Computing and Informatics, Vol. ?, ?, 1–18, V 2013-Jun-22

EFFICIENTLY USING PRIME-ENCODING FOR MINING FREQUENT ITEMSETS IN SPARSE DATA

Karam Gouda, Mosab Hassaan

Faculty of Computers & Informatics
Benha University, Benha, Egypt

e-mail: karam.gouda@fci.bu.edu.eg, mosab.hassaan@fci.bu.edu.eg

Abstract. In the data mining field, data representation turns out to be one of the major factors affecting mining algorithm scalability. Mining Frequent Itemsets (MFI) is a data mining problem that is heavily affected by this fact. The vertical approach is one of the successful data representations adopted for MFI problem. The main advantage of this approach is support for fast frequency counting via joining operations. Recently, an encoding method called prime-encoding is proposed as an enhancement for the vertical approach [10]. The performance study introduced in [10] confirmed the high quality of prime-encoding based vertical mining of frequent sequence over other vertical and horizontal ones in terms of space and time. Though sequence mining is more general than itemset mining, this paper presents a prime-encoding based vertical mining of frequent itemsets with new optimizations and a new re-encoding method that further enhance memory and speed. The experimental results show that prime encoding based vertical itemset mining is suitable for high-dimensional sparse data.

Keywords: Mining frequent itemsets, prime-block encoding, sparse data

1 INTRODUCTION

Mining Frequent Itemsets (MFI) is a fundamental and essential problem in many data mining applications, including market and customer analysis, mining web logs, patterns discovery in protein sequences, and so on. The problem is formulated as follows: Given a set of item transactions, find all frequent itemsets, where a frequent itemset is one that occurs in at least a user specified percentage of the data.

Several efficient and scalable algorithms have been proposed in the previous studies, ranging from mining frequent itemsets [3, 15, 1, 11, 16, 19, 20], with and without constraints, to mining closed and maximal itemsets [6, 12, 5, 13, 7, 9, 21, 18]. Most of the existing methods follow implicitly or explicitly the same search and pruning strategies, that are often depth or breadth first search with pruning mostly based on the downward closure property [3]. Indeed, the variant performance of current methods is due to the way of how each method represents data and counts pattern frequency. Generally, there are two main data representation approaches referred to as the horizontal and the vertical approaches. In the horizontal data representation, the data set consists of a list of tuples called transactions, where each transaction has an identifier called tid (tid stands for transaction id), followed by a list of items in that transaction. In the vertical data representation, two data formats are often used, tidsets and bitmaps. In tidset format, each data item is associated with its corresponding *tidset*, the set of all tids where it appears, whereas in bitmap, a bit is used for each transaction. The bit is set to *one* if the item appears in the corresponding transaction whereas *zero* is used to register the item absence.

In this paper, we focus on the vertical format, in particular its performance on sparse data. The main advantage of the tidset format is that tidsets offer natural pruning of irrelevant transactions as a result of joining (tids not relevant drop out) [19]. Thus, when the original and intermediate relevant data is very small as in sparse domains, tidset becomes one of the most efficient data representations. The bitmap format, on the other hand, suffers from the problem of sparseness of the bitmaps especially at lower support levels as always occurs for sparse data, though on new 64 bit-based computer systems joining bitmaps becomes very fast.

Can vertical-based algorithms be enhanced further for sparse data? Recently, an encoding method called Prime-block Encoding is proposed as an enhancement for the vertical approach [10]. The primal structure is a very elegant structure and is much more compact than bitmap and tidset. In addition, it combines the virtues of both formats– irrelevant data drops out as early as possible as a result of joining as in tidsets and joining is performed very fast as in bitmap. The performance study presented in [10] confirmed the high quality of prime-encoding based vertical mining of frequent sequence over other vertical and horizontal ones in terms of space and time. Though sequence mining is more general than itemset mining, this paper presents a prime-encoding based vertical mining of frequent itemsets with new optimizations and a new re-encoding method that further enhance memory and speed.

A systematic performance study is reported to verify the performance gain claimed by these new optimizations. To do so, the prime-encoding is integrated with Eclat [19], the state-of-the-art tidset-based frequent itemset mining method. Our enhancement is called P_Eclat. The experimental results show that the prime-encoding with the new optimizations deliver more than 4 times performance improvement over the tidset-based mining on sparse data.

The remainder of this paper is organized as follows. The problem of Mining Frequent Itemsets and preliminary work are presented in Section 2. Section 3 is devoted to P_Eclat algorithm and the new optimizations. The related work is discussed in Section 4. The experimental results are reported in Section 5. Section 6 concludes the paper.

2 PRELIMINARIES

2.1 Problem Definition

Let $\mathcal{I} = \{i_1, \ldots, i_m\}$ be a set of items, and $\mathcal{D} = \{T_1, T_2, \ldots, T_N\}$ a set of tuples called transactions, where each transaction T_i has a unique identifier (*tid*) and contains a set of items. Let $\mathcal{T} = \{1, 2, \ldots, N\}$ be the the set of tids in $\mathcal{D}$, a set $X \subseteq \mathcal{I}$ is called an *itemset*, and a set of tids $Y \subseteq \mathcal{T}$ is called a *tidset*. An itemset with k items is called a k-itemset. For convenience we write an itemset $\{A, C, W\}$ as ACW, and a tidset $\{2, 4, 5\}$ as 245. A transaction T_i is said to contain an itemset X, if X is a subset of T_i. The *absolute support* of an itemset X is defined as the number of data transactions that contain X, and the *relative support* is defined as the percentage of data transactions that contain X. Without loss of generality, we use the absolute support in the rest of the paper while in experimental study the relative support is often used. The support of itemset X in a data set $\mathcal{D}$ is denoted by $sup_{\mathcal{D}}(X)$. Given a support threshold *min_sup*, itemset X is *frequent* in a data set $\mathcal{D}$ if X is contained by at least *min_sup* transactions of $\mathcal{D}$. The problem of **Mining Frequent Itemsets** is to find all frequent itemsets in a data set $\mathcal{D}$, given a support threshold *min_sup*.

tid	T
1	ABE
2	BD
3	BC
4	ABD
5	AC
6	BC
7	AC
8	$ABCE$
9	ABC

Table 1. Horizontal Data $\mathcal{D}$

Example As an example, consider the transaction data set $\mathcal{D}$ shown in Table 1. It consists of 9 transactions, and there are five items used, i.e., $\mathcal{I} = \{A, B, C, D, E\}$.

Suppose *min_sup* = 2. The 2-itemset BD is frequent since it is contained by the second and fourth tuples of $\mathcal{D}$.

Search Space The search space for enumeration of all frequent itemsets is given by the power set $\mathcal{P}(\mathcal{I})$ which is exponential ($2^{|\mathcal{I}|}$) in the number of items. Figure 1 shows an example of a complete itemset search tree for $\mathcal{I} = \{A, B, C, D, E\}$. The root is the null itemset and each lower level k contains all of the k-itemsets, which are ordered lexicographically. Notice from the figure that the itemset search tree is huge even with very few data items. The factors which make mining frequent itemsets feasible is that the real data instances comprise of short transactions. This, indeed, bounds the deepest levels at which mining will be stopped. The uncrossed nodes of the tree represent the frequent itemsets of the example data given in Table 1 with deepest level is at most 3. Pruning is also used to cut the search space that should be considered. For example, support-based ordering of tree nodes and the *downward closure property*[1] help to narrow down the search space and prune unnecessary branches.

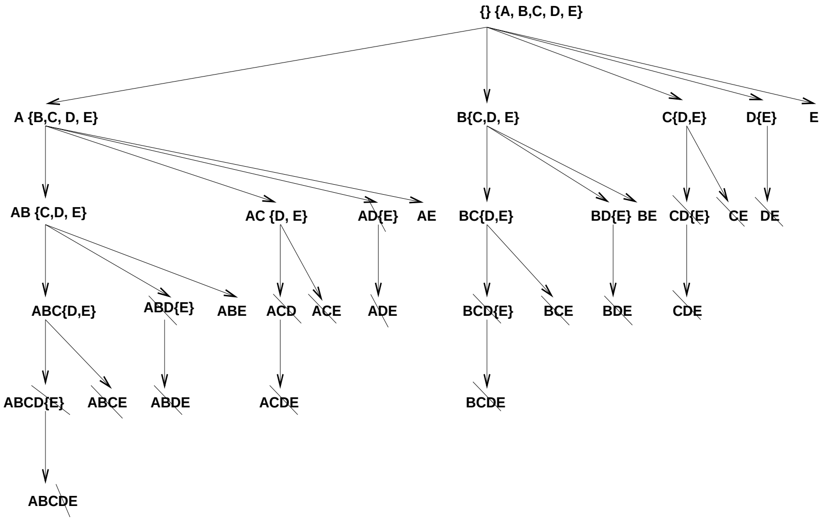

Fig. 1. Subset Search Tree

2.2 Common Data Formats

The data format given in Table 1 is the traditional horizontal data representation. Vertical format is another data representation used. Two variant of the vertical representation are *tidset* and *bitmap*. In the tidset format, we maintain for each

[1] the property that all subsets of a frequent itemset must themselves be frequent.

item its tidset, a set of all tids where it occurs, whereas in bitmap, a bit is used for each transaction. The bit is set to *one* if the item appears in the corresponding transaction whereas *zero* is used to register the item absence. For each item $i_l \in \mathcal{I}$, let $t(i_l)$ denote its tidset and $b(i_l)$ denotes its bitmap. Table 2 and Table 3 show tidset and bitmap formats, respectively. Methods use horizontal format include Apriori [1], MaxMiner [6] and DepthProject [5]. Methods based on vertical tidset format include Eclat [19], Charm [21], and Partition [15]. Methods based on vertical bitmap format include Viper [16] and Mafia [7, 8]. Our main focus is to improve upon methods that utilize the vertical format for sparse data.

$t(A)$	$t(B)$	$t(C)$	$t(D)$	$t(E)$
1	1	3	2	1
4	2	5	4	8
5	3	6		
7	4	7		
8	6	8		
9	8	9		
	9			

Table 2. Vertical Data: tidset

tid	$b(A)$	$b(B)$	$b(C)$	$b(D)$	$b(E)$
1	1	1	0	0	1
2	0	1	0	1	0
3	0	1	1	0	0
4	1	1	0	1	0
5	1	0	1	0	0
6	0	1	1	0	0
7	1	0	1	0	0
8	1	1	1	0	1
9	1	1	1	0	0

Table 3. Vertical Data: bitmap

2.3 Frequency Counting

Each node in the subset search tree can be treated as a prefix itemset, from which the set of its children can be generated by adding one item from $\mathcal{I}$. Because we are only interested in mining frequent itemsets, according to the downward closure property, we only need to grow a prefix itemset using the set of its locally frequent items. Two ways are presented in the literature to determine the prefix locally frequent items. Suppose the prefix is at level k, i.e., it is a $k-$itemset. The first way is to determine locally frequent items from the frequent itemsets at the prefix level, i.e, from frequent $k-$itemsets. They are the suffix items of those itemsets sharing the prefix in all its first $k-1$ itemes, in the same order. The class of these itemsets is referred to as the prefix equivalence class [19]. For example, the items C and E are the locally frequent items that are used for extending the prefix AB. Another way is presented in [11, 5] and explained below.

Vertical Counting In vertical-based methods, the generated itemset is tested against the data set through constructing its corresponding vertical structure. Let $X = Px_i$[2] be a prefix itemset and x_j be the locally frequent item used for extending X. The corresponding tidset of the generated itemset Xx_j is constructed through intersection on transaction ids of the two tidsets $t(Px_i)$ and $t(Px_j)$, i.e., $t(Xx_j) = t(Px_i) \cap t(Px_j)$, if the equivalence class approach is used [19] or $t(Xx_j) = t(X) \cap t(x_j)$ otherwise. On the other hand, bitwise AND ($\wedge$) on the corresponding bitmaps is enough to get $b(Xx_j)$. The new itemset support is given by the number of distinct *tids* in the corresponding tidset, or the number of ones if bitmaps are used.

Horizontal Counting Horizontal-based algorithms, on the other hand, use data scans for this task. In the first generation of horizontal algorithms–itemsets generation is based on breadth first traversal–full scans of the entire data is used to evaluate the support of generated k-itemsets [3, 4, 1, 17, 6]. In the second generation of horizontal algorithms–algorithms are based on depth first traversal–algorithms utilize proper data projection to reduce the size of the data to be scanned [5, 11]. Data projection means that, during search, the transactions containing the given prefix itemset X are collected to form the X-projected data. Then, the further search of larger itemsets can be achieved by searching only the X-projected data. In order to get the frequent extensions of X, the X-projected data is scanned to count the locally frequent items with respect to X which can be used to grow prefix X in order to get longer frequent itemsets. Then, the X-projected data is reprojected for every extension.

2.4 Primal Block Encoding

Recently, an encoding method called Prime-block Encoding is proposed as an enhancement for the vertical approach [10]. The method is explained as follows. Given $\mathcal{T} = [1 : N] = \{1, 2, \ldots, N\}$, the set of tids in the data set $\mathcal{D}$, and let G be a base set of prime numbers sorted in increasing order. Without loss of generality assume that N is a multiple of $|G|$, i.e., $N = m \cdot |G|$. Let $B \in \{0,1\}^N$ be a binary vector of length N. Then B can be partitioned into $m = \frac{N}{|G|}$ consecutive blocks, where each block $B_i = B\left[(i-1) \cdot |G| + 1 : i \cdot |G|\right]$, with $1 \leq i \leq m$. In fact, each $B_i \in \{0,1\}^{|G|}$, can be thought of as an indicator vector, representing subsets of G. Let $G[j]$ denotes the j-th prime in G. Define the *value* of B_i with respect to G as follows, $\nu(B_i, G) = \otimes\{G[j]^{B_i[j]}\}$. For example if $B_i = 1001$, and $G = \{2, 3, 5, 7\}$, then

[2] x_i is a single item and P is an itemset, where $|P| \geq 0$

$\nu(B_i, G) = 2^1 \cdot 3^0 \cdot 5^0 \cdot 7^1 = 2 \cdot 7 = 14$. Note also that if $B_i = 0000$ then $\nu(B_i, G) = 1$. Define $\nu(B, G) = \{\nu(B_i, G) : 1 \leq i \leq m\}$, as the *primal block encoding* of B with respect to the base prime set G. It should be clear that each $\nu(B_i, G) \in P(G)$, the power set of G. Note that when there is no ambiguity, we write $\nu(B_i, G)$ as $\nu(B_i)$, and $\nu(B, G)$ as $\nu(B)$. As an example, let $\mathcal{T} = \{1, 2, ..., 12\}$, $G = \{2, 3, 5, 7\}$, and $B = 100111100100$. Then there are $m = 12/4 = 3$ blocks, $B_1 = 1001$, $B_2 = 1110$ and $B_3 = 0100$. We have $\nu(B_1) = \otimes S_G(B_1) = \otimes\{2, 7\} = 2 \cdot 7 = 14$, and the primal block encoding of B is given as $\nu(B) = \{14, 30, 3\}$. Define $ones(B)$ to be the number of 1's in the binary vector B. For example $ones(100111100100) = 6$.

$p(A)$	$p(B)$	$p(C)$	$p(D)$	$p(E)$
14	210	5	21	2
70	21	210	1	7
2	2	2	1	1

Table 4. Vertical Data: primal structure

$t(AB)$	$b(AB)$	$p(AB)$
1	1	14
4	0	7
8	0	2
9	1	
	0	
	0	
	0	
	1	
	1	

Table 5. Joining tidsets, bitmaps and primal block structures.

Table 4 shows the vertical primal block encoding format of the example data given at Table 1. Note that the primal structure is compact, for example, $|p(B)| = 3$ integers, whereas $|t(B)| = 7$. Given two integers a and b. Let $gcd(a, b)$ denote the *greatest common divisor* of the two numbers a and b. The following theorem presents a way to join primal block encoding structures and get the number of ones of the generated structure.

Theorem 2.1. Let G be the base prime set, and let $A = A_1A_2 \cdots A_m$, and $B = B_1B_2 \cdots B_m$ be any two binary vectors in $\{0, 1\}^N$, with $N = m \cdot |G|$, and $A_i, B_i \in \{0, 1\}^{|G|}$. Then $\nu(A \cap B) = \{gcd(\nu(A_i), \nu(B_i)) : 1 \leq i \leq m\}$, and $ones(A \cap B) = \sum_i \|gcd(\nu(A_i), \nu(B_i))\|_G$.

Continuing our example above, let $A = 100011111000$, then $\nu(A \cap B) = \nu(100011100000) = \{\nu(1000), \nu(1110), \nu(0000)\} = \{2, 30, 1\}$. Note that $\nu(A) = \{2, 210, 2\}$, and $\nu(B) = \{14, 30, 5\}$. Applying the above theorem, we have $\nu(A \cap B) =$

$\{gcd(2, 14), gcd(210, 30), gcd(2, 1)\} = \{2, 30, 1\}$, which matches the direct computation. Also $ones(A \cap B) = \|2\|_G + \|30\|_G + \|1\|_G = 1 + 3 + 0 = 4$. Table 5 shows the structures $t(AB)$, $b(AB)$ and $p(AB)$ of the 2-itemset AB. These structures are generated by joining those of itemsets A and B.

3 P_ECLAT ALGORITHM

We integrated the prime-encoding method with Eclat [19], the state-of-the-art tidset-based frequent itemset mining method. Our enhancement is called P_Eclat. P_Eclat is outlined in Algorithm 1. Details on new optimizations are given in the following subsection. P_Eclat algorithm follows the equivalence class approach. It performs a depth first search of the subset tree. The input to the procedure is a set of class members for a subtree rooted at P. Frequent itemsets are generated by joining primal structures of all distinct pairs of itemsets and checking the support of the resulting itemset. A recursive procedure call is made with those itemsets found to be frequent at the current level. This process is repeated until all frequent itemsets have been enumerated. In terms of memory management, it is easy to see that we need memory to store intermediate primal structures for at most two consecutive levels within a class. Once all frequent itemsets at the next level have been generated, the itemsets at the current level within a class can be deleted.

Algorithm 1: P_Eclat(min_sup, $\mathcal{D}$)

Input: Transactional data set $\mathcal{D}$, *min_sup*.
output: F: Frequent itemsets in $\mathcal{D}$.

```
$F_1$ = { frequent items or 1-itemsets }
$F_2$ = { frequent 2-itemsets }
$\xi$ = { equivalence classes $[P]$ }
for all $[P] \in \xi$ do Enumerate-Frequent-Itemsets($[P]$)
Procedure Enumerate-Frequent-Itemsets($[P]$)
  for each $X_i \in [P]$ do
    for each $X_j \in [P]$, with $j > i$ do
      $R := X_i \cup X_j$
      $p(R) := \{gcd(p(X_i)_l, p(X_j)_l) : 1 \leq l \leq m\}$
      $\sigma(R) := \sum_l \|gcd(p(X_i)_l, p(X_j)_l)\|_G$
      if $\sigma(R) \geq \delta \times |\mathcal{D}|$ then
        $F_i := F_i \cup \{R\}$ // $F_i$ initially empty
    if $F_i \neq \phi$ then Enumerate-Frequent-Itemsets($F_i$)
```

3.1 Optimizations

3.1.1 Optimization 1: Removing Sparse Blocks

The primal structure size m is not fixed across all tree levels but it could be progressively decreased level by level as follows: If the prefix primal structure contains l entries of ones (i.e., l sparse blocks), then those ones could be deleted at the next level since it is assured that all structures which are formed using this prefix contain those ones at the same positions[3]. Thus, all itemsets generated by the prefix will have primal structures of size $m - l$. For example, consider the prefix D of the previous example, its primal structure $p(D)$ contains ones at the second and third positions. We can assign the new structure $p(DE)$ size 1 instead of 3 since it is guaranteed that its second and third elements will be ones and does not contribute to the support of DE. Then $p(DE) = \{1\}$. This way we save more space during computation.

3.1.2 Optimization 2: Building Initial Primal Structures

The initial primal structures can be computed from tidsets at any level of the search tree. It could also be read from disk directly using a primal version of the data. If we choose in the implementation to build primal structures for frequent items, we may have structures longer than the corresponding tidsets. The size increase is because each item's primal structure contains many ones corresponding to the sparse blocks, i.e., blocks of $\mathcal{T}$ where the item does not appear. Assume, for example, that the minimum support is set to 1 in Example 2.1, then every item having this support will be associated a primal structure of size 3. Contrast this to the corresponding tidsets of length 1. Generally, if $|G| = 4$, then all data items with support less than 25 percent of the data have primal structures longer than their corresponding tidsets. In sparse domains, most data items have this support bound.

Although the many ones exist in the prefix primal structure will be removed from the next level primal structures according to optimization 1, we have decided to construct the initial primal structures at the second tree level, i.e., for 2-itemsets. The primitive way for doing this is to first get tidsets and then construct primal structures from them using the primal encoding method. The resulting primal structures will have the original size m but they may contain more ones than before. Consider Example 2.1, $p(AB) = \{14, 7, 2\}$ could be constructed directly from $t(AB)$ by applying the encoding ν instead of joining $p(A)$ with $p(B)$. The method adopted in our implementation is the same as the primitive one but the difference is that we do not actually build the second level tidsets in memory. The intersection is done on the fly. We have also replaced the universal $\mathcal{T} = [1 : 12]$ in the transformation ν by the new one $\mathcal{T} = [1 : 6]$ - the indices of the prefix tidset $t(A)$. This is possible since we are working with equivalence classes and the prefix tidset indices will be

[3] $gcd(1, x) = 1$ for any integer $x \geq 0$

the new universal set to the class instead of original $\mathcal{T}$. As an example, consider how to build $p(AB)$. For each tid in the intersection $t(AB)$, we get its position at the prefix tidset $t(A)$. This position has its corresponding prime number according to the transformation ν with the new universal set. The positions of tids of $t(AB)$ at $t(A)$ are 1, 2, 5 and 6, their equivalent prime numbers when $|G| = 4$ are 2, 3, 2, and 3 according to the new ν. Then $p(AB) = \{6, 6\}$. Compare this primal structure with the one derived before, the size here is 2 instead of 3.

Why primal encoding at the second level of the search tree speed up mining? Building second level primal structures contributes not only in removing sparse blocks but in a high compression. At the second level, tidsets will be compressed than ever, since we build primal structures using the indices of prefix's tidset. The maximum index value is equal to the prefix support, then every generated primal structure will be of size at most $\lceil sup(a)/|G| \rceil$, where a is the prefix item. Compare this size with $\lceil |\mathcal{T}|/|G| \rceil$. In the case of sparse data, where $|\mathcal{T}| \gg sup(a), \forall a$, the primal structure becomes very short. Sorting frequent items in increasing order of support and building primal structures at the second tree level as the optimization 2 suggests will improve the storage.

3.1.3 Optimization 3: Re-encoding

Primal block structure adds to the vertical tidset approach new advantages especially for sparse data. First, it becomes memory-efficient. It is clear that primal structure is a compressed form of tidset and the compression ratio depends on the base prime set size, $|G|$. In the previous example, where G is the first four primes, $p(AB)$ achieves 50% compression on the corresponding tidset $t(AB)$ – $|p(AB)| = 2$ whereas $|t(AB)| = 4$. Alternatively, if we choose G to be the first eight primes, then $p(AB) = \{858\}$ achieves about 75% compression on $t(AB)$. Generally, we can achieve up to $10|G|$% compression ratio on tidsets. Unfortunately, $|G|$ is bounded by the memory size allowed for each integer on the computer system. Moreover, $10|G|$% reduction ratio is possible at the beginning of the encoding process, i.e., with initial primal structures, and it decreases upward until, in the worst case, the primal structures hold the same size as if tidsets were used, and this of course will be at the end of mining.

Even with small G we can keep the initial compression ratio across all search tree levels by applying the concept of *re-encoding*. The main issue here is that re-encoding is performed given primal structures not tidsets or bitmaps. Naive re-encoding works by first joining the corresponding primal structures on the fly and while joining, it decodes each generated element to its corresponding tids in order to decide their positions at the corresponding prefix tidset, then it follows the method presented at optimization 2 to re-encode. Consider for example re-encoding while joining $p(A)$ with $p(B)$ given in Table 4. Joining $p(A)$ with $p(B)$ on the fly yields $p(AB) = \{14, 7, 2\}$, decoding its elements 14, 7 and 2, we get the positions 1, 2, 5 and 6 at $t(A)$, respectively, where the positions 1 and 2 correspond to the element 14. Then by encoding these positions we get $p(AB) = \{6, 6\}$.

The main problem with naive re-encoding is that the cost of decoding elements to their corresponding tids and then re-encoding would be expensive than joining the original tidsets. The adopted re-encoding methodology works by deciding for each prefix primal structure what is called the *encoding regions*: the ranges on prefix primal structure for which the total number of tids corresponding to primal entries are at most $|G|$. These encoding regions are collected in a prefix offset array. This array guides re-encoding while joining the prefix primal structure with every primal structure in its equivalence class. Consider re-encoding $p(AB)$ according to this method. There are two encoding regions of $p(A)$, where the first region is given by the element 14 and the second is given by the two elements 70 and 2, since the element 70 has corresponding 3 tids and 2 has only one corresponding tid; these four tids represent a region to be re-encoded into one new element while joining. While joining $p(A) = \{14, 70, 2\}$ with $p(B) = \{210, 21, 2\}$, the first element is joined without re-encoding to get 14 whereas the second and third elements are joined then re-encoded. Joining 70 with 21 and 2 with 2 yield the elements 7 and 2, since these elements are at the third and fourth positions of the encoding region, these two elements are encoded into the element 35, then the resulting structure becomes $p(AB) = \{14, 35\}$.

The question arising here is that: Does the benefit of using the encoded primal structures outweigh the cost for re-encoding at every tree node? Re-encoding guarantees that every generated primal structure will be of size at most $\lceil sup(X)/|G| \rceil$, where X is the prefix itemset. This shows a good memory saving. However, re-encoding at every tree node may affect algorithm speed. In order to weigh up memory saving and speed, we use an adaptive approach to determine when to re-encode. At each node, we estimate the prefix primal structure size relative to prefix support. When that ratio reaches some threshold, re-encoding is chosen for that node and the subtree rooted at that node.

3.2 Auxiliary Data Structure

In our implementation of the primal encoding method, joining is performed very fast using a collection of pre-computed, and compact, lookup tables. Since computing the *gcd* is the prominent operation while joining primal structures, we use a pre-computed table called GCD to facilitate rapid *gcd* computations. Note that in our examples above, we used G as the first four primes. However, in our actual implementation, we assume $|G| = 8$, i.e., $G = \{2, 3, 5, 7, 11, 13, 17, 19\}$. Note that with the new G, the largest element in $\otimes P(G)$ is 9699690. In total there are $| \otimes P(G)| = 256$ possible elements.

In naive implementation, the GCD lookup table can be stored as two dimension array with cardinality 9699690 × 9699690, where $GCD(i, j) = gcd(i, j)$ for any two integer $i, j \in [1 : 9699690]$. This is clearly inefficient, since there are in fact only 256 distinct elements in $\otimes P(G)$, and we thus really need a table of size 256 × 256 to store all the *gcd* values. We achieve this by representing each element in $\otimes P(G)$ by its rank, as opposed to its value [10]. For example, suppose the rank of the two elements 210 and 42 in the set $\otimes P(G)$ are 15 and 12 respectively. Then, gcd(15,12) = 12 is equivalent to gcd(210, 42)= 42. Since $rank \in [0 : 255]$, this representation bring down the storage requirements of the GCD table to just 256 × 256 = 65536 bytes. The primal structures entries are also represented by element rank which greatly enhance storage requirement of this format.

Other lookup tables. To speed up support determination, P_Eclat maintains a one-dimensional lookup array called $CARD$ to store the number of prime factors, i.e., *factor-cardinality* for each element in the set $\otimes P(G)$. That is, for each $x \in \otimes P(G)$, we store $CARD(rank(x)) = \|x\|_G$. For example, since $\|42\| = 3$, we have $CARD(rank(42)) = CARD(12) = 3$. To speed up re-encoding, another three-dimensional lookup table called *Common_Facts* is used. Common_Facts stores for each two elements in $\otimes P(G)$ the locations of common factors to both elements.

4 RELATED WORK

Mining Frequent Itemsets problem was introduced in [2]. The extensive research performed on this problem has led to an abundance of algorithms. Each algorithm typically consists of two interleaved steps, namely generation of itemsets and frequency testing. In most algorithms, generation is done by using one of the itemset tree traversals: depth-first or breadth-first. The existing methods essentially differ in the data structures used to ”index” the data to facilitate fast enumeration.

Two popular vertical and horizontal data representations are adopted. In vertical representation, each item is associated with an inverted index called tidset or bitmap. Frequency counting is done via joining operations on tidsets or bitmaps. On the other hand, in horizontal representation, the data transactions are not indexed at all, itemset frequency is determined by directly checking in which transaction the itemset appear [2]. Data projection, which is a hybrid between the horizontal and vertical representation, is introduced to accelerate the counting process of horizontal algorithms [5, 11, 14].

In sparse domain, the main topic of this paper, where the original and intermediate relevant data are very small, the horizontal approach with projection and vertical tidset format seem to be suitable, since tidset joining offers natural pruning of irrelevant data and the projected data become more and more smaller. However, since projection requires the original data to be in memory, the horizontal approach does not scale to large sparse data sets with long transaction size.

Even-though bitmaps use bits to represent information and simple bitwise AND, the ANDing operation does not affect the bitmap size. Hence, the vertical bitmap approach suffers from the problem of sparseness of the bitmaps especially at lower support levels. Data projection is also used in order to compress bitmaps [8], which makes the bitmap-based approach competitive for mining sparse data as well. Nevertheless, the additional memory required to hold the original data set as in the horizontal approach bounds the applicability of the vertical bitmap approach to only small data sets [7, 8].

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of P_Eclat algorithm on sparse, synthetic data sets. P_Eclat is implemented with the three optimizations given in Section 3 in standard C++ and compiled with GNU GCC. All experiments were performed on a 3GHz dual Core PC with 4G memory running Linux.
Data Sets: We chose several sparse, synthetic data sets for testing the performance of P_Eclat. Synthetic data sets are generated using the IBM data generation program [3]. There are several factors that we considered while comparing algorithms on synthetic data sets. All of these factors can be specified as parameters when running the generation program. For example, a transaction data set T10I4D100k means that the data set contains 100k transactions; the average number of items per transaction is 10; and the average number of items within the maximal itemsets is 4. The number of items $|\mathcal{I}|$ is set to 1000 in all data sets unless mentioned otherwise.

5.1 Performance Study

To evaluate P_Eclat algorithm, experiments are conducted to compare it with the state-of-the-art horizontal and vertical frequent itemset mining algorithms like FP-growth [11] and Eclat [19]. The codes/executables for these methods were obtained from their authors. Two versions of P_Eclat are used in the comparisons: P_Eclat-opt2 is the P_Eclat algorithm with optimization 2, and P_Eclat-opt3 is the algorithm with optimizations 2 and 3.

Figure 2 shows the results for the data sets where all four methods can run for at least some support values. P_Eclat-opt3 shows the best performance on most data sets. The only exception is for the data set T10I4D100k where FP-growth outperforms other methods. For small data set like T10, the initial overhead needed to set up and use the vertical representation in some cases outweighs the benefit of faster counting, and because of this FP-growth run slightly faster for this small data set. However, for all other data sets which are characterized by higher values of the parameters T and I, the performance of FP-growth degrades sharply to the limit that it fail to run on our machine on very small support values. On the

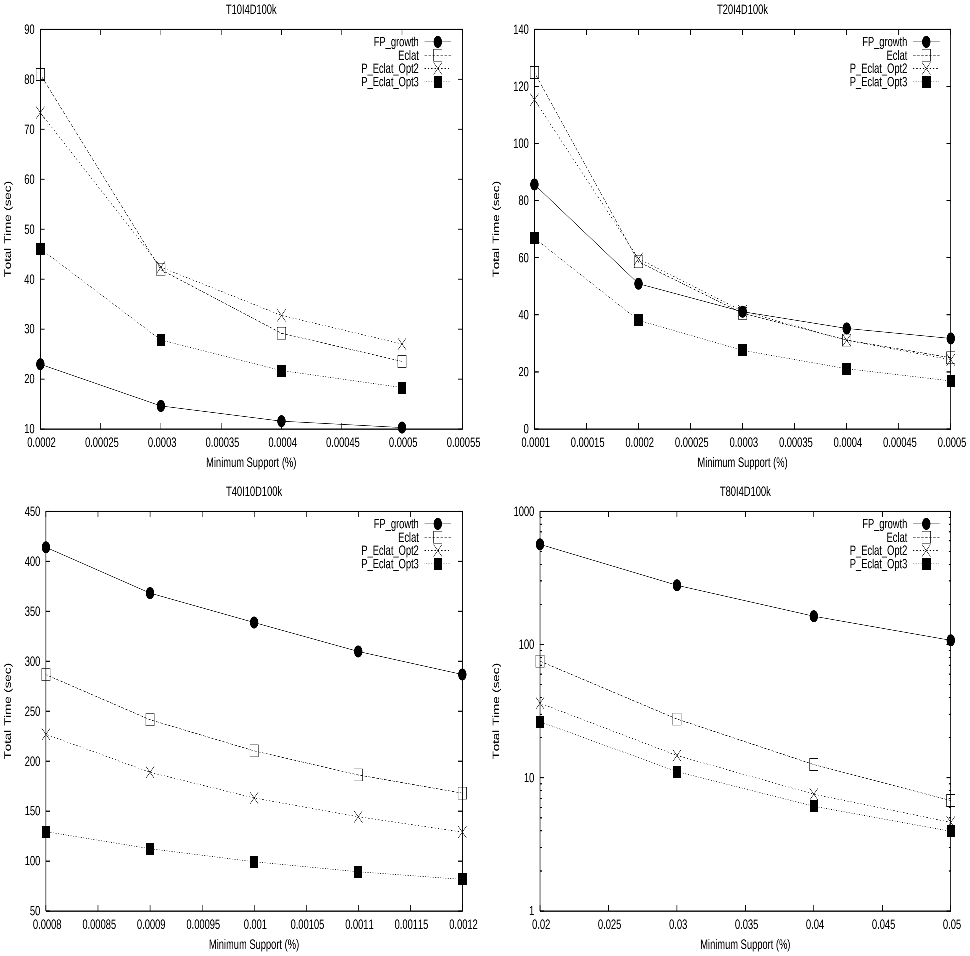

Fig. 2. Comparative Performance: FP-growth, Eclat, P_Eclat-opt2, P_Eclat-opt3

contrary, P_Eclat-opt3 performance is less sensitive to these parameters. P_Eclat-opt3 outperforms FP-growth by more than one order of magnitude on the data set T80I4D100k, and outperforms Eclat by more than three times on both T40I10D100k and T80I4D100k. You can also note that P_Eclat-opt3 outperforms P_Eclat-opt2 for all data sets. This confirms the benefit of re-encoding on the performance of P_Eclat.

Figure 3 shows the performance on the data sets T40I16D100k and T120I4D100k. These data sets are characterized by larger values of T and I. FP-growth is not shown because it fails to run on our machine for the given minsup values due to its memory consumption is well beyond the physical the memory available (4GB), and

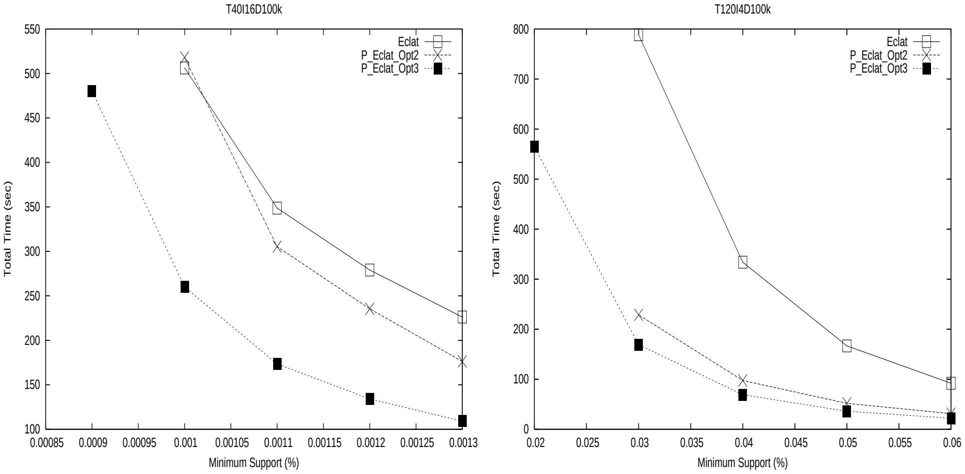

Fig. 3. Comparative Performance: Larger values of T and I

thus the program aborts when the system runs out of memory. Eclat and P_Eclat-opt2 also fail to run for the same reason for smaller minsup values (< 0.001) on the data set T40I16D100k and (< 0.03) on the data set T120I4D100k. P_Eclat-opt3 outperforms Eclat by more than four times on the larger data set T120I4D100k. This result proves the advantage of the primal encoding method with re-encoding over the tidset approach in terms of memory and speed.

Scalability: Figure 4 shows the scalability of the different methods when we vary the different data set parameters. The base values used are as follows: $T = 40$, $I = 4$, and $D = 100$k. We vary a single parameter at a time, keeping all others fixed to the default values. Figures 4 shows the effect of increasing the number of transactions from 100k to 300k, the effect of increasing the average transaction size from 10 to 120, and the effect of increasing the the average number of items within the maximal itemsets from 4 to 20. P_Eclat-opt3 scales gracefully. It shows the best performance in all experiments. The performance gain on different data set parameters is consistent with previous experiments.

6 CONCLUSIONS

Mining Frequent Itemsets (MFI) is a fundamental and essential problem in many data mining applications. The vertical approach is one of the successful data representations adopted for MFI problem. In this paper, the Prime-block Encoding method [10] is used with the vertical MFI Algorithms. New optimizations and a re-encoding method are also presented. The experimental evaluation shows that Prime-block Encoding based vertical itemset mining is suitable for high-dimensional sparse data.

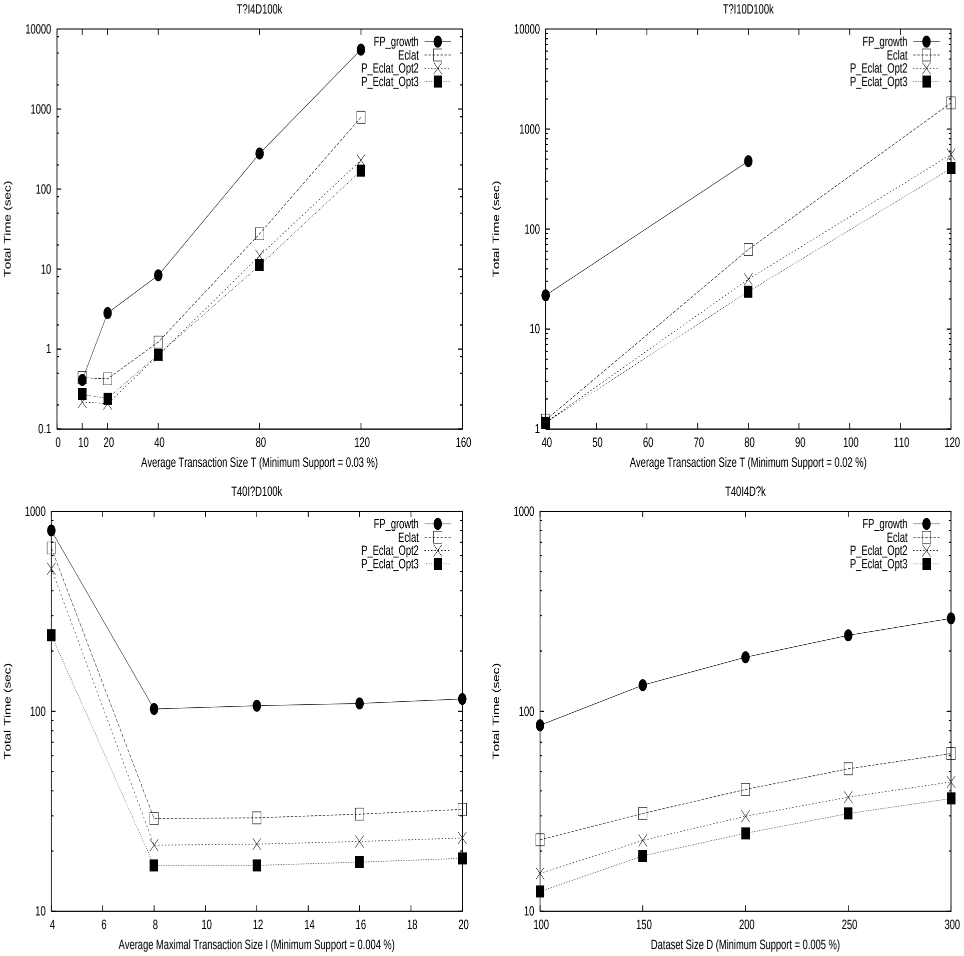

Fig. 4. Scalability with different data parameters

REFERENCES

[1] Agrawal, R. — Mannila, H. — Srikant, R. — Toivonen, H. — Inkeri Verkamo A.: Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, CA, 1996.

[2] Agrawal, R. — Imielinski, T. — Swami, A.: Mining Association Rules Between Sets of Items in Large Databases. In *Proceeding of the ACM-SIGMOD conference on Management of Data*, pages 207-216, 1993.

[3] Agrawal, R. — Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In *20th VLDB Conf.*, pp. 487-499, 1994.

[4] Agrawal, R. — Srikant, R.: Mining Sequential Patterns. In *11th Intl. Conf. on Data Engineering*, February 1995.

[5] Agrawal, R. — Aggarwal, C. — Prasad, V.V.V.: Depth First Generation of Long Patterns. In *7th Int'l Conference on Knowledge Discovery and Data Mining*, August 2000.

[6] Bayardo, R. J.: Efficiently mining long patterns from databases. In *ACM SIGMOD Conf.*, June 1998.

[7] Burdick, D. — Calimlim, M. — Gehrke, J.: Mafia: A maximal frequent itemset algorithm for transactional databases. In *17th Intl. Conf. on Data Engineering*, February 2001.

[8] Burdick, D. — Calimlim, M. — Flannick, J. — Gehrke, J. — Yiu, T.: MAFIA: A Performance Study of Mining Maximal Frequent Itemsets. In FIMI03.

[9] Gouda, K. — Zaki, M. J.: GenMax: An Efficient Algorithm for Mining Maximal Frequent Itemsets. In *Data Mining and Knowledge Discovery: An International Journal*, 11(3): 223-242, 2005.

[10] Gouda, K. — Hassaan, M. — Zaki, M. J.: Prism: An Effective Approach for Frequent Sequence Mining via Prime-Block Encoding. In *Journal of Computer and System Sciences (JCSS)*, 76(1):88-102, Feb 2010.

[11] Han, J. — Pei, J. — Yin, Y.: Mining frequent patterns without candidate generation. In *ACM SIGMOD Conf. Management of Data*, May 2000.

[12] Pasquier, N. — Bastide, Y. — Taouil, R. — Lakhal, L.: Discovering frequent closed itemsets for association rules. In *7th Intl. Conf. on Database Theory*, January 1999.

[13] Pei, J. — Han, J. — Mao, R.: Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery*, May 2000.

[14] Pei, J. — Han, J. — Mortazavi-Asl, B. — Pinto, H. — Chen, Q. Dayal, U. — Hsu, M.-C.: PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In *17th Intl. Conf. on Data Engineering*, Apr. 2001.

[15] Savasere, A. — Omiecinski, E. — Navathe, S.: An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[16] Shenoy, P. — Haritsa, J.R. — Sudarshan, S. — Bhalotia, G. — Bawa, M. — Shah, D.: Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 2000.

[17] Srikant, R. — Agrawal, R.: Mining sequential Patterns: Generalization and Performance. In *EDBT 1996*, pages 3–17 Mar. 1996.

[18] Wang, J. — Han, J. — Pei, J.: CLOSET+: Searching for the best strategies for Mining Closed Itemsets. In *9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, August 2003.

[19] Zaki, M. J.: Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, May-June 2000.

[20] Zaki, M. J. — Gouda, K.: Fast Vertical Mining using diffsets. In *9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, August 2003.

[21] Zaki, M. J. — Hsiao, C.-J.: CHARM: An efficient algorithm for closed itemset mining. In *SDM'02*, Arlington, VA, April 2002.

Karam Gouda. Associate Professor in the department of Information Systems, Faculty of computers & informatics, Benha University, Egypt. His research interest is in the field of Data Mining, and in String and Graph data Management.

Mosab Hassaan. Assistant Lecturer in the department of Information Systems, Faculty of computers & informatics, Benha University, Egypt. His research interest is in the field of Data Mining, and Graph data Management.