An Efficient Approach for Mining Association Rules from Sparse and Dense Databases

Lan Vu, Gita Alaghband
Department of Computer Science and Engineering
University of Colorado Denver
Denver, USA
{lan.vu; gita.alaghband}@ucdenver.edu

Abstract— **Association rule mining (ARM) is an important task in data mining. This task is computationally intensive and requires large memory usage. Many existing methods for ARM perform efficiently on either sparse or dense data but not both. We address this issue by presenting a new approach for ARM that runs fast for both sparse and dense databases by detecting the characteristic of data subsets in database and applying a combination of two mining strategies: one is for the sparse data subsets and the other is for the dense ones. Two algorithms, FEM and DFEM, based on our approach are introduced in this paper. FEM applies a fixed threshold as the condition for switching between the two mining strategies while DFEM adopts this threshold dynamically at runtime to best fit the characteristics of the database during the mining process, especially when minimum support threshold is low. Additionally, we present optimization techniques for the proposed algorithms to speed up the mining process, reduce the memory usage and optimize the I/O cost. We also analyze in-depth the performance of FEM and DFEM and compare them with several existing algorithms. The experimental results show that FEM and DFEM achieve a significant improvement in execution time and consume less memory than many popular ARM algorithms including the well-known Apriori, FP-growth and Eclat on both sparse and dense databases.**

Index Terms- **data mining, frequent pattern mining, association rule mining, frequent itemset, transactional database**

I. INTRODUCTION

Association rule mining (ARM) aims at discovering rules specifying the frequency co-occurrence of groups of itemsets, subsequences, or substructures in a database. For example, an association rule of retail data may be of the form "70% of customers who buy milk and butter also buy bread with confidence 90%". ARM is one of fundamental tasks in data mining. Since its first introduction for sales analysis [1], ARM has been broadly applied in many fields such as market analysis, biomedical and computational biology research, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [2]. Google uses this mining task for their query recommendation system [4].

Several studies have shown that ARM methods have typically worked well for certain types of databases. Most methods performed efficiently on either sparse or dense databases but poorly on the other [1, 5, 6, 7, 8, 9, 10, 11]. Table I presents the execution time of three well-known algorithms Apriori [1], Eclat [5] and FP-growth [6] on sparse and dense databases which shows Eclat perform best on dense data while FP-growth run fastest on the sparse ones (the best execution times among the three algorithms are underlined). Therefore, it is difficult to select a suitable algorithm for a specific application. Moreover, data mining components in database management systems and statistical software usually require mining methods that stably perform on various data types.

TABLE I. EXECUTION TIME (SEC.) ON SPARSE AND DENSE DATABASES

Databases	Type	Minsup	Apriori	Eclat	FP-growth
Chess	Dense	20%	1924	<u>77</u>	89
Connect	Dense	30%	522	<u>366</u>	403
Retail	Sparse	0.003%	18	59	<u>10</u>
Kosarak	Sparse	0.08%	4332	385	<u>144</u>

Contributions: Most databases consist of both dense and sparse data portions that can only be detected during the mining process. Applying single mining strategy for ARM will omit this feature and result in unstable performance on different data types. In this paper, we present a novel high performance approach for ARM that can self-adapt to data characteristics. The main contributions of our study include:

1) The recognition of various characteristics of databases and the fact that this characteristics may change during the mining process is an original idea. The new approach presented in this paper detects the data characteristics at various stages of the mining process, and selects one of the two mining algorithms suitable for each subset of the data remaining to be mined on the fly. Two algorithms FEM and DFEM derived from the proposed approach are discussed.
2) Effective optimization techniques are introduced for the implementation of our mining approach to further speed up the mining process, reduce the memory usage and the I/O cost.
3) The efficiency of our approach is demonstrated in both execution time and memory usage via the benchmark of our algorithms (FEM and DFEM) with six other ARM algorithms including Apriori [1], Eclat [5], FP-growth [6], FP-growth* [7], FP-array [12], AIM2 [10]. We also analyze the reasons for the performance merit of our approach.

II. BACKGROUND

A. Problem Statement

The association rule mining problem can be stated as follows: Let $I = \{i_1, i_2,\ldots, i_n\}$ be the set of n distinct items in the transactional database *D*. Each transaction T in D contains a set of items called *itemset* and a *k-itemset* is an itemset with k items. The *count* of an *itemset x* is the number of occurrences of *x* in D and the *support* of *x* is the percentage of transactions containing *x*. Given a database *D*, the ARM problem is to find all strong association rules with the form: $X \rightarrow Y \mid X, Y \subset I$, and $X \cap Y = \emptyset$ whose *support* and *confidence* satisfy two user-specified inputs a minimum support threshold (*minsup*) and a minimum confidence threshold (*minconf*). The *confidence* of a rule is defined as the percentage of transactions in D that contain X also contain Y and is computed as *confidence*$(X \rightarrow Y)$ = *support*$(X\cup Y)$/*support*(X). The ARM task involves two separate steps: (1) mining all frequent patterns (or frequent itemsets) from the original database and (2) generating rules from these frequent patterns. An *itemset* α is a frequent pattern if α*'s support* is no fewer than *minsup*.

TABLE II. SAMPLE DATASET WITH *MINSUP* = 20%

Transaction ID (TID)	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e	a,b,c,e

For example, given the database in Table II and *minsup*=20%, the frequent 1-itemsets include *a, b, c, d* and *e* while *f* is infrequent because the *support* of f is only 11%. Similarly, *ab, ac, ad, ae, bc, bd, cd, ce, de* are frequent 2-itemsets and *abc, abd, ace, ade* are the frequent 3-itemset. If *minconf* = 80%, some of all association rules include $b\rightarrow a$, $c\rightarrow a$, $d\rightarrow a$, $e\rightarrow a$, and $c\rightarrow b$ because their *confidence*s are larger or equal to 80%. In this paper, we use the terms pattern and itemset; database and dataset interchangeably.

B. Association Rule Mining Approaches

Most current approaches [1, 5, 6, 7, 8, 9, 10, 11] for finding association rules utilize the property that a *k-itemset* is frequent only if its *sub-itemset*s are frequent to significantly reduce the search space of frequent *itemsets*. First, the database D is scanned to specify all frequent items (or *1-itemsets*) in D based on the *minsup* value. After this step, only data of frequent items (e.g. the third column in Table II) are used to determine the frequent *itemsets* as well as to generate the association rules. This considerably reduces the memory usage and computation by avoiding a large amount of infrequent data from loading into memory. In next steps, the frequent *(k+1)-itemsets*, initially with *k=1*, are discovered using frequent *k-itemsets* X of the previous step. For this purpose, the datasets D_X which are subsets of D and contain frequent items Y co-occurring with X ($X \cap Y = \emptyset$) are retrieved and used to determine the frequency of *(k+1)-itemsets*. Depending on the mining methods being applied, D_X can be presented in memory in many different data structures such as TID-list [5], Bitmap Vectors [3], FP-tree [6], FP-array [12], etc. or even be obtained by re-scanning the original database D from disks as in the Apriori method [1]. The characteristics of these data structures and the behaviors of their mining methods are very different which result in their different performance for a given database. For example, algorithms like Apriori [1], FP-growth [6], H-mine [8], nonordfp [9] and those making use of FP-array data structure [12] exploit horizontal format of data and perform efficiently on sparse databases (e.g. web document data or retail data) while Eclat [5], Mafia [3], AIM2 [10] present data in vertical format and run faster on the dense ones (e.g. biological sequence data). These mining methods perform unstably on different data types as demonstrated in Table I. Furthermore, the characteristics of data subsets D_X used to mine *(k+1)-itemsets* can change from very sparse to very dense as the mining task proceeds. Hence, applying a suitable mining strategy for each D_X is essential to improve the performance of ARM.

III. A NEW DYNAMIC APPROACH FOR ASSOCIATION RULE MINING

We focus on solving the first stage of ARM, i.e., discovering all frequent patterns in a database because this task is computationally intensive and constitutes the majority of work complexity while the second stage of generating association rules is trivial in comparison.

A. Data Structures

ARM is a memory intensive task whose data presentation and manipulation have a huge impact on mining performance. In our mining approach, we apply two main data structures including FP-tree and Bit Vector for the mining task.

FP-tree is a prefix tree that compacts all sets of ordered frequent items from database into memory. This tree consists of a header table storing the frequent items with their *count*, a root node and a set of prefix sub-trees. Each node of the tree includes an *item name*, a *count* indicating the number of transactions that contain all items in the path from the root node to the current node, and a *link* to its parent node. Each linked list starting from the header table links all nodes of the same frequent item. If two itemsets share a common prefix, the shared part can be merged as long as the *count* properly reflects the frequency of each itemset in the database. Fig. 1 illustrates an FP-tree constructed from the dataset in Table II where a pair <x:y> indicates *item name* and its *count*.

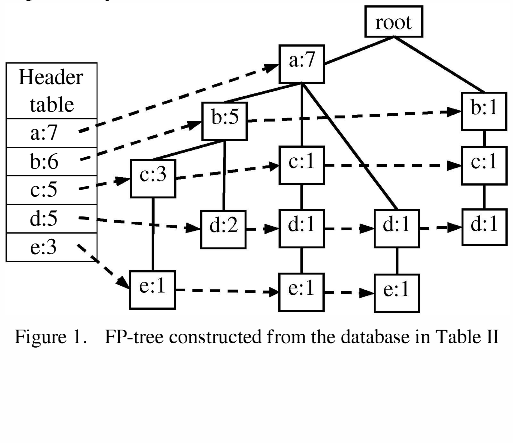


Figure 1. FP-tree constructed from the database in Table II

Bit Vector is used to store data in memory using the vertical format. This data structure includes *item name*, *count* and *vector of binary bits* associated with an item or an itemset. The i^th bit of this vector indicates if the i^th transaction in the database contains that item or itemset (1: exist, 0: does not exist). For example, the dataset in Table II can be presented in five bit vectors as in Fig. 2. The bit vector of the item f is removed because this item is infrequent. This structure does not only save memory but also enables low-cost bitwise operations for computations.

TID	Frequent Items
1	a,b,d
2	b,c,d
3	a,c,d,e
4	a,d,e
5	a,b,c
6	a,b,c
7	
8	a,b,d
9	a,b,c,e

→

Bit Vectors

a	b	c	d	e
1	1	0	1	0
0	1	1	1	0
1	0	1	1	1
1	0	0	1	1
1	1	1	0	0
1	1	1	0	0
0	0	0	0	0
1	1	0	1	0
1	1	1	0	1

Figure 2. Bit Vectors constructed from the dataset in in Table II

B. *The Proposed Approach for Association Rule Mining*

Studying many real databases and their characteristics, we observed that most consist of a group of items occurring much more frequently than the others. During the mining process, the items in this group create data subsets whose characteristic is dense while the less frequent items create subsets which are sparse. Our approach is combining two mining strategies: (1) the first one which is applied for sparse data subsets presents data as FP-tree and uses the divide and conquer approach to generate frequent patterns; (2) the second strategy which is used to mine the dense data portions stores data into Bit Vectors and performs ANDing bitwise operation on pairs of vectors to specify the frequent patterns. It has been shown that the first mining strategy works better on sparse data [6, 7, 9] and the second one is more suitable for dense one [5, 10, 11]. The proposed approach detects the characteristic of each data subset (not whole database) and applies a suitable one for this data dynamically. Based on this approach, we develop two new algorithms FEM and DFEM which are presented in Section IV and V. In general, our mining method includes three main subtasks as shown Fig. 3:

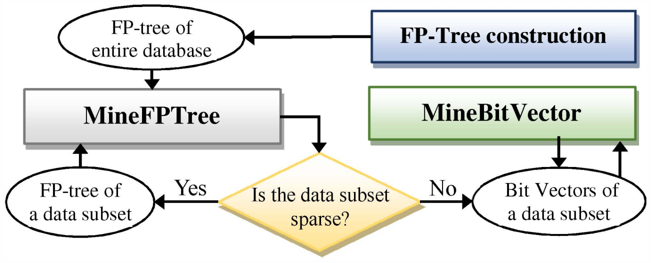

FP-tree of entire database
FP-Tree construction
MineFPTree
MineBitVector
FP-tree of a data subset
Yes
Is the data subset sparse?
No
Bit Vectors of a data subset

Figure 3. Mining model of the proposed approach.

FP-tree construction: Database is scanned for the first time to find the frequent items and create the header table. A second database scan is conducted to get frequent items of each transaction. Then, these items are sorted and inserted in the FP-tree in frequency descending order. During the top-down traversal of the tree construction, if a node presenting an item exists, its count will be incremented by one. Otherwise, a new node is added to the FP-tree.

MineFPTree generates frequent patterns by concatenating the suffix pattern of the previous step with each item α of the input FP-tree. Then, it constructs a child FP-tree called conditional FP-tree for every item α using a dataset called conditional pattern base, i.e., D_X as discussed in Section II. This dataset is extracted from the input FP-tree and consists of sets of frequent items co-occurring with the suffix pattern. The new tree is then used as the input of this recursive mining task. This mining approach explores data in the horizontal format and does not require generating a large number of candidate patterns. Hence, it performs well on sparse databases. However, unlike the related works [6, 7, 9] that perform mining on FP-tree only, *MineFPTree* can switch to the second mining strategy when it detects the current data subset is dense. In this case, the data subset is converted into bit vectors and *MineBitVector* is invoked. A weight vector *w* whose elements indicate the frequency of sets in the conditional pattern base is added as the input of *MineBitVector*.

MineBitVector generates frequent patterns by concatenating the suffix pattern with each item of the input bit vector. It then joins pairs of bit vectors using logical AND operation and computes their *support* using the weight vector to specify new frequent patterns. The resulting bit vectors are used as the input of *MineBitVector* to find longer frequent patterns. The mining process continues in a recursive manner until all frequent patterns are found. For dense data, this mining strategy is better than *MineFPtree* because the number of frequent patterns, which is usually found in dense data, make is suitable for the candidate generation and test approach of *MineBitVector*.

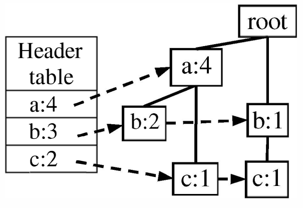
Fig. 4 illustrates an example. The conditional pattern base of item d, extracted from the FP-tree in Fig. 1, consists of the 4 sets {a:2,b:2}, {a:1, c:1}, {a:1} and {b:1, c:1} in which {a , b} occurs twice (Fig. 4-a). This base is equivalent to the dataset represented in Fig. 4-b. If *MineFPTree* is selected, the conditional FP-tree of item d is constructed as in Fig. 4-c. Otherwise, the bit vectors *a, b, c* and the weight vector w (Fig. 4-d and 4-e) are created instead to be used by *MineBitVector*.

{a:2,b:2}
{a:1, c:1}
{a:1}
{b:1, c:1}

(c) Conditional pattern base of item d

TID	Items	Frequency
1	a,b	2
2	a,c	1
3	a	1
4	b,c	1

(b) Dataset equivalent to the conditional base of item d



(c) Conditional FP-tree of item d

a	b	c
1	1	0
1	0	1
1	0	0
0	1	1

(d) Bit Vectors

w
2
1
1
1

(e) Weight vector

Figure 4. Illustration of FP-tree and Bit Vector construction

C. Switching Between Two Mining Strategies

Effective determination of how and when to switch between the two mining strategies is key in our approach to perform efficiently on different types of databases. During the mining process of *MineFPTree*, thousands or even millions of child FP-trees are constructed from the parent tree. A FP-tree is organized in such a way that the nodes of the most frequent items are closer to the top. The newly generated trees are much smaller than their parents because the less frequent items whose nodes are at bottom of the parent trees are removed. The size of a conditional pattern base which is used to construct a new FP-tree also reduces to a level where it contains mostly the most frequent items. In these cases, the conditional pattern base has the characteristic of a dense dataset. Therefore, only small conditional pattern bases are considered for transforming into bit vectors and weight vector. The size of a conditional pattern base is specified by the number of sets in that base which is similar to the number of transactions in a dataset. If this size is less than or equal to a threshold K, bit vectors and a weight vector are constructed and the mining switches to *MineBitVector*.

IV. THE FEM ALGORITHM

FEM uses the method described in Section III and includes three sub algorithms: *FEM* (Fig.5), *MineFPTree* (Fig. 6) and *MineBitVector* (Fig. 7). *FEM* first constructs the FP-tree from the database and then calls *MineFPTree* to start searching for frequent patterns and dynamically switch to *MineBitVector* if appropriate.

```
FEM algorithm
Input: Transactional database D and minsup
Output: Complete set of frequent patterns
1:  Scan D once to find all frequent items
2:  Scan D a second time to construct the FP-tree T
3:  K = 128
4:  Call MineFPTree(T,∅,minsup)
```

Figure 5. FEM algorithm

In FEM, a fixed value of threshold K is used to decide whether to apply *MineFPTree* or *MineBitVector* during the mining process. Our experimental results show that selecting a good value of K for FEM is data-specific and depends on the user-specified minimum support threshold (minsup). For FEM to perform well on many databases, we suggest to choose a value of K in the range of 0-256 based on our extensive experimental results on many real databases. In Fig.5, we select K=128 as a default value but K can be adjusted to obtain better performance for a specific database application. With K=128, the maximum size of a TID bit vector is 128 bits (16 bytes.) This is smaller than or equal to the size of just one node of FP-tree which needs at least 16 bytes for item name (4 bytes), count (4 bytes), a link to parent node (4 bytes) and a link to the next node of its linked list (4 bytes). The total memory size of all TID bit vectors is therefore not greater than the number of items in the conditional pattern base multiplied by 16 bytes. This data structure requires much less memory space than an equivalent conditional FP-tree does. Furthermore, the bitwise operations on TID bit vectors will perform faster than creating and manipulating FP-trees [13].

```
MineFPTree algorithm
Input: Conditional FP-Tree T, suffix, minsup
Output: Set of frequent patterns
1:  If T contains a single path P
2:  Then For each combination x of the items in T
3:          Output $\beta = x \cup suffix$
4:  Else For each item $\alpha$ in the header table of T
5:  {  Output $\beta = \alpha \cup suffix$
6:     Construct $\alpha$'s conditional pattern base C
7:     size = the number of nodes in the linked list of $\alpha$
8:     If size > K
9:     Then { Construct $\alpha$'s conditional FP-tree T'
10:           Call MineFPTree (T',$\beta$,minsup)}
11:    Else { Transform C into TID bit vectors V
12:                  and weight vector w
13:           Call MineBitVector (V,w,$\beta$,minsup) }
14: }
```

Figure 6. MineFPTree algorithm

```
MineBitVector algorithm
Input: Bit vectors V, weight vector w, suffix, minsup
Output: Set of frequent patterns
1:  Sort V in support-descending order of their items
2:  For each vector $v_i$ in V
3:  {  Output $\beta$ = item of $v_i \cup suffix$
4:     For each vector $v_j$ in V with $j < i$
5:     {  $u_j = v_i$ AND $v_j$
6:        $sup_j$ = support of $u_j$ computed using w
7:        If $sup_j \geq minsup$ Then add $u_j$ into U
8:     }
9:     If all $u_j$ in U are identical to $v_i$
10:    Then For each combination x of the items in U
11:            Output $\beta' = x \cup \beta$
12:    Else If U is not empty
13:    Then Call MineBitVector(U,w,$\beta$,minsup)
14: }
```

Figure 7. MineBitVector algorithm

V. THE DFEM ALGORITHM

DFEM is a major improvement of FEM. Unlike FEM, it automatically finds the dynamic value of K at runtime which helps DFEM adapt better to the data characteristic [14].

A. Computing Dynamic Value of K

FEM perform well for many databases using a value of *K=128*. However, in some cases, the best performance is not reached with this fixed selection. The second column in Table III shows the runtime of FEM for Kosarak dataset with different values of *K* and *minsup*=0.07%. As can be seen, for *K=224*, the runtime of FEM is 871 seconds, significantly faster than its runtime of 1206 seconds for *K=128*. This execution time difference becomes significantly larger when the minimum support threshold (*minsup*) is set to lower levels as required by many applications such as query recommendation for web search engine [4]. In this case, it is important to find the best possible value of K dynamically as the program runs on a specific database with the required *minsups* to gain near-optimal performance.

TABLE III. MEASUREMENTS OF FEM FOR KOSARAK (MINSUP=0.07%)

Thres. K_i	Runtime (second)	# patterns by the *MineFPTree* task (P_i)	Ratio R_i
$K_0 = 0$	3341	2776266097	N/A
$K_1 = 32$	2939	1316339679	2.1
$K_2 = 64$	2146	206479285	6.4
$K_3 = 96$	1664	26795140	7.7
$K_4 = 128$	1206	2413815	11.1
$K_5 = 160$	1005	407051	5.9
$K_6 = 192$	934	86575	4.7
$K_7 =$ **224**	**871**	**63876**	**1.4**
$K_8 = 256$	870	58304	1.1

In Table III, when K increases, the number of frequent patterns found solely by the *MineFPTree* task reduces because more mining workload is shifted to the *MineBitVector* task. Let $\{K_0, K_1, \ldots K_n\}$ be the set of all values of K where $K_i = K_{i-1} + 32$; P_i is the number of frequent patterns generated by the *MineFPTree* task when K_i is applied; and R_i is the ratio indicating the difference between P_i and P_{i-1}. R_i is computed as:

$$R_i = P_{i-1}/P_i \quad , i = 1 \ldots n \qquad (1)$$

Our intensive empirical study indicates that good mining performance is achieved with K_i that satisfies: $R_i < 2 \ni (\nexists R_j \geq 2, \forall j > i)$. In other words, FEM will perform best (near optimal) at the smallest K_i where increasing K does not result in a sharp drop in the number of frequent patterns found by the *MineFPTree* task. In the example in Table III, the K_i that satisfies this condition is 224 and its runtime is 871 seconds. While this result is promising, the challenge is that this K_i can only be specified when the mining process completes and all P_i and R_i have been computed. We have developed a practical method to predict a value of K that is close to or equal to the best K_i. The predicted value is based on all P_i's estimated dynamically at runtime as described in *UpdateK* algorithm in Fig. 8. This algorithm requires less computational need in comparison to the one we presented in [14].

```
UpdateK algorithm
Input: NewPatterns and Size
Output:  updated value of threshold K
1:  newK = 0
2:  P[0] = P[0] + NewPatterns
3:  For i = 1 to N – 1 step 1
4:  {  If  Size > i*Step
5:     {  P[i] = P[i] + NewPatterns
6:          If  P[i-1] >= 2* P[i]  Then  newK = (i+1)*Step
7:     } Else Exit Loop
8:  }
9:  i = K/Step - 1
10: If  (i>0 AND P[i-1]  < 2*P[i] ) Then  K = 0
12: If  newK  > K Then K = newK
```

Figure. 8 UpdateK algorithm

B. Algorithmic Description

DFEM uses *UpdateK* algorithm (Fig. 8) to dynamically select the value of K at runtime and using it to adapt its mining behaviors to the characteristics of processed data better than FEM. DFEM consists of four sub algorithms: *DFEM* (Fig. 9), *UpdateK* (Fig. 8), *MineFPTree** (Fig. 10) and *MineBitVector* (Fig. 6). *MineBitVector* of DFEM is similar to the one of FEM, shown in Fig. 6.

DFEM algorithm builds the FP-tree, initializes the variables used by *UpdateK* and invokes the *MineFPTree**. The variables in Lines 3-6 must be declared in a scope that *UpdateK* can access and update.

```
DFEM algorithm
Input: Transactional database D and minsup
Output: Complete set of frequent patterns
1:   Scan D once to find all frequent items
2:   Scan D a second time to construct the FP-tree T
3:   N = 9
4:   Step = 32
5:   K = 0
6:   Create P[N] and set all elements to zero
7:   items = the number of frequent items in D
8:   Call UpdateK(items, N*Step)
9:   Call MineFPTree* (T,∅,minsup)
```

Figure. 9 DFEM algorithm

MineFPTree algorithm*: This algorithm is similar to *MineFPTree* with the exception of the extra steps needed to regularly update K (Line 4-5 and Line 11-13).

```
MineFPTree* algorithm
Input: Conditional FP-Tree T, suffix, minsup
Output: Set of frequent patterns
1:  If FP-tree T contains a single path P
2:  {   For each combination x of the items in P
3:        {  Output β = x ∪ suffix  }
4:        n = the number of outputs β
5:        Call UpdateK (n,1)                        }
7:  Else
8:  {   For each item α in the header table of FP-tree T
9:     {  Output β = α ∪ suffix
10:          Construct α's conditional pattern base C
11:          n = the number of items in C
12:          size = the number of nodes in the linked list of α
13:          Call UpdateK (n,size)
14:          If  size > K    Then
15:          {  Construct α's conditional FP-tree T'
16:              Call MineFPTree*(T',β,minsup)  }
17:          Else
18:          {  Transform C into TID bit vectors V
19:                                 and weight vector w
20:              Call MineBitVector(V,w,β,minsup) }
21: } }
```

Figure. 10 MineFPTree* algorithm

VI. OPTIMIZATION TECHNIQUES

In addition to the mining strategies and their data structures, the architecture of the machine on which a frequent pattern mining program runs also has a significant impact on mining time of an ARM task. In this section, we present implementation techniques for our mining approach to optimize the use of cache, memory and I/O and reduce the mining time.

FP-tree construction: In the second database scan, FEM and DFEM pre-load the frequency descending sorted sets of frequent items into a lexicographically sorted list. One copy of similar transactions is kept with its *count*. For very large databases, the transaction list size is set at runtime to fit the available memory. We organize this list in a binary tree and maintain its order while the list grows in size. When its size limit is reached, the sets of frequent items and their *count*s are extracted from the list one by one to build the FP-tree. Therefore, the construction time of FP-tree is significantly reduced because similar itemsets are added into FP-tree only once. Moreover, the lexicographical order of the transaction list makes the FP-tree nodes most visited together to be allocated close together in memory optimizing the use of cache and speeding up the mining stage.

FP-tree mining task: We improve the technique proposed in [7] to implement an additional array associated with each FP-tree to pre-compute the *count* of new patterns. It helps to reduce the traversal cost of parent FP-trees when constructing the child FP-trees. The improvement of performance results from maximizing the locality of consistent memory access pattern. However, for the trees with a large number of frequent items, the array size will be very large which consequently consume a large amount of memory and increases the runtime. Therefore, we only enable this technique in FEM and DFEM whenever the array size does not go beyond a predefined limit; current default value is 64KB.

Memory management*:* For better memory utilization, large chunks of memory are allocated to store data of all FP-trees and bit vectors which is similar to the technique used in [7]. When all frequent patterns from a FP-tree or bit vectors and their child FP-trees or bit vectors have been found, the storage for these data structures are discarded. The chunk size is variable. This technique minimizes the overhead of allocating and freeing small pieces of data and prevents data scattered in memory.

Output processing: The most frequent output values are preprocessed and stored in an indexed table as proposed in [10]. In addition, the similar part of two frequent itemsets outputted consecutively is processed only once. This technique considerably reduces the computational time on output reporting, especially when the output size is large.

I/O optimization: Data are read into a buffer before being parsed into transactions. Similarly, the outputs are buffered and only written when the buffer is full. This technique reduces much of the I/O overhead.

VII. EXPERIMENTS AND EVALUATION

We evaluate the efficiency of our approach by benchmarking the two algorithms FEM and DFEM with six other state-of-the-art ARM on both sparse and dense real datasets.

A. *Experimental Setup*

Datasets: Eight real datasets with various characteristics and domains were selected from the Frequent Itemset Mining Implementations Repository [15]. They include four dense, three sparse and one moderate datasets (Table IV).

TABLE IV. DATASETS AND THEIR PROPERTIES

Datasets	Type	# Items	Avg. length	# Trans.
Chess	Dense	76	37	3196
Connect	Dense	129	43	67557
Mushroom	Dense	119	23	8124
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: We benchmarked FEM, DFEM and six state-of-the-art ARM algorithms: Apriori [1], Elcat [5], FP-growth [6], FP-growth* [7], FP-array [12], AIM2 [10]. FEM and DFEM are implemented using our proposed method and the optimization techniques introduced in this paper. Source codes of compared methods can be found at [15][16].

Hardware: Eight algorithms were tested on an Altus 1702 machine with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. Its running operating system is CentOS 5.3, a Linux-based distribution.

B. *Time Comparison*

The execution time of eight algorithms on eight datasets with various *minsup* are presented in Fig. 11. The experimental results show that FEM and DFEM run stably and outperform the others in almost all cases, while the other algorithms behave differently for different datasets. Apriori runs slowest on eight datasets but it does better than FP-growth* and FP-array for two dense datasets, Chess and Mushroom. For Retail the sparse dataset, Apriori has longer execution time compared to FEM, DFEM and FP-growth but run faster than the others. Eclat performs better than the others except AIM2, FEM and DFEM on the dense datasets. However, for the sparse datasets such as Retail and Kosarak, Eclat runs slower than most of the others. Compared to Eclat, three algorithms FP-growth, FP-growth* and FP-array run faster for the dense datasets but slower for the sparse ones. AIM2, a variant of Eclat, performs well for some dense and sparse datasets but worse for the other ones.

Based on the execution time in this experiment, we found that FEM and DFEM run faster than Apriori - the most popular used ARM method- from 3.4 to 5555.6 times. In comparison to Eclat and AIM2 whose mining approach use vertical data format, our algorithms run faster from 1.02 to 45.3 times. Our algorithms performed 1.2 to 23.4 times better than FP-growth, FP-growth* and FP-array which are among the best methods

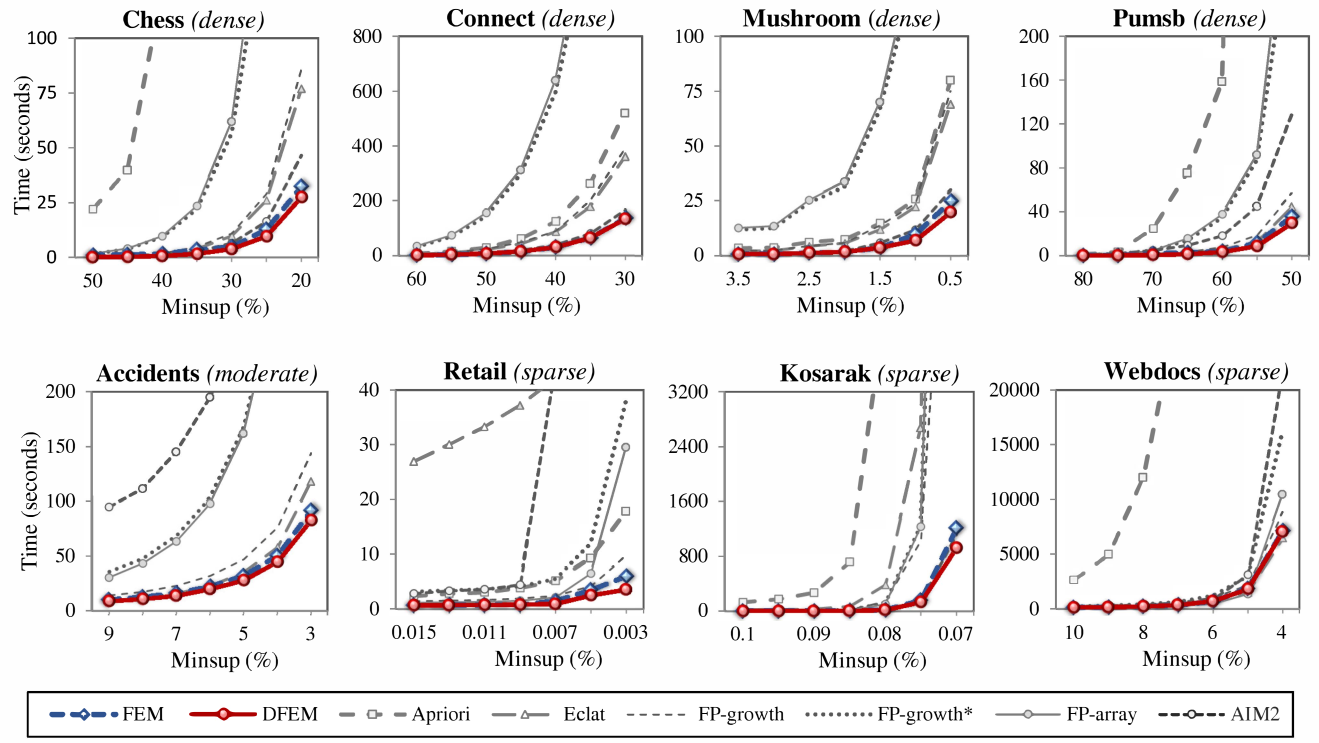


Figure. 11 Execution time of FEM, DFEM and other algorithms

for ARM. This experiment demonstrates the efficiency performance of FEM and DFEM for both sparse and dense data.

C. Memory Usage Comparison

In order to evaluate the memory usage efficiency of FEM and DFEM, we measured their peak memory usage in comparison to the other six algorithms for the eight datasets by using the *memusage* command of Linux. Table V shows the memory usage (megabytes) of all algorithms for the test cases with low minimum supports so that the result can reflect the large difference in memory usage among the algorithms. As in Table V, FEM and DFEM consume much less memory than Apriori in every case. Their memory requirements are closer to the average memory usage of Eclat and FP-growth in most cases. For the Accidents and Connect dataset, our algorithms use less memory than both Eclat and FP-growth. For the Chess dataset, FEM and DFEM need more memory because our implementation includes some additional buffers to enhance the performance. However, these buffers have fixed size and do not require much memory. Compared to FP-growth*, FEM and DFEM require more memory for the dense datasets but less memory for the sparse ones. In contrast, compared with FP-array, the memory usage of FEM and DFEM is smaller for the dense datasets but larger for the sparse ones. The memory usage of AIM2 is smallest in most cases. However, the memory usage of AIM2 for Webdocs, where memory optimization is critical due to its large memory requirements, AIM2 uses a significantly lager memory than the others do.

To sum up, the two experiments show that FEM and DFEM not only significantly improve the mining performance and outperform other existing "efficient" algorithms for both sparse and dense datasets they also compare well in memory requirements. Their memory consumption is much less than Apriori and FP-growth and is on average on par with the other algorithms. These results demonstrate the efficiency and efficacy of our algorithms. DFEM performs better than FEM, especially when *minsup* is low. Therefore, for mining application that requires low *minsup*, DFEM is a better choice.

D. Performance Impact of Our Two Mining Strategies

To study the performance merit of our mining approach, we measured the mining time of DFEM in three separated cases: (1) using *MineFPTree** only, (2) using *MineBitVector* only and

TABLE V. PEAK MEMORY USAGE (MEGABYTES) OF FEM, DFEM AND OTHER ALGORITHMS

Datasets	Minsup	FEM	DFEM	Apriori	Eclat	FP-Growth	FP-Growth*	FP-array	AIM2
Chess	20%	**4**	**4**	1139	2	3	3	33	1
Connect	30%	**11**	**11**	31	13	16	2	43	3
Mushroom	0.5%	**4**	**4**	20	3	5	2	33	1
Pumsb	50%	**15**	**15**	921	15	15	6	46	10
Accidents	3%	**181**	**181**	368	232	305	198	154	40
Retail	0.003%	**30**	**30**	1203	25	33	350	59	32
Kosarak	0.07%	**141**	**141**	16406	138	154	160	133	130
Webdocs	4%	**4707**	**4707**	24576	3996	5103	5581	4256	7544

(3) using both *MineFPTree** and *MineBitVector*, i.e., our approach. The results for DFEM on both dense and sparse data (Fig. 12) show that it outperforms the other methods where single mining strategy was used. This is explained by the contribution of dynamic combination of the two strategies, *MineFPTree and* MineBitVector to the mining task where each handles data portions it can perform best.

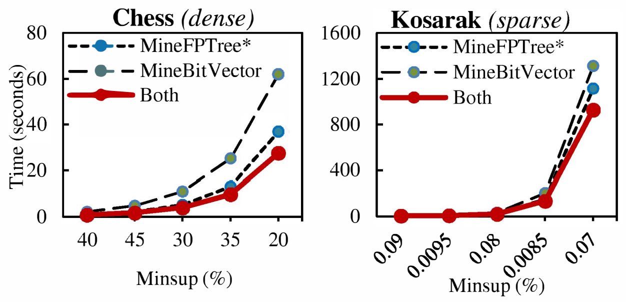


Figure 12. Execution time of using single mining strategy vs. using both

In order to understand the level that each mining strategy of our approach contributes to the overall performance, we measured *MineFPTree** and *MineBitVector* in DFEM separately; Figure 13 shows the time distributions of the two strategies. The results show that our approach automatically distributes the mining workload to its two mining strategies based on data characteristics. *MineBitVector* which is more suitable for dense data has been utilized mostly for the dense datasets like Chess, Connect, Mushroom and Pumsb. The time percentage of this strategy reduces when the data are sparse. For the sparse portions of the datasets, *MineFPTree** is a better choice because its mining approach does not require generating the very large number of infrequent candidate patterns. Hence, this strategy is used more for sparse datasets like Retail, Kosarak and Webdocs.

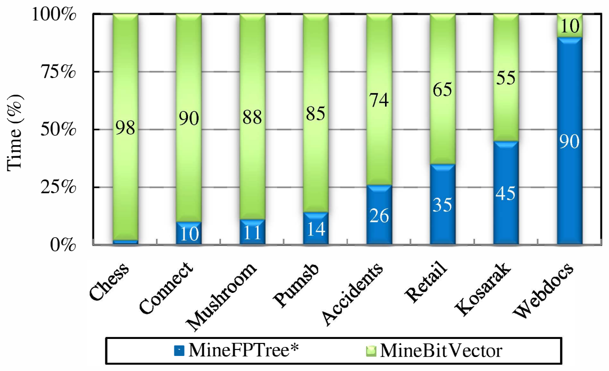


Figure 13. Time distribution of two mining strategies in DFEM

VIII. Conclusion

We have presented a new approach for association rule mining that works efficiently on both sparse and dense databases. Two algorithms FEM and DFEM derived from this approach are introduced and benchmarked with six well-known ARM algorithms. The experimental results on eight real datasets have shown that FEM and DFEM significantly improve mining time and consume much less memory than the compared methods.

References

[1] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th Int. Conf. on Very Large Databases*, pp. 487-499, 1994.

[2] J. Han, H. Cheng, D. Xin, X. Yan, "Frequent Pattern Mining: Current Status and Future Directions," in *J. Data Mining and Knowledge Discovery*, vol. 15, issue 1, pp. 55-86, Aug. 2007.

[3] D. Burdick et al., MAFIA: A Maximal Frequent Itemset Algorithm. In *IEEE Trans. on Knowledge and Data Engineering*. vol. 17, no. 11, pp. 1490–1504, Nov. 2005.

[4] H. Li, Y. Wang, D. Zhang, M. Zhang, E. Chang, PFP: Parallel FP-Growth for Query Recommendation. In *Proc. of the 2008 ACM Conf. on Recommender systems*, pp. 107-114, 2008.

[5] M. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules. In *Proc. of Knowledge Discovery and Data Mining*, pp. 283-286, 1997.

[6] J. Han, J. Pei, Y. Yin, Mining Frequent Patterns without Candidate Generation. In *Proc. of the 2000 ACM SIGMOD Int. Conf. on Mgt. of Data,* vol. 29, issue 2. pp. 1-12, Jun. 2000.

[7] G. Grahne, J. Zhu, Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. of the 2003 Workshop on Frequent Pattern Mining Implementations*, pp. 123–132, 2003.

[8] J. Pei et al. Hmine : Hyper-structure Mining of Frequent Patterns in Large Databases. In *Proc. of the IEEE Int. Conf. on Data Mining,* pp. 441–448, Nov. 2001.

[9] B. Racz, Nonordfp: An FP-growth Variation Without Rebuilding the FP-tree. In *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations,* Nov. 2004.

[10] S. Shporer, AIM2: Improved Implementation of AIM. In *Proc. of the IEEE Workshop on Frequent Itemset Mining Implementations,* Nov. 2004.

[11] L. Schmidt-Thieme, Algorithmic Features of Eclat. In *Proc. of the IEEE Workshop on Frequent Itemset Mining Implementations,* Nov. 2004.

[12] L. Liu, E. Li, Y. Zhang, Z. Tang, Optimization of Frequent Itemset Mining on Multiple-core Processor. In *Proc. of the 33rd Int. Conf. on Very Large Databases*, pp. 1275-1285, 2007.

[13] L. Vu, G. Alaghband, A Fast Algorithm Combining FP-tree and TID-List for Frequent Pattern Mining. In *Proc. of the 2011 Int. Conf. on Inf. and Knowledge Engineering,* 472-477, Jul. 2011.

[14] L. Vu, G. Alaghband, Mining Frequent Patterns Based on Data Characteristics. In *Proc. of the 2012 Int. Conf. on Information and Knowledge Engineering*, pp. 369-375, Jul. 2012.

[15] Frequent Itemset Mining Implementations Repository, *Workshop on Frequent Itemset Mining Implementation*, (2003-2004). Available at http://fimi.ua.ac.be

[16] C. Bienia, The PARSEC Benchmark Suite, *Princeton University Technical Report TR-811-08* (Jan. 2008). Available at http://parsec.cs.princeton.edu.