM ALGORITHM

A. INTUITIVE IDEA

This part describes intuitive idea of PFIM algorithm.

Generally, the number of frequent itemsets is very sensitive to the value of *minsup*. If the value of *minsup* is too small, the number of frequent itemsets will be so large that the users can become overwhelmed with too many results and it is difficult for users to find the really useful information from them. Therefore, in this paper, we assume that *there exists a lower-bound for the value of minsup in practical applications*. The lower-bound is denoted by ω in this paper. The value of ω can be determined by some domain experts, or the lowest value of the support used in the past frequent itemset mining.

On massive data, the existing algorithms often cannot meet the users' requirement, they either need to scan the table multiple times, or need a complex data structure and a high memory consumption. This is the motivation of this paper, i.e. we want to devise a highly efficient algorithm to mine the frequent itemsets on massive data quickly. Some of the existing algorithms, such as FP-tree-based methods

or vertical-representation-based methods, reuse the work that has already been done previously in the current frequent itemset mining, so they can discover frequent itemsets faster. But, when the current frequent itemset mining is done, their works are lost and the next mining still needs to be executed from scratch.

On massive data applications, data usually is stored in read/append-only mode [19]. Therefore, the overall transaction table T can be divided into two parts: the large old transaction table T_O and the relative small new transaction table T_Δ, i.e. $T = T_O \cup T_\Delta$. Usually T_Δ keeps the accumulated new transactions. When the size of T_Δ reaches to some level, for example, 5% of the size of T_O, the data in T_Δ will be merged into T_O. Since the size of T_O is much larger than that of T_Δ, we have enough confidence that the time interval of two consecutive merging operations should be long enough.

During the time interval between two consecutive merging, T_O remains unchanged and only T_Δ updates frequently. Under such circumstances, given the frequent itemset mining with varying support thresholds, why not we keep the pre-computed itemsets whose support values in T_O are no less than ω and only compute the required frequent itemsets considering the existence of T_Δ. In this way, the work done for T_O can be reused for all the frequent itemset mining in a long enough time. This is the motivation why we develop PFIM algorithm.

In the rest of this section, we first show the pre-computation operation in Section IV-B, then introduce PFIM algorithm detailedly in Section IV-C and Section IV-D. The update operation of the pre-constructed itemsets are presented in Section IV-E, and some issues are discussed in Section IV-F.

B. PRE-COMPUTATION OPERATION

This part describes the pre-computation operation to generate the required itemsets on the large old transaction table T_O whose supports are no less than ω. The required itemsets here are referred to as *quasi-frequent itemsets*, distinguishing from the frequent itemsets with the support threshold *minsup* specified by users. Let tn_o be the number of transactions in T_O and tn_Δ be the number of transactions in T_Δ. Since the size of T_O is much large, usually exceeds the size of the allocated memory. Therefore, the process of pre-computing the quasi-frequent itemsets consists of two stages: candidate generation and result refinement.

In the stage of candidate generation, we retrieve the transactions in T_O sequentially and maintain the retrieved transactions in an in-memory buffer BUF, whose size is set according to the size of the allocated memory. If *BUF* is full, we can compute the local quasi-frequent itemsets in *BUF* by the current vertical frequent itemset mining algorithms. The quasi-frequent itemsets corresponding to current *BUF* are kept in a file. Then we empty BUF and continue the sequential scan for the next iteration. The process is similarly executed until all transactions in T_O is retrieved and all local quasi-frequent itemsets are generated.

Old transaction table T_O

TID	*Items*
1	7, 8, 9
2	1, 6, 7
3	9
4	0, 4, 9
5	3, 6, 9
6	0, 1, 4, 9
7	0, 6
8	0, 1, 4, 9
9	1, 2, 3, 6, 7, 8, 9
10	1, 2, 3, 8, 9
11	0, 9
12	2, 8, 9

New transaction table T_Δ

TID	*Items*
1	4, 6, 9
2	7
3	2, 3, 4, 7, 8, 9

F_{qf}

IS	9	1	0	8	6	8,9	1,9	0,9	7	4	3	2	4,9	3,9	2,9	2,8	0,4	2,8,9	0,4,9
SUP	10	5	5	4	4	4	4	4	3	3	3	3	3	3	3	3	3	3	3

quasi frequent itemsets with SUP =2

{7,9}, {7,8}, {6,9}, {6,7}, {3,8}, {3,6}, {2,3}, {1,8}, {1,7}, {1,6}, {1,4}, {1,3}, {1,2}, {0,1}, {7,8,9}, {3,8,9}, {3,6,9}, {2,3,9}, {2,3,8}, {1,8,9}, {1,6,7}, {1,4,9}, {1,3,9}, {1,3,8}, {1,2,9}, {1,2,8}, {1,2,3}, {0,1,9}, {0,1,4}, {2,3,8,9}, {1,3,8,9}, {1,2,8,9}, {1,2,3,9}, {1,2,3,8}, {0,1,4,9}, {1,2,3,8,9},

FIGURE 2. The illustration of pre-computation operation in running example

In the stage of result refinement, we first read all the local quasi-frequent itemsets into the memory. Then another sequential scan on T_O is performed to compute support count, i.e. the absolute occurrence number, for each local quasi-frequent itemset. Then the local quasi-frequent itemsets whose support counts are no less than $\lceil tn_o \times \omega \rceil$ are maintained as the global quasi-frequent itemsets, which are stored in a file F_{qf} and will be used for the following frequent itemset mining. The schema of F_{qf} is $F_{qf}(IS, SUP)$, where IS is the quasi-frequent itemset and SUP is the corresponding support count of IS in T_O. The quasi-frequent itemsets in F_{qf} are sorted in the descending order of the support counts. In the rest of this paper, the *support level* of an itemset is referred to as its usual support (relative value) and the *support count* of an itemset is used to represent the occurrence number of the transaction in the table.

Here, it should be noted that, the pre-computation operation is executed only once. Then, as the more transactions are accumulated in T_Δ, an incremental strategy (Section IV-E) is used to update the quasi-frequent itemsets quickly.

Example 4.1: As illustrated in Figure 2, the transaction table T in the running example is split into two part: T_O with ten transactions and T_Δ with three transactions. The value of ω is set to 0.1. Given $tn_o = 12$ and $\omega = 0.1$, the support counts of quasi-frequent itemsets should be at least $\lceil tn_o \times \omega \rceil = 2$. The quasi-frequent itemsets in the running example are depicted in Figure 2 also. There are 55 quasi-frequent itemsets here, among them 19 quasi-frequent itemsets have the support count in T_O which are no less than 3.

C. BASIC PROCESS

Given the support threshold *minsup*, this part introduces the basic process that PFIM discovers the frequent itemsets on $T = T_O \cup T_\Delta$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

cnt_Δ $mas_\Delta = 2$

item	0	1	2	3	4	5	6	7	8	9
count	0	0	1	1	2	0	1	2	1	2

FIGURE 3. The illustration of cnt_Δ in running example

1) The special case

First, we discuss a special case. If T_Δ is empty, i.e. $tn_\Delta = 0$, the processing of PFIM is simple. It just needs to read the quasi-frequent itemsets in F_{qf} sequentially. $\forall t \in F_{qf}$, let t be the current element retrieved in F_{qf}. If $t.SUP \geq \lceil tn_o \times minsup \rceil$, $t.IS$ is reported as a frequent itemset. Since the elements in F_{qf} are arranged in descending order of SUP, if $t.SUP < \lceil tn_o \times minsup \rceil$, it can be guaranteed that all frequent itemsets are discovered and the sequential scan on F_{qf} terminates.

2) The general case

Of course, usually, T_Δ is not empty. Due to the existence of new transactions, we may find new frequent itemsets from T_Δ and T_O which are not contained in F_{qf}.

In the rest of this part, we describe the processing of PFIM given that T_Δ is not empty. An itemset is frequent, if there are at least $\lceil n \times minsup \rceil$ transactions in T containing it, where $n = tn_o + tn_\Delta$ and $T = T_O \cup T_\Delta$. The execution in the general case consists of four steps.

Step 1: Sequential scan on T_Δ.

PFIM first retrieves the transactions in T_Δ. $\forall t_\Delta \in T_\Delta$, let t_Δ be the currently retrieved transaction. $\forall i \in t_\Delta$, i is an item in t_Δ, we increase the count of i, whose initial value is 0, by 1. Due to its relative small size of T_Δ and the simple computation, this sequential scan can be executed quickly. We use an array cnt_Δ to keep these counts. $\forall i \in U$, $cnt_\Delta[i]$ is the count of item i in T_Δ, $cnt_\Delta[i] = 0$ if i does not appear in any transaction in T_Δ. The value of mas_Δ is the maximum support count for all items in T_Δ.

Example 4.2: The array cnt_Δ of the running example is depicted in Figure 3 and the value of mas_Δ is 2.

Step 2: Sequential scan on F_{qf}.

Then, PFIM begins to retrieve F_{qf}. $\forall t \in F_{qf}$, let t be the currently retrieved quasi-frequent itemset in F_{qf}. The quasi-frequent itemsets in F_{qf} can be divided into three classes: (1) definitely belonging to the frequent itemsets, (2) definitely not belonging to the frequent itemsets, (3) possibly belonging to the frequent itemsets. Given $|t.IS| = 1$ and $t.IS = \{i\}$, if $t.SUP + cnt_\Delta[i] \geq \lceil n \times minsup \rceil$, $t.IS$ is frequent, otherwise, $t.IS$ is not frequent. Given $|t.IS| \geq 2$, if $t.SUP \geq \lceil n \times minsup \rceil$, $t.IS$ is frequent obviously, otherwise, if $t.SUP + mas_\Delta < \lceil n \times minsup \rceil$, $t.IS$ certainly not a frequent itemset. In other cases, t may be a frequent itemset, depending on the transactions in T_Δ, and PFIM maintains t in a set ST_{CAD}.

Example 4.3: In the running example, given the support threshold $minsup = 0.2$ and the total transaction number $n = 15$, the support count of any frequent itemset should be at least 3. As depicted in Figure 2, the quasi-frequent itemsets (19 in total) in F_{qf} whose support counts are at least three can be reported as frequent itemsets directly. The other quasi-frequent itemsets (36 in total), whose support counts are 2, are stored in ST_{CAD}.

item	2	3	4	6	7	8	9
TID list	{3}	{3}	{1, 3}	{1}	{2, 3}	{3}	{1, 3}

quasi frequent itemsets in ST_{CAD} whose support counts are increased.

{7,9}: 1, {7,8}: 1, {6,9}: 1, {3,8}: 1, {2,3}: 1, {7,8,9}: 1, {3,8,9}: 1, {2,3,9}: 1, {2,3,8}: 1, {2,3,8,9}: 1

quasi frequent itemsets in ST_{CAD} whose support counts are not increased.

{1,8}, {1,7}, {1,6}, {1,4}, {1,3}, {1,2}, {0,1}, {1,8,9}, {1,6,7}, {1,4,9}, {1,3,9}, {1,3,8}, {1,2,9}, {1,2,8}, {1,2,3}, {0,1,9}, {0,1,4}, {1,3,8,9}, {1,2,8,9}, {1,2,3,9}, {1,2,3,8}, {0,1,4,9}, {1,2,3,8,9}, {6,7}, {3,6}, {3,6,9}

FIGURE 4. The illustration of execution of step 3 in running example

Step 3: Increase supports for itemsets in ST_{CAD}.

When all quasi-frequent itemsets are retrieved already, PFIM needs to increase the support counts of quasi-frequent itemsets in ST_{CAD} by their counts in T_Δ, this can be done by a sequential scan on T_Δ. Of course, if ST_{CAD} is empty, the sequential scan on T_Δ is unnecessary. Since T_Δ may be large also, FPIM retrieves B transactions from T_Δ at most in BUF_Δ every iteration. For the current iteration, the transactions maintained in memory are transformed into vertical representation, i.e. each item is associated with the list of identifiers (TID) of transactions containing the item. $\forall t \in ST_{CAD}$ and $t.IS = \{i_{j_1}, i_{j_2}, \ldots, i_{j_a}\}$, the number of transactions in BUF_Δ containing $t.IS$ is $|\bigcap_{b=1}^{a} i_{j_b}.tlist|$, where $i_{j_b}.tlist$ is the TID list corresponding to the item i_{j_b}. Therefore, in this iteration, the support count of t is increased by $|\bigcap_{b=1}^{a} i_{j_b}.tlist|$, i.e. $t.SUP$ += $|\bigcap_{b=1}^{a} i_{j_b}.tlist|$. The similar iteration continues until all transactions in T_Δ are processed already. Then, the itemsets in ST_{CAD} are traversed and the itemsets whose support counts are no less than $\lceil n \times minsup \rceil$ are reported as frequent itemsets.

Example 4.4: The vertical representation of T_Δ in the running example is illustrated in Figure 4. There are 36 quasi-frequent itemsets in ST_{CAD}, among them only 10 quasi-frequent itemsets can increase their support counts in T_Δ. Since the support count of any quasi-frequent itemset in ST_{CAD} is 2, the 10 quasi-frequent itemsets have the support counts of 3 and can be reported as frequent itemsets, while other 26 quasi-frequent itemsets can be discarded.

Step 4: Compute new frequent itemsets in T_Δ if required.

It is known that F_{qf} contains the quasi-frequent itemsets whose support counts in T_O are at least $\lceil tn_o \times \omega \rceil$. Therefore, for any itemset which is not contained in F_{qf}, its support count in T_O is at most $(\lceil tn_o \times \omega \rceil - 1)$. Besides, we have already acquired the value of mas_Δ, the maximum count for all items in T_Δ. This means that the maximum support count of any itemset in T_Δ cannot exceed max_Δ. If $(\lceil tn_o \times \omega \rceil - 1) + mas_\Delta < \lceil n \times minsup \rceil$, there are no new frequent itemsets generated from T_Δ that are not

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

contained in F_{qf}, and PFIM does not need to compute new itemsets in T_Δ. Conversely, if $(\lceil tn_o \times \omega \rceil - 1) + mas_\Delta >= \lceil n \times minsup \rceil$, PFIM still needs to discover required frequent itemsets from T_Δ whose support counts in T_Δ are at least $\lceil n \times minsup \rceil - (\lceil tn_o \times \omega \rceil - 1)$. This corresponds to a new frequent itemset mining on T_Δ with the specified support threshold $minsup_\Delta = \frac{\lceil n \times minsup \rceil - (\lceil tn_o \times \omega \rceil - 1)}{tn_\Delta}$.

Example 4.5: In the running example, the value of ω is 0.1 and $tn_o = 12$, the maximum possible support count of any itemset not in F_{qf} is $(\lceil tn_o \times \omega \rceil - 1) = 1$. Since $mas_\Delta = 2$, $(\lceil tn_o \times \omega \rceil - 1) + mas_\Delta \geq 3$, it is possible that T_Δ can generate new frequent itemsets. Step 4 should be executed in the running example. Here, the specified support threshold $minsup_\Delta = 0.67\ (\frac{2}{3})$ in T_Δ.

Since T_Δ could be large also, PFIM retrieves B transactions at most from T_Δ in BUF_Δ every iteration, computes the local frequent itemsets in BUF_Δ with the support threshold $minsup_\Delta$. The transactions in BUF_Δ first are transformed into vertical representation format and the local frequent 1-itemsets $LF_{1,\Delta}$ can be determined. In the following operation, suppose that we have acquired local frequent k-itemsets $LF_{k,\Delta}$ ($k \geq 1$). By use of $LF_{k,\Delta}$, the local frequent $(k+1)$-itemsets $LF_{k+1,\Delta}$ can be generated. $\forall IS_1, IS_2 \in LF_{k,\Delta}$, the items in IS_1 (or IS_2) are arranged in the ascending order, if $IS_1 \prec IS_2$, where $\prec$ is an operator of lexicographically smaller relation, and $\forall 1 \leq b \leq k-1$, $IS_1[b] == IS_2[b]$, i.e. the first $(k-1)$ itemsets of IS_1 and IS_2 are equal, a new $(k+1)$-itemset can be generated: $IS_{k+1} = \{IS_1[1], \ldots, IS_1[k-1], IS_1[k], IS_2[k]\}$. Here, if IS_1 and IS_2 are two 1-itemsets, we consider that their first $(k-1)$ itemsets are equal, although their first $(k-1)$ itemsets are empty. Given itemset IS, we denote by $IS.tlist$ the corresponding TID lists of IS. If $\frac{|IS_1.tlist \cap IS_2.tlist|}{|BUF_\Delta|} < minsup_\Delta$, IS_{k+1} is discarded. Otherwise, IS_{k+1} is added into $LF_{k+1,\Delta}$. If $LF_{k+1,\Delta}$ is empty, the current iteration terminates and PFIM outputs all of the local frequent itemsets. Then PFIM clears BUF_Δ and continues retrieving the transactions in T_Δ for the next iteration. This iteration continues until all transactions in T_Δ have been retrieved and processed. All the local frequent itemsets LF_Δ are the candidates of the global frequent itemsets in T_Δ. Note that for any frequent itemset computed in T_Δ, if it is contained in F_{qf}, it has been considered in the previous processing. Therefore, before computing the support counts of itemsets in LF_Δ, PFIM removes the local frequent itemsets in LF_Δ that have appeared in F_{qf}. This can reduce the counting cost for local frequent itemset in T_Δ considerably. The support counts of the local frequent itemsets in LF_Δ can be computed by another scan on T_Δ, which is in a similar way as in the description of step 3. The global frequent itemsets in T_Δ, denoted by GF_Δ, are the local frequent itemsets whose support counts are no less than $\lceil n \times minsup \rceil - (\lceil tn_o \times \omega \rceil - 1)$. The itemsets in GF_Δ needs another scan on T_O to compute their support counts in $T = T_O \cup T_\Delta$. Then, only the itemsets in GF_Δ, whose support counts are no less than $\lceil n \times minsup \rceil$, are reported as the frequent itemsets.

item	2	3	4	6	7	8	9
TID list	{3}	{3}	{1, 3}	{1}	{2, 3}	{3}	{1, 3}

frequent 1 *itemsets in* T_Δ {4}: 0.67, {7}: 0.67, {9}: 0.67
frequent 2 *itemsets in* T_Δ {4, 9}: 0.67

FIGURE 5. The illustration of execution of step 4 in running example

Example 4.6: The frequent itemsets with $minsup_\Delta = 0.67$ in T_Δ are shown in Figure 5. Since $\{4\}, \{7\}, \{9\}, \{4, 9\}$ are contained in F_{qf}, they can be removed. And PFIM has discovered all the frequent itemsets.

D. PRUNING OPERATION

In terms of the description in Section IV-C, PFIM can reuse the pre-computation result of T_O and reduce the execution cost significantly. In this part, we discuss how to improve PFIM further to speed up its execution by pruning operation.

1) Pruning in step 2

One main part of the cost in PFIM is to compute the support counts of the itemsets of ST_{CAD} in T_Δ, i.e. step 3 in Section IV-C2. Therefore, if we can reduce the number of itemsets in ST_{CAD} in step 2, the counting cost in T_Δ can be decreased.

In Section IV-C2, $\forall t \in ST_{CAD}$, it satisfies:

$$\lceil n \times minsup \rceil - mas_\Delta \leq t.SUP < \lceil n \times minsup \rceil$$

That is, we use the maximum count mas_Δ of the single item in T_Δ to determine the support count range of the possible frequent itemsets. Obviously, if we can narrow down the support count range, the size of ST_{CAD} can be reduced.

As described in the process of step 2, PFIM can determine directly whether the quasi-frequent 1-itemsets in F_{qf} are frequent itemsets. Therefore, ST_{CAD} only needs to maintain the quasi-frequent itemsets which contain at least two items.

At the end of step 2, PFIM maintains the possible frequent itemsets in ST_{CAD}. Before entering the step 3, we wonder whether the size of ST_{CAD} can be decreased further. But, it should be noted that, the reason to prune the itemsets in ST_{CAD} is that the execution cost in step 3 can be high if ST_{CAD} contains many itemsets, the cost of pruning operation should keep low also. Otherwise, the overall cost in step 2 and step 3 can still be large, which can affect the high efficiency of PFIM.

In order to prune itemsets in ST_{CAD} as many as possible with a low cost, PFIM first chooses two items of each quasi-frequent itemset in ST_{CAD} which have the smallest support counts in cnt_Δ. $\forall t \in ST_{CAD}$, we keep an item pair $(i_{t,1}, i_{t,2})$ in $t.IS$ in PIP (*P*runing *I*tem *P*air), $(i_{t,1}, i_{t,2})$ are two items with the smallest support counts in cnt_Δ among all items in $t.IS$ and $i_{t,1} \prec i_{t,2}$. PFIM computes the support counts of the item pairs in PIP in the similar operation as step 3 in Section IV-C2. Let $SUP(T_\Delta, \{i_{t_1}, i_{t_2}\})$ be the support count of $(i_{t,1}, i_{t,2})$ in T_Δ. Then the pruning rule 1 (PR1) listed below is performed.

PR1: $\forall (i_{t,1}, i_{t,2}) \in PIP$, we can determine the quasi-frequent itemset $t.IS$ corresponding to $(i_{t,1}, i_{t,2})$ in

PIP structure

{6,7}: 0, {1,8}: 0, {1,7}: 0, {1,6}: 0, {3,8}: 1, {1,4}: 0, {1,3}: 0, {3,6}: 0, {1,2}: 0, {7,9}: 1, {7,8}: 1, {0,1}: 0, {2,3}: 1, {6,9}: 1

the quasi frequent itemsets in ST_{CAD}

{7,9}, {7,8}, {6,9}, {6,7}, {3,8}, {3,6}, {2,3}, {1,8}, {1,7}, {1,6}, {1,4}, {1,3}, {1,2}, {0,1}, {7,8,9}, {3,8,9}, {3,6,9}, {2,3,9}, {2,3,8}, {1,8,9}, {1,6,7}, {1,4,9}, {1,3,9}, {1,3,8}, {1,2,9}, {1,2,8}, {1,2,3}, {0,1,9}, {0,1,4}, {2,3,8,9}, {1,3,8,9}, {1,2,8,9}, {1,2,3,9}, {1,2,3,8}, {0,1,4,9}, {1,2,3,8,9},

FIGURE 6. The illustration of execution of PR1 in running example

ST_{CAD}, (1) if $SUP\,(T_\Delta, \{i_{t_1}, i_{t_2}\}) + t.SUP < \lceil n \times minsup \rceil$, t can be removed from ST_{CAD}, (2) if $|t.IS| == 2$ and $SUP\,(T_\Delta, \{i_{t_1}, i_{t_2}\}) + t.SUP >= \lceil n \times minsup \rceil$, t can be removed from ST_{CAD} and $t.IS$ is reported as a frequent itemset.

Since the support count of 2-itemset is usually lower than the support count of any item in the 2-itemset, the pruning operation can reduce the size of ST_{CAD} further. Furthermore, because the support counts of the quasi-frequent 2-itemsets in ST_{CAD} have been computed, the quasi-frequent 2-itemsets can be removed from ST_{CAD}.

Example 4.7: The PIP structure in the running example is depicted in Figure 6. There are 14 item pairs used for PR1. After counting the occurrence number of these item pairs, the support counts of these item pairs in T_Δ are obtained. Initially, ST_{CAD} keeps 36 quasi-frequent itemsets, whose support counts in T_O are 2. The support counts of all quasi-frequent 2-itemsets (14 in total) in T can be determined and they can be removed from ST_{CAD}. For other quasi-frequent itemsets (22 in total), their corresponding item pairs can be used to perform PR1 pruning. By use of PR1, the number of quasi-frequent itemsets in ST_{CAD} can be reduced from 36 to 5. Only the quasi-frequent itemsets in Figure 6 with red color are remained after PR1 pruning.

Three aspects, i.e. the reason why we utilize item pairs to perform PR1 pruning, are considered below. The first aspect is that, $\forall t \in ST_{CAD}$, $|t.IS| \geq 2$, since the 1-itemsets have been processed by use of cnt_Δ. If we use itemsets whose sizes are more than two to prune ST_{CAD}, then the itemsets whose sizes are equal to two cannot be pruned. The second aspect is that the cost to compute the information used for pruning should be low. Thus, we only utilize the item pairs in each quasi-frequent itemset of ST_{CAD}. The computation of the support counts for these item pairs can be executed relatively fast because the number of item pairs to count is small and the item pair is short (only contain two items). The third aspect is that, the pruning effect is supposed to be good. We select two items in each quasi-frequent itemset of ST_{CAD} with the smallest support counts in cnt_Δ. Intuitively, if the support counts for the two items are small, it is expected that the support count of the 2-itemset containing these two items should be smaller.

2) Pruning in step 4

Another possible main part of the cost in PFIM is the operation to generate the new frequent itemsets in T_Δ if required, i.e. the step 4 in Section IV-C2.

Although T_Δ is much smaller compared with T_O, its absolute size may be large also. If $minsup_\Delta$ is low, the computation on T_Δ and T_O can take a relatively long execution time, this will affect the high efficiency of PFIM. Therefore, in step 4, we also need a pruning operation to speed up its execution. The pruning operation is performed on the generation of the local frequent itemsets in T_Δ because, if the number of local frequent itemsets is decreased, the cost in support counting can be reduced also.

Given the support threshold $minsup_\Delta$, suppose that, in the current iteration, we have retrieved transactions in BUF_Δ and transformed them into vertical representation. According to the description in step 4 in Section IV-C2, PFIM first determines the local frequent 1-itemsets $LF_{1,\Delta}$. Given the local frequent k-itemsets $LF_{k,\Delta} (k \geq 1)$ we have acquired, before computing the candidates of the local frequent $(k+1)$-itemsets, the pruning operation is executed. It is noted that the itemsets in $LF_{k,\Delta}$ are sorted in the lexicographical order. The itemsets in $LF_{k,\Delta}$ are first grouped according to the first $(k-1)$ items, i.e. each group contains the itemsets sharing the same first $(k-1)$ items. Here, we assume that the local frequent 1-itemsets $LF_{1,\Delta}$ belong to one group. After the grouping operation, assume that we can obtain h groups $G_1, G_2, \ldots, G_h$. Let lf_1 and lf_2 be two itemsets, $lf_1 \cup lf_2$ be the union operation of two itemsets, it generates a set of items which is in lf_1, in lf_2 or in both. The pruning operation in step 4 has two rules: pruning rule 2 (PR2) and pruning rule 3 (PR3).

- PR2: $\forall 1 \leq b \leq h$, if G_b contains only one k-itemset, G_b can be removed.
- PR3: $\forall 1 \leq b \leq h$, if the itemset $\left(\bigcup_{lf \in G_b} lf\right)$ is contained in F_{qf}, G_b can be removed.

The intuition behind PR2 is that, if G_b contains only one k-itemset, $(k+1)$-itemsets cannot be generated from G_b. The intuition behind PR3 is that, if the itemset $\left(\bigcup_{lf \in G_b} lf\right)$ is contained in F_{qf}, the possible itemset $\left(\bigcup_{lf \in G_b} lf\right)$ has already been considered in the previous steps, we do not need to consider it again.

Example 4.8: Since the size of T_Δ is relatively small, we do not illustrate the execution of PR2 and PR3 in the running example. But, the idea of the pruning operation is intuitive.

E. UPDATE OPERATION

As the description above, the new transactions are accumulated in T_Δ. When the size of T_Δ reaches a certain threshold, for example, 5% of the size of T_O, the transactions in T_Δ and T_O are merged. At this point, the quasi-frequent itemsets in F_{qf} needs to be updated also. Of course, re-construction totally is one choice, i.e. re-compute the quasi-frequent itemsets with the support threshold ω on $T = T_O \cup T_\Delta$ from scratch. But the total re-construction can be expensive. Therefore, in this paper, we propose an incremental update strategy, which utilizes the existing information computed already to speed up the update operation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

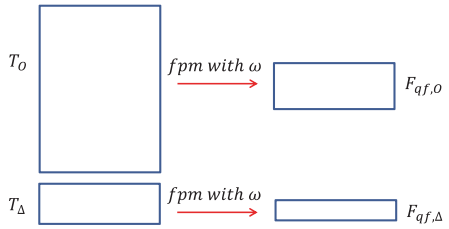


FIGURE 7. The illustration of update operation

The goal of the update operation is to generate the quasi-frequent itemsets on T given the support level ω. As illustrated in Figure 7, the local quasi-frequent itemsets of T_O are kept in $F_{qf,O}$, which is exactly the F_{qf} mentioned in Section IV-B. First we need to add the occurrences of the local quasi-frequent itemsets of $F_{qf,O}$ in T_Δ, which is similar to the execution in step 3 in Section IV-C2. Then, the local quasi-frequent itemsets in $F_{qf,O}$, whose support counts are no less than $\lceil \omega \times n \rceil$ $(n = tn_o + tn_\Delta)$, are written into the new file F_{qf}. Also, as depicted in Figure 7, the local quasi-frequent itemsets of T_Δ are kept in $F_{qf,\Delta}$. In order to avoid duplicate computation, the local quasi-frequent itemsets in $F_{qf,\Delta}$, which have been contained in $F_{qf,O}$, are removed before the support counting. The support counts of the local quasi-frequent itemsets in $F_{qf,\Delta}$ are calculated by another scan on T_O and T_Δ. This processing is similar to that in step 4 in Section IV-C2. The local quasi-frequent itemsets in $F_{qf,\Delta}$, whose support counts are no less than $\lceil \omega \times n \rceil$, are written into the new file F_{qf}. This moment, the update operation terminates and F_{qf} maintains the updated quasi-frequent itemsets of T.

According to the description above, the computation of $F_{qf,O}$ on T_O is saved. The itemsets in $F_{qf,O}$ just need to add their support counts by the relatively small T_Δ. The number of the itemsets in $F_{qf,\Delta}$ can be reduced significantly by the containment checking in $F_{qf,O}$. The computation cost of adding the the support counts of itemsets in T_Δ and T_O can be lowered accordingly. Therefore, the incremental update strategy can runs much faster than the total re-construction strategy, which also is verified in the experiments.

F. DISCUSSIONS

This paper assumes that there exists a lower-bound ω for the value of *minsup* in practical applications. The value of ω is determined by the domain experts or the lowest value of the used support levels. However, some user may submit a frequent itemset mining with the specified *minsup* which is lower than ω. Although this case should be quite unusual (too many frequent itemsets can be generated), we still hope that PFIM can deal with this case. Total re-computation on $T = T_O \cup T_\Delta$ is one choice. But this choice should be expensive and it neglects the pre-computation result of the existing F_{qf}. A proper alternative is to reuse the pre-computation result. F_{qf} maintains the quasi-frequent itemsets whose support counts in T_O are no less than $\lceil \omega \times tn_o \rceil$. We first compute the frequent itemsets on T_O with support level *minsup* (here $minsup < \omega$). This computation is similar to candidate generation stage in Section IV-B. The difference is that, the local frequent itemsets, which are contained in F_{qf}, are removed before entering result refinement stage. This moment, the newly generated frequent itemsets on T_O and the existing F_{qf} can be treated as the $F_{qf,O}$ in the incremental update strategy. Then we set ω to be *minsup* and the remaining operations can be executed as that in Section IV-E. The itemsets in the generated new F_{qf} are the required frequent itemsets. Finally, we merge T_O and T_Δ. With the new F_{qf}, the new T_O and new ω, PFIM can continue processing the coming frequent itemset mining. Since the emphasis of this paper is to efficiently compute frequent itemsets on massive data, the discussion in this part will not be explored further.

V. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETTINGS

To evaluate the performance of PFIM, we implement it in Java with jdk-8u20-windows-x64. The experiments are executed on LENOVO ThinkCentre M8400 (Intel (R) Core(TM) i7-3770 CPU @ 3.40GHz (8 CPUs) + 32G memory + 64 bit windows 7). The used data set is stored in Seagate Expansion STBV3000300 (3TB). In the experiments, the performance of PFIM is evaluated against Apriori [20] and negFIN [25]. The reason to select the two algorithms for performance evaluation is that, (1) Apriori is a classic level-wise algorithm and thus we select it as the baseline algorithm, (2) negFIN is the latest algorithm and it shows a performance advantage over the other main frequent itemset mining algorithms [25]. Since the high memory consumption, we adopt a Partition mode for negFIN, i.e. in the first stage, we use negFIN to compute local frequent itemsets, and in the second stage, support counts of the local frequent itemsets are computed to determine the global frequent itemsets. It is found that, except for the cases of very low support levels, the execution time in the first stage usually dominates the overall execution time.

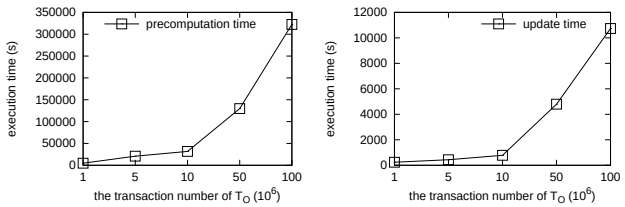
In the experiments, we evaluate the performance of PFIM in terms of several aspects: transaction number of T_O (tn_o), support level (*minsup*), the relative proportion of transaction number tn_Δ of T_Δ over tn_o. The experiments are executed on two data sets: synthetic data set and real data set. The synthetic data set is generated by the IBM Quest Data generator [1], in which the number of items is set to 1000, the number of maximal potential large itemsets is set to 2000, average item number per transaction is set to 20 and the average maximal potentially frequent itemset size is set to 6. The used parameter settings are listed in Table 2. In the experiments, the lower-bound ω for the value of *minsup* in practical applications is set to 0.001 in the synthetic data set. The real data used is kosarak data set from FIMI Dataset Repository (http://fimi.uantwerpen.be/data/).

[1]https://sourceforge.net/projects/ibmquestdatagen/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

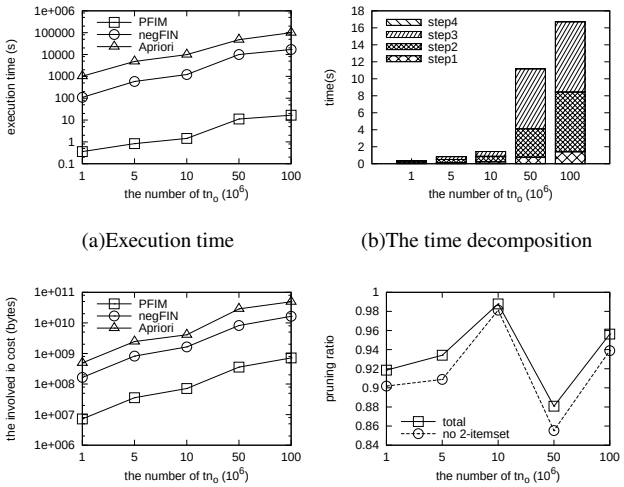
TABLE 2. Parameter Settings

Parameter	Used values
$tn_o(10^6)$ (synthetic)	1, 5, 10 (*def*), 50, 100
minsup (synthetic)	0.001, 0.005, 0.01 (*def*), 0.015, 0.02
tn_Δ/tn_o (synthetic)	$1\%, 2\%, 3\%(def), 4\%, 5\%$
minsup (real)	0.01, 0.02, 0.03, 0.04, 0.05



(a)Pre-computation operation (b)Incremental update

FIGURE 8. The result of pre-computation and update



(a)Execution time (b)The time decomposition

(c)The I/O cost (d)Pruning ratio of PR1

FIGURE 9. The effect of transaction number in old table

B. PRE-COMPUTATION OPERATION AND UPDATE OPERATION

In this part, we report the execution times of pre-computation operation and the update operation. As depicted in Figure 8(a), given that $tn_\Delta/tn_o = 0.03$, we illustrate the pre-computation cost of PFIM with the varying values of tn_o. The pre-computation time of PFIM increases quickly with the greater value of tn_o. At $tn_o = 100 \times 10^6$, the pre-computation time is 322337.466s. For one hand, this indicates that frequent itemset mining algorithms require a rather long execution time to compute frequent itemsets if the value of *minsup* is small. For the other hand, the pre-computation operation is expensive on massive data. But it should be noted that the pre-computation only is executed once from scratch and then the incremental update can be performed. After the initial pre-computation operation, PFIM can perform incremental update to speed up the construction of the required structures. As depicted in Figure 8(b), given $\frac{tn_\Delta}{tn_o} = 0.05$ and tn_o ranging from 10^6 to 100×10^6, incremental update can run 33 times faster than the pre-computation operation.

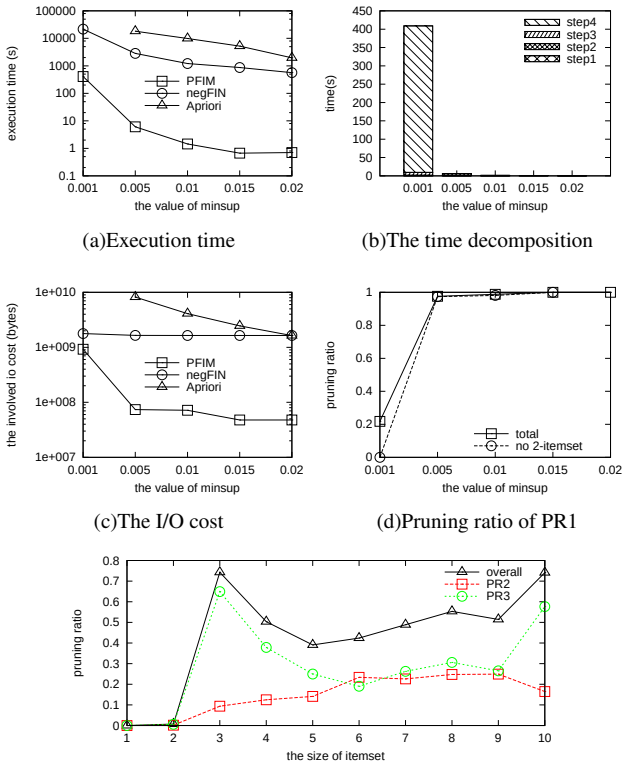
C. EXP 1: THE EFFECT OF TRANSACTION NUM IN OLD TABLE

Given $\frac{tn_\Delta}{tn_o} = 0.03$ and $minsup = 0.01$, experiment 1 evaluates the performance of PFIM on varying transaction numbers in old table. As depicted in Figure 9(a), PFIM runs 750.7 times faster than negFIN and 5134.6 times faster than Apriori. This shows the high efficiency for PFIM to compute frequent itemsets on massive data. Even at $tn_o = 100 \times 10^6$, PFIM can return frequent itemsets within 17s. The decomposition of execution time of PFIM is illustrated in Figure 9(b). Given the specified value 0.01 of *minsup*, step 4 in PFIM, i.e. computing new frequent itemsets in T_Δ, is not executed. In experiment 1, the most of the execution time of PFIM is spent on step 2 and step 3. Due to the pre-computation results, PFIM also involves much less I/O cost than other algorithms. As illustrated in Figure 9(c), PFIM involves 22.8 times less I/O cost than negFIN and 68.5 times less I/O cost than Apriori. Since the step 4 is not executed, PR2 and PR3 are not invoked also. Only the effect of PR1 is reported in experiment 1. The pruning ratio of PR1 is depicted in Figure 9(d). Let $|ST_{CAD}|_{before}$ and $|ST_{CAD}|_{after}$ be the numbers of candidates in ST_{CAD} before and after the pruning with PR1 respectively. Here, the results of solid line are computed by $\frac{|ST_{CAD}|_{before} - |ST_{CAD}|_{after}}{|ST_{CAD}|_{before}}$, while the results of dotted line are computed similarly, except that not considering 2-itemsets. Figure 9(d) shows that PR1 can discard most of the candidates in ST_{CAD} before entering step 3.

D. EXP 2: THE EFFECT OF SUPPORT LEVEL

Given $tn_o = 10 \times 10^6$ and $\frac{tn_\Delta}{tn_o} = 0.03$, experiment 2 evaluates the performance of PFIM on varying support levels. As illustrated in Figure 10(a), if the value of *minsup* decreases, the execution times of the three algorithms rise significantly. The result of Apriori at $minsup = 0.001$ is not reported in Figure 10(a), since it is rather expensive for Apriori to compute frequent itemsets at such low support level. Averagely, PFIM runs 693.29 times faster than negFIN and runs 5106.1 times faster than Apriori. The time decomposition of PFIM is depicted in Figure 10(b). At $minsup = 0.001$, the step 4 of PFIM is executed. Obviously its execution time dominates other steps since it involves the processing of the old table. Figure 10(c) compares the I/O costs of the three algorithms. The pruning ratio of PR1 is shown in Figure 10(d). At $minsup = 0.001$, PR1 only can remove the candidate 2-itemsets in ST_{CAD}. But, in other cases, PR1 can discard most of the candidates. In experiment 2, PR2 and PR3 are executed at $minsup = 0.001$. Therefore, we report their results in Figure 10(e) and the results are displayed according to their effect on the candidate itemsets of different sizes. PR2 and PR3 actually first perform a grouping operation and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access

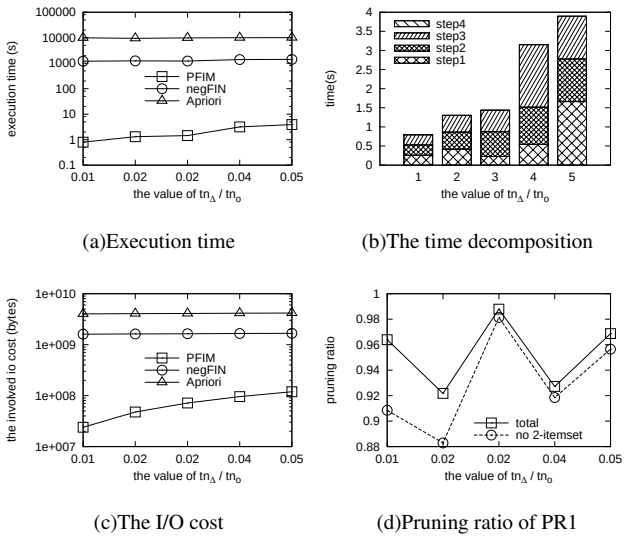


(a)Execution time (b)The time decomposition

(c)The I/O cost (d)Pruning ratio of PR1

(e)Pruning ratios of PR2 and PR3 (*minsup* = 0.001)

FIGURE 10. The effect of minimum support threshold

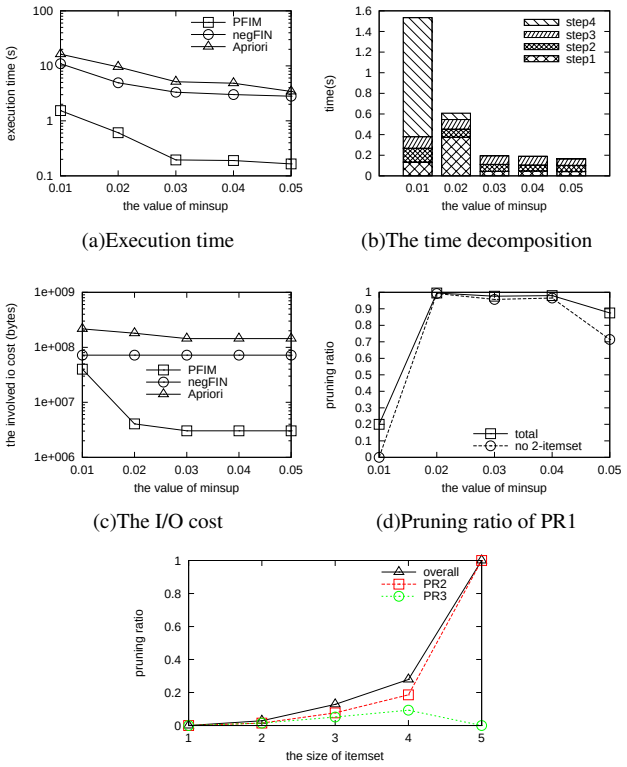


(a)Execution time (b)The time decomposition

(c)The I/O cost (d)Pruning ratio of PR1

FIGURE 11. The effect of transaction number in new table

then check candidates in each group. Obviously, when the size of the candidate itemsets is 1 or 2, the pruning effects of PR2 and PR3 are poor. But, as the size of the candidates increases, PR2 and PR3 can achieve a satisfactory overall pruning effect.



(a)Execution time (b)The time decomposition

(c)The I/O cost (d)Pruning ratio of PR1

(e)Pruning ratios of PR2 and PR3 (*minsup* = 0.01)

FIGURE 12. The effect of real data set

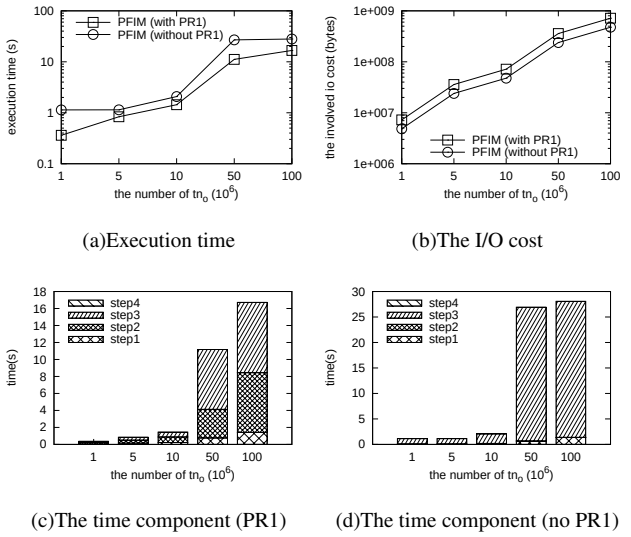
E. EXP 3: THE EFFECT OF TRANSACTION NUM IN NEW TABLE

Given $tn_o = 10 \times 10^6$ and $minsup = 0.01$, experiment 3 evaluates the performance of PFIM on varying transaction numbers in new table. As depicted in Figure 11(a), the execution time of PFIM increases gradually as more transactions in new table are involved, while the execution times of other two algorithms basically remain unchanged. This reason is that, a larger value of tn_Δ will increase the execution times in step 1, 2 and 3, but it does not affect the total transaction number significantly. PFIM runs 820.4 times faster than negFIN and 6476.8 times faster than Apriori. The time decomposition of PFIM is illustrated in Figure 11(b), the step 4 of PFIM is still not executed in experiment 3. As shown in Figure 11(c), the I/O costs of the three algorithms are similar to those in Figure 11(a) and can be explained in the same way. PFIM involves 31.1 times less I/O cost than negFIN and 77.6 times less I/O cost than Apriori. The pruning ratio of PR1 is depicted in Figure 11(d), and most of the candidate itemsets in ST_{CAD} can be removed by PR1.

F. EXP 4: THE REAL DATA SET

The real data, kosarak data set, is obtained from FIMI Dataset Repository. It contains 990002 transactions, in which the maximum length of the transactions is 2498 and the average length of the transaction is 8.1. In experiment 4, we split the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2902602, IEEE Access



(a)Execution time (b)The I/O cost

(c)The time component (PR1) (d)The time component (no PR1)

FIGURE 13. The effect of PR1 pruning

kosarak data set into two partitions by randomly selecting the specified number of transactions. The partition 1 is used as the old table while the partition 2 acts as the new table, $\frac{tn_\Delta}{tn_o}$ is set to be 0.03 in experiment 4. The value of ω is set to 0.01. The experiment 4 evaluates the performance of PFIM on real data set with varying support levels. As illustrated in Figure 12(a), PFIM runs 13.1 times faster than negFIN and 19.8 times faster than Apriori. The variation trends of the algorithms are similar with those in experiment 2 and can be explained similarly. The time decomposition of PFIM is depicted in Figure 12(b). PFIM involves an order of magnitude less I/O cost than other algorithms, 18.1 times less I/O cost than negFIN and 38.4 times less I/O cost than Apriori. The pruning effects of PR1, PR2 and PR3 and illustrated in Figure 12(d) and Figure 12(e) respectively.

G. EXP 5: THE PRUNING EFFECT

Given $\frac{tn_\Delta}{tn_o} = 0.03$ and $minsup = 0.01$, the experiment 5 evaluates the performance of PR1 in PFIM with varying transaction number in old table. For PR1, it utilizes an extra I/O operation on T_Δ to reduce the number of candidates in step 2, and therefore, the computation cost in step 3 can be saved significantly. It is described in Section IV-D1 that PR1 can speed up the execution of PFIM. This is verified in experiment 5. As depicted in Figure 13(a), PFIM with PR1 runs 2 times faster than PFIM without PR1. As shown in Figure 13(b), due to the extra I/O operation, the involved I/O cost in PFIM with PR1 is a little greater than that in PFIM without PR1. PFIM without PR1 runs faster than PFIM with PR1 in the execution of step 2, which is illustrated in Figure 13(c) and Figure 13(d). However, with the help of PR1, the execution time in step 3 for PFIM with PR1 is significantly saved. Comparatively, this time decrease in step 3 is much greater than the time increase in step2. This verifies that PR1 improves the overall performance of PFIM.

Since PR2 and PR3 do not involve extra I/O operation and the computation is not expensive, they can improve the overall performance of PFIM naturally. Given $tn_o = 10 \times 10^6$, $\frac{tn_\Delta}{tn_o} = 0.03$, $minsup = 0.001$, we evaluate the effects of PR2 and PR3. Since the other operations are the same, we only report the execution time to compute local frequent itemsets in step 4. Without PR2 and PR3, PFIM takes 299.987 seconds to compute the local frequent itemsets in T_Δ. With PR2 and PR3, PFIM only takes 84.712 seconds to do the same operation, since the fewer candidates are generated. This verifies the effects of PR2 and PR3.

VI. CONCLUSION

This paper considers the problem of computing frequent itemsets on massive data. It is found that the existing algorithms cannot perform frequent itemset mining on massive data efficiently. This paper utilizes the idea of reusing the work done previously and devises a precomputation-based PFIM algorithm to quickly acquire the frequent itemsets on massive data. The transaction table consists of two part: the large old table and the relatively small new table. By the quasi-frequent itemsets pre-computed on the old table, PFIM can report the frequent itemsets on massive data efficiently. Three pruning rules are proposed in this paper to speed up the execution of PFIM. The incremental update strategy is presented to re-construct the quasi-frequent itemsets quickly when merging the old table and the new table. The extensive experimental results show that PFIM has a significant performance advantage over the existing algorithms.

REFERENCES

[1] A. Ceglar and J.F. Roddick, "Association mining," ACM Comput. Surv., 38(2):5, 2006.
[2] H. Cheng, X. Yan, J. Han, and P.S. Yu, "Direct discriminative pattern mining for effective classification," in Proceedings of the 24th International Conference on Data Engineering, April 7-12, 2008, pp. 169–178.
[3] H. Wang, W. Wang, J. Yang, and P.S. Yu, "Clustering by pattern similarity in large data sets," in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, pp. 394–405.
[4] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, September 5-9, 2005, pp. 306–315.
[5] J.T.L. Wang, M.J. Zaki, H. Toivonen, and D.E. Shasha, editors. "Data Mining in Bioinformatics," Springer, 2005.
[6] R. Agrawal, T. Imielinski, and A.N. Swami, "Database mining: A performance perspective," IEEE Trans. Knowl. Data Eng., vol. 5, no. 6, pp.914–925, 1993.
[7] C.C. Aggarwal, "Data Mining - The Textbook," Springer, 2015.
[8] C.C. Aggarwal and J. Han, editors, "Frequent Pattern Mining," Springer, 2014.
[9] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," Data Min. Knowl. Discov., vol. 15, no. 1, pp.55–86, 2007.
[10] R. Agrawal, T. Imielinski, and A.N. Swami, "Mining association rules between sets of items in large databases," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.
[11] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, 1994, pp. 487–499.
[12] A. Savasere, E. Omiecinski, and S.B. Navathe, "An efficient algorithm for mining association rules in large databases," in VLDB'95, Proceedings of

21th International Conference on Very Large Data Bases, 1995, pp. 432–444.

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

[14] M.J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 326–335.

[15] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," IEEE Trans. Knowl. Data Eng., vol. 17, no. 10, pp.1347–1362, 2005.

[16] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," Data Min. Knowl. Discov., vol. 8, no. 1, pp.53–87, 2004.

[17] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-mine: Hyper-structure mining of frequent patterns in large databases," in Proceedings of the 2001 IEEE International Conference on Data Mining, 2001, pp.441–448.

[18] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," IEEE Trans. Vis. Comput. Graph., vol. 20, no. 12, pp.2122–2131, 2014.

[19] R.C. Fernandez, P.R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang, "Liquid: Unifying nearline and offline big data integration," in CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Online Proceedings, 2015.

[20] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.. Verkamo, "Fast discovery of association rules," in Advances in Knowledge Discovery and Data Mining, 1996, pp. 307–328.

[21] Z.H. 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.

[22] Z.H. Deng, Z. Wang, and J.J. Jiang, "A new algorithm for fast mining frequent itemsets using n-lists," SCIENCE CHINA Information Sciences, vol. 55, no. 9, pp.2008–2030, 2012.

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

[24] Z.H. Deng, "Diffnodesets: An efficient structure for fast mining frequent itemsets," Appl. Soft Comput., vol. 41, pp.214–223, 2016.

[25] N. Aryabarzan, B.M. Bidgoli, and M. Teshnehlab, "negfin: An efficient algorithm for fast mining frequent itemsets," Expert Syst. Appl., vol. 105, pp.129–143, 2018.

[26] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad, "Depth first generation of long patterns," in Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, 2000, pp. 108–118.

[27] G. Liu, H. Lu, W. Lou, Y. Xu, and J.X. Yu, "Efficient mining of frequent patterns using ascending frequency ordered prefix-tree," Data Min. Knowl. Discov., vol. 9, no. 3, pp.249–274, 2004.

XIXIAN HAN is associate professor in the School of Computer Science and Technology (SCST), Harbin Institute of Technology (HIT), China. He received his Ph.D degree from SCST, HIT in 2012. His main research interests include massive data analysis and massive data mining. He has published dozens of high-quality papers on reputable international conferences and peer-reviewed journals.

XIANMIN LIU is an associate of the School of Software at Harbin Institute of Technology, China. He received his Ph.D. degree from the School of Computer Science and Technology, Harbin Institute of Technology in 2013. His research interest includes database theory, massive data computing and data quality. He has published several papers in reputable international journals, including Journal of Combinatorial Optimization, Theoretical Computer Science, ACM Transactions on Database Systems.

JIAN CHEN is currently an undergraduate in the School of Computer Science and Technology at Harbin Institute of Technology, China. He will receive a Bachelor's degree from Harbin Institute of Technology, in 2019. His major research interest includes data mining and massive data management.

GUOJUN LAI is currently an undergraduate in the School of Computer Science and Technology at Harbin Institute of Technology, China. He will receive a Bachelor's degree in the School of Computer Science and Technology from Harbin Institute of Technology in 2019. His research interest includes massive data management.

HONG GAO is professor in the School of Computer Science and Technology at Harbin Institute of Technology, China. Prof. Gao is the principal investigator for several National Natural Science Foundation Projects and participated two the National Basic Research (973) Program. Her research interests include wireless sensor network, cyber-physical systems, massive data management and data mining.

JIANZHONG LI is Chair of the Department of Computer Science and Engineering at Harbin Institute of Technology, China. He is also a professor in the School of Computer Science and Technology, the Dean of School of Computer Science and Technology, and the Dean of School of Software at Heilongjiang University, China. His current research interests include data-intensive computing, wireless sensor networks and CPS.