Finding Minimum Representative Pattern Sets

Guimei Liu
School of Computing
National University of Singapore
liugm@comp.nus.edu.sg

Haojun Zhang
School of Computing
National University of Singapore
zhanghao@comp.nus.edu.sg

Limsoon Wong
School of Computing
National University of Singapore
wongls@comp.nus.edu.sg

ABSTRACT

Frequent pattern mining often produces an enormous number of frequent patterns, which imposes a great challenge on understanding and further analysis of the generated patterns. This calls for finding a small number of representative patterns to best approximate all other patterns. An ideal approach should 1) produce a minimum number of representative patterns; 2) restore the support of all patterns with error guarantee; and 3) have good efficiency. Few existing approaches can satisfy all the three requirements. In this paper, we develop two algorithms, MinRPset and FlexRPset, for finding minimum representative pattern sets. Both algorithms provide error guarantee. MinRPset produces the smallest solution that we can possibly have in practice under the given problem setting, and it takes a reasonable amount of time to finish. FlexRPset is developed based on MinRPset. It provides one extra parameter K to allow users to make a trade-off between result size and efficiency. Our experiment results show that MinRPset and FlexRPset produce fewer representative patterns than RPlocal—an efficient algorithm that is developed for solving the same problem. FlexRPset can be slightly faster than RPlocal when K is small.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data Mining*

Keywords

representative patterns, frequent pattern summarization

1. INTRODUCTION

Frequent pattern mining is an important problem in the data mining area. It was first introduced by Agrawal et al. in 1993 [4]. Frequent pattern mining is usually performed on a transaction database $D = \{t_1, t_2, ..., t_n\}$, where t_j is a transaction containing a set of items, $j \in [1, n]$. Let $I = \{i_1, i_2, ..., i_m\}$ be the set of distinct items appearing in D. A pattern X is a set of items in I, that is, $X \subseteq I$. If a transaction $t \in D$ contains all the items of a pattern X, then we say t supports X and t is a supporting transaction of X. Let $T(X)$ be the set of transactions in D supporting pattern X. The support of X, denoted as $supp(X)$, is defined as $|T(X)|$. If the support of a pattern X is larger than a user-specified threshold min_sup, then X is called a frequent pattern. Given a transaction database D and a minimum support threshold min_sup, the task of frequent pattern mining is to find all the frequent patterns in D with respect to min_sup.

Many efficient algorithms have been developed for mining frequent patterns [10]. Now the focus has shifted from how to efficiently mine frequent patterns to how to effectively utilize them. Frequent patterns has the anti-monotone property: if a pattern is frequent, then all of its subsets must be frequent too. On dense datasets and/or when the minimum support is low, long patterns can be frequent. All the subsets of these frequent long patterns are frequent too based on the anti-monotone property. This leads to an explosion in the number of frequent patterns. The huge quantity of patterns can easily become a bottleneck for understanding and further analyzing frequent patterns.

It has been observed that the complete set of frequent patterns often contains a lot of redundancy. Many frequent patterns have similar items and supporting transactions. It is desirable to group similar patterns together and represent them using one single pattern. Frequent closed pattern is proposed for this purpose [16]. Let X be a pattern and S be the set of patterns appearing in the same set of transactions as X, that is, $S = \{Y|T(Y) = T(X)\}$. The longest pattern in S is called a closed pattern, and all the other patterns in S are subsets of it. The closed pattern of S is selected to represent all the patterns in S. The set of frequent closed patterns is a lossless representation of the complete set of frequent patterns. That is, all the frequent patterns and their exact support can be recovered from the set of frequent closed patterns. The number of frequent closed patterns can be much smaller than the total number of frequent patterns, but it can still be tens of thousands or even more.

Frequent closed patterns group patterns supported by exactly the same set of transactions together. This condition is too restrictive. Xin et al. [25] relax this condition to further reduce pattern set size. They propose the concept of δ-covered to generalize the concept of frequent closed pattern. A pattern X_1 is δ-covered by another pattern X_2 if X_1 is a subset of X_2 and $(supp(X_1) - supp(X_2))/supp(X_1) \leq \delta$.

The goal is to find a minimum set of representative patterns that can δ-cover all frequent patterns. When δ=0, the problem corresponds to finding all frequent closed patterns. Xin et al. show that the problem can be mapped to a set cover problem. They develop two algorithms, RPglobal and RPlocal, to solve the problem. RPglobal first generates the set of patterns that can be δ-covered by each pattern, and then employs the well-known greedy algorithm [9] for the set cover problem to find representative patterns. The optimality of RPglobal is determined by the optimality of the greedy algorithm, so the solution produced by RPglobal is almost the best solution we can possibly have in practice. However, RPglobal is very time-consuming and space-consuming. It is feasible only when the number of frequent patterns is not large. RPlocal is developed based on FPClose [11]. It integrates frequent pattern mining with representative pattern finding. RPlocal is very efficient, but it produces much more representative patterns than RPglobal.

In this paper, we analyze the bottlenecks for finding a minimum representative pattern set and develop two algorithms, MinRPset and FlexRPset, to solve the problem. Algorithm MinRPset is similar to RPglobal, but it utilizes several techniques to reduce running time and memory usage. In particular, MinRPset uses a tree structure called CFP-tree [14] to store frequent patterns compactly. The CFP-tree structure also supports efficient retrieval of patterns that are δ-covered by a given pattern. Our experiment results show that MinRPset is only several times slower than RPlocal, while RPglobal is often several orders of magnitude slower than RPlocal. Algorithm FlexRPset is developed based on MinRPset. It provides one extra parameter K which allows users to make a trade-off between efficiency and the number of representative patterns selected. When $K = \infty$, FlexRPset is the same as MinRPset. With the decrease of K, FlexRPset becomes faster, but it produces more representative patterns. When K=1, FlexRPset is slightly faster than RPlocal, and it still produces fewer representative patterns than RPlocal in almost all cases.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 gives the formal problem definition. The two algorithms, MinRPset and FlexRPset, are described in Section 4 and Section 5 respectively. Experiment results are reported in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

The number of frequent patterns can be very large. Besides frequent closed patterns, several other concepts, such as generators [5], disjunction-free generators[7], δ-free sets [6], non-derivable patterns[8], maximal patterns [13], top-k frequent closed patterns [21] and redundancy-aware top-k patterns [24], have been proposed to reduce pattern set size. The number of generators is larger than that of closed patterns. Furthermore, the set of generators itself is not lossless. It requires a border to be lossless [7], so does the set of disjunction-free generators and δ-free sets. The number of non-derivable patterns can also be larger than that of closed patterns on some datasets. The number of maximal patterns is much smaller than the number of closed patterns. All frequent patterns can be recovered from maximal patterns, but their support information is lost. Another work that also ignores the support information is [3]. It selects k patterns that best cover a collection of patterns.

Frequent closed patterns preserve the exact support of all frequent patterns. In many applications, knowing the approximate support of frequent patterns is sufficient. Several approaches have been proposed to make a trade-off between pattern set size and the precision of pattern support. The work by Xin et al. [25] described in Section 1 is one such approach. Another approach proposed by Pei et al. [17] uses absolute error bound. It uses heuristic algorithms to mine a minimal condensed pattern-base, which is a superset of the maximal pattern set. All frequent patterns and their support can be restored from a condensed pattern-base with error guarantee.

Yan et al. [26] use profiles to summarize patterns. A profile consists of a master pattern, a support and a probability distribution vector which contains the probability of the items in the master pattern. The set of patterns represented by a profile are subsets of the master pattern, and their support is calculated by multiplying the support of the profile and the probability of the corresponding items. To summarize a collection of patterns using k profiles, Yan et al. partition the patterns into k clusters, and use a profile to describe each cluster. There are several drawbacks with this profile-based approach: 1) It makes contradictory assumptions. On one hand, the patterns represented by the same profile are supposed to be similar in both item composition and supporting transactions, thus the items in the same profile are expected to be strongly correlated. On the other hand, based on how the support of patterns are calculated from a profile, the items in the same profile are expected to be independent. It is hard to make a balance between the two contradicting requirements. 2) There is no error guarantee on the estimated support of patterns. 3) The proposed algorithm for generating profiles is very slow because it needs to scan the original dataset repeatedly. 4) The boundary between frequent patterns and infrequent patterns cannot be determined using profiles.

Several improvements have been made to the profile-based approach. Jin et al. [12] develop a regression-based approach to minimize restoration error. They cluster patterns based on restoration errors instead of similarity between patterns, thus their approach can achieve lower restoration error. However, there is still no error guarantee on the restored support. CP-summary [18] uses conditional independence to reduce restoration error. It adds one more component to each profile: a pattern base, and the new profile is called c-profile. The items in a c-profile are expected to be independent with respect to the pattern base. CP-summary provides error guarantee on estimated support. However, patterns of a c-profile often share little similarity, so a c-profile is not representative of its patterns any more.

Profiles can be considered as generalizations of closed patterns. Wang et al.[20] make generalization on another concise representation of frequent patterns—non-derivable patterns. They use Markov Random Field (MRF) to summarize frequent patterns. The support of a pattern is estimated from its subsets, which is similar to non-derivable patterns. Markov Random Field model is not as intuitive as profiles, and it is also expensive to learn. It does not provide error guarantee on estimated support either.

The above approaches aim to summarize frequent patterns. Mampaey et al. [15] aim to summarize data instead with a collection of non-redundant patterns. A probabilistic maximum entropy model is used in their approach.

3. PROBLEM STATEMENT

We follow the problem definition in [25]. The distance between two patterns is defined based on their supporting transaction sets.

Definition 1 ($D(X_1, X_2)$). *Given two patterns X_1 and X_2, the distance between them is defined as $D(X_1, X_2) = 1 - \frac{|T(X_1) \cap T(X_2)|}{|T(X_1) \cup T(X_2)|}$.*

Definition 2 (ϵ-covered). *Given a real number $\epsilon \in [0, 1]$ and two patterns X_1 and X_2, we say X_1 is ϵ-covered by X_2 if $X_1 \subseteq X_2$ and $D(X_1, X_2) \leq \epsilon$.*

In the above definition, condition $X_1 \subseteq X_2$ ensures that the two patterns have similar items, and condition $D(X_1, X_2) \leq \epsilon$ ensures that the two patterns have similar supporting transaction sets and similar support. Based on the definition, a pattern ϵ-covers itself.

Lemma 1. *Given two patterns X_1 and X_2, if pattern X_1 is ϵ-covered by pattern X_2 and we use $supp(X_2)$ to approximate $supp(X_1)$, then the relative error $\frac{supp(X_1)-supp(X_2)}{supp(X_1)}$ is no larger than ϵ.*

Proof. $\frac{supp(X_1)-supp(X_2)}{supp(X_1)} = 1 - \frac{supp(X_2)}{supp(X_1)} = 1 - \frac{|T(X_2)|}{|T(X_1)|} \leq 1 - \frac{|T(X_1) \cap T(X_2)|}{|T(X_1) \cup T(X_2)|} \leq \epsilon$. □

Lemma 2. *If a frequent pattern X_1 is ϵ-covered by pattern X_2, then $supp(X_2) \geq min\_sup \cdot (1 - \epsilon)$.*

Proof. Based on Lemma 1, $1 - \frac{supp(X_2)}{supp(X_1)} \leq \epsilon$, so we have $supp(X_2) \geq supp(X_1) \cdot (1 - \epsilon) \geq min\_sup \cdot (1 - \epsilon)$. □

Our goal here is to select a minimum set of patterns that can ϵ-cover all the frequent patterns. The selected patterns are called representative patterns. Based on Lemma 1, the restoration error of all frequent patterns is bounded by ϵ. We do not require representative patterns to be frequent. Based on Lemma 2, the support of representative patterns must be no less than $min\_sup \cdot (1-\epsilon)$. The problem is how to find a minimum representative pattern set? In the next two sections, we describe two algorithms to solve the problem.

4. THE MINRPSET ALGORITHM

Let $\mathcal{F}$ be the set of frequent patterns in a dataset D with respect to threshold min_sup, and $\hat{\mathcal{F}}$ be the set of patterns with support no less than $min\_sup \cdot (1 - \epsilon)$ in D. Obviously, $\mathcal{F} \subseteq \hat{\mathcal{F}}$. Given a pattern $X \in \hat{\mathcal{F}}$, we use $C(X)$ to denote the set of frequent patterns that can be ϵ-covered by X. We have $C(X) \subseteq F$. If X is frequent, we have $X \in C(X)$.

A straightforward algorithm for finding a minimum representative pattern set is as follows. First we generate $C(X)$ for every pattern $X \in \hat{\mathcal{F}}$, and we get $|\hat{\mathcal{F}}|$ sets. The elements of these sets are frequent patterns in $\mathcal{F}$. Let $S = \{C(X)|X \in \hat{\mathcal{F}}\}$. Finding a minimum representative pattern set is now equivalent to finding a minimum number of sets in S that can cover all the frequent patterns in $\mathcal{F}$. This is a set cover problem, and it is NP-hard. We use the well-known greedy algorithm [9] to solve the problem, which achieves an approximation ratio of $\sum_{i=1}^{k} \frac{1}{i}$, where k is the maximal size of the sets in S. We call this simple algorithm MinRPset.

The greedy algorithm is essentially the best-possible polynomial time approximation algorithm for the set cover problem. Our experiment results have shown that it usually takes little time to finish. Generating $C(X)$s is the main bottleneck of the MinRPset algorithm when $\mathcal{F}$ and $\hat{\mathcal{F}}$ are large because we need to find $C(X)$s over a large $\mathcal{F}$ for a large number of patterns in $\hat{\mathcal{F}}$. We use the following techniques to improve the efficiency of MinRPset: 1) consider closed patterns only; 2) use a structure called CFP-tree to find $C(X)$s efficiently; and 3) use a light-weight compression technique to compress $C(X)$s.

4.1 Considering closed patterns only

A pattern is closed if it is more frequent than all of its supersets. If a pattern X_1 is non-closed, then there exists another pattern X_2 such that $X_1 \subset X_2$ and $supp(X_2) = supp(X_1)$.

Lemma 3. *Given two patterns X_1 and X_2 such that $X_1 \subseteq X_2$ and $supp(X_1) = supp(X_2)$, if X_2 is ϵ-covered by a pattern X, then X_1 must be ϵ-covered by X too.*

The above lemma directly follows from Definition 2. It implies that instead of covering all frequent patterns, we can cover frequent closed patterns only, which leads to the following lemma.

Lemma 4. *Let $\mathcal{F}$ be the set of frequent patterns in a dataset D with respect to a threshold min_sup. If a set of patterns R ϵ-covers all the frequent closed patterns in $\mathcal{F}$, then R ϵ-covers all the frequent patterns in $\mathcal{F}$.*

Lemma 5. *Given two patterns X_1 and X_2 such that $X_1 \subseteq X_2$ and $supp(X_1) = supp(X_2)$, if a pattern X is ϵ-covered by X_1, then X must be ϵ-covered by X_2 too.*

This lemma also directly follows from Definition 2. It suggests that we can use closed patterns only to cover all frequent patterns.

The number of frequent closed patterns can be orders of magnitude smaller than the total number of frequent patterns. Consider only closed patterns improves the efficiency of the MinRPset algorithm in two aspects. On one hand, it reduces the size of individual $C(X)$s since now they contain only frequent closed patterns. On the other hand, it reduces the number of patterns whose $C(X)$ needs to be generated as now we need to generate $C(X)$s for closed patterns only.

4.2 Using CFP-tree to find $C(X)$s efficiently

The CFP-tree structure is specially designed for storing and querying frequent patterns [14]. It resembles a set-enumeration tree [19]. We use an example dataset D in Table 1 to illuminate its structure. Table 2 shows all the frequent patterns in D when $min\_sup = 3$. The CFP-tree constructed from these frequent patterns is shown in Figure 1. The CFP-tree is constructed using a pattern-growth approach. The root node contains all frequent items sorted in ascending frequency order. Each item i in the root node points to a subtree, and this subtree stores all the frequent patterns discovered from item i's conditional database.

Each node in a CFP-tree is a variable-length array. If a node contains multiple entries, then each entry contains exactly one item. If a node has only one entry, then it is called a *singleton node*. Singleton nodes can contain more than one item. For example, node 2 in Figure 1 is a singleton node with two items m and a. An entry E stores several pieces of information: (1) m items ($m \geq 1$), (2) the support

Table 1: An example dataset D

TID	Transactions
1	a, c, e, f, m, p
2	b, e, v
3	a, b, f, m, p
4	d, e, f, h, p
5	a, c, d, m, v
6	a, c, h, m, s
7	a, f, m, p, u
8	a, b, d, f, g

Table 2: Frequent patterns (min_sup=3)

ID	Itemsets	ID	itemsets	ID	itemsets
1	a:6	9	ac:3	17	acm:3
2	b:3	10	af:4	18	afm:3
3	c:3	11	am:5	19	afp:3
4	d:3	12	ap:3	20	amp:3
5	e:3	13	cm:3	21	fmp:3
6	f:5	14	fm:3	22	afmp:3
7	m:5	15	fp:4		
8	p:4	16	mp:3		

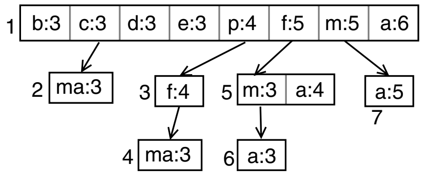


Figure 1: CFP-tree constructed on the frequent patterns in Table 2

of E, (3) a pointer pointing to the child node of E and (4) the id of the entry which is assigned using preordering. In the rest of this paper, we use *E.items*, *E.support*, *E.child* and *E.preorder* to denote the above fields.

The CFP-tree structure allows different patterns to share the storage of their prefixes as well as suffixes. Prefix sharing is easy to understand. For example, patterns $\{f, m\}$ and $\{f, a\}$ share the same prefix $\{f\}$ in Figure 1. In a multi-entry CFP-tree node, only the items after an entry E can appear in the subtree pointed by E, and these items are called *candidate extensions* of E. Suffix sharing occurs when a candidate extension of an entry E occurs in the same set of transactions as the pattern represented by E. Let i be a candidate extension of E and X be a pattern represented by E. If $supp(X) = supp(X \cup \{i\})$, then for any pattern Z, we must have $supp(X \cup Z) = supp(X \cup \{i\} \cup Z)$. In other words, X and $X \cup \{i\}$ have the same extensions. In CFP-tree, a singleton node containing item i is created to enable the sharing between X and $X \cup \{i\}$. In Figure 1, patterns $\{p\}$ and $\{p, f\}$ have the same support, so a singleton node containing item f is created, which is node 3, to allow $\{p\}$ and $\{p, f\}$ to share the same subtree.

Every entry in a CFP-tree represents one or more patterns with the same support, and these patterns contain the items on the path from the root to the entry. Items contained in singleton nodes are optional. Let E be an entry, X_m be the set of items in the multiple-entry nodes and X_s be the set of items in the singleton nodes on the path from the root to the parent of E respectively. The set of patterns represented by E is $\{X_m \cup Y \cup Z | Y \subseteq X_s, Z \subseteq E.items, Z \neq \emptyset\}$. The longest pattern represented by E is $X_m \cup X_s \cup E.items$. Let us look at an example. Node 4 contains only one entry. For this entry, we have $X_m = \{p\}$, $X_s = \{f\}$ and $E.items = \{m, a\}$. Hence node 4 represents 6 patterns: $\{p, m\}$, $\{p, a\}$, $\{p, m, a\}$, $\{p, f, m\}$, $\{p, f, a\}$ and $\{p, f, m, a\}$. We use *E.pattern* to denote the longest pattern represented by E.

The above feature makes CFP-tree a very compact structure for storing frequent patterns. The number of entries in a CFP-tree is much smaller than the total number of patterns stored in the tree. For each entry, we consider its longest pattern only based on Lemma 3 and Lemma 5. For an entry E, only its longest pattern can be closed. Other patterns of E that are shorter than the longest pattern cannot be closed based on the definition of closed patterns. If the longest pattern of an entry is not closed, then we call the entry a *non-closed entry*.

The CFP-tree structure has the following property.

PROPERTY 1. *In a multiple-entry node, the item of an entry E can appear in the subtrees pointed by entries before E, but it cannot appear in the subtrees pointed by entries after E.*

For example, in the root node of Figure 1, item p is allowed to appear in the subtrees pointed by entries b, c, d and e, but it is not allowed to appear in the subtrees pointed by entries f, m and a. This property implies the following lemma.

LEMMA 6. *In a CFP-tree, the supersets of a pattern cannot appear on the right of the pattern. They appear either on the left of the pattern or in the subtree pointed by the pattern.*

4.2.1 Finding one $C(X)$

Given a pattern X, $C(X)$ contains the subsets of X that can be ϵ-covered by X. CFP-tree supports efficient retrieval of subsets of patterns. To find the subsets of a pattern X, we simply traverse the CFP-tree and match the items of the entries against X. For an entry E in a multiple-entry node, if its item appears in X, then entry E represents some subsets of X and the search is continued on its subtree. Otherwise, entry E and its subtree is skipped because all the patterns in the subtree of E contain $E.items \notin X$, and these patterns cannot be subsets of X. An entry E in a singleton node can contain items not in X, and these items are simply ignored.

Algorithm 1 shows the pseudo-codes for retrieving $C(X)$. Initially, *cnode* is the root node of the CFP-tree. Parameter Y contains the set of items to be searched in *cnode*. It is set to X initially. Once an entry E is visited, the item of E is removed from Y when Y is passed to the subtree of E (line 8, 18). The item of E is also excluded when Y is passed to the entries after E (line 21). This is because the item of E cannot appear in the subtrees pointed by entries after E based on Property 1.

During the search of $C(X)$s, we also mark non-closed patterns. If the longest pattern of E is a proper subset of X and *E.support*=$supp(X)$, then E is marked as non-closed (line 2-3, 12-13), and it is skipped in subsequent search.

The *early termination* technique. If a pattern is ϵ-covered by X, then its support must be no larger than $\frac{supp(X)}{(1-\epsilon)}$ based on Definition 2. We use this requirement to further improve the efficiency of Algorithm 1. Given an entry E in a multiple-entry node, after we visit the subtree of E, if we find $supp(E.pattern \cup Y) > \frac{supp(X)}{(1-\epsilon)}$, where Y is the set of items that is passed to E, then there is no need

Algorithm 1 Search_CX Algorithm

```
Input:
    $cnode$ is a CFP-tree node; //$cnode$ is the root node initially.
    $Y$ is the set of items to be searched in $cnode$; //$Y$=$X$ initially.
    $supp(X)$ is the support of $X$;
Output:
    $C(X)$;
Description:
 1: if $cnode$ contains only one entry $E$ then
 2:     if $E.support == supp(X)$ AND $E.pattern \subset X$ then
 3:         Mark $E$ as non-closed;
 4:     if $E$ is not marked as non-closed AND $E$ is frequent then
 5:         if $E.items \bigcap Y \neq \emptyset$ AND $E.support \leq \frac{supp(X)}{(1-\epsilon)}$ then
 6:             Put $E.preorder$ into $C(X)$;
 7:         if $E.child \neq NULL$ AND $Y - E.items \neq \emptyset$ then
 8:             Search_CX($E.child$, $Y - E.items$, $supp(X)$);
 9: else if $cnode$ contains multiple entries then
10:     for each entry $E \in cnode$ from left to right do
11:         if $E.items \in Y$ AND $E$ is frequent then
12:             if $E.support == supp(X)$ AND $E.pattern \subset X$ then
13:                 Mark $E$ as non-closed;
14:             if $E$ is not marked as non-closed then
15:                 if $E.support \leq \frac{supp(X)}{(1-\epsilon)}$ then
16:                     Put $E.preorder$ into $C(X)$;
17:                 if $E.child \neq NULL$ AND $Y - E.items \neq \emptyset$ then
18:                     Search_CX($E.child$, $Y - E.items$, $supp(X)$);
19:                 if $supp(E.pattern \cup Y) > \frac{supp(X)}{(1-\epsilon)}$ then
20:                     return ;
21:                 $Y$=$Y - E.items$;
```

to visit the subtrees pointed by entries after E (line 19-20). The reason being that all the subsets of X in these subtrees must be subsets of $(E.pattern \cup Y)$, and their support must be larger than $\frac{supp(X)}{(1-\epsilon)}$ too based on the anti-monotone property. We call this pruning technique *early termination.*

4.2.2 *Finding $C(X)$s of all closed patterns*

Algorithm 2 shows the pseudo-codes for generating all $C(X)$s. It traverses the CFP-tree in depth-first order from left to right. Using this traversal order, the supersets of a pattern X that are on the left of X are visited before X. If the support of X is the same as one of these supersets, then X should be marked as non-closed when *Search_CX* is called for that superset. If X is not marked as non-closed when X is visited, it means that X is more frequent than all its supersets on its left. Based on Lemma 6, the supersets of a pattern appear either on the left of the pattern or in the subtree pointed by the pattern. If X is also more frequent than its child entries, then X must be closed. The conditions listed at line 2 and line 3 ensure that Algorithm 2 generates $C(X)$s for only closed patterns.

In Algorithm 2, if an entry E is marked as non-closed because it has the same support as one of its supersets on its left, then all the patterns in the subtree pointed by E are non-closed. We can safely skip E and its subtree in subsequent traversal (line 2). The same pruning is done in Algorithm 1 (line 4, 14). This pruning technique has been used in almost all frequent closed pattern mining algorithms to prune non-closed patterns [11, 22].

4.3 Compressing $C(X)$s

In a CFP-tree, each entry E has an id, which is denoted as $E.preorder$. In Algorithm 1, we put the id of entry E into

Algorithm 2 DFS_Search_CXs Algorithm

```
Input:
    $cnode$ is a CFP-tree node; //$cnode$ is the root node initially.
Output:
    $C(X)$s;
Description:
 1: for each entry $E \in cnode$ from left to right do
 2:     if $E$ is not marked as non-closed then
 3:         if $E$ is more frequent than its child entries then
 4:             $X$=$E.pattern$;
 5:             $C(X)$ = Search_CX($root$, $X$, $E.support$);
 6:         if $E.child \neq NULL$ then
 7:             DFS_Search_CXs($E.child$);
```

$C(X)$ if E is ϵ-covered by X (line 6, 16). Each id takes 4 bytes. The total number of $C(X)$s generated by Algorithm 2 grows with the number of frequent (closed) patterns. When the number of frequent closed patterns is large, the total size of $C(X)$s can be very large. If the main memory cannot accommodate all $C(X)$s, the greedy set cover algorithm becomes very slow.

To alleviate this problem, we compress $C(X)$s using a light-weight compression technique [23]. Each entry id occupies one or more bytes depending on its value. To reduce the number of bytes needed for storing entries ids, we sort the entry ids in ascending order and store the differences between consecutive ids instead. Our experiment results show that this compression technique can reduce the space needed for storing $C(X)$s by about three quarters in many cases.

Algorithm 3 MinRPset Algorithm

```
Description:
 1: Mine patterns with support $\geq min\_sup \cdot (1-\epsilon)$ and store
    them in a CFP-tree; let $root$ be the root node of the tree;
 2: DFS_Search_CXs($root$);
 3: Remove non-closed entries from $C(X)$s;
 4: Apply the greedy set cover algorithm on $C(X)$s to find representative patterns;
 5: Output representative patterns;
```

Algorithm 3 shows the pseudo-codes of MinRPset, and it calls Algorithm 2 to find $C(X)$s. Note that we store all patterns, including non-closed patterns, with support no less than $min\_sup \cdot (1-\epsilon)$ in a CFP-tree (line 1). Non-closed patterns are identified during the search of $C(X)$s. Hence it is possible that some $C(X)$s contains some non-closed entries. These non-closed entries are removed from $C(X)$s (line 3) before the greedy set cover algorithm is applied.

5. THE FLEXRPSET ALGORITHM

When the number of frequent patterns is large on a dataset, the MinRPset algorithm may become very slow since it needs to search subsets over a large CFP-tree for a large number of patterns. Furthermore, the set of $C(X)$s may become too large to fit into the main memory. To solve this problem, instead of searching $C(X)$s for all closed patterns, we can selectively generate $C(X)$s such that every frequent pattern is covered a sufficient number of times, in the hope that the greedy set cover algorithm can still find a near-optimal solution. Intuitively, the fewer the number of $C(X)$s generated, the more efficient the algorithm is. This is the basic idea of the FlexRPset algorithm.

The FlexRPset algorithm uses a parameter K to control the minimum number of times that a frequent pattern needs

Algorithm 4 Flex_Search_CXs Algorithm

```
Input:
    cnode is a CFP-tree node; //cnode is the root node initially.
    K is the minimum number of times that a frequent closed
    pattern needs to be covered;
Output:
    C(X)s;
Description:
1: for each entry $E \in cnode$ from left to right do
2:   if $E$ is not marked as non-closed then
3:     if $E.child \neq NULL$ then
4:       Flex_Search_CXs($E.child$);
5:     if $E$ is more frequent than its child entries then
6:       if ($E$ is frequent AND $E$ is covered less than $K$ times)
         OR ($\exists$ an ancestor entry $E'$ of $E$ such that $E'$ is frequent,
         $E'$ can be $\epsilon$-covered by $E$ and $E'$ is covered less
         than $K$ times) then
7:         $X$=$E.pattern$;
8:         $C(X)$ = Search_CX($root$, $X$, $E.support$);
```

to be covered. Algorithm 4 shows how FlexRPset selectively generates $C(X)s$. The other steps of FlexRPset are the same as those of MinRPset.

Algorithm 4 still uses the depth-first order to traverse a CFP-tree from left to right. It traverses the subtree of an entry E first (line 3-4) before it processes E (line 5-8), which means that when E is processed, all the supersets of E have been processed already based on Lemma 6, and E cannot be covered any more except by itself. If E is frequent and it is covered less than K times, then we generate $C(E.patterns)$ to cover E (the first condition at line 6). If E has already be covered at least K times when E is visited, then we look at the ancestor entries of E. For an ancestor entry E' of E, most of its supersets are already processed too when E is visited, hence not many remaining entries can cover E'. If E' is frequent, E' can be ϵ-covered by E and E' is covered less than K times, then we also generate $C(E.patterns)$ to cover E' (the second condition at line 6).

With the increase of parameter K, more information are gathered, hence less representative patterns are generated. However, the running time of FlexRPset becomes longer. How can users make a trade-off between running time and result size conveniently? We observe that the $C(X)$s generated at a smaller K value can be re-used at a larger K value. This leads to the incremental FlexRPset algorithm. It starts from K=1 and works like FlexRPset. If the user is happy with the number of representative patterns generated at K=1, then it stops. Otherwise, it increases K to 10 and generates $C(X)$s in a way similar to Algorithm 4. The only difference is that the set of $C(X)$s generated at K=1 are taken into consideration. During the traversal of the CFP-tree, the times that an entry E is covered is decided not only by the $C(X)$s generated in the current traversal of the CFP-tree, but also by the $C(X)$s generated in previous traversal. This process is repeated until either users are happy with the number of representative patterns, or K has reached a certain limit. In every iteration, we increase K by 10 times.

6. EXPERIMENTS

In this section, we study the performance of our algorithms. The experiments were conducted on a PC with 2.33Ghz Intel Duo Core CPU and 3.25GB memory. Our algorithms were implemented using C++. We downloaded the source codes of RPlocal from the IlliMine system package [2]. All source codes were compiled using Microsoft Visual Studio 2005.

6.1 Datasets

The datasets used in the experiments are shown in Table 3. They are obtained from the FIMI repository [1]. Table 3 shows the number of transactions (#trans) and items (#items), the maximal and average length of transactions (MaxTL, AvgTL) on the datasets.

Table 3: Datasets

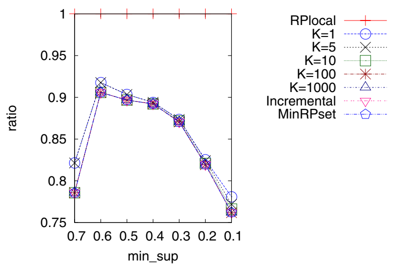
dataset	#trans	#items	MaxTL	AvgTL
accidents	340183	468	52	33.81
chess	3196	75	37	37.00
connect	67557	129	43	43.00
mushroom	8124	119	23	23.00
pumsb	49046	2113	74	74.00
pumsb_star	49046	2088	63	50.48

6.2 Comparison with RPlocal

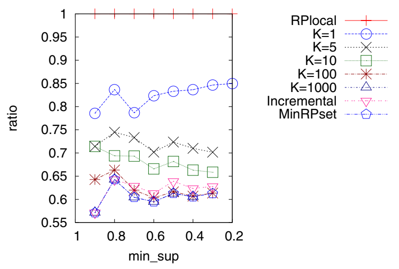
The first experiment compares MinRPset and FlexRPset with RPlocal. Let N be the number of representative patterns generated by an algorithm. Figure 2 shows the ratio of N to the number of representative patterns generated by RPlocal when min_sup is varied. Obviously, the ratio is always 1 for RPlocal. ϵ is set to 0.2 on mushroom and to 0.1 on other datasets. In [25], the authors have shown that the number of representative patterns selected by RPlocal can be orders of magnitude smaller than the number of frequent closed patterns. MinRPset further reduces the number of representative patterns by 5%-65%. The FlexRPset algorithm generates a similar number of representative patterns with MinRPset when K is large. When K gets smaller, the number of representative patterns generated by FlexRPset increases. When K=1, FlexRPset still generates less representative patterns than RPlocal in most of the cases.

Figure 3 shows the running time of the several algorithms when min_sup is varied. The running time of MinRPset and FlexRPset includes the time for mining frequent patterns. The running time of all algorithms increases with the descrease of min_sup. MinRPset has similar running time with RPlocal on mushroom. On accidents and pumsb_star, when min_sup is relatively high, the running time of MinRPset and RPlocal is similar too. In other cases except for dataset pumsb, MinRPset is several times slower than RPlocal. RPglobal is often hundreds of times slower than RPlocal as shown in [25]. This indicates the techniques used in MinRPset is very effective in reducing running time. On pumsb, MinRPset is more than 10 times slower than RPlocal when $min\_sup \leq 0.7$, but it achieves the greatest reduction in the number of representative patterns on this dataset.

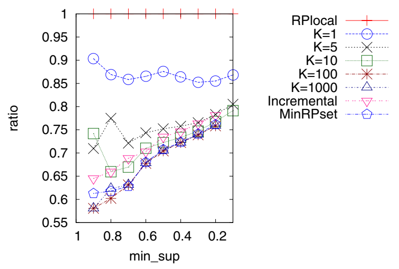
FlexRPset has similar running time with RPlocal when K is small. When K increases, FlexRPset becomes slower. For the incremental FlexRPset algorithm, we set the maximum value of K to 1000. The incremental algorithm tries 4 values of K incrementally: 1, 10, 100 and 1000. Figure 3 shows that the running time of the incremental algorithm (denoted as "incremental" in the figures) is slightly longer than that of FlexRPset with K=1000. The reason being that the most costly steps in FlexRPset are constructing the CFP-tree and generating $C(X)$s, and these two steps are shared among different K values in the incremental algorithm.



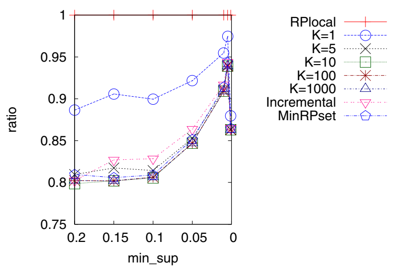
(a) accidents, ϵ=0.1



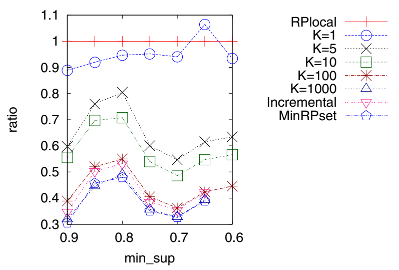
(b) chess, ϵ=0.1



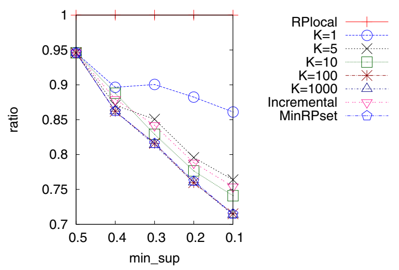
(c) connect, ϵ=0.1



(d) mushroom, ϵ=0.2

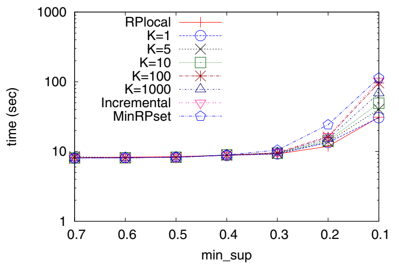


(e) pumsb, ϵ=0.1

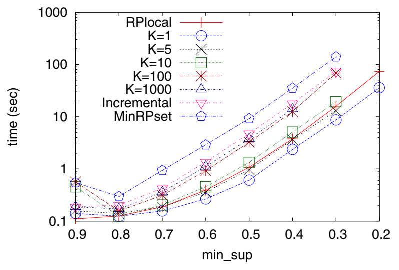


(f) pumsb_star, ϵ=0.1

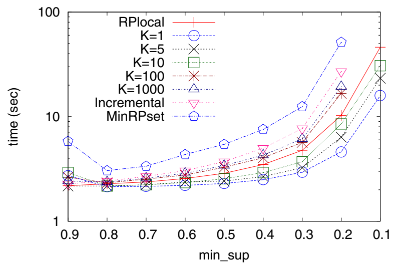
Figure 2: Number of representative patterns when varying ***min_sup***



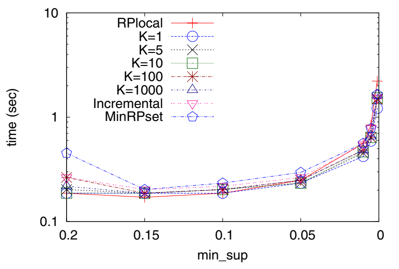
(a) accidents, ϵ=0.1



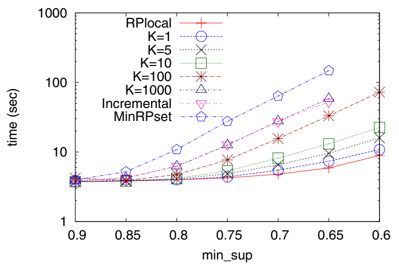
(b) chess, ϵ=0.1



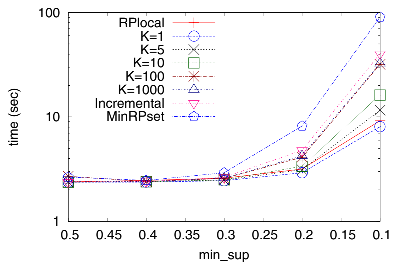
(c) connect, ϵ=0.1



(d) mushroom, ϵ=0.2



(e) pumsb, ϵ=0.1



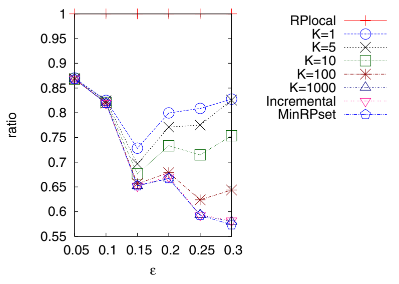
(f) pumsb_star, ϵ=0.1

Figure 3: Running time when varying ***min_sup***

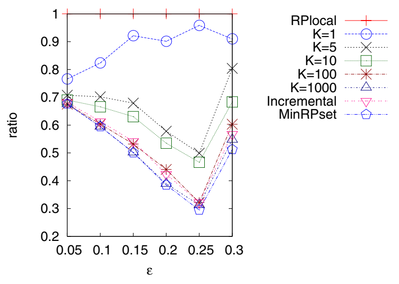
Figure 4 compares the number of representative patterns generated by the several algorithms when ϵ is varied. When ϵ increases, MinRPset achieves greater reduction in the number of representative patterns. However, its running time increases quickly too as shown in Figure 5. The running time of RPlocal is relatively stable with respect to ϵ, so is the running time of FlexRPset when $K \leq 10$. When $K \geq 100$, the running time of FlexRPset increases more obviously with ϵ.

6.3 Effect of the early termination technique

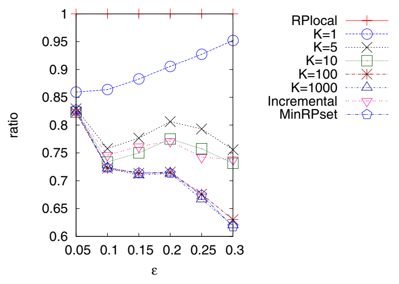
In Algorithm 1, we use an early termination technique (described at the end of Section 4.2.1) to improve the efficiency of Algorithm 1. Table 4 shows the effect of the early termination technique on the running time of MinRPset. Columns "W/O (sec)" and "With (sec)" are the running time of MinRPset without and with the early termination technique respectively. The last column is the ratio of "With (sec)" to



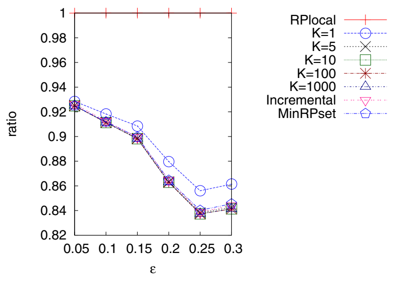
(a) accidents, *min_sup*=0.2



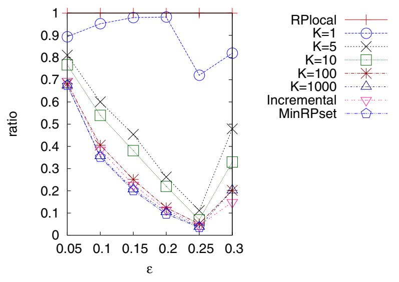
(b) chess, *min_sup*=0.6



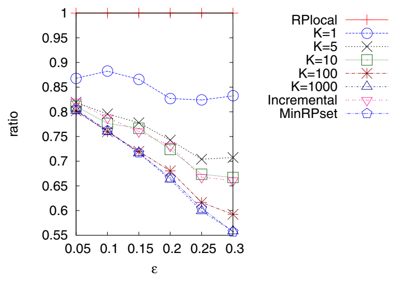
(c) connect, *min_sup*=0.4



(d) mushroom, *min_sup*=0.001

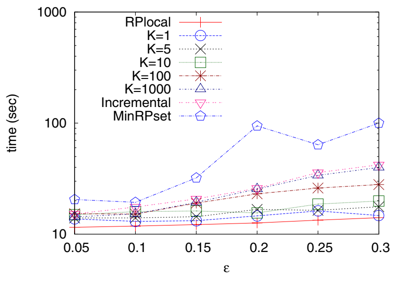


(e) pumsb, *min_sup*=0.75

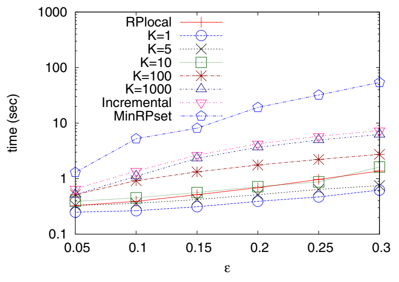


(f) pumsb_star, *min_sup*=0.2

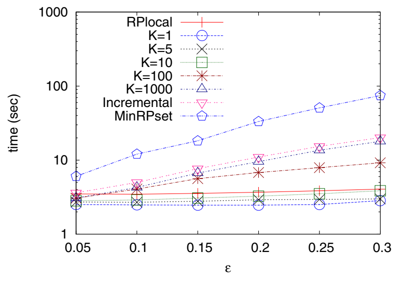
Figure 4: Number of representative patterns when varying ϵ



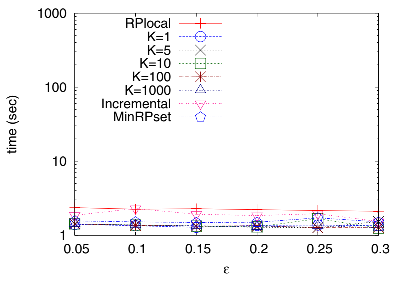
(a) accidents, *min_sup*=0.2



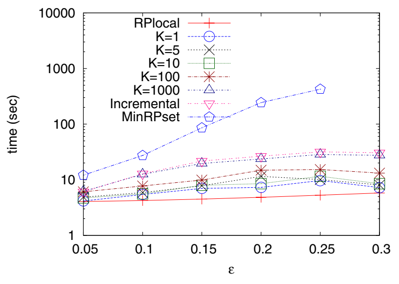
(b) chess, *min_sup*=0.6



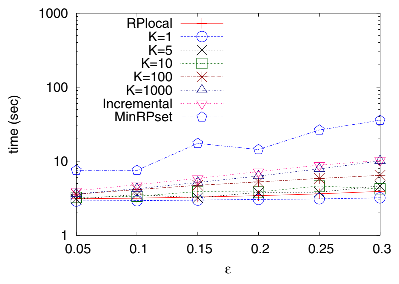
(c) connect, *min_sup*=0.4



(d) mushroom, *min_sup*=0.001



(e) pumsb, *min_sup*=0.75



(f) pumsb_star, *min_sup*=0.2

Figure 5: Running time when varying ϵ

"W/O (sec)". On dataset pumsb, the early termination technique achieves the lowest reduction in running time. This is one reason why MinRPset is more than 10 times slower than RPlocal on pumsb. On other datasets, the early termination technique can reduce the running time by 5-15 times. Similar results are observed for FlexRPset.

The early termination technique is more effective when ϵ is smaller. This is because when ϵ is smaller, fewer subsets of X can be ϵ-covered by X, and more subsets of X that do not satisfy the support constraint are pruned by the early termination technique.

7. DISCUSSION AND CONCLUSION

In this paper, we have described two algorithms, MinRPset and FlexRPset, for finding minimum representative pattern sets. Both algorithms generate less representative

Table 4: Running time of MinRPset with and without the early termination technique.

dataset	min_sup	ϵ	W/O(sec)	With(sec)	ratio
accidents	0.2	0.1	12.139	2.406	19.8%
accidents	0.2	0.05	10.280	1.640	16.0%
chess	0.3	0.1	323.964	48.107	14.8%
chess	0.3	0.05	240.312	22.392	9.3%
connect	0.2	0.1	104.444	15.014	14.4%
connect	0.2	0.05	88.492	5.625	6.4%
mushroom	0.001	0.2	3.312	0.312	9.4%
mushroom	0.001	0.1	0.281	3.266	8.6%
mushroom	0.001	0.05	0.265	3.266	8.1%
pumsb	0.6	0.1	160.670	242.33	66.3%
pumsb	0.6	0.05	34.687	106.388	32.6%
pumsb_star	0.1	0.1	109.796	24.904	22.7%
pumsb_star	0.1	0.05	88.148	13.934	15.8%

patterns than previous work RPlocal. FlexRPset takes one extra parameter K, which allows users to make a trade-off between result size and efficiency. With the increase of K, FlexRPset produces less representative patterns, but its running time increases. When K is small, FlexRPset can be slightly faster than RPlocal even though RPlocal integrates frequent pattern mining with representative pattern finding, while FlexRPset first mines frequent patterns and then finds representative patterns in a post-processing step.

Definition 2 allows a pattern to cover its subsets only. This condition allows users to restore the support of a pattern by searching the supersets of the pattern in the representative pattern set, and then using the highest support of the supersets to approximate the support of the pattern. Without this condition, it is impossible to estimate the support of a pattern as we do not know which representative pattern covers it. In MinRPset and FlexRPset, all frequent patterns are stored in a CFP-tree compactly. Users can retrieve the support of patterns from the CFP-tree directly. The set of representative patterns merely provides a concise view of all patterns. In this situation, the subset condition becomes unnecessary. We can relax Definition 2 by removing condition $X_1 \subseteq X_2$ to further reduce the number of representative patterns. This will be our future work.

8. ACKNOWLEDGMENT

This work is supported in part by Singapore Agency for Science, Technology and Research grant SERC 102 101 0030.

9. REFERENCES

[1] Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data/.

[2] Illimine system package. http://illimine.cs.uiuc.edu/download/.

[3] F. N. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent sets. In *KDD*, pages 12–19, 2004.

[4] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.

[5] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *Proc. of Computational Logic Conference*, pages 972–986, 2000.

[6] J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 7(1):5–22, 2003.

[7] A. Bykowski and C. Rigotti. A condensed representation to find frequent patterns. In *PODS*, 2001.

[8] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *PKDD*, pages 74–85, 2002.

[9] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[10] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *Proc. of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[11] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[12] R. Jin, M. Abu-Ata, Y. Xiang, and N. Ruan. Effective and efficient itemset pattern summarization: regression-based approaches. In *KDD*, pages 399–407, 2008.

[13] R. J. B. Jr. Efficiently mining long patterns from databases. In *SIGMOD Conference*, pages 85–93, 1998.

[14] G. Liu, H. Lu, and J. X. Yu. Cfp-tree: A compact disk-based structure for storing and querying frequent itemsets. *Inf. Syst.*, 32(2):295–319, 2007.

[15] M. Mampaey, N. Tatti, and J. Vreeken. Tell me what i need to know: succinctly summarizing data with itemsets. In *KDD*, pages 573–581, 2011.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.

[17] J. Pei, G. Dong, W. Zou, and J. Han. Mining condensed frequent-pattern bases. *Knowledge and Information Systems*, 6(5):570–594, 2004.

[18] A. K. Poernomo and V. Gopalkrishnan. Cp-summary: a concise representation for browsing frequent itemsets. In *KDD*, pages 687–696, 2009.

[19] R. Rymon. Search through systematic set enumeration. In *KR*, pages 539–550, 1992.

[20] C. Wang and S. Parthasarathy. Summarizing itemset patterns using probabilistic models. In *KDD*, pages 730–735, 2006.

[21] J. Wang, J. Han, Y. Lu, and P. Tzvetkov. Tfp: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Trans. Knowl. Data Eng.*, 17(5):652–664, 2005.

[22] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *KDD*, pages 236–245, 2003.

[23] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[24] D. Xin, H. Cheng, X. Yan, and J. Han. Extracting redundancy-aware top-k patterns. In *KDD*, pages 444–453, 2006.

[25] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.

[26] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *KDD*, pages 314–323, 2005.