The algorithm of mining frequent closed itemsets based on index array

Haitao He, Shasha Feng, Jiadong Ren, Qian Wang
College of Information Science and Engineering, Yanshan University,
Qinhuangdao City, Hebei Province, P.R.China, 066004
haitao@ysu.edu.cn, fengshasha624@126.com, jdren@ysu.edu.cn, wangqianysu@qq.com

Abstract
The set of frequent closed itemsets determines exactly the complete set of all frequent itemsets and is usually much smaller than the latter. In this paper, an algorithm based on index array for mining frequent closed itemsets, Index-FCI is proposed. The vertical BitTable is adopted to compress the dataset for counting fast the support. To make use of the horizontal BitTable, the index array corresponding to the database is constructed and a new concept GFI (Great Frequent Itemset) is defined, which can be quickly found from the index array to reduce the closed itemsets checking. The hash table whose hash function value is the support of the itemset is created to remove any frequent but "non-closed" items using the hash pruning. In Index-FCI, the database is firstly compressed into BitTable; secondly, the index array corresponding to the dataset is constructed; thirdly, GFI is found from the index array and the hash table is created to store frequent closed itemsets; finally, the hash table is traversed to obtain all frequent closed itemsets. Experimental results show that Index-FCI is suitable for mining frequent closed itemsets.

Keywords: Frequent closed itemsets, Index array, BitTable

1. Introduction
Frequent Itemsets Mining (FIM) is a very important issue in data mining. It is mainly applied in association rules mining, correlation analysis, sequential patterns mining [1], multi-dimensional patterns mining [2] and so on.
The algorithm based on prefix-tree, FP-growth [3] used the FP-tree to store the frequency information of the database, and adopted a recursive divide-and-conquer method and the database projection approach to find all frequent itemsets. FP-growth had a good compression, but it needed to create conditional FP-trees recursively and traverse paths on those trees. IFP-growth [4] employed an address-table structure to lower the complexity of forming the entire FP-tree and a new tree FP-tree+ was used to reduce the need for rebuilding conditional FP-trees. IFP-growth required relatively little memory space, however, it might cause potential performance problems in traversing FP-tree+, in which the infrequent nodes didn’t be eliminated. APT [5] adopted an array-based prefix-tree to mine frequent itemsets in top-down traversal strategy, and unfiltered pseudoconstruct conditional database, which could improve computational performance. The APT had a distinct feature that the space requirement could be predictable in advance and the traversal cost was lower. The BitTable-based algorithm BitTableFI [6] applied BitTable horizontally and vertically to compress database for generating quickly candidate itemsets and counting fast the support, respectively. However, in BitTableFI, the length-2 candidate itemsets were generated in the same way as Apriori. To address the issue, Index-BitTableFI [7] used index array to find quickly those frequent itemsets having the same support as frequent 1-items, and mined all frequent itemsets in depth-first manner.
The above algorithms are used to mine all frequent itemsets, in which the resulting itemsets and association rules are often surprisingly large, it is difficult to be understood and used. The set of frequent closed itemsets is the lossless expression of all frequent itemsets. It can uniquely determine the exact supports of all frequent itemsets, and it can be orders of magnitude smaller than the set of all frequent itemsets. Therefore, mining frequent closed itemsets have been studied.
Pasquier et al. proposed the concept of frequent closed itemsets and an Apriori-based mining algorithm A-Close [8], which needed to scan the database more times. To address the problem, CHARM [9] adopted the vertical format of the transaction identifier to express the supports of the
itemsets, which saved a lot of time. However, the storage space was still large and the efficiency of projection was not high. The algorithm based on pattern-growth, CLOSET [10] scanned the database twice and mined frequent closed itemsets in depth-first search, but it needed to construct condition FP-trees and the overhead of CPU and storage were very large. FP-Close [11] was proposed, in which CFI-tree (Closed Frequent itemset Tree) was constructed, the array was used to mine frequent itemsets, and conditional FP-trees were built to perform the closed itemsets checking. PCP-Miner [12] found at first all frequent 2-itemsets in the database, and then a frequent patterns tree was created to generate recursively frequent closed itemsets in a depth-first search manner. Nonetheless, when the number of frequent itemsets was getting larger and larger, it was difficult to be loaded into main memory. IsTa [13] used a prefix-tree structure and a recursive procedure to find quickly frequent closed itemsets, which saved a lot of time. However, these algorithms spent a large number of time and space in constructing of prefix-tree.

In order to mine more effectively frequent closed itemsets with less time, the algorithm based on index array, Index-FCI is proposed. The database is compressed into the vertical BitTable for counting the support quickly; GFI (great frequent itemset) is defined and found from the index array corresponding to the database, which reduces the closed itemsets checking; the hash table is created, in which the support is used as the hash function value, to mine all frequent closed itemsets using the hash pruning.

The remaining of the paper is organized as follows. Section 2 presents the problem definitions. In Section 3, we describe the proposed algorithm. In Section 4, the performance of Index-FCI is reported. We conclude this study in Section 5.

2. Problem definitions

Let \( I = \{i_1, i_2, \ldots, i_m\} \) be a set of items, \( D = \{T_1, T_2, \ldots, T_n\} \) be a transaction database, where each transaction \( T_i \) contains a set of items in \( I \). We call a subset \( X \subseteq I \) an itemset and call \( X \) a \( k \)-itemset if \( X \) contains \( k \) items. The support of the itemset \( X \), denoted as \( sup(X) \), is the number of transactions in \( D \) that contain all items in \( X \). Let \( minsup \) be the minimum support threshold specified by user. If \( X.sup \geq minsup \), \( X \) is called a frequent itemset. If the itemset \( X \) is frequent, and there exists no its superset \( Y \) having the same support as \( X \), then \( X \) is a frequent closed itemset.

**Definition 1. (Great frequent itemset)** If an itemset \( X \) is the intersection of all transactions which contain frequent 1-item, then we call the itemset \( X \) as a great frequent itemset, denoted as \( GFI \).

**Definition 2. (Hash pruning)** Suppose \( X \) and \( Y \) be two frequent itemsets, if \( sup(X) = sup(Y) \) and \( X \subseteq Y \), then \( X \) will be deleted.

BitTable is a set of integer whose every bit represents an item. BitTable can be used to compress candidate itemset and database. When the candidate itemset is compressed with BitTable, if the candidate itemset contains the item \( i \), the bit \( i \) of BitTable’s element is marked as one; otherwise, it will be marked as zero. When the database is compressed, the BitTable is used vertically. If the item \( i \) appears in transaction \( T \), the bit \( i \) of BitTable’s \( T \) element is marked as one.

**Definition 3. (Index array)** Index array is an array with size \( m \), where \( m \) is the number of frequent 1-item. Each element of the array corresponds to a two tuple \((item:cnt, subsume)\), where \( item \) is a frequent 1-item, \( cnt \) is the support of \( item \), \( subsume(item) \) is the maximal itemset that co-occurs with \( item \) and has the same support with \( item \). For each element of index array, we call \( item \) the representative item, and \( subsume(item) \) the subsume index.

Based on BitTable and index array structure, the process of generating index array [5] is as follows.

First, the database \( D \) is scanned once to delete infrequent items, and the frequent 1-items are sorted in supports ascending order as \( a_1, a_2, \ldots, a_m \).

Second, the sorted frequent 1-items and their supports are assigned to elements of index array as representative items and their supports one by one.

Third, the database \( D \) is scanned once again, and represented with BitTable.

Fourth, the operation is conducted by intersection of the transactions containing index[j].item to obtain the candidate of index[j].subsume. If bit \( i \) of the candidate is 1, then index[i].item is add into index[j].subsume.

Fifth, write out the index array.
Given a transaction database $D$ in table 1 and a minimum support of 2. The database $D$ is firstly scanned to obtain frequent 1-items and their supports. The infrequent item $G$ is deleted and frequent 1-items are sorted in support ascending order as $\{D:2, A:3, B:4, C:4, E:4, F:4\}$. The sorted frequent 1-items and their supports will be assigned to each representative item $\text{index}[j].\text{item}.\text{cnt}$, respectively, so we can gain the following representative items and their supports: $\text{index}[1].\text{item}.\text{cnt} = D:2$, $\text{index}[2].\text{item}.\text{cnt} = A:3$, $\text{index}[3].\text{item}.\text{cnt} = B:4$, $\text{index}[4].\text{item}.\text{cnt} = C:4$, $\text{index}[5].\text{item}.\text{cnt} = E:4$, $\text{index}[6].\text{item}.\text{cnt} = F:4$. The dataset deleted infrequent items will be represented with BitTable, as shown in table 2. Then the computation is done by intersection of the transactions that contain frequent 1-item one by one to obtain the candidate of $\text{index}[j].\text{subsume}$. We take $\text{index}[1].\text{item}.\text{cnt} = D:2$ as example, candidate $= 1 \cap 4 = 111101$, where 1 and 4 are tids. There are five 1 in the candidate, since the first bit corresponds to $D$, the items, corresponding the second, the third, the fourth and the last bit, constitute the subsume index of $D$, that is $\text{index}[1].\text{subsume} = ABCF$. We can iterate the process similarly, and finally the index array is $(D:2, ABCF), (A:3, \Phi), (B:4, CF), (C:4, BF), (E:4, \Phi), (F:4, BC)$.

**Lemma 1.** [5] Let $\text{item}$ be a representative item, $\text{subsume}(\text{item}) = a_1, a_2, \ldots, a_m$, if $\text{item}$ is combined with the $2^m-1$ nonempty subsets of $a_1, a_2, \ldots, a_m$, if any itemset of resulting itemsets are all $\text{sup}(\text{item})$.

**Lemma 2.** [5] Let $\text{index}[j].\text{item}$ be a frequent item with $\text{sup}(\text{index}[j].\text{item}) = \text{minsup}$, then there exists no item $\text{index}[j].\text{item} \sim i$ ("$\sim$" according to the support ascending order) and $i \notin \text{index}[j].\text{subsume}$, such that $\text{index}[j].\text{item} \cup i$ is a frequent itemset.

**Theorem 1.** If the itemset is a $\text{GFI}$, then it must be a frequent closed itemset.

**Proof.** According to the definition 1, we know $\text{GFI}$ is the intersection of all transactions which contain frequent 1-item, thus $\text{sup}(\text{GFI})$ is equal to the support of frequent 1-item. Then $\text{GFI}$ is maximal frequent itemset contained frequent 1-item. Therefore, if an itemset is a $\text{GFI}$, then it must be a frequent closed itemset.

**Theorem 2.** If the item $X$ is a frequent 1-item and its corresponding subsume index is $\Phi$, then $X$ is a frequent closed item.

**Proof.** Based on the structure of index array, we know the subsume index of frequent 1-item $X$ is the maximal itemset that co-occurs with $\text{item}$ and has the same support with $\text{item}$. If $\text{subsume}(X) = \Phi$, then the superset having the same support as $X$ no exists. Thus $X$ is a frequent closed item.

**Theorem 3.** If the support of frequent 1-item $X$, $\text{sup}(X)$, is equal to $\text{minsup}$, then $X \cup \text{subsume}(X)$ is a frequent closed itemset and $\text{sup}(X \cup \text{subsume}(X)) = \text{minsup}$.

**Proof.**
1) From Lemma 1, we know $\text{sup}(X \cup \text{subsume}(X)) = \text{sup}(X) = \text{minsup}$.
2) According to Lemma 2, if the support of frequent 1-item $X$, $\text{sup}(X)$, is equal to $\text{minsup}$, then there exists no item $i \notin \text{subsume}(X)$ to make the itemset $X \cup i$ be a frequent itemset, thus $X \cup \text{subsume}(X)$ is a $\text{GFI}$ that contains frequent 1-item $X$. Therefore, according to theorem 1, $X \cup \text{subsume}(X)$ is a frequent closed itemset.

From 1) and 2), we know the itemset $X \cup \text{subsume}(X)$ is a frequent closed itemset and $\text{sup}(X \cup \text{subsume}(X)) = \text{minsup}$.

### 3. The Algorithm based on index array for mining frequent closed itemsets

In order to mine frequent closed itemsets more effectively with less time, we will process the index array which consists of two cases.

1) If $\text{sup}(\text{index}[j].\text{item}) = \text{minsup}$, according to theorem 3, $\text{index}[j].\text{item} \cup \text{index}[j].\text{subsume}$ is a frequent closed itemset.

2) If $\text{sup}(\text{index}[j].\text{item}) > \text{minsup}$, there are two kinds of situations.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C D E F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A E G</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>B C E F</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>A B C D F</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>B C E F</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>26</td>
<td>23</td>
<td>23</td>
<td>29</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 1. Transaction database $D$  Table 2. Example of compressing database into BitTable
i) If $index[j].subsume = \Phi$, according theorem 2, $index[j].item$ is a frequent closed item. Then the operation checking whether $t(index[j].item)$ is $\Phi$ or not will be performed. If $t(index[j].item) = \Phi$, the operation of $index[j].item$ ends; otherwise, $index[j].item$ will be combined with each nonempty subset of $t(index[j].item)$ to form new itemset $Y$, and $sup(Y)$ will be counted by the vertical BitTable. The subset checking will be conducted between $Y$ and the existing itemset having the same support as $Y$. If $Y$ has been contained in the hash table, then it will be deleted; otherwise, it will be added into the hash table.

ii) If $index[j].subsume \neq \Phi$, then the subset checking will be conducted between $index[j].item \cup index[j].subsume$ and the existing itemsets having the same support as $index[j].item \cup index[j].subsume$. If $index[j].item \cup index[j].subsume$ has been contained in the hash table, then it will be deleted; otherwise, it will be added into the hash table. Then the operation checking whether $t(index[j].item)$ is $\Phi$ or not will be performed. If $t(index[j].item) = \Phi$, the operation of $index[j].item$ finishes; otherwise, the following operations will be performed. The items contained by $index[j].subsume$ will be deleted from $t(index[j].item)$, and $t(index[j].item)$ processed will be denoted as $tail$. The itemset $index[j].item \cup index[j].subsume$ will be combined with each nonempty subset of $tail$ to form new itemset $Y$, and $sup(Y)$ will be counted with the vertical BitTable. The subset checking will be conducted between $Y$ and the existing itemset having the same support as $Y$ in the hash table. If $Y$ has been contained, then it will be deleted; otherwise, it will be added into the hash table.

Based on the above analysis, the algorithm based on index array for mining frequent closed itemsets is shown in algorithm 1.

**Algorithm 1.** The algorithm for mining frequent closed itemsets based on index array

**Input:** index array, minsup

**Output:** frequent closed itemsets $FCI$

[1] construct a hash table $H$, $H = \Phi$ and $FCI = \Phi$;
[2] for each element $index[j]$ of index array do {
  [3] if $(sup(index[j].item) = minsup)$ then
  [4] add $index[j].item \cup index[j].subsume$ into $H$;
  [5] else
  [6] if $(index[j].subsume = \Phi)$ then
    [7] add $index[j].item$ into $H$;
  [8] Depth_first($index[j].item,t(index[j].item)$);
  
  // according to support ascending order
  [9] else
  [10] add $index[j].item \cup index[j].subsume$ into $H$ using the hash pruning strategy;
  [11] $tail\leftarrow t(index[j].item)\backslash items$ in $index[j].subsume$;
  
  // delete items included by $index[j].subsume$ from $t(index[j].item)$
  [12] Depth_first($index[j].item \cup index[j].subsume,tail)$;
  [13].
  [14].
[15] Scan the hash table $H$ to record frequent closed itemsets into $FCI$;

**Procedure** Depth_first (itemset, tail)

[16] if $tail = = \Phi$ then return;
[17] for each $i \in tail$ do {
[18] $f$-itemset $\leftarrow$ itemset $\cup i$;
[19] if $sup(f$-itemset $) \geq minsup$ then
[20] add $f$-itemset into $H$ using the hash pruning strategy;
[21] $tail \leftarrow tail \backslash i$;
[22] Depth_First ($f$-itemset, tail);

In algorithm 1, the hash table $H$ is firstly constructed, and assigned to $\Phi$ (Step 1). In the mining process, frequent closed itemsets will be stored in $H$, the subset checking will be conducted through the hash pruning, and final frequent closed itemsets will be outputted to $FCI$. Then we process the index array (Step 2-22). If $sup(index[j].item)$ is equal to $minsup$, the itemset $index[j].item \cup index[j].subsume$ is added into the hash table whose hash function value is equal to the support of $index[j].item$ (Step 2, 3). If $sup(index[j].item)$ is great than $minsup$ and $index[j].subsume$ is $\Phi$, the following operations will
be performed: first of all, index[j].item is added into H (Step 6, 7); second, index[j].item will be combined with each nonempty subset of t(index[j].item); finally, the subset checking will be conducted (Step 8). If sup(index[j].item) is great than minsup and index[j].subsume is not Φ, the following operations will be performed: index[j].item ∪ index[j].subsume is firstly added into H according to the hash pruning (Step 10); and the items contained in index[j].subsume will be deleted from t(index[j].item) to obtain tail (Step 11); then index[j].item ∪ index[j].subsume will be combined with each nonempty subset of tail and the subset checking will be performed (Step 12). Finally, H is traversed to record all frequent closed itemsets to FCI.

We illustrate the conducting process of Index-FCI using the database in table 1. The index array corresponding to the database is: (D:2, ABCF), (A:3, Φ), (B:4, CF), (C:4, BF), (E:4, Φ), (F:4, BC). The representative item index[1].item = D is at first processed, since sup(D) = minsup = 2, according to theorem 3, D ∪ ABCF = ABCDF:2, the itemset ABCDF is a frequent closed itemset and added into H[2].

Then, for the element of index array whose representative item is index[2].item = A, because sup(A) = 3 > minsup and index[2].subsume = Φ, according to theorem 2, so A is a frequent closed item and added into H[3]. According to support ascending order, the set of items after A is t(index[2].item) = BCEF. Then A is combined with each nonempty subset of t(index[2].item) to form the new itemset Y, which will be added into H[sup(Y)] using the hash pruning. Therefore, AE will be added into H[sup(Y)].

Next, for the representative item index[5].item = E, sup(E) = 4 > minsup, Index-FCI exploits the similar processes as that of A. Thus, E is added into H[4].

Then, the representative item index[3].item = B is processed, as sup(B) = 4 > minsup and index[3].subsume ≠ Φ, according to the hash pruning, index[3].item ∪ index[3].subsume = BCF will be added into H[4]. The set of items after B is t(index[3].item) = EF, and F is in index[3].subsume, so F will be deleted from t(index[3].item) to gain tail = E. The itemset index[3].item ∪ index[3].subsume will be combined with each nonempty subset of tail, and the itemset BCEF will be obtained and added into H[sup(BCEF)] using the hash pruning.

Next, for the remaining elements of index array whose representative items are C and F, Index-FCI iterates the above-mentioned processes similarly. Thus, the ultimate hash table is shown in table 3. Finally, the hash table is traversed to obtain all frequent closed itemsets: {ABCDF:2, AE:2, A:3, BCEF:3, E:4, BCF:4}.

### Table 3. The hash table H

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FCI</td>
<td>ABCDF</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>AE</td>
<td>BCEF</td>
<td>BCEF</td>
</tr>
</tbody>
</table>

4. Performance evaluation

To test the performance of Index-FCI, we compare it with two algorithms FPClose and Index-BitTableFI over real and synthetic datasets. FPClose is a closed itemsets mining algorithm which shows better performance. Index-BitTableFI is recently developed algorithm based on index array for mining frequent itemsets. Index-FCI was written in C++ and compiled with C++6.0. The main purpose of the experiment is to not only show the effectiveness of Index-FCI but also to testify its scalability.

### Table 4. The characteristics of the datasets

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Size</th>
<th>#Trans</th>
<th>#Items</th>
<th>A(Mm)L.L.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>12.14M</td>
<td>65,557</td>
<td>150</td>
<td>43(43)</td>
</tr>
<tr>
<td>Pumsb</td>
<td>14.75M</td>
<td>49,046</td>
<td>2113</td>
<td>74(74)</td>
</tr>
<tr>
<td>T10I4Dx</td>
<td>5.06-50.6M</td>
<td>100k-1000k</td>
<td>1000</td>
<td>10(31)</td>
</tr>
</tbody>
</table>

Table 4 shows the characteristics of the databases used for performance evaluation. Connect is a dense database, and it is derived from its game steps. Pumsb is census dataset. These two real databases can be obtained from the UCI machine learning repository (http://www.ics.uci.edu/~mlearn) and the frequent itemsets mining dataset repository (http://fimi.cs.helsinki.fi/data/). The synthetic T10I4Dx
datasets were generated from the IBM dataset generator. Relative to the real datasets, the synthetic datasets are sparse, which contain from 100k to 1000k transactions. The experiments were conducted on a Windows XP PC equipped with an AMD Athlon II * 4 640 CPU, a 3.00GHz processor, 2 GB of main memory and 500GB hardware.

Figures 1 and 2 show the number of itemsets and runtime comparison results on the real dataset Connect among three algorithms, where the minimum support varies between 0.4% and 0.2%. In figure 1, the number of frequent closed itemsets generated in Index-FCI is equal to those in FPClose, and more and more frequent itemsets are generated with the minimum support decreasing. In figure 2, obviously, Index-FCI runs faster than Index-BitTableFI. This is because Index-BitTableFI finds frequent itemsets through combining each representative item in index array with each nonempty subset of its corresponding subsume index; however, in Index-FCI each representative item only needs to be combined with its corresponding subsume index to generate frequent closed itemsets. Moreover, while the minimum support decreases, Index-FCI runs faster than FPClose. This is due to it needs plenty of time to create conditional FP-trees in FPClose with the minimum support decreasing. We can not see the number of itemsets of Index-BitTableFI in the figure 1 because Index-BitTableFI mines a huge number of itemsets with these parameter settings.

Figures 3 and 4 are comparison results on database Pumsb among three algorithms, where the minimum support varies between 0.7% and 0.5%. From the figure 3, we can see that the number of frequent closed itemsets in Index-FCI is smaller than those of frequent itemsets in Index-BitTableFI. Figure 4 shows when the minimum support becomes lower, the performance of Index-FCI is priori to FPClose.

We performed a scalability test on the T10I4Dx synthetic datasets. The runtime is shown in figure 5 with regard to the number of transactions from 100k to 1000k. From the figure 5, we know Index-FCI has much better scalability in terms of the number of transactions.

In summary, Index-FCI can mine frequent closed itemsets efficiently, and has much better scalability. This is because it exploits index array, in which the great frequent itemset can be quickly
found to reduce the close itemset checking. Besides, the hash pruning is adopted in Index-FCI to remove frequent but “non-closed” itemsets.

![Image of a graph showing scalability test results](Image)

**Figure 5.** Scalability test (T10I4Dx datasets)

5. Conclusions

In this paper, the algorithm based on index array, Index-FCI, is proposed for mining frequent closed itemsets. In Index-FCI, the database is firstly compressed into BitTable in support ascending order. The vertical BitTable is adopted for counting support quickly. Secondly, the index array corresponding to the database is constructed through the horizontal BitTable. Thirdly, the great frequent itemset (GFI) is defined and found through combining the representative item and its corresponding subsume index in index array, which reduces the closed itemsets checking. Fourthly, the hash table whose hash function value is the support of the itemset is constructed to perform the subset checking to delete “non-closed” itemsets. Finally, all frequent closed itemsets can be obtained by traversing the hash table. Experimental results demonstrate that Index-FCI is effective for mining frequent closed itemsets.

6. Acknowledgment

This work is supported by the Natural Science Foundation of Hebei Province P. R. China under Grant No.F2010001298 and the National Natural Science Foundation of China under Grant No. 61170190. The authors also gratefully acknowledge the valuable comments and suggestions of the reviewers.

7. References

[10] Jian Pei, Jiawei Han, Runying Mao, “CLOSEST: An Efficient Algorithm for Mining Frequent Closed Itemsets”, In Proceeding of ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery, pp. 21-30, 2000.