Lan Vu* and Gita Alaghband

Efficient Algorithms for Mining Frequent Patterns from Sparse and Dense Databases

Abstract: In this article, we present a new approach for frequent pattern mining (FPM) that runs fast for both sparse and dense databases. Two algorithms, FEM and DFEM, based on our approach are also introduced. FEM applies a fixed threshold as the condition for switching between the two mining strategies; meanwhile, 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 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 FPM algorithms including the well-known Apriori, FP-growth, and Eclat.

Keywords: Data mining, frequent pattern mining, association rule mining, frequent itemset, transactional database.

2010 Mathematics Subject Classification: 62-07, 68P99, 97R50.

DOI 10.1515/jisys-2014-0040
Received February 24, 2014

1 Introduction

Frequent pattern mining (FPM) is a fundamental task in data mining that is used to find many types of relationships among variables in large databases such as associations [1], correlations [6], causality [26], sequential patterns [2], episodes [19], and partial periodicity [12]. Moreover, it helps in data indexing, classification, clustering, and other data mining tasks as well [14]. Thus, FPM has become a focused research with numerous practical applications including consumer market-basket analysis, web mining, similarity search of complex structured data, network intrusion detection, and many others [8, 14].

The FPM problem aims at searching for groups of itemsets, subsequences, or substructures that co-occur in a database with their frequency no less than a user-specified minimum support threshold. For example, a set of items (itemset), such as milk and bread, that appear frequently together in a database is a frequent itemset or frequent pattern. In a typical transactional database, the number of distinct single items and their combinations are usually very large. For a small minimum support threshold, the number of generated itemsets can be extremely large. Hence, it is a great challenge to design algorithms for mining frequent patterns that scale with memory size and run in reasonable time [17, 30].

Many exiting methods for FPM have typically worked well for certain types of databases. Most methods performed efficiently on either sparse or dense databases but poorly on the other [3–5, 9, 11, 13, 22–24]. Table 1

***Corresponding author: Lan Vu,** Department of Computer Science and Engineering, University of Colorado Denver, 1380 Lawrence Street, Denver, CO 80204, USA, e-mail: lan.vu@ucdenver.edu
Gita Alaghband: Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO, USA

Table 1. Execution Time 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>

Execution time is in seconds. The best execution times among the three algorithms are underlined.

presents the execution time of three well-known algorithms Apriori [1], Eclat [34], and FP-growth [13] on sparse and dense databases and shows that Eclat performs best on dense data whereas FP-growth runs 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 like Oracle RDBMS, MS. SQL Server and IBM DBS2 and statistical software like R, SAS and SPSS Clementine which support data mining tasks [16, 28, 32] usually require mining methods that stably perform on various data types.

1.1 Contributions

Most databases consist of both sparse and dense data portions that can only be detected during the mining process. Applying single mining strategy for FPM will omit this feature and result in unstable performance on different data types. In this article, we present a novel approach for FPM and two algorithms, FEM and DFEM, based on this approach 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 article 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 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 FPM algorithms including Apriori [1], Eclat [34], FP-growth [13], FP-growth* [11], FP-array [18], AIM2 [25]. We also analyze the reasons for the performance merit of our approach.

2 Background

2.1 Problem Statement

The FPM problem can be stated as follows: let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all distinct items in the transactional database D. The *count* of an itemset x (a set of items) is the number of occurrences of x in D and the *support* of x is the percentage of transactions containing x. A k-itemset x, which consists of k items from I, is frequent if the *support* of x is at least minsup, where minsup is a user-specified minimum support threshold. Given a database D and a *minsup*, the problem statement is to find the complete set of frequent itemsets (or frequent patterns) in D. In this article, we use the terms “pattern” and “itemset” as well as “database” and “data set” interchangeably.

For example, given the data set in Table 2 and minsup = 20%, the frequent 1-itemsets include *a*, *b*, *c*, *d*, and *e*, whereas *f* is infrequent because its *support* is only 11%. Similarly, *ab*, *ac*, *ad*, *ae*, *bc*, *bd*, *cd*, *ce*, and *de* are frequent 2-itemsets, and *abc*, *abd*, *ace*, and *ade* are the frequent 3-itemset.

Table 2. Sample Data Set with Minsup = 20%.

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*,*c*	*a*,*d*,*c*
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*

FPM is an important part of many data mining tasks, especially in association rule mining (ARM) [3], a problem of finding all strong association rules with the form: $X \to Y \mid X, Y \subset I$ and $X \cap Y = \varnothing$, whose *support* and *confidence* satisfy a minimum support threshold (*minsup*) and a minimum confidence threshold (*minconf*). The *confidence* of a rule is the percentage of transactions in D that contain X also contain Y. An example of such a rule might be that 60% of customers who purchase flour also buy sugar with a confidence 80%. ARM consists of two steps. The first step is finding all frequent itemsets that are computationally intensive and is the focus of our research. The second step is generating strong association rules from the frequent itemsets.

2.2 Frequent Pattern Mining Approaches

Mining frequent patterns is nontrivial because of its exponential search space and its large amount of data. Many algorithms have been developed to do this mining on large databases, and they typically perform efficiently on either sparse or dense databases but not both [1, 3–5, 9, 11, 13, 15, 22–25, 34].

Apriori [1] was the first sequential algorithm that sharply cuts down the search space to make this mining task feasible. Some variants of Apriori include direct hashing and pruning (DHP) [21], sampling technique [27], dynamic itemset counting (DIC) [7], BitApriori [9], and Hybrid search-based ARM [10]. Because of the breath-first strategy with multiple I/O scanning times and candidate generate-and-test approach, Apriori-like methods usually suffer from large computational time and memory usage.

Eclat-like algorithms load all data of the frequent items into memory, present them in vertical format, and apply the depth-first strategy to search for all frequent patterns. These algorithms run significantly faster than Apriori-like ones because they require at most two database scans and reduce memory usage sharply. Eclat and their variants have been shown to be the better choice for long patterns and/or dense databases [9, 24, 25, 33].

FP-growth is known as one of the most efficient FPM algorithms [13]. It compresses the data of the frequent items into an FP-tree in memory and recursively mines all frequent patterns from this data structure without candidate generation requirement. Because of its efficiency, this algorithm is used in large-scale applications such as Google's query recommendation [17]. FP-growth* [11], H-mine [22], nonordfp [23], FP-growth with database partition projection [20] are some improvements of FP-growth. Although FP-growth has shown to outperform previously developed methods including Eclat and Apriori [4, 11, 13, 22, 23], it has been shown not to perform as well as Eclat-like algorithms [9, 24, 25] for dense databases or mining with low minimum support.

Recently, a new algorithm named FP-array has been proposed [18]. This algorithm inherits the mining features of FP-growth and presents data as arrays for cache optimization. FP-array was shown to achieve a significant performance improvement compared with FP-growth [13], nonordfp [23], and AIM2 [25].

3 A New Self-Adaptive Approach for Frequent Pattern Mining

This section lays the groundwork for our algorithms. We present our observations and analysis of the characteristics of various databases with respect to frequent patterns to motivate our new approach. We present a high-level description of our method, the relevant data structures used, and the data transformation needed to switch between the two mining tasks of our approach.

3.1 Data Structures

FPM 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 subtrees. 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. Figure 1 illustrates an FP-tree constructed from the data set in Table 2, where a pair <*x*:*y*> indicates item name and its *count*.

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 data set in Table 2 can be presented in five bit vectors as in Figure 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.

3.2 The Proposed Approach

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, whereas the less frequent items create subsets that 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

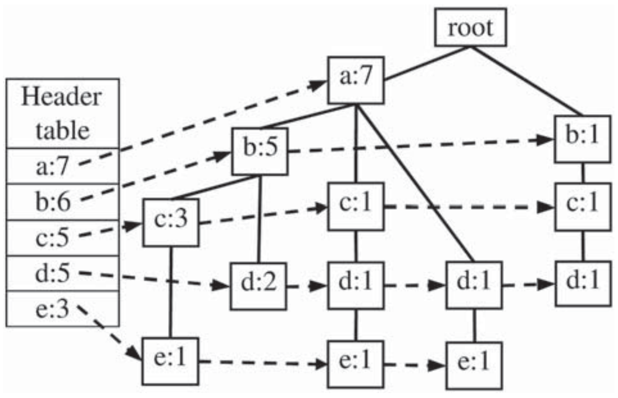
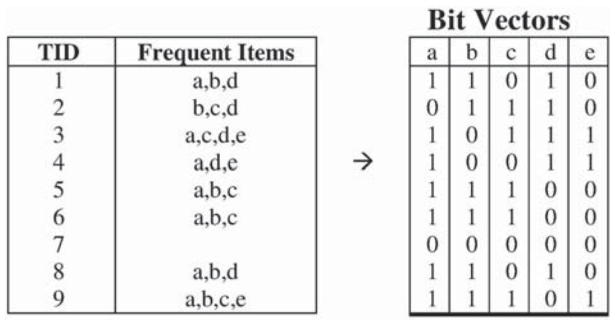


Figure 1. FP-Tree Constructed from the Database in Table 2.



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 Data Set in in Table 2.

pairs of vectors to specify the frequent patterns. It has been shown that the first mining strategy works better on sparse data [11, 13, 22, 23] and the second one is more suitable for dense one [9, 24, 25, 34]. The proposed approach detects the characteristic of each data subset (not whole database) and applies a suitable one for these data dynamically. Based on this approach, we develop two new algorithms, FEM and DFEM, which are presented in Sections 4 and 5. In general, our mining method includes three main subtasks as shown in Figure 3:

FP-tree construction: the 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 x of the input FP-tree. Then, it constructs a child FP-tree called conditional FP-tree for every item x using a data set called conditional pattern base. This data set 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 [11, 13, 22, 23] 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

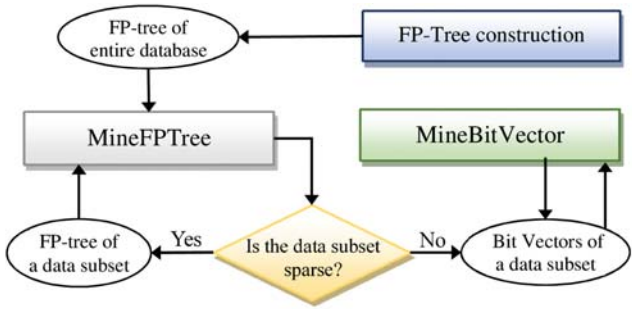
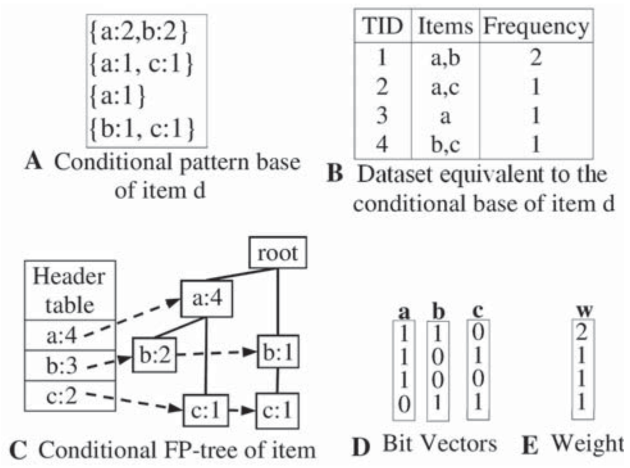

FP-tree of entire database
FP-Tree construction
MineFPTree
MineBitVector
FP-tree of a data subset
Yes
Is the data subset sparse?
No
Bit Vectors of a data subset

Figure 3. Mining Model of the Proposed Approach.


{a:2,b:2}
{a:1, c:1}
{a:1}
{b:1, c:1}
A Conditional pattern base of item d
B Dataset equivalent to the conditional base of item d
root
Header table
a:4
b:3
c:2
a:4
b:2
b:1
c:1
c:1
C Conditional FP-tree of item
a b c
w
D Bit Vectors
E Weight

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

a	b	c	w
1	1	0	2
1	0	1	1
1	0	0	1
0	1	1	1

Figure 4. Illustration of FP-Tree and Bit Vector Construction. A: Conditional pattern base of item d; B: Dataset equivalent to the conditional base of item d; C: Conditional FP-tree of item d; D: Bit Vectors; E: Weight vector.

number of frequent patterns, which is usually found in dense data, make is suitable for the candidate generation and test approach of *MineBitVector*.

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

3.3 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*, a very large number 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 data set. 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 that is similar to the number of transactions in a data set. 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*.

4 The FEM Algorithm

4.1 Algorithmic Descriptions

FEM uses the method described in Section 3 and includes three subalgorithms: FEM (Figure 5), MineFPTree (Figure 6), and MineBitVector (Figure 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. In FEM, a fixed value of threshold K is used to decide whether MineFPTree or MineBitVector is applied during

```
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($T$,$\varnothing$,$minsup$)
```

Figure 5. FEM Algorithm.

```
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.

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 $K = 128$ (line 3). K can be adjusted to obtain better performance for a specific database application. An in-depth analysis on a range of values for K is presented in Section 4.2.

4.2 Selection of Threshold *K*

We further investigate the impact of varying values of K of different databases in this section. Selecting a good value of K also depends on the minimum support specified by the user. This raises the question of how to select a good value of K without testing all possible values? To answer this question, we conducted an experiment from which we suggest a good default value of K for FEM. In this experiment, we measure the performance of FEM on eight real data sets for varying values of K. The eight data sets selected for these test cases consist of a mix of four dense, three sparse, and one moderate to represent a variety of database characteristics. The detailed characteristics of these data sets are presented in Table 4. Values of K were selected in the range of 0–256 as multiples of 32 so that the maximum size of transaction ID (TID) bit vectors are also multiples of 32 (4 bytes) for better memory utilization. The running time of FEM for values of K for each data set are presented in Figure 8.

As the results show, FEM performs better with K in the 32–128 range for seven of the data sets. For the Kosarak data set, FEM with a value of K in the 128–256 range performs best. This result reflects the excecution time with the selected minsup values. Based on extensive tests not presented here, we observe that when *minsup* varies, the range of K for each data set to produce best performance also changes, but it remains within the overall range of 0–256. For this reason, we recommend $K = 128$ as a default value for good performance on various databases and different minsup values. For $K = 128$, the maximum size of a 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 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 Bit Vector will perform faster than creating and manipulating FP-trees.

5 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 [29, 31].

5.1 Computing Dynamic Value of *K*

FEM performs 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 3 shows the running time of FEM for Kosarak data set with different values of K and *minsup* = 0.07%. As can be seen, for $K = 224$, the running time of FEM is 871 s, significantly faster than its running time of 1206 s 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 [17]. 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.

In Table 3, 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 $R_i = P_{i-1}/P_i$, $i = 1, \ldots, n$.

Our intensive empirical study indicates that good mining performance is achieved with K_i that satisfies: $R_i < 2 \ni (\nexists R_j < 2, \forall j > i)$. 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*

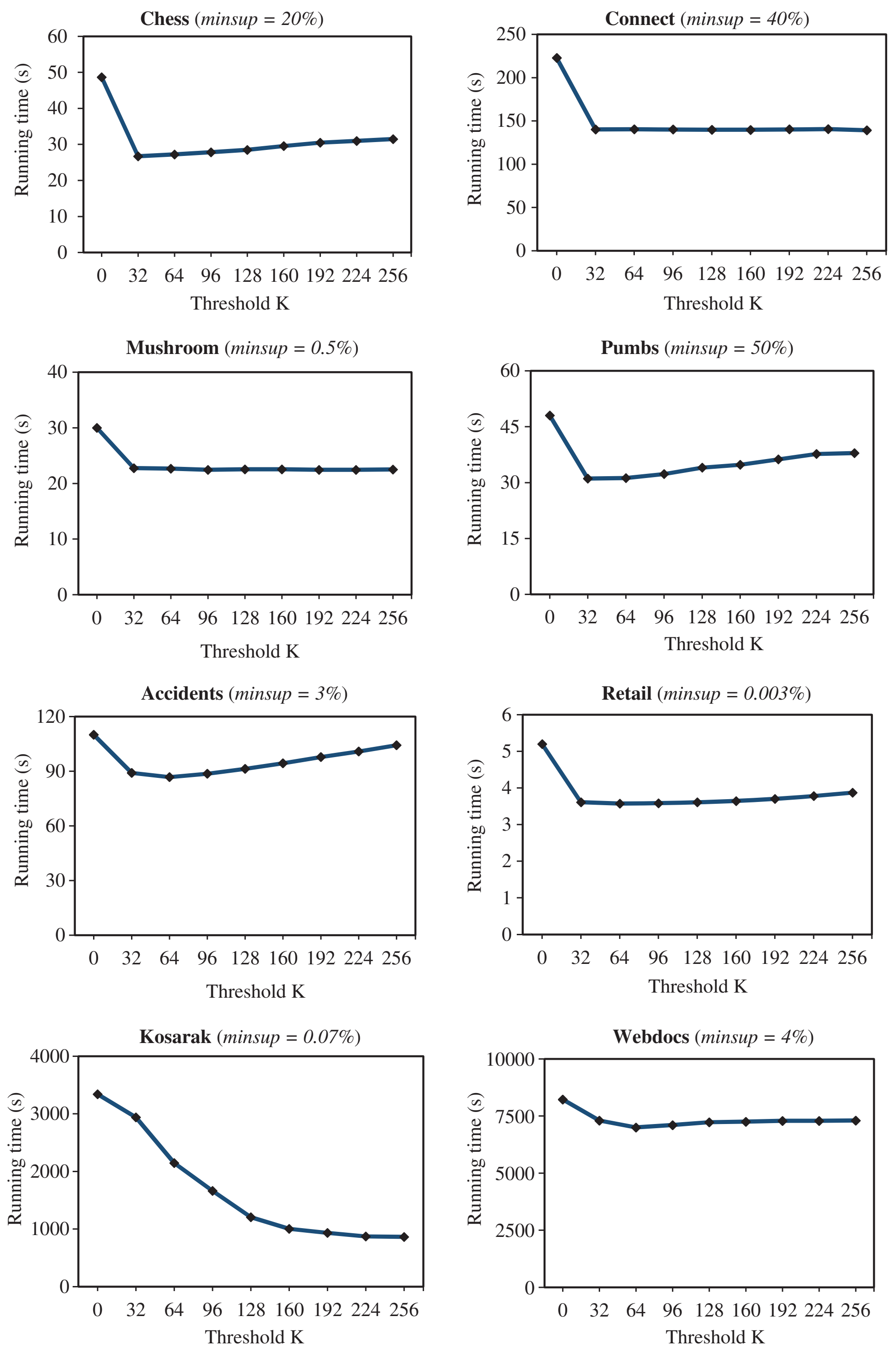


Figure 8. Running Time of FEM with Different Values of *K*.

task. In the example in Table 3, the K_i that satisfies this condition is 224, and its runtime is 871 s. Although 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 Figure 9. In this algorithm, K is the global threshold initialized to zero,

Table 3. Measurements of FEM for Kosarak (Minsup = 0.07%).

Threshold K_i	Runtime (s)	No. patterns by the MineFPTree task (P_i)	Ratio R_i
$K_0 = 0$	3341	2,776,266,097	N/A
$K_1 = 32$	2939	1,316,339,679	2.1
$K_2 = 64$	2146	206,479,285	6.4
$K_3 = 96$	1664	26,795,140	7.7
$K_4 = 128$	1206	2,413,815	11.1
$K_5 = 160$	1005	407,051	5.9
$K_6 = 192$	934	86,575	4.7
$K_7 =$ **224**	**871**	**63,876**	**1.4**
$K_8 = 256$	870	58,304	1.1

Bold text presents the good K value for FEM performing fast. N/A, Not available.

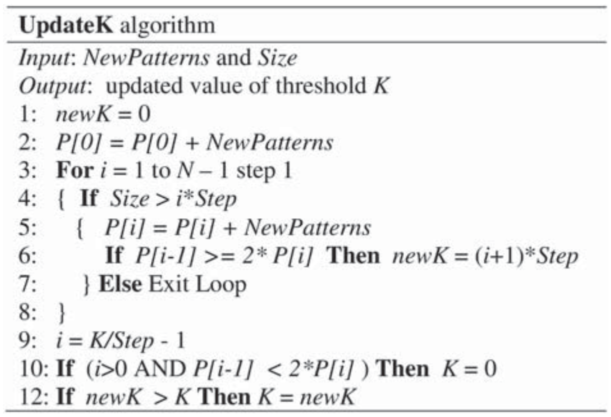
N is the number of different threshold values, K_i, that are being considered (default value *N*=9), *Step* is the distance between K_i and K_{i-1} (default value *Step*=32); and *P[N]* is an array whose the *i*th element stores the number of frequent patterns P_i estimated from all conditional pattern bases of the *MineFPTree* task. The elements of *P[N]* are initialized to zero and are regularly updated using the *UpdateK* algorithm everytime a new conditional pattern base is processed. *K* is then updated with a new value *newK* if *newK* satisfies the condition stated above. In Figure 9, *NewPatterns* indicates the number of new frequent patterns and is equal to the number of items in *C* (conditional pattern base); *Size* is the size of the *C*.

5.2 Algorithmic Description

DFEM uses UpdateK algorithm (Figure 9) 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 (Figure 10), UpdateK (Figure 9), MineFPTree* (Figure 11), and MineBitVector (Figure 6). MineBitVector of DFEM is similar to the one of FEM, shown in Figure 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.

The MineFPTree* algorithm is similar to MineFPTree with the exception of the extra steps needed to regularly update *K* (lines 4–5 and lines 11–13 in Figure 10).



```
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 9. UpdateK Algorithm.

```
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 10. DFEM Algorithm.

6 Optimization Techniques

In addition to the mining strategies and their data structures, the architecture of the machine on which a FPM program runs also has a significant impact on mining time of an FPM 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 preload 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 counts 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

```
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 $\beta = x \cup suffix$  }
4:        n = the number of outputs $\beta$
5:        Call UpdateK (n,1)                    }
7:  Else
8:  {   For each item $\alpha$ in the header table of FP-tree T
9:      {  Output $\beta = \alpha \cup suffix$
10:          Construct $\alpha$'s conditional pattern base C
11:          n = the number of items in C
12:          size = the number of nodes in the linked list of $\alpha$
13:          Call UpdateK (n,size)
14:          If  size > K   Then
15:          {  Construct $\alpha$'s conditional FP-tree T'
16:             Call MineFPTree*(T',$\beta$,minsup)   }
17:          Else
18:          {  Transform C into TID bit vectors V
19:                                 and weight vector w
20:             Call MineBitVector(V,w,$\beta$,minsup) }
21: } }
```

Figure 11. MineFPTree* Algorithm.

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.

MineFPTree Task: We improve the technique proposed in [11] to implement an additional array associated with each FP-tree to precompute 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 64 KB.

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 [11]. When all frequent patterns from an 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 [25]. 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.

7 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 FPM on both sparse and dense real data sets.

7.1 Experimental Setup

7.1.1 Data Sets

Eight real data sets with various characteristics and domains were selected from the Frequent Itemset Mining Implementations Repository [35]. They include four dense, three sparse, and one moderate data sets (Table 4).

7.1.2 Software

We benchmarked FEM, DFEM, and six state-of-the-art FPM algorithms: Apriori [1], Elcat [34], FP-growth [13], FP-growth* [11], FP-array [18], and AIM2 [25]. FEM and DFEM are implemented using our proposed method and the optimization techniques introduced in this article. Source codes of the compared methods can be found in [5, 35].

7.1.3 Hardware

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

Table 4. Data Sets and Their Properties.

Data sets	Type	No. items	Average length	No. transactions
Chess	Dense	76	37	3196
Connect	Dense	129	43	67,557
Mushroom	Dense	119	23	8124
Pumsb	Dense	2113	74	49,046
Accidents	Moderate	468	33.8	340,183
Retail	Sparse	16,470	10.3	88,126
Kosarak	Sparse	41,271	8.1	990,002
Webdocs	Sparse	52,676,657	177.2	1,623,346

7.2 Time Comparison

The execution time of eight algorithms on eight data sets with various minsup are presented in Figure 12. The experimental results show that FEM and DFEM run stably and outperform the others in almost all cases, whereas the other algorithms behave differently for different data sets. Apriori runs slowest on eight data sets, but it does better than FP-growth* and FP-array for two dense data sets, Chess and Mushroom. For Retail, the sparse data set, Apriori has longer execution time compared with FEM, DFEM, and FP-growth but runs faster than the others. Eclat performs better than the others except AIM2, FEM, and DFEM on the dense data sets. However, for the sparse data sets such as Retail and Kosarak, Eclat runs slower than most of the others. Compared with Eclat, three algorithms, FP-growth, FP-growth*, and FP-array, run faster for the dense data sets but slower for the sparse ones. AIM2, a variant of Eclat, performs well for some dense and sparse data sets 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 FPM 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

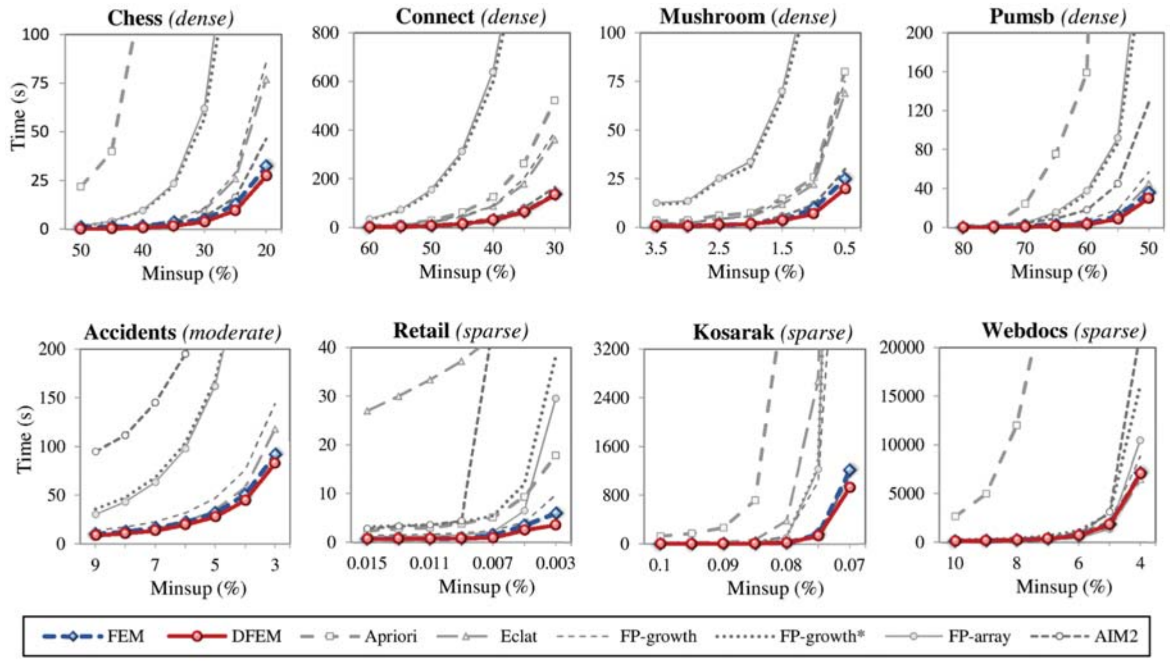


Figure 12. Execution Time of FEM, DFEM, and Other Algorithms.

1.2 to 23.4 times better than FP-growth, FP-growth*, and FP-array, which are among the best methods for FPM. This experiment demonstrates the efficiency performance of FEM and DFEM for both sparse and dense data.

7.3 Memory Usage Comparison

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 data sets using the memusage command of Linux. Table 5 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 5, 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 data set, our algorithms use less memory than both Eclat and FP-growth. For the Chess data set, 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 with FP-growth*, FEM and DFEM require more memory for the dense data sets 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 data sets 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 in Sections 7.2 and 7.3 show that FEM and DFEM not only significantly improve the mining performance and outperform other existing "efficient" algorithms for both sparse and dense data sets, they also compare well in memory requirements. Their memory consumption is much less than Apriori and FP-growth and is, on average, at 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.

7.4 Performance Impact of the 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 (3) using both MineFPTree* and MineBitVector, i.e., our approach. The results for DFEM on both dense and sparse data (Figure 13) 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.

Table 5. Peak Memory Usage of FEM, DFEM, and Other Algorithms.

Data sets	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**	16,406	138	154	160	133	130
Webdocs	**4**	**4707**	**4707**	24,576	3996	5103	5581	4256	7544

Peak memory usage is in megabytes. Bold text presents memory usage of FEM and DFEM, the proposed algorithms.

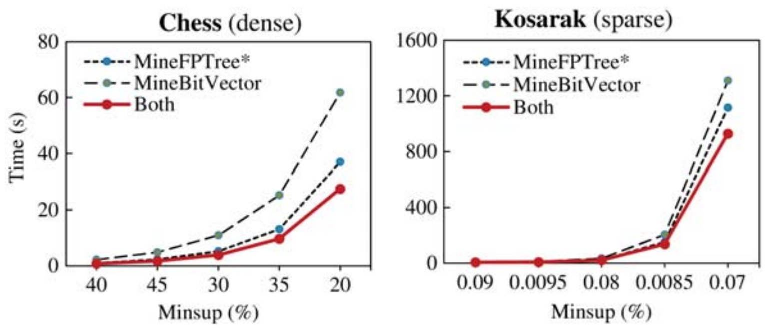


Figure 13. Execution Time of Using Single Mining Strategy vs. Using Both.

7.5 Analyzing the Self-adaptive Capability of Our Mining Approach to Data Characteristics

To understand the self-adaptive capability of our mining to data characteristics, we measured the time of MineBitVector and MineFPTree* in DFEM separately to observe the level that each mining strategy contributes to the overall performance for different databases and for various minsup input values.

Figure 14 shows the time distribution of MineBitVector and MineFPTree* for eight databases whose data characteristics range from very sparse to very dense. 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 data sets like Chess, Connect, Mushroom, and Pumsb (85–99% of total mining time). The time percentage of this strategy reduces when the data are sparse. MineFPTree* is increasingly used for sparse data sets like Retail, Kosarak, and Webdocs (33–90% of total mining time). For the sparse portions of the data sets, MineFPTree* is a better choice because its mining approach does not require generating the very large number of infrequent candidate patterns.

The ability to self-adapt to data characteristics shows not only for different databases but also for different minsup values for a given database. This is demonstrated by the time distribution of MineFPTree* and MineBitVector as minsup varies. Figure 15 presents the time distribution for Accident data set with various minsup values. The results show the lower minsup, the more mining workload and time were handled by MineBitVector. For the smaller values of minsup, a large number of frequent patterns are generated because more patterns satisfy the condition to be frequent. This situation is similar to mining on dense database. Our mining approach, therefore, automatically detects this feature and adjust its mining behavior to have more mining workload handled by MineBitVector and help improve the overall performance.

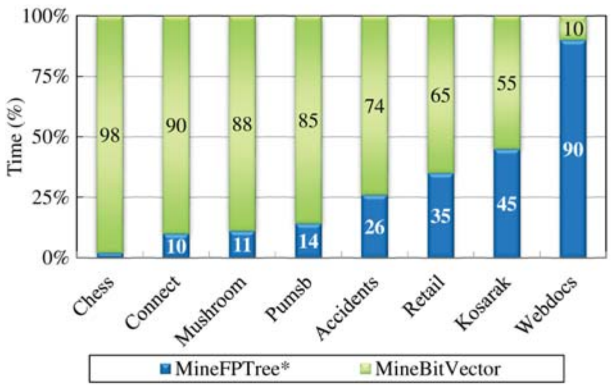


Figure 14. Time Distribution of Two Mining Strategies in DFEM.

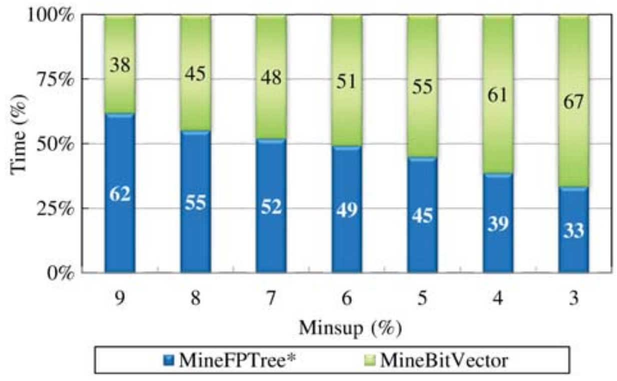


Figure 15. Time Distribution of Two Mining Strategies in DFEM for Accidents Dataset with Different minsup Input Values.

8 Conclusion

We have presented FEM and DFEM, two new FPM algorithms that work efficiently on both sparse and dense databases. DFEM is different from FEM, as it dynamically selects the value of threshold K used to switch between the two mining tasks. We have also introduced a combination of several optimization techniques for the implementation of FEM and DFEM to enhance their performance. The experimental results show that FEM and DFEM significantly improve the performance of mining frequent patterns and outperform other efficient algorithms for both sparse and dense databases. They consume much less memory than Apriori and FP-growth and, on average, are at par with the other memory-optimized algorithms. Comparing the two, DFEM runs faster than FEM in most test cases. In future work, we will study parallel approaches for implementing FEM and DFEM on parallel and distributed systems because the experiments show that memory limitation and computational time are the major obstacles to deploying any sequential FPM algorithm on very-large-scale databases.

Bibliography

[1] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in: *Proceedings of the 20th International Conference on Very Large Databases*, pp. 487–499, 1994.
[2] R. Agrawal and R. Srikant, Mining sequential patterns, in: *Proceedings of Data Engineering*, pp. 3–14, 1995.
[3] R. Agrawal, T. Imielinski and A. Swami, Mining association rules between sets of items in large databases, *Proc. ACM SIGMOD Manage. Data* **22** (1993), 207–216.
[4] C. Borgelt, An implementation of the FP-growth algorithm, in: *Proceedings of OSDM Frequent Pattern Mining Implementations*, August 2005.
[5] C. Borgelt, *Frequent pattern mining implementations*. http://www.borgelt.net.
[6] S. Brin, R. Motwani and C. Silverstein, Beyond market basket: generalizing association rules to correlations, *Proc. ACM SIGMOD Manage. Data* **26** (1997), 265–276.
[7] S. Brin, R. Motwani, J. D. Ullman and S. Tsur, Dynamic itemset counting and implication rules for market basket analysis, *Proc. ACM SIGMOD Manage. Data* **26** (1997), 255–264.
[8] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke and T. Yiu, MAFIA: a maximal frequent itemset algorithm, *IEEE Trans. Knowledge Data Eng.* **17** (2005), 1490–1504.
[9] A. Fiat and S. Shporer, AIM: another itemset miner, in: *Proceedings of Frequent Itemset Mining Implementations*, 2003.
[10] A. Ghanem and H. Sallam, Hybrid search based association rule mining, in: *Proceedings of the 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 608–612, 2011.
[11] G. Grahne and J. Zhu, Efficiently using prefix-trees in mining frequent itemsets, in: *Proceedings of Frequent Pattern Mining Implementations*, pp. 123–132, 2003.
[12] J. Han, G. Dong and Y. Yin, Efficient mining of partial periodic patterns in time series database, in: *Proceedings of the IEEE Data Engineering*, pp. 106–115, March 1999.

[13] J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation, *Proc. Manage. Data* **29** (2000), 1–12.

[14] J. Han, H. Cheng, D. Xin and X. Yan, Frequent pattern mining: current status and future directions, *J. Data Mining Knowledge Discov.* **15** (2007), 55–86.

[15] N. Le, T. Nguyen and T. C. Chung, BitApriori: an apriori-based frequent itemsets mining using bit streams, in: *Proceedings of Knowledge Discovery and Data Mining*, pp. 1–6, 2010.

[16] W. Li and A. Mozes, Computing frequent itemsets inside Oracle 10g, in: *Proceedings of the 30th International Conference on Very Large Databases*, pp. 1253–1256, 2004.

[17] H. Li, Y. Wang, D. Zhang, M. Zhang and E. Chang, PFP: parallel FP-growth for query recommendation, in: *Proceedings of the 2008 ACM Recommender Systems*, pp. 107–114, 2008.

[18] L. Liu, E. Li, Y. Zhang and Z. Tang, Optimization of frequent itemset mining on multiple-core processor, in: *Proceedings of the 33rd International Conference on Very Large Databases*, pp. 1275–1285, 2007.

[19] H. Mannila, H. Toivonen, and A. I. Verkamo, Discovery of Frequent Episodes in Event Sequences, *J. Data Mining Knowledge Discov.* **1** (1997), 259–289.

[20] R. Moriwal, FP-growth tree for large and dynamic data set and improve efficiency, *J. Inform. Comput. Sci.* **9** (2014), 83–90.

[21] J. S. Park, M. S. Chen and P. Yu, An effective hash-based algorithm for mining association rules, *Proc. ACM SIGMOD Manage. Data* **24** (1995), 175–186.

[22] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, Hmine: hyper-structure mining of frequent patterns in large data-bases, in: *Proceedings of IEEE Data Mining*, pp. 441–448, November 2001.

[23] B. Racz, nonordfp: an FP-growth variation without rebuilding the FP-tree, in: *Proceedings of IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2004.

[24] L. Schmidt-Thieme, Algorithmic features of Eclat, in: *Proceedings of IEEE Frequent Itemset Mining Implementations*, November 2004.

[25] S. Shporer, AIM2: Improved implementation of AIM, in: *Proceedings of IEEE Frequent Itemset Mining Implementations*, November 2004.

[26] C. Silverstein, S. Brin, R. Motwani and J. Ullman, Scalable techniques for mining causal structures, *J. Data Mining Knowledge Discov.* **4** (2000), 163–192.

[27] H. Toivonen, Sampling large databases for association rules, in: *Proceedings of Very Large Databases*, pp. 134–145, 1996.

[28] C. Utley, Introduction to SQL server 2005 data mining, *Microsoft SQL Server 9.0 technical articles* (2005). http://technet.microsoft.com/en-us/library/ms345131.aspx.

[29] L. Vu and G. Alaghband, A fast algorithm combining FP-tree and TID-list for frequent pattern mining, in: *Proceedings of Information and Knowledge Engineering*, pp. 472–477, July 2011.

[30] L. Vu and G. Alaghband, High performance frequent pattern mining on multi-core cluster, in: *Proceedings of 2012 IEEE International Conference on Collaboration Technologies and Systems*, pp. 630–633, May 2012.

[31] L. Vu and G. Alaghband, Mining frequent patterns based on data characteristics, in: *Proceedings of 2012 International Conference on Information and Knowledge Engineering*, pp. 369–375, July 2012.

[32] T. Yoshizawa, I. Pramudiono and M. Kitsuregawa, SQL based association rule mining using commercial RDBMS (IBM db2 UBD EEE), in: *Proceedings of Data Warehousing and Knowledge Discovery*, pp. 301–306, 2000.

[33] M. J. Zaki and K. Gouda, Fast vertical mining using diffsets, in: *Proceedings of ACM SIGKDD Knowledge Discovery and Data Mining*, pp. 326–335, 2003.

[34] M. Zaki, S. Parthasarathy, M. Ogihara and W. Li, New algorithms for fast discovery of association rules, in: *Proceedings of Knowledge Discovery and Data Mining*, pp. 283–286, 1997.

[35] Frequent itemset mining implementations repository, in: *Workshop on Frequent Itemset Mining Implementation*, 2003–2004. http://fimi.ua.ac.be.