ISSN:0975-9646

Mining frequent item sets without candidate generation using FP-Trees

G.Nageswara Rao M.Tech, (Ph.D) **Suman Kumar Gurram (M.Tech I.T)**

Aditya Institute of Technology and Management, Tekkali, Srikakulam (DT), A.P, India.

Abstract

There are so many algorithms for extracting frequent item sets.. These are very important for mining association rules as well as for many other data mining tasks . So many methods have been implemented for mining frequent item sets using a prefix tree structure known as frequent Pattern Tree (FP-Tree) for storing all the information about frequent item sets. In this paper we propose a new technique called fp-array technique based on FP-Tree Data structure, that reduces the traverse time of FP-Trees and we can improve the performance of FP-Tree based algorithms. This FP-array technique will give the good results for sparse data sets. It consumes more memory when we use the sparse data sets , consumes less memory for dense data sets and the performance of this algorithm is very well when the minimum support is low**.**

Index Terms – frequent Pattern - Tree, frequent item sets, association rules.

1.INTRODUCTION

Mining of frequent item sets (FIS) is a fundamental problem for mining association rules [1] [2]. It also plays an important role in other data mining tasks such as sequential patterns, episodes, multi dimensional patterns[7] [8].etc.The description of problem is as follows , Let I={i1,i2….in} be a set of items and D be a multi set of transactions, where each transaction T is a set of items such that T .For **any** ,we say that a transaction T contains X if **X T**.The set X is called an item set. The set of all **X I** (the powerset of I) naturally forms a lattice, called the item set lattice. The count of an item set X is the number of transactions in D that contain X.The support of an item set X is the proportion of transactions in D that contain X. Thus ,if the total number of transactions in D is n, then the support of X is divided by n.100 percent. An item set X is called frequent if its support is greater than or equal to some given percentage s, where s is called the minimum support.

When a transaction database is very dense and the minimum support is very low, i.e., when the database contains a significant number of large frequent item sets, mining all frequent item sets might not be a good idea. For example, if there is a frequent item set with size l, then all 2l nonempty subsets of the item set have to be generated. However, since frequent item sets are downward closed in the item set lattice, meaning that any subset of a frequent item set is frequent, it is sufficient to discover only all the maximal frequent item sets (MFIs). A frequent item set X is called maximal if there does not exist frequent item set Y such that X subset Y . Mining frequent item sets can thus be reduced to mining a “border” in the item set lattice. All item sets above the border are infrequent and those that are below the border are all frequent. Therefore, some existing algorithms only mine maximal frequent item sets.

However, mining only MFIs has the following deficiency: From an MFI and its support s, we know that all its subsets are frequent and the support of any of its subset is not less than s, but we do not know the exact value of the support. For generating association rules, we do need the support of all frequent item sets. To solve this problem, another type of a frequent item set, called closed frequent item set (CFI), was proposed in . A frequent item set X is closed if none of its proper supersets have the same support. Any frequent item set has the support of its smallest closed superset. The set of all closed frequent item sets thus contains complete information for generating association rules. In many cases, the number of CFIs is greater than the number of MFIs, although still far less than the number of FIs.

1.1 Mining FIS

The first algorithm Apriori for mining frequent item sets was proposed by Agarwal et al. It is a bottom-up breadth first search algorithm. This uses hash-trees to store frequent item sets and candidate item sets. It needs l database scans if the size of the largest frequent item set is l .

The Next algorithm FP-Growth method (novel algorithm) for mining frequent item sets was proposed by Han et al. It is a bottom-up depth first search algorithm. This uses FP-Tree to store frequency information of the original data base in a compressed form . It needs only 2 database scans and no candidate generation is required.

1.2 Mining Maximal frequent Item sets

Several algorithms have been proposed for Mining Maximal frequent item sets. These algorithms are different mainly in the adopted main memory data structures and in the strategy to visit the search space.. They are MAFIA, GenMax ..etc and these algorithms have some drawbacks . Smart Miner , also a depth-first algorithm, uses a technique to quickly prune candidate frequent item sets in the item set lattice. The technique gathers "tail" information for a node used to find the next node during depth-first mining in the lattice. Items are dynamically reordered based on the tail information. Smart Miner is about 10 times faster than MAFIA and GenMax.

1.3 Mining Closed frequent Item sets

Several algorithms have been proposed for Mining Maximal frequent item sets are A-Close , CHARM and etc.. . a different algorithm CLOSET for mining CFIs was proposed by pei at al. In this The FP-tree

structure was used and some optimizations for reducing the search space were proposed. The experimental results reported in showed that CLOSET is faster than CHARM and A-close. CLOSET was extended to CLOSET+ by Wang et al. in to find the best strategies for mining frequent closed item sets. CLOSET+ uses data structures and data traversal strategies that depend on the characteristics of the data set to be mined. Experimental results in showed that CLOSET+outperformed all previous algorithms.

1.4 Contributions

One of the important contributions of our work is a FP-array technique that uses a special data structure, called an FP-array, to greatly improve the performance of the algorithms on FP-trees. The FP-tree has been shown to be a very efficient data structure for mining frequent patterns [3], [4], [10], [11], [16] and its variation has been used for "iceberg" data cube computation [9**].** We first demonstrate that the FP-array technique drastically speeds up the FP-growth method on sparse data sets, since it now needs to scan each FP-tree only once for each recursive call emanating from it. This technique is then applied to our previous algorithm FP-max for mining maximal frequent item sets[3][4]. We call the new method FP max*. For checking maximal frequent item sets used MFI-Tree and for checking closedness of frequent item sets used a tree called CFI-Tree.

2 DISCOVERING frequent Item set's

2.1 The FP-Tree Structure and FP-Growth Algorithm

The algorithm Frequent Pattern-Growth method[3] [4] (novel algorithm) for mining frequent item sets was proposed by Han et al. It is a bottom-up depth first search algorithm. This uses FP-Tree to store frequency information of the original data base in a compressed form . Compression is achieved by building the tree in such a way that overlapping item sets share prefixes of the corresponding branches.

The FP-growth method relies on the following principle: If X and Y are two item sets, the count of item set X U Y in the database[5] is exactly that of Y in the **restriction** of the database to those transactions containing X. This restriction of the database is called the conditional pattern base of X and the FP-tree constructed from the conditional pattern base is called X0s conditional FP-tree, which we denote by TX. We can view the FP-tree constructed from the initial database as T;, the conditional FP-tree for the empty item set. Note that, for any item set Y that is frequent in the conditional pattern base of X, the **set X** □ Y is a frequent item set in the original database .

Let's an item i in Tx.header, by following the linked list starting at i in Tx. header, all branches that contain item i are visited. The portion of these branches from i to the root forms the conditional pattern base of X U {I} (X union i), so the traversal obtains all frequent items in this conditional pattern base. The FP-growth method then constructs the conditional FP-tree $\mathbf{T}_{x\ U\ \{i\}}$ by first initializing its header table based on the frequent items found, then revisiting the branches of TX along the linked list of i and inserting the corresponding item sets in $\mathbf{T}_{xU\{i\}}$. Note that the order of items can be different in TX and $\mathbf{T}_{xU\{i\}}$.. As an example, the conditional pattern base of ffg and the conditional FP-tree Tffg for the database in Fig. 1a is shown in Fig. 1c. The above procedure is applied recursively, and it stops when the resulting new FPtree contains only one branch. The complete set of frequent item sets can be generated from all single-branch FP-trees.

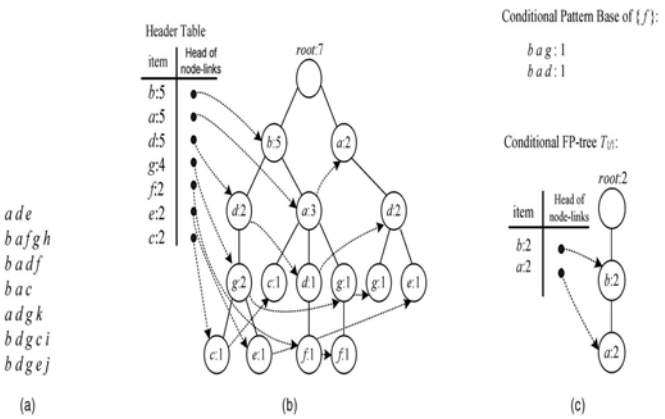

Fig. 1. An FP-tree example. (a) A database. (b) The FP-tree for the database (minimum support = 20 percent).

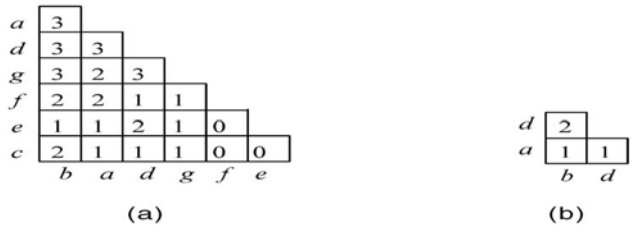

Fig. 2. Two FP-array examples. (a) $A_{;}$. (b) $A_{\{g\}}$.

2. The frequent Pair -Array Technique

In fp-growth method we can construct conditional FP-Trees after constructing FP-Tree. We found out that 80% of time was used for traversing FP-trees. To reduce the traversal time we can use a data structure called FP-Array. for each item j in the header of a conditional FP-tree Tx, two traversals of Tx are needed for constructing the new conditional FP-tree $T_{XU\{j\}}$ The first traversal finds all frequent items in the conditional pattern base of X U {j} and initializes the FP-tree $T_{XU\{j\}}$ by constructing its header table. The second traversal constructs the new tree $T_{XU\{j\}}$. We can omit the first scan of TX by constructing a frequent pairs array Ax while building Tx. We initialize Tx with an attribute Ax.

Definition. Let T be a conditional FP-tree and I= {j1;i2; . . im} be the set of items in T header. A frequent pairs array (FP- array) of T is a (m—1)x (m-1) matrix, where each element of the matrix corresponds to the counter of an ordered pair of items in I.

Obviously, there is no need to set a counter for both item Pairs(ij,ik) and (ik; ij). Therefore, we only store the counters for all pairs (ik; ij) such that k < j.

We use an example to explain the construction of the FP-tree supposing minimum support is 20 percent. We sort the frequent items b:5,a:5, d; a, g:4, f:2, e:2, c:2. This is the order of items is the header table T_b. During the second scan of the database, we will construct T_b and an FP-array A_b, as shown in Fig. 2a. All cells in the FP-array are initialized to 0.

According to the definition of an FP-array, in A_b, each cell is a counter of a pair of items. Cell A_b[c, b] is the counter item set {c, b}, cell A_b[c,a]and so forth. During the second scan for constructing Tb, for each transaction, all frequent items in the transaction are extracted. Suppose these items form item set J. To insert J into Tb, the items in J are sorted according to the order in Tb.header. When we insert J into Tb, at the same time Ab[i,j] incremented by 1 if {I, j} is contained in J. For instance, or the second transaction,{b, a, f, g} is extracted (item h is infrequent) and sorted as b, a, g, f. This item set is inserted into Tb; as usual and, at the same time, Ab[f,b], Ab[f,a], Ab[f,g], Ab[g, b], Ab[g, a], A[a, b] are all incremented by 1. After the second scan, the FP-array Ab, contains the countsof all pairs of frequent items, as shown in Fig. 2a.

Next, the FP-growth method is recursively called to mine frequent item sets for each item in Tb.header. However, now for each item i, instead of traversing Tb, along the linked list starting at i to get all frequent items in i's conditional pattern base, A; gives all frequent items for i. For example, by checking the third line in the table for A;, frequent items b; a; d for the conditional pattern base of g can be obtained. Sorting them according to their counts, we get b, d, a. Therefore, for each item i in Tb, the FP-array Ab makes the first traversal of Tb, unnecessary and each Tb can be initialized directly from Ab.

For the same reason, from a conditional FP-tree Tx, when we construct a new conditional FP-tree for X U {i} , for an item i, a new FP-array A xU {i}g is calculated. During the construction of the new FP-tree TxU{i}, the FP-array AX U {i} is filled. As an example, from the FP-tree in Fig. 1b, if the conditional FP-tree T{g} is constructed, the FP-rray A{g} will be in Fig. 2b. This FP-array is constructed as follows: From the FP-array Ab, we know that the frequent items in the conditional pattern base of {g} are, in descending order of their support, b, d, a. By following the linked list of g, from the first node, we get {b, d} :2, so it is inserted as (b :2;d :2) into the new FP-tree T{g}. At the same time, A{g}[b, d] is incremented by 1. From the second node in the linked list, {b, g} :1 is extracted and it is inserted as (b :1;a :1) into T{g}. At the same time, A{g}[b,d] is incremented by 1. From the third node in the linked list, {a, d} :1 is extracted and it is inserted as (d :1;a :1) into T{g}. At the same time, A{g}[d, a]is incremented by 1. Since there are no other nodes in the linked list, the construction of T{g} is finished and FP-arrayA{g} is ready to be used for construction of FP-trees at the next level of recursion. The construction of FP-arrays and FP-trees continues until the FP-growth method terminates.

Based on the foregoing discussion, we define a variant of the FP-tree structure in which, besides all attributes given

in [14], an FP-tree also has an attribute, FP-array, which contains the corresponding FP-array.

Let us analyze the size of an FP-array first. Suppose the number of frequent items in the first FP-tree Tb is n. Then, the size of the associated FP-array is proportional to $\sum_{i=1}^{n-1} i = n(n-1)/2$, which is the same as the number of candidate large 2-item sets in Apriori in [7]. The FP-trees constructed from the first FP-tree have fewer frequent items, so the sizes of the associated FP-arrays decrease. At any time when the space for an FP-tree is freed, so is the space for its FP-array. There are some limitations for using the FP-array technique. One potential problem is the size of the FP-array. When the number of items in Tb is small, the size of the FP-array is not very big. For example, if there are 5,000 frequent items in the original database and the size of an integer is4 bytes, the FP-array takes only 50 megabytes or so. However, when n is large, **n(n-1)/2** becomes an extremely large number. At this case, the FP-array technique will reduce the significance of the FP-growth method, since the method mines frequent item sets without generating any candidate frequent item sets. Thus, one solution is to simply give up the FP-array technique until the number of items in an FP-tree is small enough. Another possible solution is to reduce the size of the FP-array. This can be done by generating a much smaller set of candidate large two-item sets as in [15] and only store in memory cells of the FP-array corresponding to a two-item set in the smaller set. However, in this paper, we suppose the main memory is big enough for all FP-arrays.

The FP-array technique works very well, especially when the data set is sparse and very large. The FP-tree for a sparse data set and the recursively constructed FP-trees will be big and bushy because there are not many shared common prefixes among the FIs in the transactions. The FP-arrays save traversal time for all items and the next level FP-trees can be initialized directly. In this case, the time saved by omitting the first traversals is far greater than the time needed for accumulating counts in the associated FP-arrays.

Even for the FP-trees of sparse data sets, the first levels of recursively constructed FP-trees for the first items in a header table are always conditional FP-trees for the most common prefixes. We can therefore expect the traversal times for the first items in a header table to be fairly short, so the cells for these items are unnecessary in the FP-array. As an example, in Fig. 2a, since b, a, and d are the first three items in the header table, the first two lines do not have to be calculated, thus saving counting time. Note that the data sets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a data set is sparse or dense, during the construction of each FP-tree, we count the number of nodes in each level of the tree. Based on experiments, we found that if the upper quarter of the tree contains less than 15 percent of the total number of nodes, we are most likely dealing with a dense data set. Otherwise, the data set is likely to be sparse.

2.2 FP-growth*: An Improved FP-Growth Method

Fig. 3 contains the pseudo code for our new method FP-growth*. The procedure has an FP-tree T as parameter. T has attributes: *base*, *header*, *and FP-array*. T.base contains the item set X for which T is a conditional FP-tree, the attribute header contains the header table, and T:FP-array contains the FP-array A*x*.
In FP-growth*, line 6 tests if the FP-array of the current FP-
tree exists. If the FP-tree corresponds to a sparse data set, its FP-array exists, and line 7 constructs the header table of the new conditional FP-tree from the FP-array directly. One FP-tree traversal is saved for this item compared with the FP-growth method in [7]. In line 9, during the construction, we also count the nodes in the different levels of the tree in order to estimate whether we shall really calculate the FP-array or just set TY :FP-array as undefined.

PROCEDURE FP growth *

Input : A conditional FP-Tree T

Output : The complete set of all FI's corresponding to T.

Method:

1. ***If T only contains a single branch B***
2. ***For each subset Y of the set of item in B***
3. ***Output item set Y U T.base with count = smallest count of nodes in Y:***
4. ***Else for each i in T.header do begin***
5. ***Output Y =T.base U{i} with i count ;***
6. ***If T.FP-array is defined***
7. ***Construct a new header table for Y's FP-Tree from FP-array.***
8. ***Else construct a new header from the table T.***
9. ***Construct Y's conditional FP-Tree Ty and possible its FP-array Ay;***
10. ***If T≠ Φ***
11. ***Call FP-growth *(Ty);***
12. ***end***

figure 3

2.3 FP-MAX*: MINING MFI'S

In [6], we developed FP-max, another method that mines maximal frequent item sets using the FP-tree structure. Since the FP-array technique speeds up the FP-growth method for sparse data sets, we can expect that it will be useful in FP-max too. This gives us an improved method, FP-max*. Compared to FP-max, in addition to the FP-array technique, the improved method FP-max* also has a more efficient maximality checking approach, as well as several

other optimizations. It turns out that FP-max* outperforms FP-max for all cases we discussed in [6].

Fig. 4 gives algorithm FP-max*. In the figure, three attributes of T, T:base, T:header, and T:FP-array, are the same as the attributes we used in FP-growth*. The first call of FP-max* will be for the FP-tree constructed from the original database, and it has an empty MFI-tree. Before a recursive call FP-max*(T,M), we already know from line 10 that the set containing T:base and all items in T is not a subset of any existing MFI. During the recursion, if there is only one branch in T, the path in the branch together with T:base is anMFI of the database. In line 2, theMFI is inserted MIf the FP-tree is not a single-branch tree, then for each item i in T.header, we start preparing for the recursive call FP-max*(Ty ,;My), for Y = T.base U{i}. The items in the header table of T are processed in increasing order of frequency, so that maximal frequent item sets will be found before any of their frequent subsets. Lines 5 to 8 use the FP- array if it is defined or traverse Tx. Line 10 calls function *maximality_checking* to check if Y together with all frequent items in Y 's conditional pattern base is a subset of any existing MFI in M (thus, we do superset pruning here). *If maximality checking* returns false, FP-max* will be called recursively, with (Ty , My). The implementation of function *maximality_checking* will be explained shortly.

Procedure *FP-max *(T,M)*

Input : T an FP-tree

M, the MFI-Tree for T.base
Output: Updated M

Method:

1. ***if*** *T only contains a single branch B*
2. *insert B into M*
3. ***else for each*** *i in T.header* ***do begin***
4. *set Y= T.base U {i}*
5. ***if*** *T.FP-array is defined*
6. *Let tail be the set of frequent items for i in T.FP array*
7. ***else***
8. *let tail be the set of frequent items in i's condition pattern base;*
9. *sort tail decreasing order of the item's counts;*
10. ***if not*** *maximality_checking (Y U tail, M)*
11. *Construct Y's coditinal FP_tree Ty and possibly its FP_array Ay;*
12. *Initialize Y's conditional MFI-tree My;*
13. *Call FP-max * (Ty,My)*
14. *Merge My with M.*
15. ***end***

Figure 4: FP-max* Algorithm

set containing T:base and all items in T is not a subset of any existing MFI. During the recursion, if there is only one branch in T, the path in the branch together with T:base is anMFI of the database. In line 2, theMFI is inserted into M. If the FP-tree is not a single-branch tree, then for each item i in T:header, we start preparing for the recursive call FP-max* (Ty ,My) , for Y =T.base U {i}. The items in the header table of T are processed in increasing order of frequency, so that maximal frequent item sets will be found before any of their frequent subsets. Lines 5 to 8 use the FP-array if it is defined or traverse TX. Line 10 calls function maximality_checking to check if Y together with all frequent items in Y 's conditional pattern base is a subset of any existing MFI in M (thus, we do superset pruning here). If maximality_checking returns false, FP-max* will be called recursively, with (Ty , My) Note that, before and after calling maximality_checking,if Y U *tail* is not a subset of any MFI, we still do not know whether Y U *tail* is frequent. If, by constructing Y 's conditional FP-tree Ty , we find out that Ty only has a single branch, we can conclude that Y U *tail* is frequent. Since Y U *tail* was not a subset of any previously discovered MFI, it is maximal and will be inserted into My. The function maximality_checking works as follows: Suppose *tail* = i1, i2; ... ik, in decreasing order of frequency according to M:header. By following the linked list of ik, for each node n in the list, we test if tail is a subset of the ancestors of n. Here, the level of n can be used for saving comparison time. First, we test if the level of n is smaller thank. If it is, the comparison stops because there are not enough ancestors of n for matching the rest of tail. This pruning technique is also applied as we move up the branch and toward the front of tail. The function maximality_checking returns true if tail is a subset of an existing MFI otherwise, false is returned.

Unlike an FP-tree, which is not changed during the execution of the algorithm, an MFI-tree is dynamic. At line 12, for each Y , a new MFI-tree My is initialized from the preceding MFI-tree M. Then, after the recursive call, M is updated on line 14 to contain all newly found frequent item sets. In the actual implementation, we however found that it was more efficient to update all MFI-trees along the recursive path, instead of merging only at the current level. In other words, we omitted line 14, and instead on line 2,Bis inserted into the current M, and also into all preceding MFI-trees that the implementation of the recursion needs to store in memory in any case.

In details, at line 12, when an MFI-tree My_j for Yj = $i_1 i_2 \ldots i_j$ is created for the next call of FP-max*, we know that conditional FP-trees and conditional MFI-trees for Yj-1 = $i_1 i_2 \ldots i_{j-1}$, Yj-2= $i_1 i_2 \ldots i_{j-2}$,…., $Y_1 = i_1$, and $Y_0 = \Phi$ are all in memory. To make Myj store all already found MFIs that contain Yj, Myj is initialized by extracting MFIs from My_{j-1}. The initialization can be done by following the linked list for ij from the header table of My_{j-1} and extracting the maximal frequent item sets containing i_j. Each such found item set I is sorted according to the order of items in Myj .header (the same

item order as in Tyj .header) and then inserted into Myj. On line 2, we have found a new MFI B in Tyj ,so B is inserted into Myj . since y_j U B also contains yj-1, ...,y_1, y_0 and the trees My $_{j-1}$, ... My_1,My_0 are all in memory, to make these MFI-trees consistently store all already discovered MFIs that contain their corresponding item set, for each k =0, 1,2, ... j , the MFI B U(yj_— yk) is inserted into the corresponding MFI-tree My_k . At the end of the execution of FP-max*, the MFI-tree My_0 (i.e., M_Φ) contains all MFIs mined from the original database. Since *FP-max* is a depth-first algorithm*, it is straightforward to show that the maximality checking is correct. Based on the correctness of the FP-max method, we can conclude that FP-max* returns all and only the maximal frequent item sets in a given data set.

In FP-max*, we also used an optimization for reducing the number of recursive calls. Suppose that, at some level of the recursion, the item order in T.*header* is $i_1, i_2, \ldots, i_k$. Then, starting from ik, for each item in the header table, we may need to do the work from line 4 to line 14. If for any item, say i_m where $m \leq k$, its maximal frequent item set contains items $i_1, i_2, \ldots, i_{m-1}$, i.e., all the items that have not yet called FP-max* recursively, these recursive calls can be omitted. This is because any frequent item set found by uch a recursive call must be a subset of $\{i_1, i_2, \ldots, i_{m-1}\}$ thus, it could not be maximal.

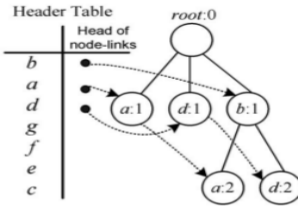

Figure 5.Size reduced maximal frequent item set tree.

2.4 FPCLOSE:MINING CFI'S

Recall that an item set X is closed if none of the proper supersets of X have the same support. For mining frequent closed item sets, FPclose works similarly to FP-max*. They both mine frequent patterns from FP-trees. Whereas FP-max* needs to check that a newly found frequent ite-emset is maximal, FPclose needs to verify that the new frequent item set is closed. For this, we use a closed frequent item sets tree (CFI-tree), which is another variation on the FP-tree. 4.1 The CFI-Tree and Algorithm FPclose as in algorithm FP-max*, a newly discovered frequent item set can be a subset only of a previously discovered CFI. Like anMFI-tree, a CFI-tree depends on an FP-tree T*x* and is denoted as C*x*. The item set X is represented as an attribute of T,T.*base*. The CFI-tree CX always stores all already found CFIs containing item set X and their counts.

A newly found frequent item set Y that contains X only needs to be compared with the CFIs in C*x*. If there is no proper superset of Y in *Cx* with the same count as Y , the set Y is closed.In a CFI-tree, each node in the subtree has four fields:*item-name, count, node-link, and level*. Here, *leve*l is still used for subset testing, as in MFI-trees. The *count* field is needed because when comparing Y with a set Z in the tree, we are trying to verify that it is not the case that Y *subset of* Z and Y and Z have the same count. The order of the items in a CFI-tree's header table is the same as the order of items in header table of its corresponding

The insertion of a CFI into a CFI-tree is similar to the insertion of a transaction into an FP-tree, except now the count of a node is not incremented, but replaced by the maximal count up-to-date. Fig. 6 shows some snapshots of the construction of a CFI-tree with respect to the FP-tree in Fig. 1b. The item order in the two trees are the same because they are both for base. Note that insertions of CFIs into the top level CFI-tree will occur only after recursive calls have been made. In the following example, the insertions would be performed during various stages of the execution, not in bulk as the example might suggest. In Fig. 6, a node *x : l : c* means that the node is for item x, that its level is l and that its count is c. In Fig. 6a, after inserting the first six CFIs into the CFI-tree, we insert (d, g) with count 3. Since (d , g) shares the prefix d with (d , e), only node g is appended and, at the same time, the count for node d is changed from 2 to 3. The tree in Fig. 6b contains all CFIs for the data set in Fig. 1a. Fig. 8 gives algorithm FPclose. Before calling FPclose with some (T , C), we already know from line 8 that there is no existing CFI X such that 1) T.base *subset of* X and 2) T:base and X have the same count (this corresponds to optimization 4 in [13]). If there is only one single branch in T, the nodes and their counts in this single branch can be easily used to list the T.*base-local* closed frequent item sets. These item sets will be compared with the CFIs in C. If an item set is closed, it is inserted into C. If the FP-tree T is not a single-branch tree, we execute line 6. Lines 9 to 12 use the FP-array if it is defined, otherwise, T is traversed. Lines 4 and 8 call function closed checking (Y ,C) to check whether a frequent item set Y is closed. Lines 14 and 15 construct Y 's conditional FP-tree TY and CFI-tree CY . Then, FPclose is called recursively for Ty and Cy .

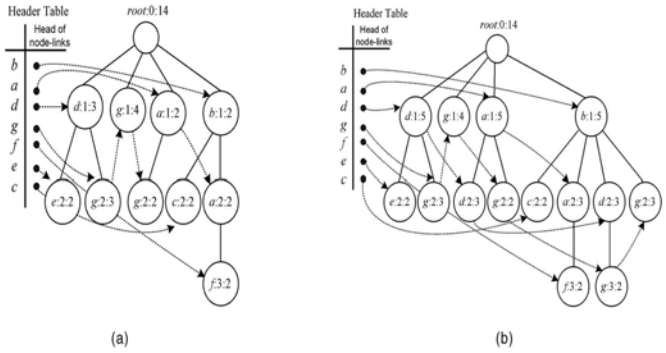

Figure 6

The implementation of function closed checking is almost the same as the implementation of function maximality checking, except now we also consider the count of an item set. Given an item set Y ={ i1, i2, . . . , ik} with count c, suppose the order of the items in header table of the current CFI-tree is i1, i2, . . . , ik. Following the linked list of ik, for each node in the list, we first check if its count is equal to or greater than c. If it is, we then test if Y is a subset of the ancestors of that node. Here, the level of a node can also be used for saving comparison time. The function closed checking returns true only when there is no existing CFI Z in the CFI-tree such that Z is a superset of Y and the count of Y is equal to or greater than the count of Z.

Procedure *FPclose*(T, C)
Input: T, an FP-tree
C, the CFI-tree for $T.base$
Output: Updated C
Method:
1. **if** T only contains a single branch B
2. generate all CFI's from B;
3. **for each** CFI X generated
4. **if not** *closed_checking*(X, C)
5. insert X into C;
6. **else for each** i in $T.header$ **do begin**
7. set $Y = T.base \cup \{i\}$;
8. **if not** *closed_checking*(Y, C)
9. **if** $T.FP\text{-}array$ is defined
10. let $tail$ be the set of frequent items for i in $T.FP\text{-}array$
11. **else**
12. let $tail$ be the set of frequent items in i's conditional pattern base;
13. sort $tail$ in decreasing order of items' counts;
14. construct the FP-tree T_Y and possibly its FP-array A_Y;
15. initialize Y's conditional CFI-tree C_Y;
16. call *FPclose*(T_Y, C_Y);
17. merge C_Y with C;
18. **end**

Figure 7 FP Close Algorithm

By a analysis, we can estimate the total memory requirement for running FPclose on a data set. If the tree that contains all CFIs needs to be stored in memory, the algorithm needs space approximately equal to the sum of the size of the first FP-tree and its CFI-tree. In addition, as for mining MFIs, for each CFI, by inserting only part of the CFI into CFI-trees, less memory is used and the implementation is faster than the one that stores all complete C FIs. Fig. 9 shows the size-reduced CFI-tree for ; corresponding to the data set in Fig. 1.In this CFI-tree, only 6 nodes are inserted, instead of 15 nodes in the complete CFI-tree in Fig. 6b.

3. FI Mining

In the section , we studied the performance of FP-growth* by comparing it with the original FP-growth method [3], [4], kDCI [13], dEclat [15], Apriori and PatriciaMine [16]. To see the performance of the FP-array technique, we implemented the original FP-growth method on the basis of the paper [14]. The Apriori algorithm was implemented by Borgelt in [12] for FIMI '03. The source codes of the other algorithms were provided by their authors.

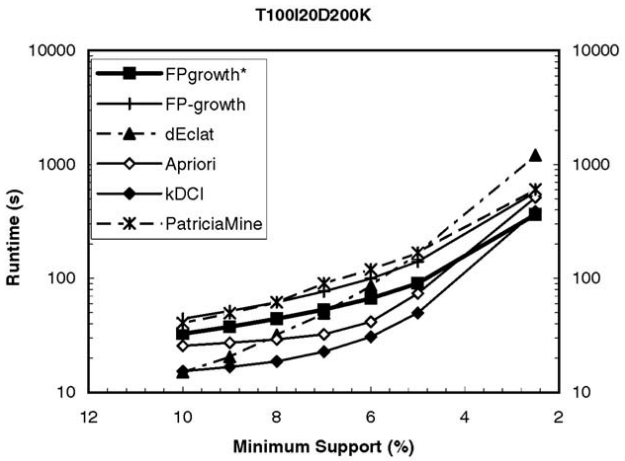

Figure 8

Fig. 8 shows the time of all algorithms running on **T100I20D200K**. In the figure, FP-growth* is slower than kDCI, Apriori, and dEclat for high minimum support. For low minimum support, FP-growth* becomes the fastest. The algorithm which was the fastest, dEclat, now becomes the slowest. The FP-array technique also shows its great improvement on the FP-growth method. FP-growth* is always faster than the FP-growth method and it is almost two times faster than the FP-growth method for low minimum support. When the minimum support is low, it means that the FP-tree is bushy and wide and the FP-array technique saves much time for traversing the FP-trees.

We can see that FP-growth* and the FP-growth method

unfortunately use the maximum amount of memory. Their memory consumption is almost four times greater than the data set size. Since the FP-growth* and FP-growth methods consume almost the same amount of memory, it means that thememoryspent onthe FP-array technique is negligible. The memory is mainly used by FP-trees constructed in the FP-growth method. Fig. 9 shows the peak memory consumption of the algorithms on the synthetic data set. The FP-growth* and the FP-growth method consume almost the same memory, their curves overlap again. In the figure, kDCI uses the lowest amount of memory when the minimum support is high. The algorithm dEclat also consumes far less memory than the other algorithms except kDCI.

.

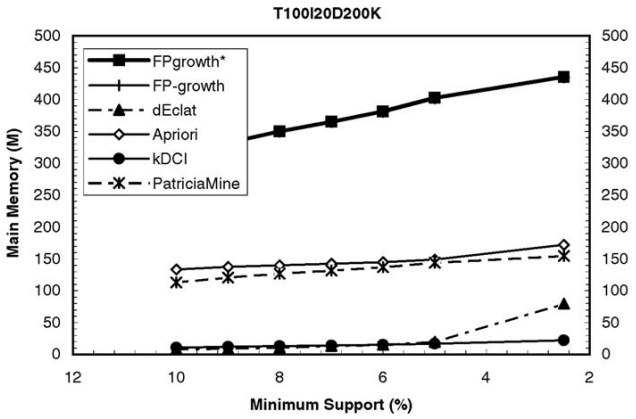

Fig 9 Memory consumption of mining of FIs on T100120D200K.

4 COMPARISON

Comparing Fig. 8 with Fig. 9, we also can see that FP-growth* and the FP-growth method still have good speed
even when they have to construct big FP-trees. We also can see from the figures that *PatriciaMine* consumes less memory than FP-growth*. This is because we implemented FP-growth* by a standard trie. On the other hand, in *PatriciaMine*, the FP-tree structure was implemented as a Patricia trie, which will save some memory.

5 SCALABILITY

Fig 10 shows the speed scalability of all algorithms on synthetic data sets. The same data sets were used in for testing the scalability of all algorithms for mining all frequent item sets. Fig 10 shows that FP-max* is also a scalable algorithm. Runtime increases almost five times when the data size increases five times. The figures also demonstrate that other algorithms have good scalability. No algorithms have exponential runtime increase when the data set size increases.

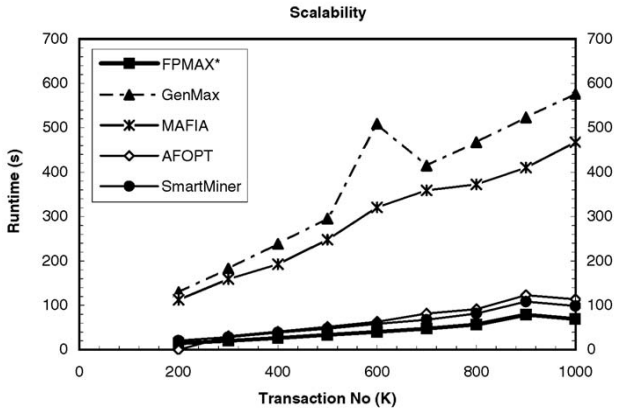

Fig 10 Scalability of runtime of mining MFIs

FP-max* possesses good scalability of memory consumption as well. Memory consumption grows from 76 megabytes to 332 megabytes when data size grows from 16 megabytes to 80 megabytes. All algorithms have similar scalability on synthetic data sets.

6 CONCLUSIONS AND FUTURE WORK

Support	Execution time of AprioriT	Execution time of DIC	Execution time of FP-Growth
50	187ms	226754ms	94ms
60	110ms	184297ms	74ms
70	78ms	161265ms	46ms
80	47ms	106953ms	32ms
90	32ms	74984ms	31ms

We introduced a Frequent Pattern -array technique which allows using frequent Pattern -trees more efficiently when mining frequent item sets. Our technique mainly reduces the time spent on traversing FP-trees, and works especially well for sparse data sets when compared to dense data . By using the FP-array technique into the FP-growth method, the FP-growth* algorithm for mining frequent item sets has been introduced. Then we have been presented some new algorithms for mining maximal and closed frequent item sets. To mine maximal frequent item sets, we have used our earlier algorithm FP-max to FP-max*. FP-max* not only uses the FP- array technique, but also an effective maximality checking approach. For the maximality testing, a variation on the FP-tree, called an MFI-tree was introduced for keeping track of all MFIs. In FP-max*, a newly found FI is always compared with a small set of

MFIs that are stored ina local MFI-tree, thus making maximality-checking very efficient. For mining closed frequent item sets we gave the FPclose algorithm. In this algorithm, a CFI-tree, another variation of the FP-tree, is used for testing the closedness of frequent item sets. For all of our algorithms we have introduced several optimizations for further reducing their running time a nd memory consumption.
Both our experimental results and the results of the independent experiments conducted by the organizers of FIMI '03 [12] show that FP-growth*, FP-max*, and FPclose are Among the best algorithms for mining frequent item sets.

The algorithms are the fastest algorithms for many cases. For sparse data sets, the algorithms need more memory than other algorithms because the FP-tree structure needs a large amount of memory in these cases. The experimental results given in this paper show the success of our algorithms, the problem that FP-growth*, FP-max* and FPclose consume lots of memory when the data sets are very sparse still needs to be solved. Consuming too much memory decreases the scalability of the algorithms, a *Patricia Trie* to implement the FP-tree data structure could be a good solution for the problem.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," Proc. ACMSIGMOD Int'l Conf. Management of Data, pp. 207-216, May 1993.

[2]R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," Proc. Int'l Conf. Very Large Data Bases, pp. 487-499, Sept. 1994.

[3] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," Proc. ACM-SIGMOD Int'l Conf. Management of Data, pp. 1-12, May 2000.

[4] J. Wang, J. Han, and J. Pei, "CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Item sets," Proc. Int'l Conf. Knowledge Discovery and Data Mining, pp. 236-245, Aug. 2003.

[5] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," Data Mining and Knowledge Discovery, vol. 8, no. 1, pp. 53-87, 2004.

[6] B Babcock, S Babu, M Datar,et al·Models and issues in datastreams systems [C]·The 21st ACM SIGACT-SIGMODSIGART Symp on Priciples of Database Systems, Madison,2002.

[7] H. Mannila, H. Toivonen, and I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," Data Mining and Knowledge Discovery, vol. 1, no. 3, pp. 259-289, 1997.

[8] M. Kamber, J. Han, and J. Chiang, "Meta rule-Guided Mining of Multi-Dimensional Association Rules Using Data Cubes," Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining, pp. 207-210, Aug. 1997.

[9] D. Xin, J. Han, X. Li, and B.W. Wah, "Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration," Proc. Int'l Conf. Very Large Data Bases, pp. 476-487, Sept. 2003.

[10] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Item set Algorithm for Transactional Databases," Proc. Int'l Conf. Data Eng., pp. 443-452, Apr. 2001.

[11] J. Pei, J. Han, and R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Item sets," Proc. ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery, pp. 21-30, May 2000.

[12] Bart Goethals FIMI: Workshop on Frequent Item set Mining Implementations, The 3rd IEEE International conference Melbourne, FL, Nov 2003

[13] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri,"kDCI: A Multi-Strategy Algorithm for Mining Frequent Sets," Proc. IEEE ICDM Workshop Frequent Item set Mining Implementations, CEUR Workshop Proc., vol. 80, Nov. 2003.

[15] M.J. Zaki and K. Gouda, "Fast Vertical Mining Using Diffsets," Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining, pp. 326-335, Aug. 2003.

[16] A. Pietracaprina and D. Zandolin, "Mining Frequent Item sets Using Patricia Tries," Proc. IEEE ICDM Workshop Frequent Item set Mining Implementations, CEUR Workshop Proc., vol. 80, Nov. 2003.