PAPER

Mining Noise-Tolerant Frequent Closed Itemsets in Very Large Database

Junbo CHEN[†], *Member*, **Bo ZHOU**[†a)], *Nonmember*, **Xinyu WANG**[†], *Member*, **Yiqun DING**[†], *and* **Lu CHEN**[††], *Nonmembers*

SUMMARY *Frequent Itemsets(FI)* mining is a popular and important first step in analyzing datasets across a broad range of applications. There are two main problems with the traditional approach for finding frequent itemsets. Firstly, it may often derive an undesirably huge set of frequent itemsets and association rules. Secondly, it is vulnerable to noise. There are two approaches which have been proposed to address these problems individually. The first problem is addressed by the approach *Frequent Closed Itemsets(FCI)*, *FCI* removes all the redundant information from the result and makes sure there is no information loss. The second problem is addressed by the approach *Approximate Frequent Itemsets(AFI)*, *AFI* could identify and fix the noises in the datasets. Each of these two concepts has its own limitations, however, the authors find that if *FCI* and *AFI* are put together, they could help each other to overcome the limitations and amplify the advantages. The new integrated approach is termed *Noise-tolerant Frequent Closed Itemset(NFCI)*. The results of the experiments demonstrate the advantages of the new approach: (1) It is noise tolerant. (2) The number of itemsets generated would be dramatically reduced with almost no information loss except for the noise and the infrequent patterns. (3) Hence, it is both time and space efficient. (4) No redundant information is in the result.

key words: *noise-tolerant frequent closed itemsets, closed itemsets, approximate frequent itemsets, association rules*

1. Introduction

It has been well recognized that *FI* mining [13] plays an essential role in many important data mining tasks and provides the basis for deriving association rules, clustering data, and building classifiers from relational databases.

There are two main problems of the *FI* mining. Firstly, it may often derive an undesirably huge set of frequent itemsets and association rules. Secondly, it is vulnerable to noise.

The first problem is addressed by the approach named *FCI*, which is proposed by [2]. *FCI* is a condensed representation of all the frequent itemsets that guarantees no information loss. No information loss is one of the advantages of *FCI*. However, it turns out to be a defect if noise is presented. It is because *FCI* keeps all the noise information and redundant information caused by the noise: The recent theoretical results [1] state that, in the presence of even low levels of noise, large frequent itemsets are broken into fragments of logarithmic size. These small fragments are similar to each other and they would be redundant information once the large frequent itemsets were found.

The second problem is addressed by the approach termed *AFI*, which is proposed by [9]. *AFI* could identify and fix the noises in the datasets. However, the number of AFIs is even larger than the number of FIs. The redundant information contained by AFIs will lead to both time and space inefficiency.

The authors find that *FCI* and *AFI* could work together to help each other to overcome the limitations and amplify the advantages. The novel integrated approach is termed *Noise-tolerant Frequent Closed Itemset(NFCI)*. In this new approach: (1) *AFI* could fix the noise in the datasets, hence, it could help *FCI* to overcome its noise vulnerability while still amplifying the advantages of *FCI* because all the small fragments of the large frequent itemsets which caused by the noise could be identified and removed from the results as redundant information. (2) *FCI* could help *AFI* to overcome the time/space inefficiency problem because the search space is reduced dramatically (in one of our experiments, the number of generated itemsets is reduced from 4,129,839 to 14) by *FCI*, since *FCI* removes all the redundant information.

This paper proposes a novel algorithm called *Noise-tolerant Frequent Closed Itemsets Miner*. In the rest of the paper, it is referred to as *NFCIM*. The advantages of this approach include: (1) It is noise tolerant. (2) The number of itemsets generated would be dramatically reduced with almost no information loss. (3) Hence, it is both time and space efficient. (4) No redundant result will be generated.

1.1 Related Work

Two fields of the research are related to the concept *ACFI*. The *Frequent Closed Itemset* mining and the *Approximate Frequent Itemsets* mining. And as far as the authors know there is no effort that has been carried out to integrate the ideas from these two fields as this paper proposes.

1.1.1 Frequent Closed Itemsets

The concept of frequent closed itemsets mining was first proposed in [2]. Since then, extensive studies have been carried out in this area. Many fast algorithms have been proposed, they are divided into three categories:

Manuscript received July 25, 2008.
Manuscript revised March 7, 2009.

[†]The authors are with Computer College of Zhejiang University, Hangzhou, Zhejiang, 310027, China.
[††]The author is with State Street Corporation, Hangzhou, Zhejiang, 310027, China.
a) E-mail: bzhou@zju.edu.cn

DOI: 10.1587/transinf.E92.D.1523

(1) Both A-Close [2] and TITANIC [15] exploit a level-wise process to discover closed itemsets through a breadth-first search strategy. Usually these algorithms are required to scan the whole dataset many times.
(2) In contrast, CLOSET [4] and CLOSET+ [5] traverse the itemset lattice in a depth first manner. With the help of high compact data structure FP-Tree, they could achieve better performance than the first kind of algorithms. FPClose [6] is an improved algorithm of CLOSET+.
(3) CHARM [3] exploits hybrid techniques which explores both the closed itemset space and transaction space simultaneously. DCI-Closed [12], LCM [14] and CFII [11] could be considered as improvement algorithms of CHARM.

1.1.2 Approximate Frequent Itemsets

This topic is relatively new in the data mining community, and limited efforts have been carried out in this field.

Yang *et al.* [7] proposed two error-tolerant models, termed weak error-tolerant itemsets and strong error-tolerant itemsets. Jouni K. *et al.* [8] proposed to mine the dense itemsets in the presence of noise where the dense itemsets are the itemsets with a sufficiently large sub-matrix that exceeds a given density threshold of attributes present. Liu *et al.* [9] developed a general model for mining approximate frequent itemsets which controls errors of two directions in matrices formed by transactions and items. Selim *et al.* [10] proposed an algorithm that is obtained by modifying a hierarchical agglomerative clustering algorithm and takes advantage of the speed that bit operations afford.

All the proposed approximate algorithms share the same problems: (1) The number of approximate frequent itemsets is even more huge than that of the frequent itemsets. (2) The Apriori Property [13] will no longer hold for the approximate frequent itemsets, so there's no efficient search space pruning technique [†]. Hence, The performance of these algorithms is even worse than the Apriori algorithm [13] itself. So, they are not suitable for very large database either.

The rest of this paper is organized as follows. Section 2 gives the formal definition of the problems. Section 3 presents the proposed NFCIM algorithm. The performance evaluation is depicted in Sect. 4. Finally, Sect. 5 is the conclusion of this paper.

2. Problem Definition

2.1 Frequent Closed Itemset

Let $\mathcal{T}$ be the universal set of all the *Transactions*, $\mathcal{I}$ be the universal set of all the *Items*, $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{I}$ is a binary relation between $\mathcal{T}$ and $\mathcal{I}$. Then the triple $(\mathcal{T}, \mathcal{I}, \mathcal{R})$ is called a *Formal Context*. If $T \subseteq \mathcal{T}$, then T is named *tidset*; If $I \subseteq \mathcal{I}$, then I is named *itemset*.

We define two functions:

Table 1 NFCIM().

```
1:  procedure NFCIM()
2:    R' = R;
3:    do{
4:      C = all the NFCI in R'
5:      P = ∅;
6:      for any similar I1, I2 ∈ C, where I1 ⊂ I2
7:        for all (t, i) ∈ (I2 × g(I1))
8:          if ((t, i) ∉ R')
9:            add (t, i) to P
10:     R' = R' ∪ P
11:   }while(P is not empty)
12:   return C
```

$$f(T) = \{i \in \mathcal{I} \mid (t,i) \in \mathcal{R}, \forall t \in T\}$$

$$g(I) = \{t \in \mathcal{T} \mid (t,i) \in \mathcal{R}, \forall i \in I\}$$

Where T is a tidset and I is an itemset. $(t,i) \in \mathcal{R}$ if and only if transaction t contains i. $f(T)$ represents all the items which is contained by every transaction in T. $g(I)$ represents all the transactions which contains every item in I.

A pair (T, I) is called a *Formal Concept* if and only if $f(T) = I$ and $g(I) = T$. T is called the *Extent* and I is called the *Intent* of the concept (T, I).

One common representation for the formal context is a binary matrix as shown in Table 1. Rows in the matrix correspond to transactions, while columns represent various items. The binary value of each matrix entry indicates the presence (1) or the absence (0) of an item for a given transaction in $\mathcal{R}$.

Let $(I \times T)$ denote the sub-matrix in the dataset which has I as the universal set of items and T as the universal set of transactions.

Definition 1 *Let I be an item set, the support $supp(I)$ is defined as the number of transactions which contains I, that is, $supp(I) = \left\|\{t \in \mathcal{T} \mid (t,i) \in \mathcal{R}, \forall i \in I\}\right\| = \left\|g(I)\right\|$*

Definition 2 *An itemset I is said to be closed if and only if $C(I) = f(g(I)) = f \circ g(I) = I$ where the composite function $C = f \circ g$ is called a Galois operator or a closure operator.*

Definition 3 *A closed item set is called frequent if and only if the support of it exceeds a given threshold S*

2.2 Noise-tolerant Frequent Closed Itemset

Definition 4 *Let $\epsilon_c, \epsilon_r \in [0,1]$, $T \subseteq \mathcal{T}$ is a tidset, $I \subseteq \mathcal{I}$ is an itemset. $\mathcal{R}(t,i) = 1$ if and only if $(t,i) \in \mathcal{R}$.*
We say that the sub-matrix $(I \times T)$ satisfies ϵ_r condition if and only if:

$$\forall t \in T, \frac{1}{I} \sum_{i \in I} \mathcal{R}(t,i) \geq (1 - \epsilon_r)$$

We say that the sub-matrix $(I \times T)$ satisfies ϵ_c condition if

[†]Although [9] proposed an approximate Apriori property. However, it's not efficient enough, the search space is till very large.

and only if:

$$\forall i \in I, \frac{1}{T}\sum_{t\in T}\mathcal{R}(t,i) \geq (1-\epsilon_c)$$

ϵ_c and ϵ_r are called *slack parameters*.

Definition 5 *Itemset $I \subseteq \mathcal{I}$ is called approximate frequent itemset(AFI) if there exists a tidset $T \subseteq \mathcal{T}$ with $\|T\| > S$ such that $(I \times T)$ follow both ϵ_r condition and ϵ_c condition.*

Definition 4 and 5 are cited from [9].

Definition 6 *If $\exists I_1, I_2$, satisfies:*
(1) I_1, I_2 are frequent closed itemsets.
(2) $I_1 \subset I_2$.
(3) The sub-matrix $(I_2 \times g(I_1))$ follows the ϵ_r, ϵ_c conditions in Definition 4.
*Then, I_1 and I_2 are called **similar frequent closed itemsets**.*

Similar frequent closed itemsets is referred to as **similar FCIs** in the rest of the paper.

Definition 7
*(1) Let I_1, I_2 be similar FCIs, then all the '0' entries in the sub-matrix $(I_2 \times g(I_1))$ are called **Noise Elements**.*
*(2) The **Noise Matrix**, denoted as $\mathcal{P}$, is defined as $\mathcal{P}(t,i) = 1$ if and only if (t,i) is a noise element.*
*(3) The **Fixed Matrix**, denoted as $\mathcal{R}'$, is defined as $\mathcal{R}' = R \cup \mathcal{P}$, that is $\mathcal{R}'(t,i) = 1$ if and only if $\mathcal{R}(t,i) = 1$ or $\mathcal{P}(t,i) = 1$.*

The functions $f()$ and $g()$ could be redefined with two parameters, the second parameter $\mathcal{R}^*$ could be any dataset:

Definition 8

$$f'(T,\mathcal{R}^*) = \{i \in \mathcal{I} \mid (t,i) \in \mathcal{R}^*, \forall t \in T\}$$
$$g'(I,\mathcal{R}^*) = \{t \in \mathcal{T} \mid (t,i) \in \mathcal{R}^*, \forall i \in I\}$$

It is easy to see that the original version of $f()$ and $g()$ are special cases of $f'()$ and $g'()$ since

$$f(T) = f'(T,\mathcal{R})$$
$$g(I) = g'(I,\mathcal{R})$$

That is, $f()$ and $g()$ could only be applied to the original dataset $\mathcal{R}$, and the new version $f'()$ and $g'()$ could also be applied to the fixed matrix $\mathcal{R}'$.

Definition 9 *An itemset I is said to be a **Noise-tolerant Frequent Closed Itemset(NFCI)** if and only if $C'(I,\mathcal{R}') = f'(g'(I,\mathcal{R}'),\mathcal{R}') = f' \circ g'(I,\mathcal{R}') = I$ and the support of I exceeds the threshold S.*

According to definition 9, NFCIs are simply FCIs in the fixed matrix $\mathcal{R}'$. So, we could define *Similar NFCIs* just like the *Similar FCIs*:

Definition 10 *If $\exists I_1, I_2$, satisfies:*
(1) I_1, I_2 are noise-tolerant frequent closed itemsets.
(2) $I_1 \subset I_2$.
(3) The sub-matrix $(I_2 \times g'(I_1,\mathcal{R}'))$ follows the ϵ_r, ϵ_c conditions.
Then, I_1 and I_2 are called similar noise-tolerant frequent closed itemsets.

2.3 The Main Structure of the Algorithm

Initially, $\mathcal{P}$ is empty, so $\mathcal{R}' = \mathcal{R} \cup \mathcal{P} = \mathcal{R}$, after that, $\mathcal{P}$ and $\mathcal{R}'$ will be updated iteratively until there is no similar (N)FCIs could be found. It could be recognized that NFCIs are actually FCIs in dataset $\mathcal{R}'$. So, we may refer to "NFCI" as "FCI" without explicit declarations in the rest of this paper.

This idea is expressed in the pseudo code in Table 4. Actually, it is an intuitive implementation of NFCIM. The improved algorithm will be given in Sect. 3.

Explanations about Table 4:

- $\mathcal{P}$ is initially empty. So, in line 2, the algorithm set $\mathcal{R}' = \mathcal{R}$.
- In line 4, the algorithm find all the NFCIs in $\mathcal{R}'$, and put them to the set C.
  This procedure could be implemented by any "exact" FCI mining algorithm which satisfies the following two conditions: Firstly, the FCIs should be well-organized by the algorithm; Secondly, the algorithm should explore the item set space and transaction set space simultaneously, so that NFCIM could measure ϵ_c, ϵ_r to find similar (N)FCIs. The algorithm chosen to produce all the "exact" FCIs is termed *BaseMiner*.
  In this paper, CAHRM [†] is chosen as the BaseMiner because it's simple enough to avoid wasting too much time talking about the BaseMiner itself. However, the simplicity is only an optional condition, and it's easy to replace CHARM with more complicated algorithm like LCM [14], DCI-Close [12], FCII [11] etc.
- In line 5-9. If there's any similar NFCIs in C, the algorithm tries to merge them and put all the missing entries in the sub-matrix $(I_2 \times g(I_1))$ to $\mathcal{P}$. For instance, in the running example in the next section, the algorithm will find that $(bef \times 2345)$ and $(abef \times 345)$ are similar. so the missing entry $(2,a)$ will be added to $\mathcal{P}$.
- If there's any similar NFCIs found, $\mathcal{R}'$ will be updated in line 10, and the next iteration will begin. If there's no similar NFCIs found, the iteration will break in line 11, and the NFCIs in C are all the itemsets the algorithm is looking for.

2.4 Running Example

The essential concept of the proposed algorithm is *Similar (N)FCIs* defined in Definition 6 and Definition 10. They are actually the small fragments of the large frequent itemsets broken by the noise. Table 1 shows a running example. Here it's used to illustrate what the *similar (N)FCIs* looks like.

$(abef \times 345)$ and $(bef \times 2345)$ are two closed itemsets and corresponding tidsets in the synthetic dataset. If

[†]Refer to Sect. 3.1.

Table 2 Running Example.

	a	b	c	d	e	f
1	1		1	1	1	
2		1	1	1	1	1
3	1	1	1		1	1
4	1	1		1	1	1
5	1	1	1	1	1	1
6			1	1		1
7			1	1	1	
8				1	1	1
9						

Table 3 Sub Matrix.

	a	b	e	f
2		1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1

Table 4 Number of itemsets in running example.

Support	#FI	#FCI	#AFI	#NFCI
3	38	21	64	2

we zoom in to see the sub-matrix that only contains these itemsets, then we have Table 2.

From Table 2, it could be found that $(abef \times 345)$ and $(bef \times 2345)$ are similar intuitively. Because they are largely overlapped. Also, they follows the conditions given in Definition 6, hence, they are similar FCIs. In this way, the entry $(2, a)$ is detected as a **noise element** and should be added to the **noise matrix** $\mathcal{P}$. As a result, the **fixed matrix** $\mathcal{R}' = \mathcal{R} \cup \mathcal{P}$ contains the entry $(2, a)$ and the two "exact" closed itemsets, $(abef \times 345)$ and $(bef \times 2345)$, could be merged to generate a **noise-tolerant frequent closed itemset** and its corresponding tidset $(abef \times 2345)$.

To illustrate how many itemsets could be reduced, Table 3 lists the numbers of Frequent Itemsets (#FI), Frequent Closed Itemsets (#FCI), Approximate Frequent Itemsets (#AFI) and Noise-tolerant Frequent Closed Itemsets (#NFCI) in the synthetic dataset if the support threshold is 3 and the slack parameters of AFI and NFCI are set as $\epsilon_r = 0.4, \epsilon_c = 0.3$.

3. NFCIM

One of the biggest problem of approximate frequent itemsets mining is that the Apriori Property is no longer hold. So there is no efficient search space pruning technique available. The novel concept of *Noise-tolerant Frequent Closed Itemsets* deals with the problem in a quite naturally way. By the definition, the process of finding NFCIs requires no information from any infrequent closed itemsets. So, any NFCI which has support less than the threshold, S, could be pruned safely. That is, Apriori Property still hold for noise-tolerant frequent closed itemsets mining.

Since NFCIM does not store any infrequent FCI, it is possible that the algorithm could lose some chances to discover "noise elements". However the "missed" noise elements could be discovered in the following two ways:

(1) The "missed" noise elements could be found by the "side-effect" of other similar frequent closed itemsets. For instance, in the running example, $(abdef \times 45)$ is a closed itemset (and its corresponding transaction set), $(abef \times 345)$ is another closed itemset. The sub-matrix $(abdef \times 345)$ follows both the ϵ_c and ϵ_r condition, so $(d, 3)$ should be discovered as a noise element. Because the minimum support in the running example is "3", and "abdef" is pruned by the BaseMiner, hence, the chance of detecting $(d, 3)$ as a noise element is lost. However, $(d, 3)$ is not really "missed" because it is discovered by another pair of similar frequent closed itemsets $(def \times 2458)$ and $(ef \times 23458)$. We call this as "side effects" of the similar FCIs $(def \times 2458)$ and $(ef \times 23458)$.
Based on the above observations, we propose the following hypothesis: The "side effects" could find a large portion of the "missed" noise elements. This hypothesis could be verified in the experiment shown in Sect. 4.2.

(2) The "missed" noise elements could be discovered by set a smaller "minimum support" threshold. For instance, if we set "2" as the minimum support in the running example, then, $(abdef \times 45)$ and $(abef \times 345)$ will be found as similar frequent closed itemsets and $(d, 3)$ could be detected as noise element.
If the user is not satisfied with the quality of the output, then (s)he could simply set a little bit smaller threshold to detect more "noise elements".

3.1 CHARM

From the definitions given in Sect. 2, it could be concluded that when mining NFCI, the algorithm requires the information from both the tidset space and the itemset space. CHARM [3] is a well-known algorithm which explores these two dimensions simultaneously, so we choose CHARM as the BaseMiner of NFCIM.

This section gives a short introduction about CHARM. The CHARM algorithm is given in Table 5, Table 6 and Table 7.

- CHARM assumes all the items are ordered according to a total order f.
- CHARM provides a data structure called IT-tree. Each node in the tree is represented by an itemset-tidset pair, $(I \times T)$, where $T = g(I)$.
- The root of the IT-tree is initialized as $(C(\emptyset) \times \mathcal{T})$, and every item i is added to *root* as a child in the ascending order of f. As in Table 5, line 2-5.
- The input of the procedure process(*node*) is a node in the IT-tree, the children of which are the candidate generators [12] of closed itemsets based on it. In this procedure, the algorithm tries to find the closed itemset generated from each of the children of *node*.

Table 5 CHARM().

```
1:  procedure CHARM()
2:    root = (C(∅) × T);
3:    for all i ∈ I (in order of f)
4:      if(||i|| ≥ S)
5:        add (i × g(i)) as a child of root;
6:    process(root);
```

Table 6 process(*node*).

```
1:  process(node)
2:    for all child of node, denoted as ch
3:      if(ch is a duplicated node)
4:        delete ch;
5:        return;
6:      for all right sibling of ch, denoted as rs
7:        check(node, ch, rs);
8:      if(∃ child under ch)
9:        process(ch)
```

Table 7 check(*node*, *ls*, *rs*).

```
1:  check(node, ls, rs)
2:    T = ls.T ∩ rs.T
3:    if(||T|| < S)
4:      return;
5:    if(ls.T = rs.T)
6:      ls.I = ls.I ∪ rs.I;
7:      delete rs;
8:      return;
9:    else if(ls.T ⊂ rs.T)
10:     ls.I = ls.I ∪ rs.I;
11:     return;
12:   else if(ls.T ⊃ rs.T)
13:     delete rs;
14:     add (ls.I ∪ rs.I × T) as a child of ls;
15:     return;
16:   else
17:     add (ls.I ∪ rs.I × T) as a child of ls;
18:     return;
```

- In line 3 in Table 6, it checks whether *ch* is duplicated. If a node n_1 is duplicated, that means $\exists n_2 \in$ IT-tree, which satisfies that $n_2.I \supset n_1.I$ and $n_2.T = n_1.T$, which leads to the conclusion that $n_1.I$ is not a closed itemset. So, a duplicated node could be deleted from the tree safely. Refer to [3] to see the details of how to detect the duplicated nodes.
- In line 6-7 in Table 6, the algorithm tries to find the closed itemset generated from $ch.I$, meanwhile pruning the search space in IT-tree. The procedure check() is supported by Theorem 1 given bellow, which is proved in [3].
- There're two kinds of nodes in the IT-tree. One of them has been checked by the code in line 6-7 in Table 6. The itemset of this kind of node is proved to be a closed itemset. The other kind of nodes has not been checked yet, so the itemset of this kind of node is only a generator itemset which is not guaranteed to be a closed itemset. We call the second kind of nodes **Shadow Nodes**, since they are not proved to be a closed itemset yet.

Theorem 1 *Let l, r be two child nodes under the same parent, and l is the left sibling of r. The following four properties* † *hold*

(1) If $l.g(I) = r.g(I)$, then $C(l.I) = C(r.I) = C(l.I \cup r.I)$
(2) If $l.g(I) \subset r.g(I)$, then $C(l.I) \neq C(r.I)$, but $C(l.I) = C(l.I \cup r.I)$
(3) If $l.g(I) \supset r.g(I)$, then $C(l.I) \neq C(r.I)$, but $C(r.I) = C(l.I \cup r.I)$
(4) Else, then $C(l.I) \neq C(r.I) \neq C(l.I \cup r.I)$

The result IT-Tree of the running example in Table 1 (set support = 3) is shown in Fig. 1 ††.

The nodes in the IT-tree have internal structures. The definition of each part of it is given in the following †††.

Definition 11 *Let n be a node in the IT-tree, p be the parent of the n, then we define*

- *ext_pid* of n as: $ext\_pid = \min_f\{n.I - p.I\}$, that is, *ext_pid* is the minimum item in the itemset $\{n.I - p.I\}$ according to the total order f.
- *gen* of n as: $gen = p.I \cup n.ext\_id$. This is the generator of the closed itemset in n, which satisfies that $g(n.gen) = g(n.I)$
- *ext_ids* of n as: $ext\_ids = n.I - n.gen$. It could be proved that $\forall i \in n.ext\_ids, g(i) \supseteq g(n.gen)$
- *preset* of n as: $preset = \{i \in \mathcal{I} \mid i \prec n.ext\_id, i \notin n.I\}$
- *posset* of n as: $posset = \{i \in \mathcal{I} \mid i \succ n.ext\_id, i \notin n.I\}$

The itemset of every shadow node in the tree is actually a generator [12] of a closed itemset. The generator, *n.gen* is constructed by combining the closed itemset of it's parent, $p.I$, and the extended primary id *ext_pid*, which is an item in $p.posset$. After the shadow child is checked by the code in Table 6, line 6-7, $C(n.gen)$ is found, and the algorithm set $n.I = C(n.gen)$. For example, in Fig. 1, the generator of the node ($bef \times 2345$) is b, and after the check() procedure, the algorithm finds that $C(b) = bef$.

[12] proved that in this way, all the closed itemsets could be found.

3.2 NFCIM

Definition 12 *Let n be a node in IT-tree, then L_Tree(n) is defined as a set of nodes, each element n_l in this set has the following property: exists an ancestor node of n_l, denoted as p_l, and exists an ancestor node of n, denoted as p. which satisfies that p_l is a left sibling of p.*

Visually, we could draw a line from the node n to the root of the tree. All the nodes in the *left* of this line are in $L\_Tree(n)$. If the IT-tree is explored in depth first way as NFCIM does, and n is the current node that the algorithm is working on, then it could be proved that all the nodes in $L\_Tree(n)$ and all the ancestors of n have been explored already.

†They are called CHARM Properties in the rest of the paper.
††We assume the total order f is $a \prec b \prec c \prec d \prec e \prec f$.
†††This definition is quoted from [12].

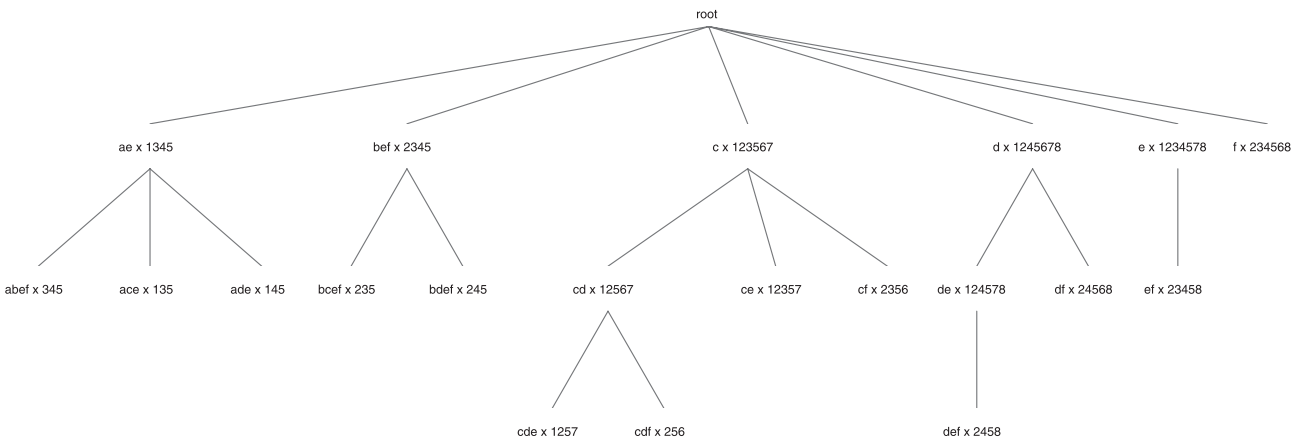


Fig. 1 CHARM Result.

Definition 13 *Let n be a node in IT-tree, then R_Tree(n) is defined as a set of nodes, each element n_r in this set has the following property: $n_r \notin L\_Tree$ and n_r is not an ancestor node of n.*

It could also be proved that, if the IT-tree is explored in depth first way as NFCIM does, and n is the current node that the algorithm is working on, then there is no node in $R\_Tree(n)$ have been explored. That is, all the nodes in $R\_Tree(n)$ must be shadow nodes.

The intuitive implementation of NFCIM is given in Table 4. However, this implementation is inefficient. For example, in Fig. 1, $(ae \times 1345)$ and $(ace \times 135)$ are similar, so the algorithm merges them and $(4, c)$ is added to $\mathcal{R}'$. The interesting thing is that the algorithm does not need to build the whole IT-tree to find these similar nodes [†]. Actually, all the nodes in the R_Tree of $(ace \times 135)$ are not necessary.

Hence, the improved algorithm tries to update $\mathcal{R}'$ without building the whole IT-tree. This is a strategy that whenever similar nodes are found, they are merged immediately and $\mathcal{R}'$ is updated as soon as possible. The implementation of the improved algorithm is the same as shown in Table 5, 6 and 7 except that the procedure process() should be rewrite as in the following:

Explanations:

(1) Line 2-7 is exactly the same as in Table 5. Line 8-9 tries to merge *node* and *ch* if they are similar. The procedure, merge(), will be described in detail in Sect. 3.2.2.
(2) The procedure find_similar_supset() [††] invoked in line 10 tries to find a node in $L\_Tree(node)$, denoted as n_{sup}, which is similar with *node* and satisfies that $n_{sup}.I \supset node.I$. If there's such kind of node found, the algorithm merges them and returns [†††].

The improved algorithm has the following advantages:

- The merge procedure is invoked for the current working node as soon as possible. Hence, the algorithm will not waste time to calculate the unnecessary information of the nodes in R_Tree of the current node.
- After $\mathcal{R}'$ is updated, the improved algorithm will not recalculate the whole IT-tree. It only requires to fixing the related node in L_Tree of the current node.
- Every time $\mathcal{R}'$ is updated, the number of NFCIs in it is reduced. Hence, the size of the IT-tree is reduced. So, by updating the $\mathcal{R}'$ as soon as possible, the search space is reduced as soon as possible.

3.2.1 Find the Superset Nodes

Definition 14 *Let $n_1, n_2 \in$ IT-tree. If $n_2.I \supset n_1.I$, we say that n_2 is a Superset Node of n_1.*

If two nodes are similar, the itemset of one of them must be a superset of the itemset of the other. Hence, the first step of finding a similar node of the current node would be to find all the superset node of it. However, the number of such nodes could be very large in the IT-tree. Hence, we introduce the concept of *Direct Superset Node* in the following:

Definition 15 *Let $n_1, n_2 \in$ IT-tree. If $n_2.I \supset n_1.I$, and $\nexists n_3 \in$ IT-tree, satisfies that $n_2.I \supset n_3.I \supset n_1.I$. We say that n_2 is a Direct Superset Node of n_1.*

It could be proved that the algorithm only needs to find the direct superset nodes of the current node. For example, if there're 3 nodes, $n_1.I \subset n_2.I \subset n_3.I$. Hence n_3 is not a direct superset node of n_1. So, even if n_1 and n_3 are similar nodes, they could not be merged directly. However, n_3 and n_2 are similar nodes too and n_3 is a direct superset node of n_2, so they could be merged as a new node n_4. After that, n_4 is a direct superset node of n_1 and they could be merged too. The conclusion is that it has the same power as merge n_1 and n_3 directly.

There're two kinds of directly superset nodes of the current node: (1) This kind of node n_{sup} satisfies that $\nexists i \in$

[†]Two nodes is similar if and only if the closed itemsets of them are similar.

[††]Refer to Sect. 3.2.1 for detailed information about this procedure.

[†††]If *node* merges with n_{sup} which is from $L\_Tree(node)$, the algorithm will delete *node*. Hence in line 12, the procedure returns directly. For detailed information, refer to Sect. 3.2.2.

Table 8 process(*node*).

```
1:  process(node)
2:    for all child of node, denoted as ch
3:      if(ch is a duplicated node)
4:        delete ch;
5:        return;
6:      for all right sibling of ch, denoted as rs
7:        check(node, ch, rs);
8:        if(node, ch are similar)
9:          merge(node, ch);
10:     if((n_sup = find_similar_supset(node)) != null)
11:       merge(n_sup, n);
12:       return;
13:     if(∃ child under ch)
14:       process(ch)
```

Table 9 find(I, i_{pre}).

```
1:  procedure find(I, i_pre)
2:    cur_node = root;
3:    I = I ∪ i_pre;
4:    pid = min_f{I};
5:    while(∃child of cur_node, child.ext_pid = pid ){
6:      I = I − child.I;
7:      if(I is empty) return child;
8:      pid = min_f{I};
9:      cur_node = child;
10:   }
11:   return null;
```

$n_{sup}, i \in c\_node.preset$. (2) This kind of node n_{sup} satisfies that $\exists i \in n_{sup}, i \in c\_node.preset$.

It could be proved that the first kind of direct superset node must be a child node of c_node. So, the algorithm need to check whether the current node is similar with its children, as shown in Table 8, line 8-9.

For the second type, the algorithm tries to use a divide and conquer strategy: for each item $i_{pre} \in c\_node.preset$, it tries to find a direct superset node that contains i_{pre} and none of the items $i \prec_f i_{pre}$ is contained. For example, in Fig. 1, the preset of the node ($ce \times 12357$) is $\{a, b, d\}$. Hence, the algorithm tries to find the direct superset node which (1) contains a, (2) contains b, but not a, (3) contains d, but not a and b.

These procedure is given in the pseudo code in Table 9.

The input parameters are (1) I, the itemset of the current node, $n.I$; (2) i_{pre}, an item id in $n.preset$. If there's no child found in line 5, then, either the direct superset includes i_{pre} is not frequent, or there must be another item $i \prec i_{pre}$ exists in the direct superset node.

For example, in Fig. 1, the algorithm tries to find the direct superset of ($df \times 24568$) which contains a. In line 3, the algorithm sets $I = adf$, the minimum element of I is a, and the cur_node is root. So in the first iteration, the *child* node is ($ae \times 1345$), In line 6, $I = adf - ae = df$, the minimum element of I is d, so in the second iteration, the *child* is ($ade \times 145$), then $I = df - ade = f$, now there's no child under ($ade \times 145$) which has f as it's ext_pid. Hence, there's no direct superset of df which contains a in the IT-tree. Actually, adf is an infrequent itemset.

The pseudo code of find_similar_supset() is listed in Table 10.

Table 10 find_similar_supset(c_node).

```
1:  procedure find_similar_supset(c_node)
2:    for each i ∈ c_node.preset;
3:      n_sup = find(c_node.I, i);
4:      if(n_sup != null && n_sup and c_node are similar)
6:        return n_sup;
11:   return null;
```

Table 11 merge(n_1, n_2).

```
1:  procedure merge(n_1, n_2)
2:    if(n_1 is parent of n_2){
3:      n_1.I = n_2.I
4:      delete_node(n_2)
5:    }else{
6:      n_2.T = n_1.T
7:      delete_subtree(n_1)
8:    }
9:    update R';
10:   fix_existed_nodes();
```

3.2.2 Merge Similar Nodes

Similar nodes will be merged. The merge procedure is given in Table 11, here, we assume that $n_1.I \subset n_2.I$.

Explanations:

- If n_1 is the parent of n_2, then the original process is: (1) delete n_1, (2) put n_2 at the position of n_1 used to be, (3) set $n_2.T = n_1.T$. Actually, this is equivalent with the code in line 3-4.
- The algorithm need the procedure fix_existed_nodes() in line 10, because since $\mathcal{R}'$ is updated, the related nodes in the IT-tree should be updated too.

There're a lot of trivial operations to be carried out in the procedure fix_existed_nodes(). This paper will not list the pseudo code of it since it's long and tedious. Instead, we just give an introduction about the main structure of it in the following:

Denote the current node the algorithm is processing as c_node. This procedure mainly deals with the situation when the tidset of the generator of c_node is expanded, that is, $g'(c\_node.gen, \mathcal{R}')$ is expanded. There're 4 kinds of sub-situations need to be concerned:

(1) Restore of a deleted node: If there exists an ancestor node n_a of c_node, satisfies that before the update of $\mathcal{R}'$, $n_a.T$ is a superset of $g'(n_a.parent.I \cup c\_node.ext\_pid, \mathcal{R}')$. According to CHARM Property 3, there's a child node of $n_a.parent$ is deleted in this kind of situation. Denote the deleted node as n_d, we have $n_d.ext\_pid = c\_node.ext\_pid$ and $n_d.gen = n_a.parent.I \cup c\_node.ext\_pid$. If after $\mathcal{R}'$ is updated, $n_a.T \not\supseteq g'(n_d.gen, \mathcal{R}')$, then, the Charm Property 3 is not applicable any longer. Hence the node n_d should be restored.

(2) Expand ext_ids: If there exists left sibling, denote as n_l, of c_node, before $\mathcal{R}'$ is updated, $n_l.T \not\subseteq c\_node.T$. If

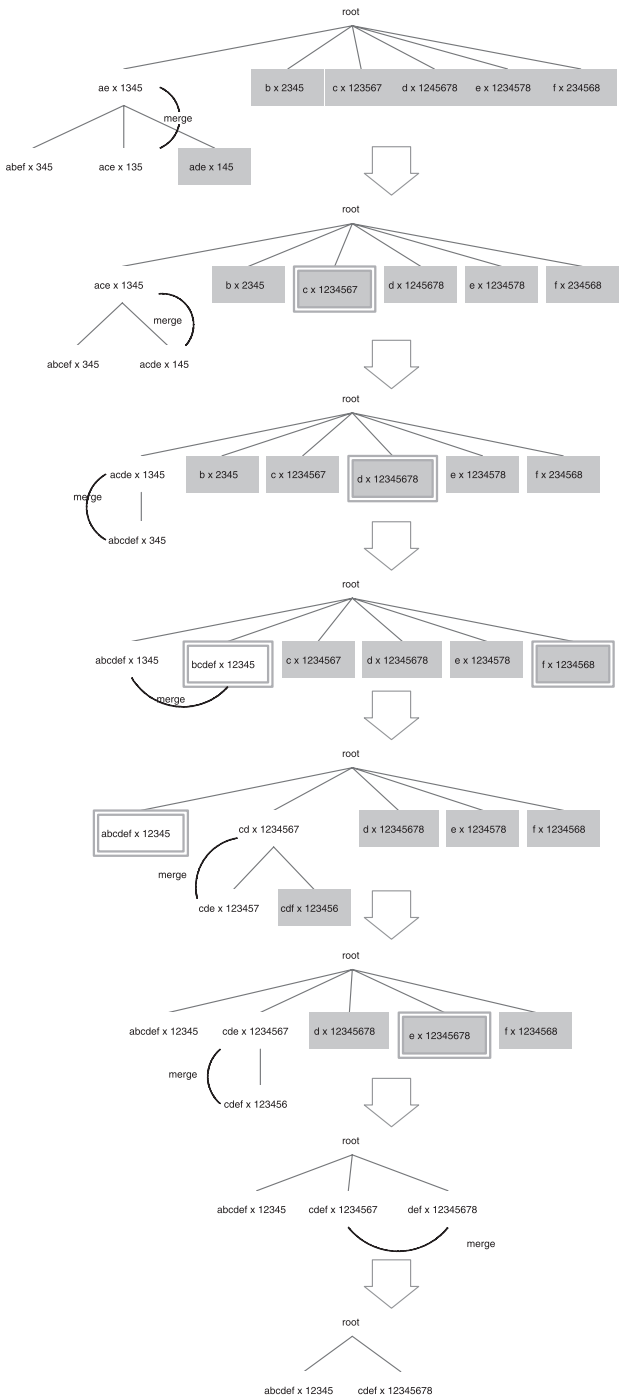


Fig. 2 NFCIM Result.

after $\mathcal{R}'$ is updated, $n_l.T \subset c\_node.T$. Then, according to CHARM Property 2, $c\_node.ext\_pid$ could be added to $n_l.I$, that is, $n_l.I = n_l.I \cup c\_node.ext\_pid$.

(3) Split node: If there exists right sibling, denoted as n_r, of c_node, before $\mathcal{R}'$ is updated, $n_r.T \supset c\_node.T$, then, according to CHARM Property 2, $n_r.pid \in c\_node.I$. If after $\mathcal{R}'$ is updated, $n_r.T \not\supseteq c\_node.T$. Hence, CHARM Property 2 is no longer applicable. So, c_node should be split up to remove $n_r.ext\_pid$ from the itemset of it.

(4) Delete duplicated node: If there exists right sibling, denoted as n_r, of c_node, before $\mathcal{R}'$ is updated, $n_r.T \not\subseteq c\_node.T$. If after $\mathcal{R}'$ is updated, $n_r.T \subset c\_node.T$. Hence, CHARM Property 3 is applicable, so n_r should be deleted from the IT-tree.

3.2.3 The Running Example

In this section, we present how the IT-tree is build based on the running example introduced in Sect. 1. Let $\epsilon_r = 0.3$, $\epsilon_c = 0.4$ and $S = 3$. The algorithm proceeds as shown in Fig. 2 [†].

In step 1, NFCIM finds that ($ae \times 1345$) and ($ace \times 135$) are similar nodes and should be merged. The IT-tree after merge is displayed in step 2. The updated node ($c \times 1234567$) is marked with a grey border. In step 2, NFCIM finds that ($ace \times 1345$) and ($acde \times 145$) are similar nodes. The algorithm will keep merging similar nodes, until there's no similar nodes found. At last, there're only two NFCIs found in the running example.

4. Performance Evaluation

In this section, we report our performance study of the three FCI algorithms: CHARM, FP-Close and NFCIM. CHARM [3] is a base algorithm of NFCIM, and FP-Close [6] is a state of arts algorithm of mining frequent closed itemsets.

We do not compare NFCIM to FI and AFI algorithms because it's well recognized that FCI algorithms outperform FI and AFI algorithms.

The experiments were conducted on a Windows XP PC equipped with a 1.7 GHz Pentium IV and 1024 MB of RAM memory.

We test the three algorithms on various datasets, including synthetic ones generated by the standard procedure described in [13], and two popular real datasets.

- synthetic dataset T25I20D10k with 1000 items: In this dataset, the average transaction size and average maximal potentially frequent itemset size are set to 25 and 20, respectively, while there are totally 10k transactions. This dataset is sparse. Most of frequent itemsets are closed.
- Real dataset Connect-4: This dataset is from the UC-Irvine Machine Learning Database Repository. It is compiled from the Connect-4 game state information. The total number of transactions is 67,557, while each transaction is with 43 items. It is a dense dataset with a lot of long frequent itemsets.
- Real dataset KDD-CUP 2007: This dataset comes from KDD-Cup 2007. The dataset provide a list of 100,000 (user_id, movie_id) pairs where the users and movies are drawn from the Netflix Prize training data set which consists of more than 100 million ratings from over 480

[†]Each grey node in the tree is a shadow node.

Table 12 Number of itemsets in dataset Connect.

Support	#FI	#FCI	#NFCI	$\frac{\#FI}{\#AFIC}$	$\frac{\#FCI}{\#NFCI}$
64179(95%)	2,205	812	7	315.0	116.0
60801(90%)	27,127	3,486	7	3875.3	498.0
54046(80%)	533,975	15,107	12	44497.9	1258.9
47290(70%)	4,129,839	35,875	14	294988.5	2562.5

Table 13 The Precision and Recall of NFCIM and AFI.

s(%)	precision		recall	
	AFI	NFCIM	AFI	NFCIMs
0.6	90.04	100	76.70	72.73
0.8	90.46	100	78.61	74.15
1.0	91.69	100	78.30	75.27
2.0	97.52	100	81.71	77.80
3.0	99.40	100	81.61	78.82

thousand randomly-chosen, anonymous customers on nearly 18 thousand movie titles. In this dataset, a transaction corresponds to a user, an item corresponds to a movie.

4.1 Reduction of the Size of Itemsets Using NFCI

Our experiments show that the number of frequent itemsets which need to be represented in mining can be reduced by several orders of magnitude in a dense database if they are represented by NFCI. Table 12 lists the numbers of Frequent Itemsets(# FI), Frequent Closed Itemsets(# FCI) and Noise-tolerant Frequent Closed Itemsets(# NFCI), as well as their ratios, in dataset Connect-4 [†].

4.2 Quality Testing with Synthetic Data

In this section, we compare the quality of the output of NFCIM with another state-of-art noise tolerant algorithm, AFI [9]. The synthetic data T10.I4.D20K is treated as the dataset without any noise, denoted as $\mathcal{R}$. The noise is introduced by flipping each element in the dataset with probability p. The dataset with noise is denoted as $\mathcal{R}_n$.

Let FI_e represent all the exact frequent itemsets in $\mathcal{R}$ discovered by the exact FI mining algorithms; FI_n represent all the "approximate" frequent itemsets in $\mathcal{R}_n$ by the noise tolerant mining algorithms. Then the following two metrics are used to measure the quality of the result of the noise tolerant algorithms.

$$presicion = \frac{||FI_e \cap FI_n||}{||FI_n||}, reccall = \frac{||FI_e \cap FI_n||}{||FI_e||}$$

Let $p = 0.05\%$, $\epsilon_c = \epsilon_r = 0.2$ for both NFCIM and AFI, then table 13 shows the experiments results for these two algorithms.

As demonstrated in Table 13, NFCIM has better precision but worse recall than AFI. It's because NFCIM discovers less noise elements than AFI, as a result, the chance for NFCIM to generate false positive patterns is slimmer than AFI, meanwhile the chance for it to generate false negative patterns is greater than AFI.

Table 13 also shows that the recall of NFCIM is close to AFI. Hence, this experiment verifies the "side effects" hypothesis proposed in Sect. 3.

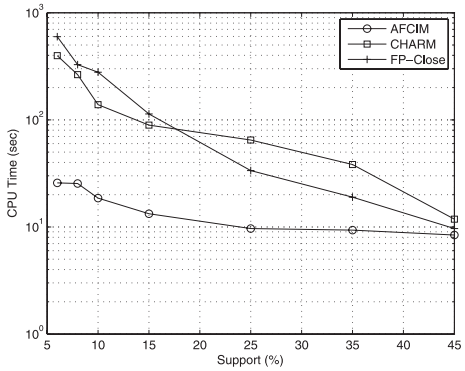


Fig. 3 Connect-4.

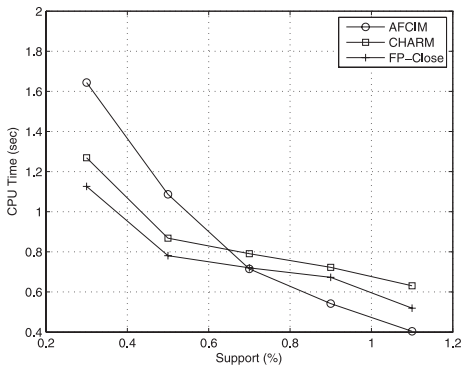


Fig. 4 T25I20D10K.

4.3 Runtime Comparison

Figure 3 shows the CPU Runtime of the algorithms on dataset Connect-4. The slack parameter of ACFIM are $\epsilon_c = \epsilon_r = 0.3$.

Figure 4 shows the CPU Runtime of the algorithms on dataset T25I20D10K. The slack parameter of ACFIM are $\epsilon_c = \epsilon_r = 0.2$.

Figure 5 shows the CPU Runtime of the algorithms on dataset KDD-CUP 2007. The slack parameter of ACFIM are $\epsilon_c = \epsilon_r = 0.2$.

The results of the performance evaluation show that NFCIM is about two orders of magnitude faster than CHARM and FP-Close on Connect-4 dataset, while NFCIM is a little bit slower than CHARM and FP-Close on the other two datasets. Hence, we could conclude that the NFCIM has much better performance than the state of arts algorithms such as CHARM and FP-Close on dense datasets, meanwhile it is as efficient as them on sparse datasets.

It could be found that NFCIM is inefficient on sparse datasets comparing to the performance on the dense

[†]Let the slack parameters of NFCIM be $\epsilon_r = \epsilon_c = 0.15$.

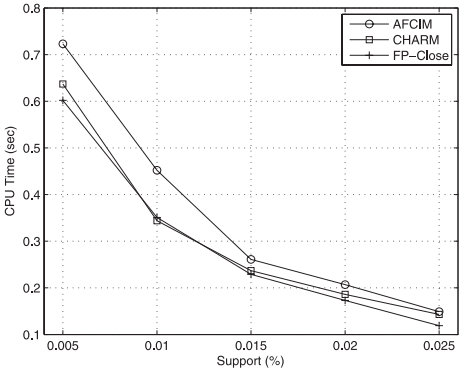


Fig. 5 KDD-CUP 2007.

datasets. The reason of it is that the reduction of the number of generated itemsets (the search space) is not as significant as on the dense datasets. For example, the search space is reduced thousands of times on dense datasets, while the search space is reduced about 50% on the sparse datasets.

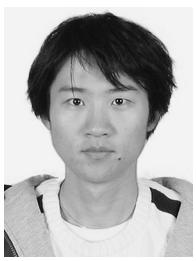
5. Conclusions and Future Work

Frequent itemsets mining is one of the most popular data mining tools ever invented and extensive efforts have been carried out in this area. However, it has two main problems. Firstly, it may often derive an undesirably huge set of frequent itemsets. Secondly, it is vulnerable to noise.

This paper proposes a novel approach, Noise-tolerant Frequent Closed Itemsets, which could address these two problems simultaneously. The new approach is noise tolerant. The number of generated itemsets is proved to be dramatically reduced with almost no information loss except for the noise and infrequent patterns. The performance experiments demonstrate that the new approach is about two orders of magnitude faster than the state-of-art FCI mining algorithms on dense datasets meanwhile it is as efficient as them on sparse datasets.

References

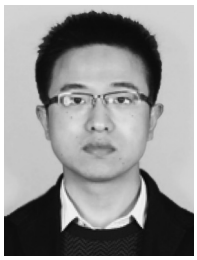
[1] X. Sun and A.B. Nobel, "Significance and recovery of block structures in binary matrics with noise," 2005 Tech. Rep. Department of Statistics and Operation Research, UNC Chapel Hill.
[2] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," Proc. ICDT'99, Jan. 1999.
[3] M.J. Zaki, C.-J. Hsiao, "CHARM: An efficient algorithm for closed itemsets mining," Proc. SIAM ICDM Conf., 2002.
[4] J. Pei, J. Han, and R. Mao, "CLOSET: An efficient algorithm for mining frequent closed itemsets," Proc. DMKD Conf., May 2000.
[5] J. Wang, J. Han, and J. Pei, "CLOSET+: Searching for the best strategies for mining frequent closed itemsets," Proc. KDD Conf., 2003.
[6] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," Proc. FIMI, ICDM Conf., 2003.
[7] C. Yang, U. Fayyad, and P.S. Bradley, "Efficient discovery of error-tolerant frequent itemsets in high dimensions," Proc. International Conference on Very Large Database, 2001.
[8] J. Seppänen and H. Mannila, "Dense itemsets," Proc. KDD Conf., 2004.
[9] J. Liu, S. Paulsen, X. Sun, W. Wang, A. Nobel, and J. Prins, "Mining approximate frequent itemsets in the presence of noise: Algorithm and analysis," Proc. SIAM ICDM Conf., 2006.
[10] M. Selim and A.S. Dan, "Clustering and approximate identification of frequent itemsets," Proc. AAAI Conf., 2007.
[11] C. Junbo, Z. Bo, D. Yiqun, and C. Lu, "Finding cloesed frequent itemsets in linear time," Proc. DMIN Conf., 2008.
[12] C. Lucchese, S. Orlando, and R. Perego, "DCI closed: A fast and memory efficient algorithm to mine frequent closed itemsets," Proc. ICDM Conf., 2004.
[13] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," Int'l Conf. on Very Large Databases, 1994.
[14] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "LCM: An efficient algorithm for enumerating frequent closed item sets," Proc. ICDM Conf., 2003.
[15] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal, "Computing iceberg concept lattices with TITANIC," J. Knowledge and Data Engineering (KDE), vol.2, no.42, pp.189–222, 2002.



Junbo Chen received the B.E. degree in the department of Computer Science and Engineering from Zhejiang University of Technology (China) in 2002. He became a Ph.D. student in 2003 and since then he's been a research assistant in Zhejiang University (China). His primary research interests is machine learning and data mining.

Bo Zhou received his Ph.D. degree of computer science at Department of Computer Science and Engineering, Zhejiang University in 1995. Since 1996, he has been a faculty member and associate professor of Department of Computer science and Engineering at Zhejiang University. His research interests include database systems, software engineering,and software architecture.

Xinyu Wang received the B.E. degree in the department of Computer Science and Engineering from Zhejiang University (China) in 2002. From 2002 to 2007, he was a research assistant in the Zhejiang University, where he got the D.E. degree from the college of Computer Science. Since the June 2007, he has been the lecturer in the same university. His primary research interests include Software engineering, distributed software architecture and distributed computing. He is a member of IEEE CS.



Yiqun Ding received the B.E. degree in the department of Computer Science and Engineering from Zhejiang University (China) in 2004. He became a PhD candidate in 2004 and since then he's been a research assistant in Zhejiang University. His primary research interests is high dimensional data analysis and its application areas, such as document analysis, collaborative filtering and web usage mining.

Lu Chen received the B.E. degree in the department of Computer Science and Engineering from Zhejiang University (China) in 2003. From 2003 to 2006, she was a research assistant in the Zhejiang University, where she got the Master degree from the college of Computer Science. Since April 2006, she has been employee in State Street as an IT Project Manager for Foreign Exchange.