LCM ver.3: Collaboration of Array, Bitmap and Prefix Tree for Frequent Itemset Mining

Takeaki Uno
National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku
Tokyo, JAPAN, 101-8430
uno@nii.jp

Masashi Kiyomi
National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku
Tokyo, JAPAN, 101-8430
masashi@grad.nii.ac.jp

Hiroki Arimura
Information Science and Technology, Hokkaido University
Kita 14-jo Nishi 9-chome
060-0814 Sapporo, JAPAN
arim@ist.hokudai.ac.jp

ABSTRACT

For a transaction database, a frequent itemset is an itemset included in at least a specified number of transactions. To find all the frequent itemsets, the heaviest task is the computation of frequency of each candidate itemset. In the previous studies, there are roughly three data structures and algorithms for the computation: bitmap, prefix tree, and array lists. Each of these has its own advantage and disadvantage with respect to the density of the input database. In this paper, we propose an efficient way to combine these three data structures so that in any case the combination gives the best performance.

1. INTRODUCTION

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining, inductive databases, and query expansion. For these applications, fast implementations of frequent itemset mining problems are needed. In this paper we propose a new data structure for decreasing the computational cost, and implemented it to obtain the third versions of LCM, LCMfreq, for enumerating all frequent closed itemsets and all frequent itemsets, respectively. LCM is an abbreviation of *Linear time Closed itemset Miner*.

According to the computational experiments in FIMI03 and FIMI04[6], the heaviest task in the process of frequent itemset mining is the frequency counting, which is to compute the number of transactions including the candidate itemsets. Thus, many techniques and data structures were proposed for frequency counting. Among these, the bitmap [4, 5, 10, 11], the prefix tree [1, 2, 7, 8, 9, 13], and "occurrence deliver" with array lists [14, 16] are popular (see Figure 1).

The bitmap stores the transaction database in a 01 matrix, such that the ij element of the matrix is 1 if and only if item i is included in jth transaction. Each cell can be represented by 1 bit, thus we can save memory, especially in the case that the database is dense. The itemset is also represented by the bitmap, so that the intersection and the union of two itemsets or transactions can be done in short time, since a 32bit CPU operates 32 bits at once. Roughly speaking, the bitmap is efficient if the input database is dense, and the minimum support is not small, i.e., larger than 5% of the number of transactions.

The prefix tree is a popular data structure to store strings or sequences. It is a rooted tree such that any string (or sequence) is represented as a path from a leaf to the root, and any common prefix of two strings is the common subpath of the representative paths of them (see Figure 1). Thus, the common prefixes save memory. The prefix tree is strong if the data is structured, and the minimum support is not too small, hence it is efficient in practice. We can also save computations for frequency counting with respect to the common prefixes. The disadvantage of the prefix tree is the high cost for its reconstruction in the recursive calls.

Occurrence deliver is a technique for efficiently computing the frequencies of many itemsets at once in short time. In an iteration of mining algorithms, some candidate itemsets are generated, and their frequencies are computed. The occurrence deliver stores the transaction database by array lists, and compute the frequencies by scanning the lists once. The occurrence deliver is efficient especially for sparse databases. With a database reduction, it is also efficient for dense databases, but still weak for quite dense databases.

These three techniques have advantages, but also disadvantages. To avoid the disadvantages, we propose a new data structure of a combination of these three techniques. The main part of the data structure is the array list, but for constant number of items, we use a bitmap and a complete prefix tree (see an example in Figure 5). The complete prefix tree is a prefix tree including all the possible itemsets. Mining algorithms are generally recursive, and reduce the database recursively. Thus, the reduced databases usually include a constant number of items in the bottom levels of the recursion. For such small databases, we can use the efficiency of the bitmap and the prefix tree, for frequency counting. Since the computation time in these bottom levels dominates the total computation time, the increase of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OSDM'05, August 21, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-210-0/05/08 ...$5.00.

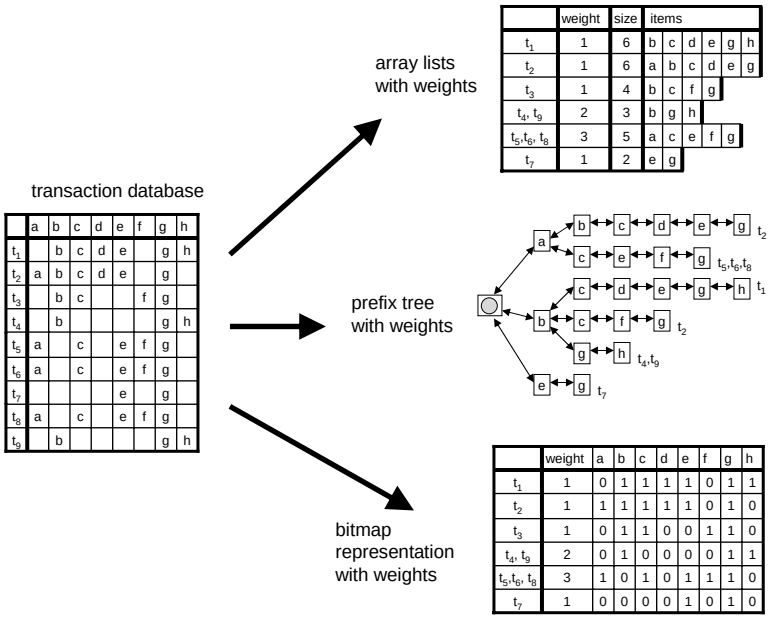


Figure 1: Popular data structures to store transaction databases in memory

the speed in the bottom levels has a drastic effect. The disadvantage of the prefix tree is avoided by using the complete prefix tree and by reusing the complete prefix tree in every iteration.

Moreover, many large databases are in the power law in practice, thus only few items are included in many transactions. Our data structure compresses the part of the databases with respect to such items by the bitmap.

We here list up the advantages of our data structure.

· fast construction
· fast frequency counting for high frequency items
· fast database reduction by using the bitmap
· no need of reconstruction of prefix trees
· applicable to many existing algorithms

However, when the database is very sparse and the minimum support is very small, i.e., less than 10, the computation time is sometime slow. In these cases, the computation time is 3 or 4 times longer than the fastest implementation in FIMI repository. This is because of the cost for changing the strategy. Note that in these cases the fastest implementation is the previous versions of LCM.

Our data structure can be applied to backtrack algorithms (depth-first algorithm) and also apriori type algorithms. It can also be applied to mining algorithms for any problem; frequent itemset mining, maximal frequent itemsets mining, and frequent closed itemset mining. We implemented codes for both frequent itemset mining and frequent closed itemset mining. The computational experiments shows that in almost cases our implementation is the fastest in the implementations proposed in FIMI03 and FIMI04. A part of the results are shown in section 4.

2. PRELIMINARIES

Let $\mathcal{E} = \{1, ..., n\}$ be the set of *items*. A *transaction database* on $\mathcal{E}$ is a set $\mathcal{T} = \{T_1, \ldots, T_m\}$ such that each T_i is included in $\mathcal{E}$. Each T_i is called a *transaction*. We denote by $||\mathcal{T}||$ the sum of sizes of all transactions in $\mathcal{T}$, that is, the size of database $\mathcal{T}$. A set $P \subseteq \mathcal{E}$ is called an *itemset*. The maximum element of P is called the *tail* of P, and denoted by $tail(P)$. An itemset Q is a *tail extension* of P if and only if both $Q \setminus P = \{e\}$ and $e > tail(P)$ hold for an item e. An itemset $P \neq \emptyset$ is a tail extension of Q if and only if $Q = P \setminus tail(P)$, hence Q is unique, i.e., any non-empty itemset is a tail extension of a unique itemset.

For itemset P, a transaction including P is called an *occurrence* of P. The *denotation* of P, denoted by $Occ(P)$ is the set of the occurrences of P. $|Occ(P)|$ is called the *frequency* of P, and denoted by $frq(P)$. In particular, for an item e, $frq(\{e\})$ is called the frequency of e. For given constant θ, called a *minimum support*, itemset P is *frequent* if $frq(P) \geq \theta$. If a frequent itemset P is included in no other frequent itemset, P is called *maximal*. We define the *closure* of itemset P in $\mathcal{T}$, denoted by $clo(P)$, by $\bigcap_{T \in Occ(P)} T$. An itemset is called *closed itemset* if $P = clo(P)$. For any two itemsets P and Q, the following properties hold.

(1) $Occ(P \cup Q) = Occ(P) \cap Occ(Q)$
(2) If Q is a tail extension of P,
$Occ(Q) = Occ(P \cup \{e\})$ for an item e
(3) If $P \subseteq Q$, $frq(Q) \leq frq(P)$
(4) If Q is a tail extension of P, $frq(Q) \leq frq(P)$.

Through this paper, let c be a given constant, which we can choose by looking the property of the input database. For transaction T, we call the set of items in T greater than $n-c$ *constant suffix* of T, and denote it by T^s. Similarly, we call $T \setminus T^p$ *constant prefix* of T, and denote it by T^p. We recall that n is the largest item.

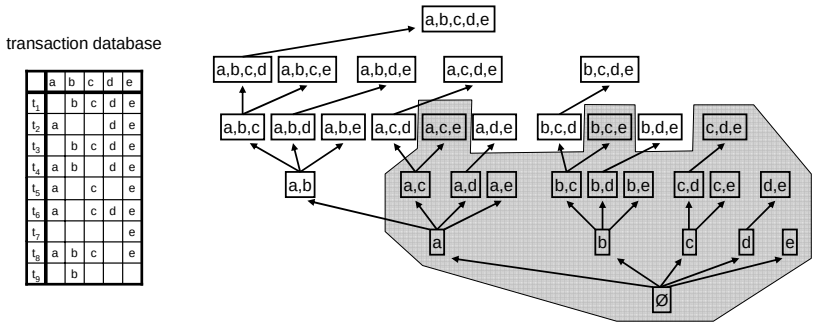


Figure 2: A tree induced by tail extensions: backtrack algorithm traverses the tree in a depth-first way

3. EXISTING ALGORITHMS AND DATA STRUCTURES

In the following subsections, we explain the popular techniques used in frequent itemset mining.

3.1 Backtrack Algorithm

Any itemset included in a frequent itemset is itself frequent. Thereby, the property "frequent" is anti-monotone. In particular, any frequent itemset is a tail extension of a frequent itemset. From this, we can see that any frequent itemset can be constructed from the empty set by generating tail extensions recursively. A backtrack algorithm is a depth-first version of this generation[3, 8, 13, 14, 15, 16].

Backtrack algorithm is based on recursive calls. An iteration of a backtrack algorithm inputs a frequent itemset P, and generates all tail extensions of P. Then, for each extension being frequent, the iteration generates a recursive call with respect to it. We describe the framework of backtrack algorithms below. Figure 2 shows an example of the execution of a backtrack algorithm.

ALGORITHM Backtrack (P:itemset)
1. **Output** P
2. **For each** $e \in \mathcal{E}, e > tail(P)$ **do**
3. **If** $P \cup \{e\}$ is frequent **then call** Backtrack ($P \cup \{e\}$)

We note that here an *iteration* of the algorithm is the computation from step 1 to 3 in a recursive call, except for the computation done in the recursive calls generated in step 3. The advantage of the algorithm is that it needs no memory for storing the frequent itemsets previously obtained, even if the number of the frequent itemsets is huge. Since backtrack algorithms generate a number of recursive calls in an iteration, the number of iterations at the kth level of the recursion is small if k is closed to 1, and increases exponentially as the increase of k. We call this property *bottom-wideness*. From the bottom-wideness, we can see that the total computation time of a backtrack algorithm is dominated by the computation in the bottom levels of the recursion.

3.2 Frequency Counting

As we can see, the heaviest part of the computation in a backtrack algorithm is in step 3, the computation of $frq(P \cup \{e\})$ to check whether $P \cup \{e\}$ is frequent or not. We call this computation *frequency counting*. Frequency counting can be done by taking the intersection of $\mathcal{T}(P)$ and $\mathcal{T}(\{e\})$, however it takes long time in a straightforward way. To reduce the computation time, there are many approaches; bitmap, prefix tree, occurrence deliver, and conditional database. The conditional database is orthogonal to the others, hence we can combine it to one of the others.

The approach of the bitmap is to use a bitmap for representing $\mathcal{T}(P)$ and $\mathcal{T}(\{e\})$. Then, we can take intersection by binary operation "and", thus 32 or 64 operations can be done in one step. However, if the database is sparse and the minimum support is small, then the bitmap representation includes so many 0's in it, which can be omitted in the other ways.

In the following, we explain the conditional database, the prefix tree, and the occurrence deliver.

3.3 Conditional Database

A *conditional database* of an itemset P is a database used in an iteration inputting P, and is obtained by removing items and transactions which are not necessary in the iteration and its recursive calls. Using conditional databases we can reduce the tail extensions to be checked, and the computation time for frequency counting.

For an itemset P, its conditional database, denoted by $\mathcal{T}_P$ is obtained from $\mathcal{T}$ by the following way (see Figure 3):

1. remove transactions not including P
   (database becomes equal to $Occ(P)$)
2. remove items no larger than $tail(P)$
3. remove items included in less than θ transactions of $Occ(P)$
4. remove items e included in all transactions of $Occ(P)$,
   and record that "e is included in all transactions"
5. after removing items as 2, 3, and 4, remove duplicated transactions, i.e., if there are k same transactions, remove $k-1$ of them, and set the weight of T to k for keeping that there were $k-1$ more transactions equal to it.

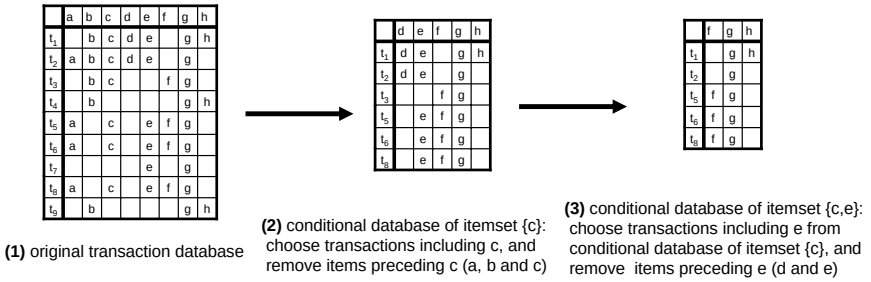


Figure 3: Conditional Database: example of construction

For any tail extension $P \cup \{e\}$, $Occ(P \cup \{e\})$ is obtained by choosing all transactions in $\mathcal{T}_P$ including e. The frequency of $P \cup \{e\}$ can be computed in a similar way. Thus we can do all operations in the recursive call with respect to P only with the conditional database of P. As the increase of the size of the itemset P, the size of its conditional database decreases exponentially. Thus the computation time for frequency counting is reduced in the bottom level. From the bottom-wideness, the total computation time for mining can also be drastically shortened. If a transaction has no weight, we consider the weight of the transaction is 1. Thus, through this paper, we assume that any transaction has a weight.

Actually, the term "conditional database" is used for many kinds of restricted databases in other papers, thus its definition is different from those in the other papers.

3.4 Prefix Tree

A *Prefix tree* is a tree shaped data structure for storing strings. Itemsets and transactions can be considered as strings composed of items, thus we can use prefix trees to store them. A vertex of the tree has an alphabet, thus a path of the tree gives a string. Each string is represented as the path from a representative vertex to the root. A vertex has no two children having the same alphabet, thus the position of the representative vertex of any string is unique. For any two strings having a common prefix, the paths representing them share the path corresponding to the prefix. See an example in Figure 1. The prefix tree is sometime called *frequent pattern tree*, or FP-tree in short, since it is also used in some algorithms to store the obtained frequent itemsets. Note that we sort items in each transaction to be stored in the order of their frequencies, so that many transactions share prefixes with others.

To compute the occurrences and the frequency of a tail extension $P \cup \{e\}$, we look up all vertices having e on them, and go down (opposite direction to the root) the tree from the vertices to find all the vertices assigned a transaction including P. The prefix tree can be also used for storing the conditional database. In the case, any occurrence of $P \cup \{e\}$ is assigned to a descendant of a vertex having e, and vice versa. Thus, the computation can be efficiently done, and computation time becomes short. This technique is popular, and many recent implementations use this technique[1, 2, 7, 8, 9, 13].

3.5 Occurrence Deliver

For an itemset P, the occurrence deliver computes the frequency and occurrences of all tail extensions of P at once. The occurrence deliver stores the transaction database by array lists. An array list is a list represented by an array such that the elements of the list is stored in the array. In an array list representation, we store each transaction of the input database in an array list so that the items in the array is sorted. See an example of array lists in Figure 1. For the initialization, the occurrence deliver assign an empty bucket for each item. Then, the occurrence deliver scans each transaction T_i in $Occ(P)$, and insert T_i to the bucket of each item included in T_i. After scanning all transactions in $Occ(P)$, the bucket of item e is equal to $Occ(P \cup \{e\})$.

The occurrence deliver works in the conditional database. The process of the occurrence deliver is equivalent to the transpose of the matrix of the conditional database, represented by array lists. We write the pseudo code of the conditional database version of the occurrence deliver below. We can also see an example of its execution in Figure 4.

ALGORITHM OccurrenceDeliver (P:itemset, $\mathcal{T}_P$:conditional database)
1. Set $Bucket[e] := \emptyset$ for each item e in $\mathcal{T}_P$
2. **For each** transaction $T_i \in \mathcal{T}_P$ **do**
3. **For each** item $e \in T_i$ **do**
4. Insert T_i to $Bucket[e]$
5. **End for**
6. **End for**
7. **Output** $Bucket[e]$ for all items e

LEMMA 1. *We can get the conditional database $\mathcal{T}_{P\cup\{e\}}$ by merging the transactions of $Bucket[e]$ including exactly the same items into one, and put the weights.*

Since array list is a compact form for sparse databases, the occurrence deliver is efficient for sparse databases[14, 15, 16]. With the use of conditional databases, it is also efficient for some dense databases, however, it takes much cost for quite dense databases[16].

4. NEW DATA STRUCTURE

Although the existing algorithms and data structures are efficient, they have their own disadvantage coming from the density and the structure of databases. The motivation of our new data structure is that we would have a good data

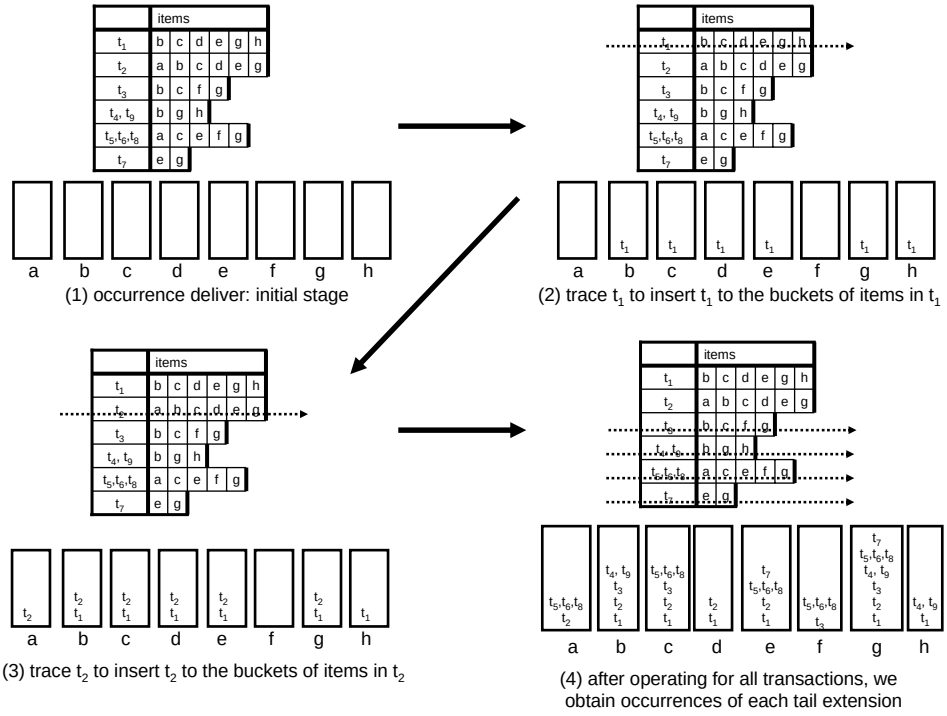


Figure 4: Occurrence deliver: example of its process

structure and a fast frequency counting algorithm by carefully combining them.

Our new data structure is very simple. When we input a transaction database, we choose a constant c. We can choose c so that the memory usage is minimal. Then, we use one integer $bit(T)$ and an integer array $ary(T)$ of size $|T^p|$. The ith bit of $bit(T)$ is 1 if and only if T^s includes item $n-i$, thus $bit(T)$ is a bitmap representation of T^s. Each item of the constant prefix T^p of T is stored in $ary(T)$ in the increasing order of items, thus $ary(T)$ is an array list representation of T^p.

For efficient frequency counting, we have a complete prefix tree for items greater than c. The complete prefix tree is composed of vertices i for $0 \leq i < 2^{n-c}$, so that vertex i corresponds to the transactions T such that $bit(T) = i$. For each vertex i, we have two integers $w(i)$ and $h(i)$. $w(i)$ is the *weight* of i, and used to keep the number (the sum of the weights) of transactions T such that $bit(T) = i$. $h(i)$ is the highest bit in i set to 1, that is, the smallest item in the constant suffix represented by i. The parent vertex of vertex i is the vertex $i - 2^{h(i)}$, where $i - 2^{h(i)}$ is the number obtained by setting $h(i)$th bit of i to zero. Similarly, any child vertex j of vertex i is obtained by setting hth bit of i to 1 for $h > h(i)$. In contrast to the usual prefix trees, we need neither pointers indicating the parent vertex and the child vertices.

Further, we have a bucket for each item, which is used by the occurrence deliver. The size of the bucket of item i is 2^{n-i} if $i > c$, and $|Occ(\{i\})|$ otherwise. We reuse the complete prefix trees and the buckets in every iteration, thus we have to allocate memory for them only once at when we input the database.

For the memory and time efficiency, we renumber the items of the input database so that the items of higher frequencies have larger indices. We can see an example of how to construct our data structure from an array list represented transaction database in Figure 5. Note that the items are sorted in the order of their frequencies so that items with high frequencies are left (it corresponds to renumbering). We can also see the complete prefix tree for the itemset $\{a, b, c, d\}$ in Figure 6.

The size of the complete prefix tree increases exponentially as the increase of c, and the memory space needed by array lists and the bitmap can decrease by the increase of c. Thus, we choose c so that the memory use is optimal, and larger than a specified minimum threshold such as 12, for speeding up. The optimal c can be found in $O(||\mathcal{T}||)$ time since the memory use can be computed from the frequencies of items. If $c = 12$, the complete prefix tree has 4096 vertices. It is not so large. However, if $c = 32$, the number of vertices grows up to 4,000 million.

We have some motivations and ideas on the new data structure. We briefly explain them below.

1. Many datasets in the real world are in the power law, thus the bitmap is efficient for and only for a small number of items.
Roughly speaking, the bitmap is efficient if items are included in at least 1% of transactions. Under the power law, databases include few, constant number of, such items. The part with respect to such items are dense part of the database, and the bitmap can compress it. For the sparse

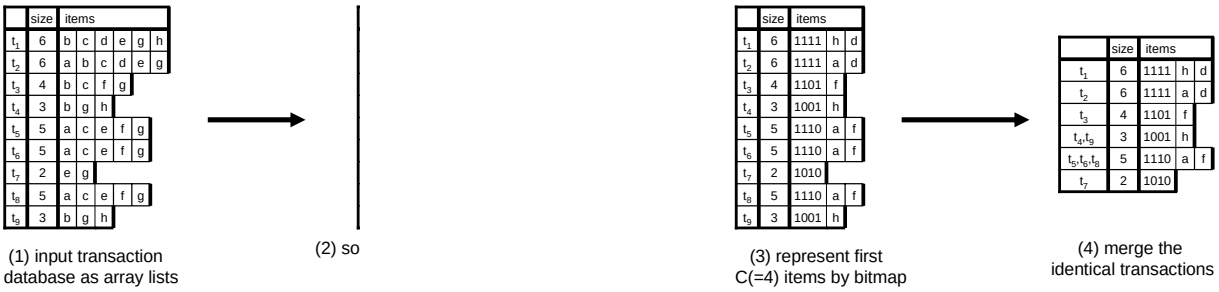


Figure 5: Example of the construction of our database

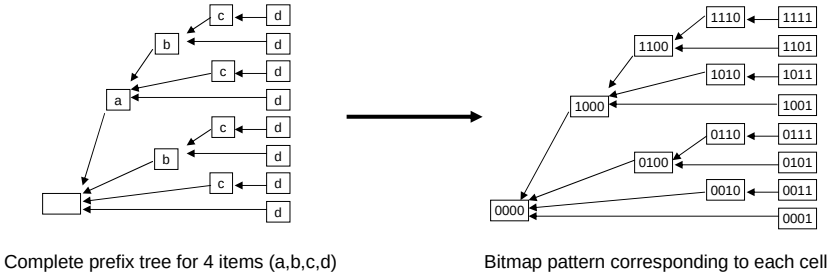


Figure 6: The complete prefix tree: each vertex of the tree on the right-side has the bitmap pattern corresponding to the string which the vertex representing.

(non-dense) part of databases, array lists are efficient. The prefix tree does not compress sparse databases as well as dense databases, but takes much cost. Hence, we do not use it for storing the transactions.

2. For the constant number of items, the size of the complete prefix tree is also constant.
Under the power law, the database has few items, usually bounded by a constant, with high frequencies. The use of the prefix tree for such items is efficient for frequency counting. Although the number of the items in the complete prefix tree is small, the computation time is drastically reduced because of bottom-wideness.

3. The complete prefix tree can be re-used.
A disadvantage of the prefix tree is that they need pointer operations and reconstructions. The complete prefix tree includes a vertex corresponding to any pattern composed of items larger than c, thus it needs no reconstruction, but only initialization of accessed vertices in the previous iterations. Thus, by using the complete prefix tree and reused it again and again in every iterations, we can avoid this disadvantage. It shortens the computation time of the iterations in the bottom level of the recursion, especially if the input database is dense.

4. In many iterations databases are small and dense
By sorting and re-numbering the items in the increasing order of their frequency, the computation time for frequency counting decreases, since the sizes of the conditional databases become smaller, more dense, and more structured on average. According to bottom-wideness, in many iterations, the conditional database includes at most c items. Hence, we can use the bitmap and the complete prefix tree, so that the computation time is reduced especially in the case that the input database is dense.

5. Constructing conditional databases becomes easy
The heaviest task in the constructing conditional databases for array lists is to find the duplicated transactions. Actually, it can be done by applying radix sort to the transactions. By using the bitmap, we can omit the computation of the radix sort for items with high frequencies. Thus, the computation time is reduced.

4.1 Frequency Counting with Complete Prefix Tree

By using the complete prefix tree, we can get both the frequency and the conditional databases for all extensions of the current itemset P by a sweep on the complete prefix tree.

Suppose that we have an itemset P and want to compute the frequencies and the conditional databases for all extensions of P. First, we initialize the complete prefix tree and the buckets, so that the weight of any vertex is zero and any bucket is empty. It can be done by initializing the vertices v with $h(v) > tail(P)$ of non-zero weights, and non-empty buckets of items $e > tail(P)$, hence we do not need initialize all the vertices and buckets.

After the initialization, for each transaction T in the conditional database $\mathcal{T}_P$, we add the weight of T to vertex $bit(T)$. Then, set the weight of vertex to the sum of weights of its

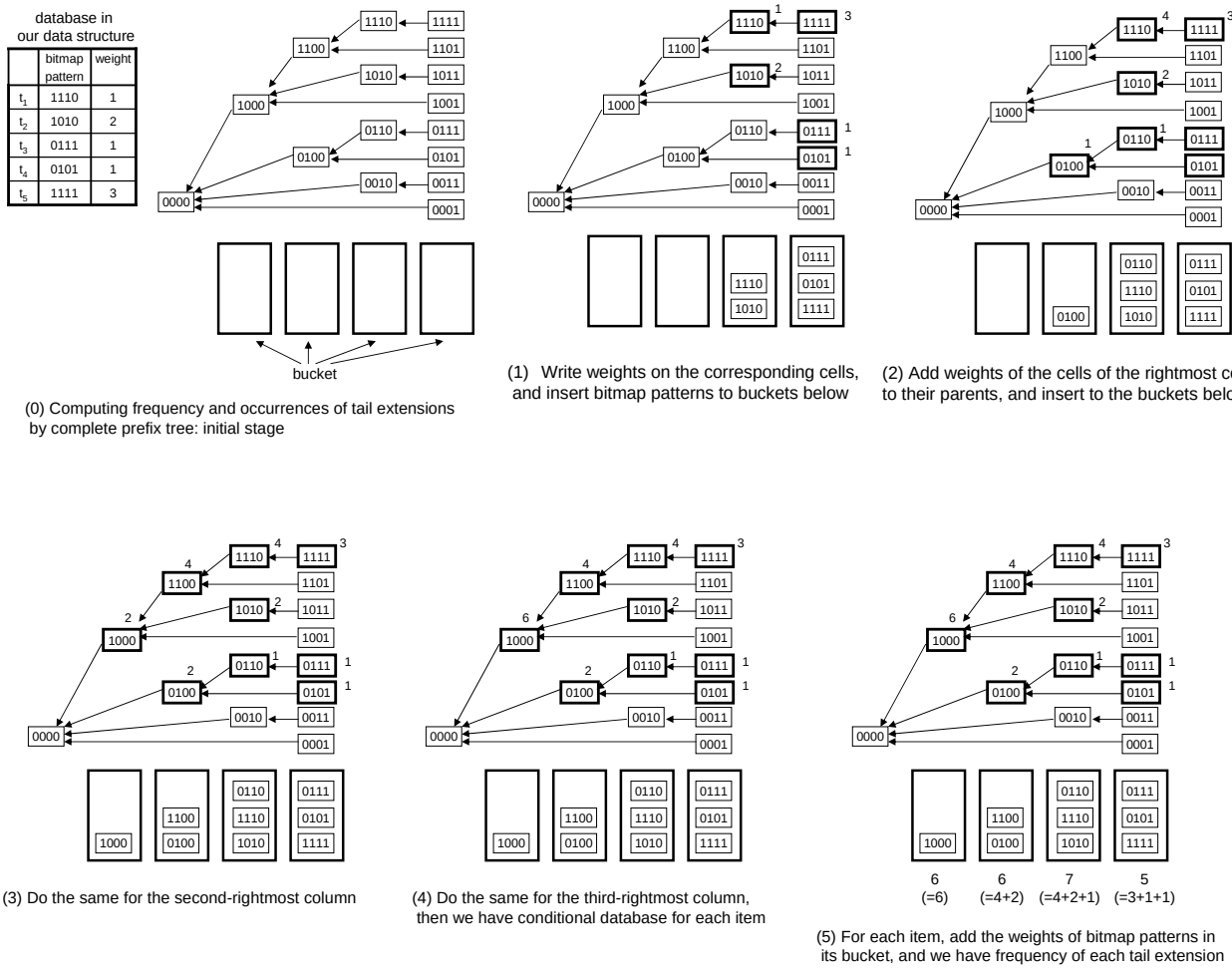


Figure 7: Example of complete prefix tree: re-use and frequency counting

descendants, and insert each vertex i of non-zero weight to the $(n-h(i))$th bucket. The former operation can be done in linear time of $|\mathcal{T}_P|$. In a bottom up way, or sweep the buckets in the increasing order, the latter computation can be done in time linear to the number of vertices with non-zero weights.

LEMMA 2. *After this operation, for any $i > c$, the ith bucket is the bitmap representation of the conditional database $\mathcal{T}_{P\cup\{i\}}$, i.e., each vertex and its weight in ith bucket are the bitmap representation of a transaction in $\mathcal{T}_{P\cup\{i\}}$ and its weight.*

From the lemma, we can see that the sum of the weights of the vertices in the ith bucket is the frequency of $P\cup\{i\}$ for any $i > c, i \notin P$. See an example in Figure 7.

4.2 Reuse of Complete Prefix Tree

To use prefix trees and the conditional database for frequency counting, we have to construct a prefix tree for the conditional database. We avoid this construction by reusing the complete prefix tree in every iteration, with the use of the rightmost sweep proposed in the first version of LCM[14].

Suppose that we have an itemset P, and completed the sweep described in the previous subsection. Then, we make recursive calls with respect to each tail extension $P\cup\{e\}$, in the decreasing order of items. In the recursive call with respect to $P\cup\{e\}$, the buckets of items no greater than e never be accessed. The vertices i with $n-h(i) \leq e$ never be also accessed. This shows that during the recursive call, the buckets and weights of vertices in the buckets are preserved for any item smaller than e. Therefore, we can reuse the complete prefix trees and buckets without disturbing the later recursive calls. This saves the memory space and computation time to allocate new memory space.

4.3 Prefix Maintenance for Closure and Maximality Checking

To apply our data structure to closed itemset mining algorithms and maximal frequent itemset mining algorithms, we have to modify the data structure so that we can compute the closure and checking the maximality in short time. In LCM ver.2[16], we propose a technique for efficiently

maintaining the prefixes of the transactions in conditional databases. Here the prefix of a transaction T in a conditional database $\mathcal{T}_P$ is the set of items in T no larger than $tail(P)$.

Suppose that transactions $T_1, \cdots, T_k$ are merged into one transaction T in $\mathcal{T}_P$. For taking closure, we put to T the intersection of the prefixes of $T_1, \cdots, T_k$.

For the check of the maximality, we take union of $T_1, ..., T_k$, and put it to T. We further put a weight to each item i of the prefix, where the weight is the number of transactions in $T_1, ..., T_k$ containing i. If the transactions are already merged in the operations previously done, the weight of i is given by the sum of the weights of i over $T_1, ..., T_k$.

These operations can be easily treated by our new data structure, just by doing them when we compute the frequencies and conditional databases of tail extensions in the way described in the previous subsection. Thus, our new data structures are efficiently applicable to closed itemset mining and maximal frequent itemset mining.

5. COMPUTATIONAL EXPERIMENTS

In this section, we show the results of our computational experiments. We implemented our new data structure and apply it for the second version of LCM and LCMfreq, which are for frequent itemset mining and frequent closed itemset mining. They are coded by ANSI C, and complied by gcc, and the machine was a PC with Pentium4 3.20E GHz CPU and 2GB memory. Due to the time limitation, we have no implementation for LCMmax for maximal frequent itemset mining. The performance of LCM algorithms are compared with the algorithms which marked good score on FIMI 03 or FIMI 04: fpgrowth[7], afopt[9], aim2[1], MAFIA[4, 5], kDCI and DCI-closed[10, 11], nonodrfp[2], and PATRICIA-AMINE[13]. We note that aim2, nonodrfp and PATRICIA-AMINE are only for all frequent itemset mining.

From the performances of implementations, the instances were classified into four groups, in each of which the results are similar. Due to the space limitation, we show one instance as a representative for each group.

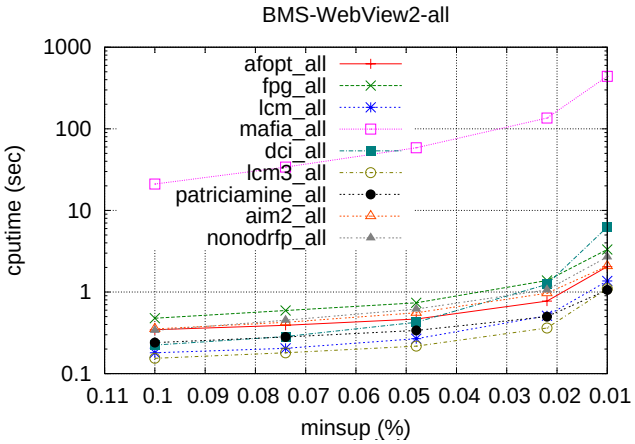
The first group is composed of BMS-WebView1, BMS-WebView2, T10I4D100K, and retail. These datasets have many items and transactions but are sparse, and even if the minimum support is very small, such as 10, the number of frequent itemsets is not so huge, i.e., frequent itemsets are enumeratable. We call these datasets *very sparse datasets*. We chose BMS-WebView2 as the representative.

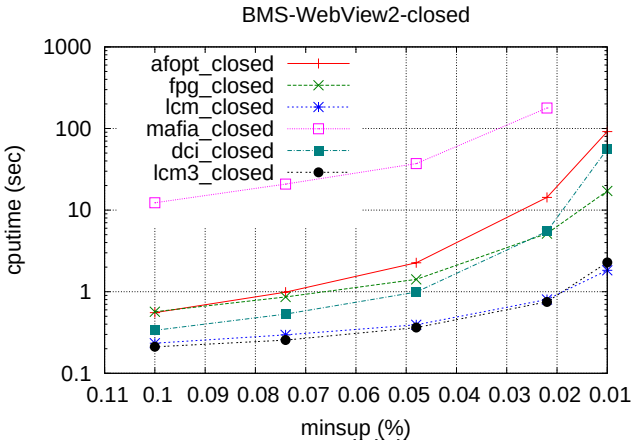
The second group is of mushrooms, BMS-POS, kosarak, Webdoc, and T40I10D100K. They are also sparse, but the number of frequent itemsets is quite huge when the minimum support is very small. We call them *sparse datasets*. We chose kosarak as the representative.

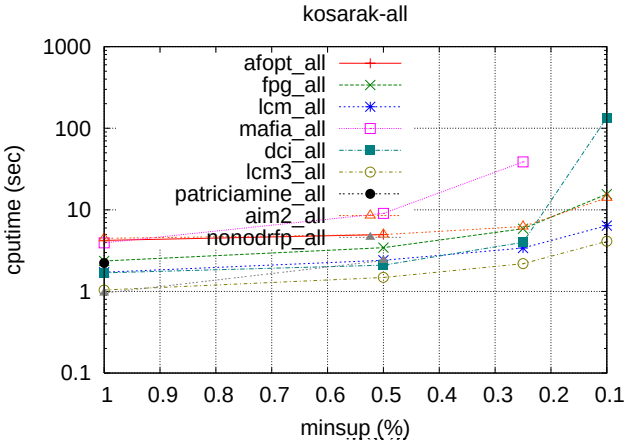
The third group is composed of connect, chess, pumsb, and pumsb-star. These datasets are generated in mathematical ways but are not natural data, thus they are structured. They have many transactions but few items. Transactions have many items, so the dataset is dense. Moreover, the number of frequent closed itemsets is very much smaller than the number of frequent itemsets for smaller minimum supports. We call these datasets *structured dense datasets*. As a representative, we show the result of chess.

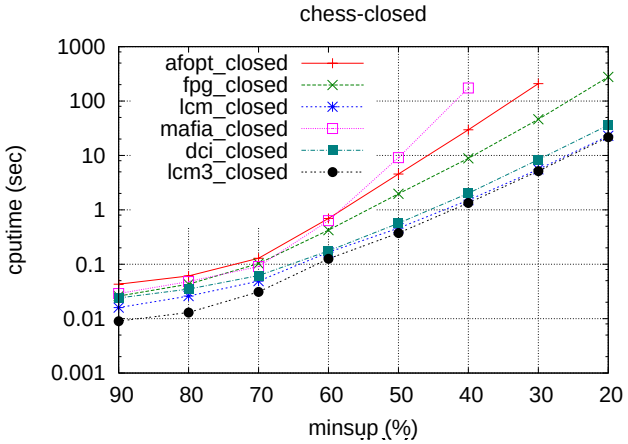
The fourth group is composed only of accidents, since its density is different from any other dataset. It has huge number of transactions, but few items. Each transaction includes many items, so the dataset is very dense, and is not structured so that the number of frequent itemsets and the number of frequent closed itemsets are almost equal for any enumeratable minimum support. We call this dataset *dense dataset*.

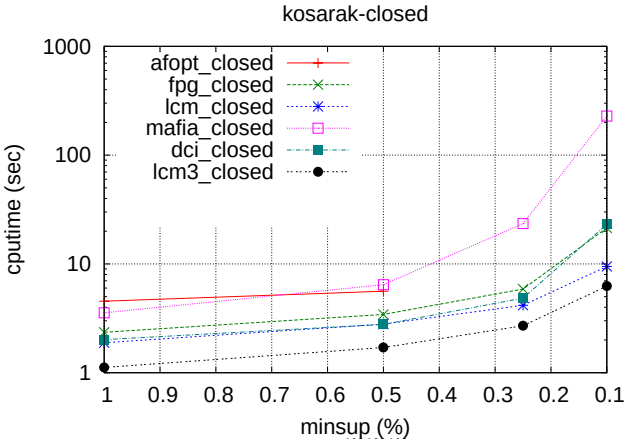
In the following, we show the results for representatives. The new implementation in this paper is written as "LCM3". To reduce the time for experiments, we stop the execution when an implementation takes more than 10 minutes. We do not plot if the computation time is over 10 minutes, or abnormal termination.

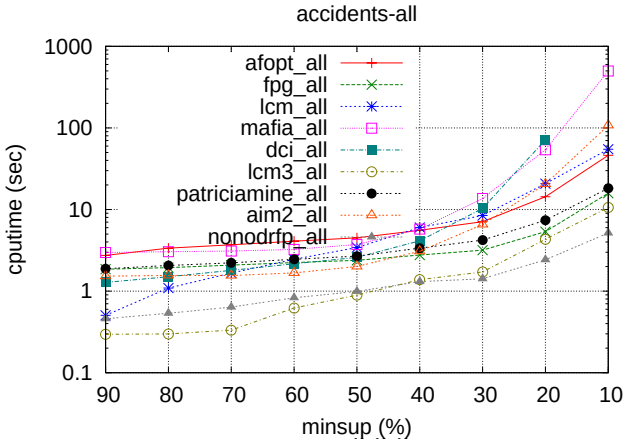


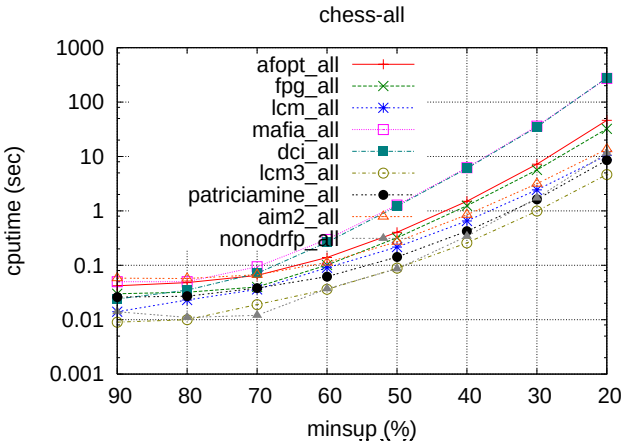


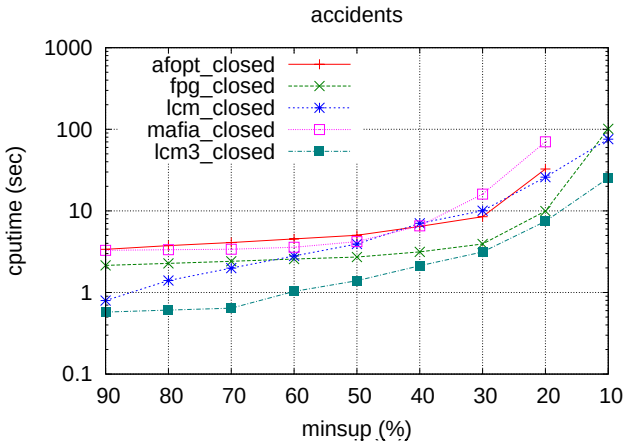












In almost instances and minimum supports, LCM3 performs the best. For unstructured dense datasets, and dense datasets with large minimum supports, nonodrfp sometime outperforms LCM3. For quite small minimum supports, LCM3 algorithms are fast but not the fastest. This is because of the cost for changing the strategy, which is from array lists to the prefix tree. If the cost per a frequent itemset is large, i.e., iterations generates few recursive calls on average, then LCM3 is slow. Moreover, in some sparse datasets, only few transactions of conditional databases share prefixes on average. In such cases, frequency counting with a prefix tree

takes longer time than that with array lists, hence the first version and the second version of LCM are fast.

For very sparse datasets, the memory usage is almost the same as array list, which is used in LCM ver.2. However, for the other datasets, the memory usage decreases to half in average.

6. CONCLUSION

In this paper, we proposed a new data structure for frequent itemset mining algorithms. We applied our data structure to the second version of LCM algorithms, and gave implementations of them. We show by computational experiments that our implementations perform above the other implementations for almost almost datasets in any minimum support, except for very small minimum support. For these very small minimum support, the second version of LCM is the fastest, thus we conclude that we can get a fast implementation by a combination of the second version and the new version of LCM algorithms.

For very huge datasets such that 10 or 100 times larger than the fimi datasets, optimizing the memory usage is quite important. In this sense, our data structure is not the best: perhaps the combination of array list and prefix trees (patricia trees[13]) might be the best. In particular, if the database is quite structured so that many transactions share prefixes, but includes no item of a high frequency. For maximal frequent itemsets mining and frequent closed itemset mining, if the number of output itemsets is small, then the checking the maximality and taking closure take long time rather than the way with storing all the itemsets obtained. Our data structure is not weak in these cases, but the next goal is how to perform better than others for such cases.

Acknowledgment

We sincerely thank to the organizers of FIMI03 and FIMI04, and all the authors gave implementations to FIMI, especially for Prof. Bart Goethals for his working for constructing and maintaining FIMI repository. This research is supported by Grant-in-Aid for Scientific Research of Japan and joint-research funds of National Institute of Informatics.

7. REFERENCES

[1] A. Fiat and S. Shporer, "AIM2: Improved implementation of AIM" In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[2] B. Racz, "nonordfp: An FP-growth variation without rebuilding the FP-tree," In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[3] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", In *Proc. SIGMOD'98*, pp. 85–93, 1998.

[4] D. Burdick, M. Calimlim, J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," In *Proc. ICDE 2001*, pp. 443-452, 2001.

[5] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A Performance Study of Mining Maximal Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[6] B. Goethals, *the FIMI repository*, `http://fimi.cs.helsinki.fi/`, 2003.

[7] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[8] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," *SIGMOD Conference 2000*, pp. 1-12, 2000

[9] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wang Wei, and Xiangye Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[10] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[11] C. Lucchese, S. Orlando and R. Perego, "DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets," In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[12] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Efficient Mining of Association Rules Using Closed Itemset Lattices*, *Inform. Syst.*, 24(1), 25–46, 1999.

[13] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[14] T. Uno, T. Asai, Y. Uchida, H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[15] T. Uno, T. Asai, Y. Uchida, H. Arimura, "An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases," *Lecture Notes in Artificial Intelligence* 3245, pp. 16–31, 2004.

[16] T. Uno, M. Kiyomi, H. Arimura, "LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets", In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.