Trends in Applied Sciences Research

ISSN 1819-3579

Academic
Journals Inc.

www.academicjournals.com

Trends in Applied Sciences Research 1 (4): 402-415, 2006
ISSN 1819-3579

MAXFP: A Multi-strategy Algorithm for Mining Maximum Frequent Patterns and Their Support Counts

R.S. Thakur, R.C. Jain and K.R. Pardasani
[1]Department of Master in Computer Application, UIT, RGPV, Bhopal (MP), India
[2]Department of Master in Computer Application, SATI, Vidisha (MP), India
[3]Department of Mathematics, Maulana Azad National Institute of Technology, Bhopal (MP), India

Abstract: The problem of efficiency is the main crux of most data mining problems, such as mining frequent patterns. This problem is mainly concerned with number of operations required for counting pattern supports in the large database. In this study we propose a Multi-strategy based new algorithm, which combines Pincer-search and counting Inference approaches for discovering maximum frequent patterns along with support count of all their subsets. This algorithm works in both directions, bottom-up as well as top-down. The main search direction is still bottom-up but a restricted search is conducted in the top-down direction. The important characteristic of the algorithm is that, it is not necessary to explicitly examine every frequent itemset. Counting inference allows us to perform as few support counts as possible. Using this method, the support of a pattern is determined without accessing the database whenever possible using the supports of some of its sub-patterns called key patterns. MAXFP method performs well even when some maximal frequent itemsets are long. It reduces cost of the frequent itemsets discovery process, that is minimizes support count operations as well as database scans and it count support of all frequent patterns which are generated by maximum frequent patterns, without accessing database.

Key words: Pattern counting inference, key patterns, maximal frequent itemsets

Introduction

Association rule mining finds interesting association among a large set of data items. With massive amount of data continuously being collected and stored. Many industries are becoming interested in mining association rules from their databases. The discovery of interesting association relationship among huge amount of business transaction records can help in many business decision making process, such as catalogue design, cross marketing and loss leader analysis.

The problem of mining frequent patterns first arose as a sub-problem of mining association rules (Agrawal and Srikant, 1994). A frequent pattern is a set of binary attributes (items) which support, i.e., the number of objects in the database containing it, is at least equal to a minimum support threshold minsup defined by the user. It then was discovered that frequent patterns are involved in a variety of problems (Han *et al*., 2000): mining sequential patterns (Agrawal and Srikant, 1995), episodes (Mannila *et al*., 1997), correlations (Brin *et al*., 1997; Silverstein *et al*., 1998), multi-dimensional patterns (Kamber *et al*., 1997; Lent *et al*., 1997), maximal patterns (Lin and Kedem 1998; Bayardo 1998; Zaki *et al*., 1997), closed patterns (Pasquier *et al*., 1999a; Pei *et al*., 2000; Taouil *et al*., 2000).

Corresponding Author: R.S. Thakur, Department of Master in Computer Application, UIT, RGPV, Bhopal (MP), India

Typical algorithms for finding the frequent pattern, i.e., the set of all frequent itemsets (Arrawal and Srikant, 1994; Mannila *et al*., 1994), operate in a bottom-up breadth-first fashion. In other words, the computation starts from frequent 1-itemsets (minimal length frequent itemsets at the bottom) and then extends one level up in every pass until all maximal (length) frequent itemsets are discovered. All frequent itemsets are explicitly examined and discovered by these algorithms. When all maximal frequent itemsets are small, these algorithms perform reasonably well. However, performance drastically decreases when any of the maximal frequent itemsets becomes larger. This is due to the fact that a maximal frequent itemset of size l implies the presence of 2^l-2 non-trivial frequent itemsets (its nontrivial subsets) as well, each of which is explicitly examined by such algorithms.

Of course the set of the maximal (length) frequent itemsets (called maximum frequent pattern) uniquely defines the entire frequent itemsets: frequent itemsets are precisely all the non-empty subsets of its elements. Thus trivially, discovering the maximum frequent pattern implies immediate discovery of all the maximal frequent itemsets. In many situations, it suffices to know the supports of the maximal frequent itemsets and the supports of all subsets of the maximal frequent itemsets. For such situations this MAXFP (Maximum Frequent Pattern) algorithm provides efficient solution.

In this study we propose Fast MAXFP algorithm to find the maximal frequent itemsets as well as support of all its subsets. Not only the this study contains the process of finding maximum frequent patterns but also support of all frequent itemsets which are subsets of maximal frequent itemsets discovered by MAXFP algorithm. This new concept also reduces number of database passes as compared to other approaches (Lin and Kedem, 1998).

The Fast MAXFP algorithm is based on Pincer-Search (Lin and Kedem, 1998) and Counting Inference approach (Bastide *et al*., 2000). As in Pincer-Search (Lin and Kedem, 1998), our approach use MFP (maximum frequent pattern) and search MFP from both bottom-up and top-down directions. This Fast MAXFP algorithm performs well even when the maximal frequent itemsets are long.

In Fast MAXFP algorithm, the bottom-up search follows concept of pattern Counting Inference method presented in (Bastide *et al*., 2000), were the Pascal-Gen algorithm was introduced, that minimizes as much as possible the number of pattern support counts performed when extracting frequent patterns. This method relies on the concept of key patterns, where a key pattern is a minimal (with respect to set inclusion) pattern of an equivalence class gathering all patterns common to the same objects of the database relation. Hence, all patterns in an equivalence class have the same support and the supports of the non-key patterns of an equivalence class can be determined using the supports of the key patterns of this class. With pattern counting inference, only the supports of the frequent key patterns (and some infrequent ones) are determined from the database, while supports of the frequent non-key patterns are derived from those of the frequent key patterns. In this approach, as in Apriori (Agrawal and Srikant, 1994), frequent patterns are extracted in a levelwise manner: During each iteration, candidate patterns of size k are created by joining the frequent patterns of size k-1, their supports are determined and infrequent ones are discarded. Using counting inference, if a candidate pattern of size k is a non-key pattern, then its support is equal to the minimal support among the patterns of size k- 1 that are its subsets. This allows to reduce the number of patterns considered during each database pass while counting supports and even more important, to reduce the total number of passes. This optimization is valid since key patterns have a property that is compatible with the pruning of Apriori: all subsets of a key pattern are key patterns and all supersets of a non-key pattern are non-key patterns.

The top-down search is implemented efficiently by introducing a set that we call the maximum-frequent-candidate-pattern or MFCP. This set consists of the minimum number of itemsets such that the union of all their subsets contains all the known (discovered so far) frequent itemsets but not any of the known infrequent itemsets. The known frequent and infrequent itemsets are those determined by the supports of the itemsets that have been discovered until the most recent pass. Thus obviously

at any point of the algorithm MFCP is a superset of MFP. When the algorithm terminates, MFCP and MFP are equal. Unlike the bottom-up search that goes up one level in each pass, MFCP set can "move down" many levels in one pass.

In the presesnt study, we use the MFCP idea, as in (Lin and Kedem, 1998) along with counting inference (Bastide *et al*., 2000), to discover frequent itemsets in market basket data. In most cases, our algorithm not only reduces the number of passes of reading the database but also reduce the number of candidates (for whom support is counted). In such cases, both I/O time and CPU time are reduced by: i) eliminating the candidates that are subsets of maximal frequent itemsets found in MFCP. ii) using Pascal-Gen (Bastide *et al*., 2000), determine as much support count as possible without accessing the database by information gathered in previous passes.

The Problem of Mining Frequent Patterns

Definition 1

Let P be a finite set of items, O a finite set of objects (e. g., transaction ids) and $R \subseteq O \times P$ a binary relation (where $(o, p) \in R$ may be read as item p is included in transaction o). The triple $D = (O, P, R)$ is called dataset.

Each subset P of P is called a pattern. We say that a pattern P is included in an object $o \in O$ if $(o, p) \in R$ for all $p \in P$. Let f be the function which assigns to each pattern $P \subseteq P$ the set of all objects which include this pattern: $f(P) = \{o \in O \mid o \text{ includes } P\}$.

The support of a pattern P is given by $supp(P) = card(f(P))/card(O)$. For a given threshold $minsup \in (0,1)$, a pattern P is called frequent pattern or frequent itemsets if $supp(P) \geq minsup$.

The problem of association rule mining consists of two stages (Agrawal and Srikant,1994): i) the discovery of frequent itemsets, followed by ii) the generation of association rules.

The maximum frequent pattern (MFP) is the set of all the maximal frequent itemsets. (An itemset is a maximal frequent itemset if it is frequent and no proper superset of it is frequent.) Obviously, an itemset is frequent if and only if it is a subset of a maximal frequent itemset. Thus, it is necessary to discover only the maximum frequent pattern during the first stage. Of course, an algorithm for that stage may explicitly discover and store some other frequent itemsets as a necessary part of its execution-but minimizing such effort may increase efficiency. If the maximum frequent pattern is known, one can easily generate the required subsets and count their supports by reading the database once, which is quite straightforward.

It follows, that in the vast majority of cases, the discovery of the maximum frequent pattern dominates the performance of the whole process. Therefore, first we find maximum frequent patterns and then calculate support of all frequents patterns.

Key Features of the Counting Inference Approach

Here, we give the theoretical basis of Counting Inference Approach as in (Bastide *et al*., 2000). Like Apriori, Counting Inference traverses the powerset of P levelwise: At the kth iteration, the algorithm generates first all candidate k-patterns.

Definition 2

A k-pattern P is a subset of P with $card(P) = k$. A candidate k-pattern is a k-pattern where all its proper sub-patterns are frequent.

For the candidate k-patterns one database pass is used to determine their support. Then infrequent patterns are pruned.

Key Patterns

Counting Inference approach is based on the observation that patterns can be considered as equivalent if they are included in exactly the same objects. We describe this fact by the following equivalence relation θ on patterns.

Definition 3

For patterns P, $Q \subseteq$P, we let $P \ \theta \ Q$ if and only if $f(P) = f(Q)$. The set of patterns which are equivalent to a pattern P is given by $[P] = \{Q \subseteq P | \ P \ \theta \ Q\}$.

In the case of patterns P and Q with $P \ \theta \ Q$, both patterns obviously have the same support:

Lemma 1

Let P and Q be two patterns.

(i) $P \ \theta \ Q \Rightarrow \text{supp}(P) = \text{supp}(Q)$

(ii) $P \subseteq Q \wedge \text{supp}(P) = \text{supp}(Q) \Rightarrow P\theta Q$

Proof

(i) $P \ \theta \ Q \Leftrightarrow f(P) = f(Q) \Rightarrow \ \text{supp}(P) = \text{card}(f(P))/\text{card}(O) =$
$\text{card}(f(Q))/\text{card}(O) = \text{supp}(Q)$.

(ii) Since $P \subseteq Q$ and f is monotonous decreasing, we have $f(P) \supseteq f(Q)$.

$\text{supp}(P) = \text{supp}(Q)$ is equivalent to card $(f(P)) = \text{card}(f(Q))$ which implies with the former $f(P) = f(Q)$ and thus $P \ \theta \ Q$.

Hence if we knew the relation θ in advance, we would need to count the support of only one pattern in each equivalence class. Of course we do not know the relation in advance, but we can construct it step by step. Thus, we will (in general) need to determine the support of more than one pattern in each class, but not of all of them. If we already have determined the support of a pattern P in the database and pass later a pattern $Q \in [P]$, then we need not access the database for it because we know that $\text{supp}(Q) = \text{supp}(P)$. The first patterns of an equivalence class that we reach using a levelwise approach are exactly the minimal patterns in the class:

Definition 4

A pattern P is a key pattern if $P \in \min([P])$. A candidate key pattern is a pattern where all its proper sub-patterns are frequent key patterns.

Observe that all candidate key patterns are also candidate patterns.

Pattern Counting Inference

In the algorithm, we apply the pruning strategy to both candidate patterns and candidate key patterns. This is justified by the following theorem as in (Bastide *et al*., 2000).

Theorem 1

If Q is a key pattern and $P \subseteq Q$, then P is also a key pattern.

If P is not a key pattern and $P \subseteq Q$, then Q is not a key pattern either.

Proof

Let $P \subseteq Q$ and P is not a key pattern. Then, $\exists P' \in \min([P])$ with $P' \subset P$. From $f(P') = f(P)$ it follows that $f(Q) = f(Q \backslash (P \backslash P'))$. Hence Q is not minimal in $[Q]$ and thus by definition not a key pattern. (i) is a direct logical consequence of (ii).

The algorithm determines, at each iteration, the key patterns among the candidate key patterns by using (ii) of the following theorem:

Theorem 2

Let P be a pattern.

(I) Let $p \in P$. Then $P \in (P\backslash\{p\}]$ if and only if supp(P) = supp($P\backslash\{p\}$).

(ii) P is a key pattern if and only if supp(P) $\neq \min_{p\in P}$ (supp($P\backslash\{p\}$)):

Proof

(I) The if part follows from Lemma 1.(ii). The only if part is obvious.

(ii) From (i) we deduce that P is a key pattern if and only if supp(P) $\neq$ supp($P\backslash\{p\}$), for all $p\in P$. Since supp is a monotonous decreasing function, this is equivalent to (ii).

Since all candidate key patterns are also candidate patterns, when generating all candidate patterns for the next level we can at the same time determine the candidate key patterns among them. If we reach a candidate k-pattern which is not a candidate key pattern, then we already passed along at least one of the key patterns in its equivalence class in an earlier iteration. Hence we already know its support. Using the following theorem, we determine this support without accessing the database:

Theorem 3

If P is a non-key pattern, then

supp(P) = $\min_{p\in P}$(supp($P\backslash\{p\}$)):

Proof

"$\leq$" follows from the fact that supp is a monotonous decreasing function. "$\geq$": If P is not a key pattern then there exists $p\in P$ with $P \theta P\backslash\{p\}$. Hence supp($P$) = supp($P\backslash\{p\}$) $\geq$ $\min_{q\in P}$ (supp($P\backslash\{q\}$)).

Thus the database pass needs to count the supports of the candidate key patterns only.

A Fast MAXFP Algorithm for Discovering the Maximum Frequent Pattern and Support Count

We propose a Fast Multi-Strategy algorithm which combines top-down and bottom-up search. It relies on a new data structure during its execution, the maximum-frequent-candidate-pattern (MFCP), which is defined below.

Definition 5

Consider some point during the execution of an algorithm for finding MFP. Some itemsets are frequent, some infrequent and some unclassified. The maximum-frequent-candidate-pattern (MFCP) is a minimum cardinality set of itemsets such that the union of all the subsets of its elements contains all the frequent itemsets but does not contain any infrequent itemsets, that is, it is a minimum cardinality set satisfying the conditions

$$\text{FREQUENT} \subseteq \bigcup \{2^X \mid X \in \text{MFCP}\}$$
$$\text{INFREQUENT} \bigcap \{2^X \mid X \in \text{MFCP}\} = \emptyset$$

where FREQUENT and INFREQUENT, stand respectively for all frequent and infrequent itemsets.

Thus obviously at any point of the algorithm MFCP is a superset of MFP. When the algorithm terminates, MFCP and MFP are equal. The computation of our algorithm follows the bottom-up search approach. In each pass, in addition to counting supports of the candidates in the bottom-up direction by Pascal-Gen (Bastide *et al*., 2000), the algorithm also counts supports of the itemsets in MFCP: this set is adapted for the top-down search. This will help in pruning candidates, but will also require changes in candidate generation.

Consider now some pass k, during which itemsets of size k are to be classified. If some itemset that is an element of MFCP, say X of cardinality greater than k is found to be frequent in this pass, then all its subsets must be frequent. Therefore, all of its subsets of cardinality k can be pruned from the set of candidates considered in the bottom-up direction in this pass. These itemsets and their supersets will never be candidates throughout the rest of the execution, potentially improving performance. But of course, as the maximum frequent pattern is finally computed, they will not be forgotten.

Similarly, when a new infrequent itemset is found in the bottom-up direction, the algorithm will use it to update MFCP. The subsets of MFCP must not contain this infrequent itemset. By using the MFCP, we will be able to discover some maximal frequent itemsets in early passes. This is especially significant when the maximal frequent itemsets discovered in the early passes are long.

MFCP Updating Algorithm

Consider some itemset Y that has been just classified as infrequent and assume that it is a subset of some itemset that is an element of MFCP. To update MFCP, we replace X by $|Y|$ itemsets, each obtained by removing from X a single item (element) of Y. We do this for each newly discovered infrequent itemset and each of its supersets that is an element of MFCP. Formally, we have the following MFCP-gen algorithm (shown here for pass k).

Algorithm

MFCP-Gen (From (Lin and Kedem, 1998)

Input

Old MFCP and the infrequent set S_k discovered in pass k

Output

New MFCP

1. for all itemsets $s \in S_k$ begin
2. for all itemsets $m \in$ MFCP begin
3. if s is a subset of m begin
4. MFCP := MFCP\{m}
5. for all items $e \in$ itemset s
6. if $m\backslash\{e\}$ is not a subset of any itemset in the MFCP set
7. MFCP := MFCP $\cup$ $\{m\backslash\{e\}\}$
8. end
9. end
10. end
11. return MFCP

Example

Suppose {{1,2,3,4,5,6}} is the current (old) value of MFCP and two new infrequent itemsets {1,6} and {3,6} are discovered. Consider first the infrequent itemset {1,6}. Since the itemset {1,2,3,4,5,6} (element of MFCP) contains items 1 and 6, one of its subsets will be {1,6}.By removing item 1 from itemset {1,2,3,4,5,6}, we get {2,3,4,5,6} and by removing item 6 from itemset {1,2,3,4,5,6} we get {1,2,3,4,5}. After considering itemset {1,6}, MFCP becomes {{1,2,3,4,5},{2,3,4,5,6}}. Itemset {3,6} is then used to update this MFCP. Since {3,6} is a subset of {2,3,4,5,6}, two itemsets {2,3,4,5} and {2,4,5,6} are generated to replace {2,3,4,5,6}. The itemset {2,3,4,5} is a subset of the itemset {1,2,3,4,5} in the new MFCP and it will be removed from MFCP. Therefore, MFCP becomes {{1,2,3,4,5}, {2,4,5,6}}. The top-down arrows in Fig. 1 show the updates of MFCP.

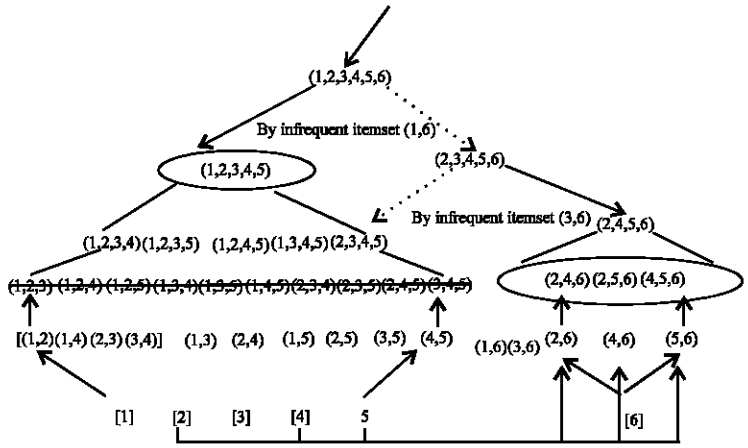

Fig. 1: Two-way search with Counting Inference approach, *Bold style represents candidates that are frequent, *Underline style represents candidates that are in frequent, *Italic style represents the candidates that are pruned by using MFCS, *Strikethrough itemsets represent the candidates that their supports are counted but will not be used to generate new candidates, *Itemsets in ellipses are the maximum frequent itemsets, *Itemsets within the [] are key pattern

A New Candidate Generation Method

Our method follows Counting Inference approach in bottom-up direction for generating candidate itemsets. for this purpose use Pascal-Gen algorithm.

Algorithm

Pascal-Gen From (Bastide *et al*., 2000) the join procedure of counting inference algorithm.

Input

L_k, the set of frequent k-patterns p with their support p.supp and the p.key flag.

Output

C_k, the set of candidate k+1-patterns c each with the flag c.key, the value c.pred_supp and the support c.supp if c is not a key pattern.

1) for i from 1 to $|L_k\text{-}1|$ begin
2) for j from i+1 to $|L_k|$ begin
3) if L_k.itemset$_i$ and L_k.itemset$_j$ have the same (k-1)-prefix
4) $C_{k+1} := C_{k+1} \cup \{L_k.\text{itemset}_i\ L_k.\text{itemset}_j\}$
5) else
6) break
7) end;
8)end;
9) forall $c \in C_{k+1}$ do begin
10) c.key := true; c.pred_supp := $+\infty$
11) forall k-subsets s of c do begin
12) if $s \notin L_k$ then
13) delete c from C_{k+1}

```
14)    else begin
15)        c.pred_supp := min(c.pred_supp, s.supp)
16)        if not s.key then c.key := false
17)    end
18) end
19) if not c.key then c.supp := c.pred_supp
20) end
21) return C_{k+1}.
```

The way Pascal-Gen operates is basically known from the generator function Apriori-Gen which was introduced in (Agrawal and Srikant, 1994). In addition to Apriori-Gen's join and prune steps, Pascal-Gen makes the new candidates inherit the fact of being or not a candidate key pattern (step 16) by using Theorem 1 and it determines at the same time the support of all non key candidate patterns (step 19) by using Theorem 3.

In working of Pascal-Gen is shown by example. For our example we use dataset in Fig. 4, the algorithm performs first one database pass to count the support of the 1-patterns (Fig. 2). As {5} has the same support as the empty set, {5} is marked as a non-key pattern and call Pascal-Gen to generate candidate pattern.

At the next iteration, all candidate 2-patterns are created and stored in C_2 (Fig. 3). At the same time, the support of all patterns containing {5} as sub-pattern is computed. Then a database pass is performed to determine the supports of the remaining ten candidate patterns:

Similarly at the third iteration, it turns out in Pascal-Gen that each newly generated candidate pattern contains at least one sub-pattern, which is not a key pattern. so their supports are determined directly in Pascal-Gen. From there, the database will be accessed only for one candidate pattern. In the fourth and fifth iteration (if needed), all supports are determined directly in Pascal-Gen.

In our algorithm, simultaneously Pascal-Gen, at each pass MFCP updates and after a maximal frequent itemset is added to MFCP, all of its subsets in the frequent set (as computed so far) will be removed. We show by example that if the Pascal-Gen join procedure is applied some of the needed itemsets could be missing from the candidate set. Consider Fig. 1. Suppose the original frequent itemset L_3 is {1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, {2,4,6}, {2,5,6}, {3,4,5}, {4,5,6}. Assume itemset {1,2,3,4,5} in MFCP is determined to be frequent. Then all 3-itemsets of the original frequent set L_3 will be removed from it by our algorithm, except for {2,4,6}, {2,5,6} and {4,5,6}. Since the join procedure uses a (k-1)-prefix test on the frequent set to generate new candidates and no two itemsets in the current frequent set {2,4,6}, {2,5,6}, {4,5,6} share a 2-prefix, no candidate will be generated by applying the join procedure on this frequent set. However, the correct candidate set should be {2,4,5,6}.

The full set of required candidates can be obtained by restoring some itemsets to the current frequent set. They will be extracted from MFCP, which implicitly maintains all frequent itemsets discovered so far. The itemsets that need to be restored are precisely those k-itemsets that have the same (k-1)-prefix as an itemset in the current frequent set.

Consider then in pass k, an itemset X in MFCP and an itemset Y in the current frequent set such that $|X|>k$. Suppose that the first k-1 items of Y are in X and the (k-1)'st item of Y is equal to the j'th item of X. We determine the k-subsets of X that have the same (k-1)-prefix as Y by taking one item of X that has an index greater than j and combining it with the first k-1 items of Y to get one of these k-subsets. After these k-itemsets are found, we generate candidates by combining them with the itemset Y. When we recovered new itemsets calculate their pred_supp, supp and key values. This is specified by the following recovery procedure.

L_1	Supp	key
{1}	3/5	t
{2}	4/5	t
{3}	4/5	t
{4}	1	t
{6}	2/5	f
		t

Fig. 2: Frequent 1-Itemsets

C_2	pred_supp	key	supp
{1,2}	3/5	t	?
{1,3}	3/5	t	?
{1,4}	3/5	t	?
{1,5}	3/5	f	3/5
{1,6}	2/5	t	?
{2,3}	4/5	t	?
{2,4}	4/5	t	?
{2,5}	4/5	f	4/5
{2,6}	2/5	t	?
{3,4}	4/5	t	?
{3,5}	4/5	f	4/5
{3,6}	2/5	t	?
{4,5}	4/5	f	4/5
{4,6}	2/5	t	?
{5,6}	2/5	f	2/5

L_2	supp	key
{1,2}	2/5	t
{1,3}	3/5	f
{1,4}	2/5	t
{1,5}	3/5	f
{2,3}	3/5	t
{2,4}	4/5	f
{2,5}	4/5	f
{2,6}	2/5	f
{3,4}	3/5	t
{3,5}	4/5	f
{4,5}	4/5	f
{4,6}	2/5	f
{5,6}	2/5	f

Fig. 3: Candidate pattern C_2 and Frequent 2-Itemsets L_2

Object	Items
T1	1,3,5
T2	2,3,4,5,6
T3	1,2,3,4,5
T4	2,4,5,6
T5	1,2,3,4,5

Fig. 4: Transaction Table

Algorithm

The recovery procedure

Input

C_{k+1}, L_k and current MFP containing the maximal frequent itemsets discover until pass k

Output

A complete candidate set C_{k+1}

1. for all itemsets l in L_k
2. for all itemsets m in MFP
3. if the first k-1 items in l are also in m
4. /* suppose $m.\text{item}_j = l.\text{item}_{k-1}$ */
5. for i from $j + 1$ to $|m|$
6. $C_{k+1} := C_{k+1} \cup \{\{l.\text{item}_1, l.\text{item}_2, ..., l.\text{item}_k, m.\text{item}_i\}\}$

9) forall new discovered itemsets $c \in C_{k+1}$ do begin
10) c.key := true; c.pred_supp := $+\infty$
11) forall k-subsets s of c do begin
12) c.pred_supp := min(c.pred_supp, s.supp)
13) if not s.key then c.key := false
14) end
15) end
16) if not c.key then c.supp := c.pred_supp
17) return C_{k+1}.

Example

Fig. 1 shows that MFCP is {{1,2,3,4,5}} and the current frequent set is {{2,4,6}, {2,5,6},{4,5,6}}. The only 3-subset of {{1,2,3,4,5}} that needs to be restored for the itemset {2,4,6} to generate a new candidate is {2,4,5}. This is because it is the only subset of {{1,2,3,4,5}} that has the same length and the same 2-prefix as the itemset {2,4,6}. By combining {2,4,5} and {2,4,6}, we recover the missing candidate {2,4,5,6}. No itemset needs to be restored for itemsets {2,5,6} and {4,5,6}. As stated earlier, our algorithm will not consider the subsets of a maximal frequent itemset as candidates. Therefore, the prune procedure in our new candidate generation algorithm will remove those subsets.

Algorithm

The new prune procedure

Input

Current MFP and C_{k+1} generated from Pascal-Gen and recovery procedures above

Output

Final candidate set C_{k+1}

1. for all itemsets c in C_{k+1}
2. if c is a subset of any itemset in current MFP
3. delete c from C_{k+1}

we eliminate the candidates that are subsets of elements of current MFP (line 3).

In summary, our candidate generation process contains the following three steps:

1.call the *Pascal-Gen* join procedure
2. call the *recovery* procedure if necessary
3. call the new *prune* procedure

The Fast MAXFP Algorithm

We now present our complete algorithm; it relies on the combined approach for determining the maximum frequent pattern.

Algorithm

The MAXFP Algorithm

Input

A database and a user-defined minimum support

Output

MFP which contains all maximal frequent itemsets

1. $\emptyset$ .supp := 1; $\emptyset$ key := true
2. L_0 := 0; k := 1
3. C_1 := {{i}| i ∈ I}
4. MFCP := {{1, 2, ….,n}}
5. MFP := 0
6. while C_k ≠0 begin
7. if k = 1 then
8. {read database and count supports for itemsets in C_k and MFCP
9. MFP := MFP ∪ {frequent itemsets in MFCP}
10. L_k := {frequent itemsets in C_k}\{subsets of itemsets in MFP }
11. for all l ∈ L_1 do l.pred-supp := 1; l.key := (l.supp ≠1)
12. }
13. else {read database and count supports for itemsets in C_k call count_support_procedure and MFCP
14. MFP := MFP ∪ {frequent itemsets in MFCP}
15. L_k :={ frequent itemsets in C_k call frequent_pattern_procedure}\{subsets of itemsets in MFP}
16. }
17. S_k := {infrequent itemsets in C_k}
18. call the Pascal-Gen to generate C_{k+1}
19. if any frequent itemset in C_k is a subset of an itemset in MFP
20. call the recovery procedure to recover candidates to C_{k+1}
21. call new prune procedure to prune candidates in C_{k+1}
22. call the MFCP-gen algorithm when S_k is not empty
23. k := k + 1
24. end
25. return MFP

The algorithm starts with the empty set, which always has a support of 1 and which is (by definition) a key pattern (line 1 and 2). In line 8, frequent 1-patterns are determined they are marked as key patterns unless their support is 1(line 11). Pascal-Gen is called to compute the candidate pattern (line 18). MFCP is initialized to contain one itemset, which consists of all the database items. MFCP is updated whenever new infrequent itemsets are found (line 22). If an itemset in MFCP is found to be frequent, then its subsets will not participate in the subsequent support counting and candidate set generation steps. Line 10,15 and line 21 will exclude those itemsets that are subsets of any itemset in the current MFP set, which contains the frequent itemsets found in the MFCP. If some itemsets in L_k are removed, the algorithm will call the recovery procedure to recover missing candidates (line 20). Line 13 call count_support_procedure for counting support of candidate itemsets(when k>1). frequent_pattern_procedure call (when k>1) for finding frequent pattern from new generated candidate itemsets(line 15).

Algorithm

The count_support_procedure

Input

C_k candidate set generated from the Pascal-Gen and recovery procedures above with each candidate c have flag c.key.

Output

Candidate set C_k in which each candidate c does count support c.supp.

1. if $\exists c \in C_k \mid c$.key then
2. forall $o \in$ D do begin
3. C_o := subset(C_k; o)
4. forall $c \in C_o \mid c$.key do
5. c.supp ++
6. end

Algorithm

The frequent_pattern_procedure

Input

C_k, the set of candidate k-patterns c each with the flag c.key, the value c.pred_supp and the support c.supp.

Output

L_k, the set of frequent k-patterns p with their support p.supp and the p.key flag.

1. forall $c \in C_k$ do
2. if c.supp $\geq$ minsup then begin
3. if c.key and c.supp = c.pred_supp then
4. c.key := false
5. $L_k := L_k \cup \{c\}$
6. end
7. end

Counting the Supports of All Frequent Itemsets

The MAXFP algorithm extracts all maximal frequent patterns from database. The non-empty subsets of these maximal frequent patterns gives all frequent patterns but in those cases when we need to know the supports of all frequent patterns, there is no efficient algorithm for counting supports of all frequent itemsets.

Most of the previous algorithms (Lin and Kedem, 1998) require extra (one more) database pass for counting supports of all frequent itemsets which is generated by maximum frequent patterns. They build hash-tree for the maximum frequent patterns, which stores all the subsets of the maximal frequent patterns. For counting support of all frequent patterns they use Apriori algorithm with minor change and add a field for storing the supports of internal nodes. The support in every node is incremented whenever the node is visited in the forward direction.

Here we introduce new concept for counting support of all frequent patterns. In our MAXFP approach, Pascal-Gen algorithm uses concept of key patterns in this case each item has three fields, key, pred-supp and supp. The supp field used for storing support of itemsets, so there is no need to add extra field as in Lin and Kedem (1998). In some cases there is no need for counting support of all frequent itemsets when all itemsets generated during extracting maximum frequent patterns. But when above mentioned cases do not occur then we will use Pascal-Gen with minor changes for counting supports of those patterns which have not occurred during extracting maximum frequent patterns. It counts support of any frequent itemset by accessing record of frequent itemsets which have been extracted by MAXFP, or its subsets support (by property of key pattern). Using concept of key pattern modified Pascal-Gen counts the support of all frequent patterns without additional database access.

Performance Evaluation

According to both (Agrawal and Srikant, 1994) and our experiments, a large fraction the 2-itemsets will usually be infrequent. These infrequent itemsets will cause MFCP to go down the levels very fast, allowing it to reach some maximal frequent itemsets after only a few passes. Indeed, in our experiments, we have found that, in most cases, many of the maximal frequent itemsets are found in MFCP in very early passes.

Running Example

We have illustrated the new algorithm on the dataset shows in Fig. 4, for minsup = 2/5:

For finding maximum frequent pattern in above dataset our new MAXFP algorithm needs three database passes in which the algorithm counts the supports of 7+11+2 = 20 patterns. Pincer (Lin and Kedem, 1998) would have needed three database passes for counting the supports of 7+16+15 = 38 patterns for the same dataset. This algorithm performs well, when bottom-up approach totally turns out in Pascal-Gen then, there is no need of accessing database for support count of candidate patterns. If at any stage MFCP is not updated the algorithm reduces the number of database passes. On the other side if bottom up approach turns out to be Pascal-Gen then no additional database scan is required. Because all frequent patterns are generated during extracting maximum frequent patterns, so there is no need for counting support of all frequents patterns.

Conclusions

The maximum frequent pattern provides a unique representation of all the frequent itemsets. In many situations, it suffices to discover the maximum frequent pattern and once it is known, all the required frequent patterns can be easily generated.

The present study, Fast MAXFP algorithm discovers the maximum frequent pattern as well as counting the supports of all frequent patterns without accessing database very efficiently. This new algorithm reduces both the number of database passes and the number of candidates considered. By using the MFCP, we will be able to discover some maximal frequent itemsets in early passes. This early discovery of the maximal frequent itemsets reduces the number of candidates and the passes of reading the database, which in turn can reduce the CPU time and I/O time. This is especially significant when the maximal frequent itemsets discovered in the early passes are long. In bottom-up direction we have used pattern counting inference approach, which is based on the notion of key patterns of equivalence classes of patterns. It allows to count from the dataset the support of some frequent patterns only, the frequent key patterns, rather than counting the support of all frequent patterns.

For counting supports of all frequent patterns, it has following advantages.

- There is no need of extra process to add new field with each itemsets.
- It does not require additional database pass.
- This algorithm can be extended with minor modifications to perform incremental mining of association rules.

References

Agrawal, R. and R. Srikant, 1994. Fast algorithms for mining association rules in large databases. Proc. VLDB Conf., pp: 478-499.

Agrawal, R. and R. Srikant, 1995. Mining sequential patterns. Proc. ICDE Conf., pp: 3-14.

Bayardo, R.J., 1998. Efficiently mining long patterns from databases. Proc. SIGMOD Conf., pp: 85-93.

Brin, S., R. Motwani and C. Silverstein, 1997. Beyond market baskets: Generalizing association rules to correlation. Proc. SIGMOD Conf., pp: 265-276.

Bastide, Y., R. Taouil, N. Pasquier, G. Stumme and L. Lakhal. 2000, Mining frequent patterns with counting inference. ACM SIGKDD Explorations Newslett, 2: 66-75.

Han, J., J. Pei and Y. Yin, 2000. Mining frequent patterns without candidate generation. Proc. SIGMOD Conf., pp: 1-12.

Kamber, M., J. Han and J.Y. Chiang, 1997. Metarule-guided mining of multi-dimensional association rules using data cubes. Proc. KDD Conf., pp: 207-210.

Lin, D. and Z.M. Kedem, 1998. Pincer-Search: A new algorithm for discovering the maximum frequent pattern. Proc. EBDT Conf., pp: 105-119.

Lent, B., A. Swami and J. Widom, 1997. Clustering association rules. Proc. ICDE Conf., pp: 220-231.

Mannila, H., H. Toivonen and A.I. Verkamo, 1994. Improved methods for finding association rules. In Proceeding of AAAI Workshop on Knowledge Discovery, pp: 181-192.

Mannila, H., H. Toivonen and A.I. Verkamo, 1997. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, 1: 259-289.

Pasquier, N., Y. Bastide, R. Taouil and L. Lakhal, 1999a. Efficient mining of association rules using closed itemset lattices. Information Sys., 24: 25-46.

Pasquier, N., Y. Bastide, R. Taouil and L. Lakhal. 1999b, Mining frequent closed itemsets for association rules. Proc. ICDT Conf., pp: 398-416.

Pei, J., J. Han and R. Mao, 2000. Closet: An efficient algorithm for mining frequent closed itemsets. Proc. ACM SIGMOD DMKD'00, pp: 21-30.

Silverstein, C., S. Brin and R. Motwani, 1998. Beyond market baskets: Generalizing association rules to dependence rules. Data Mining and Knowledge Discovery, 2: 39-68.

Taouil, R., N. Pasquier, Y. Bastide and L. Lakhal, 2000. Mining bases for association rules using closed sets. Proc. ICDE Conf., Poster, Presentation, pp: 307.

Zaki,M.J., S. Parthasarathy, M. Ogihara and W. Li,1997. New algorithms for fast discovery of association rules. Proc. KDD Conf., pp: 283-286.