International Journal of Innovative
Computing, Information and Control
Volume **9**, Number **8**, August **2013**

 ISSN 1349-4198
pp. 3115–3132

MINING FREQUENT PATTERNS BY THE GROUPING COMPRESSION TREE

Ray-I Chang and Chi-Cheng Chuang

Department of Engineering Science
National Taiwan University
No. 1, Sec. 4, Roosevelt Road, Taipei 10617, Taiwan
{ rayichang; d95525005 }@ntu.edu.tw

Received June 2012; revised October 2012

Abstract. *Traditional association rule methods that are used to discover large item sets must set the minimum support (MS) value in advance. However, users cannot in reality, determine an optimal value for the MS. Therefore, multiple frequent pattern mining processes with different MSs are often required to obtain satisfactory results. In this study, the authors propose a novel Grouping Compression Tree (GCT) data structure and develop an efficient GCT based mining method. Once a GCT is built, frequent pattern mining with different supports can be performed without rebuilding the tree. In designing the method, the authors have modified and synthesized a number of useful concepts including prefix trees, grouping compression techniques, and a projection approach. The method has been tested using several widely used test datasets. The performance study shows that the new algorithm not only speeds up the mining performance but also saves the memory usage in some cases.*

Keywords: Data mining, Association rule, Grouping compression, Projection, Adjustable threshold of support

1. **Introduction.** Discovering association rules that identify relationships among sets of items is an important problem in data mining that has been applied to many areas including e-commerce, prediction, security, and recommendation [1-4]. The problem of mining association rules can be divided into two phases: discovering the large item sets and using the large item sets to generate association rules [5]. The overall performance of mining association rules is determined primarily by the first step, which has therefore attracted significant research attention [6-9]. Existing algorithms must set the minimum support (MS) value in advance and then generate frequent patterns, depending on the MS. However, users could not, in realistic scenarios, determine an optimal value for the MS. The most difficult problem is how to select a good MS threshold because the MS is usually query-dependent. Multiple frequent pattern mining processes with different supports are often required to obtain satisfactory results.

To resolve this problem, several new approaches [9-11] have been proposed that do not require the MS value to be set in advance and can postpone the setting to the second phase, which is the stage in which the association rules are generated. The benefit of this type of approach is that it avoids the recurrent constructions of candidate item sets because of a change in the MS. These proposed data structures store in memory all of the transaction data that are not influenced when the MS is altered and thus, rescanning the database is not required. However, there are three persisting problems: 1. the process of construction and data structure is too complicated; 2. the ordering of the data is inconsistent, so that it will affect the search speed; 3. the activities of mining are too complicated. Due to above problems the performance is inefficient.

In this article, the authors propose a novel Grouping Compression Tree (GCT or GC Tree) data structure that groups and compresses frequent patterns that have the same prefix items, and that develops an efficient GCT based mining method, GC-mining, for mining the complete set of large patterns by projection techniques. The authors' goals that are to solve above problems are as follows. First, assure that the items are arranged concerted to reduce mining complications. Second, simplify the construction processes so that the building time can decrease. Finally, raise the performance by adopting the projection technique.

The remainder of this paper is organized as follows: Section 2 surveys related studies. Section 3 introduces the GCT structure, the algorithm for building it and the GC-mining algorithm. Section 4 presents some experimental results. Conclusions are given in Section 5.

2. **Related Works.** Association rules for a set of transactions in which each transaction is a set of items bought by a customer were first studied in [12]. Since then, this problem has been recognized as one of the important types of data mining problems and many algorithms have been proposed to improve its performance. The most important problem of mining for association rules is the method that can be adopted for efficiently generating the frequent item sets. The strategies for discovering frequent item sets can be divided into two categories [13]. The first category is based on the candidate generation-and-test approach. Apriori [6] and its several variations [14-16] belong to this category. The algorithm from Apriori computes the frequent item sets in the database through several iterations. Iteration i computes all of the frequent i-item sets (item sets with i elements) [5]. In this approach, a set of candidate item sets of length $i + 1$ is generated from the set of item sets of length I, and then, each candidate item set is checked to see whether it meets the support threshold. The second approach is the pattern-growth methods. These methods adopt a divide-and-conquer philosophy to project and partition databases based on the currently discovered frequent patterns and to grow such patterns into longer patterns in the projected database [17]. Algorithms in this category include FP-Growth [8], H-Mine [18], Tree Projection [19], and Opportune Project [20].

The candidate generation-and-test approach suffers from poor performance when mining dense datasets because the database must be scanned many times to test the support of candidate item sets. The alternative pattern growth approaches reduce the cost by combining pattern generation with support counting [13]. However, both of the strategies must set the MS in advance, to find out the large patterns. The generation process is usually iterative because the user must try a variety of thresholds to obtain a satisfactory result. Therefore, this time-consuming process must be repeated several times. To resolve this problem, several studies [9-11] have proposed solutions. They do not need to set the MS in advance and can postpone the setting to the stage when association rules are generated. Accordingly, the very large costs caused by recurrent re-construction operations can be reduced.

Woon et al. [10] proposed a SOTrieIT algorithm to hold pre-processed transactional data. This algorithm can generate large 1-item sets and 2-item sets quickly without scanning the database and can generate candidates for the second time. The efficiency of generating large 1-item sets and 2-item sets in the SOTrieIT algorithm improves the performance dramatically. However, it is difficult to use in an interactive mining system because the change in support can lead to repetition of the mining processes. Lin and Lee [11] presented a knowledge base assisted algorithm to take advantage of the knowledge that has been previously computed and to generate a knowledge base for further queries about sequential patterns with various support values. Nevertheless, the knowledge base

requires enormous spaces. Moreover, a suitable problem is the sequential pattern mining and not the association rules.

In [9], a Compressed and Arranged Transaction Sequence Tree (CATS Tree) is introduced. In contrast to existing approaches that need repeated construction processes, all of the transactions are mapped onto the CATS Tree structure. Once the tree is built, the algorithm must identify frequent item sets; it enables mining with different supports without rebuilding the tree structure. The benefit of "build once, mine many" increases with the amount number of frequent pattern mining that is performed. Nevertheless, the construction processes of the CATS Tree are too complicated because of the inconsistency in the order of items. Furthermore, this algorithm mines frequent item sets by constructing the conditional tree repeatedly and traverses both up and down to include all of the frequent items. The procedure is very elaborate and time-consuming. In this paper, a novel Grouping Compression Tree (GCT) data structure is proposed to improve the above-mentioned deficiencies. This proposed method can, in some cases, not only speed up the mining performance but also save the memory usage.

3. **GCT Approach.** In this study, a new data structure, GCT, and algorithm, GC-mining, are proposed to improve the efficiency of mining large item sets. The GCT is a compressed prefix tree that contains all of the transactions in the dataset. Paths from the root to the leaves in the GCT represent sets of transactions. Figure 1 shows the processes in the proposed method.

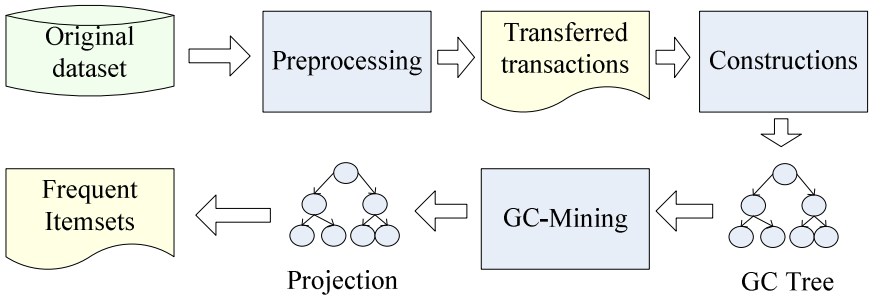

FIGURE 1. The processes in the GCT method

3.1. **GCT.** In the dataset, a very large number of transactions can contain the same set of items. The transaction volume can be reduced by grouping transactions that share common items. This operation could be performed using a modified prefix tree. In this article, the authors have adopted the grouping compression technique [13] to construct a GCT that can reduce the number of nodes in the original prefix tree. The technique groups together the nodes with common items. In a GCT, the last node of each pattern attaches a record to keep the compressed information. This record is to count the entries for each subset. Each count entry has two fields: the level of the starting item of the subset and its count. Figure 2 illustrates a GCT for the sample database.

In Figure 2, the record $(1, 1)$ of the extreme left node $\{n5\}$ means that there is one transaction with items starting at level 1, $\{n2\}$, for this node. The transactions correspond to the item set $\{n2, n3, n4, n5\}$. Each record represents a unique item set in the database. The starting node of all of the item sets must be the extreme left branch of the tree to avoid the duplication of item sets in the tree.

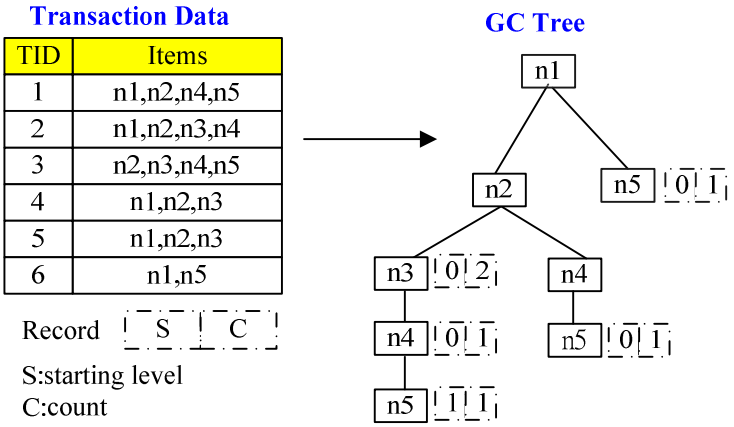

FIGURE 2. GCT for the sample database

3.2. **Pre-processing.** As mentioned in Section 2, the construction and mining processes of the CATS Tree method are very complicated because there is no pre-processing operation for the dataset. However, many industries in reality have a long span that can perform some pre-processing operations, for example, banks, supermarkets, and department stores. These industries have approximately ten hours off per day, which can be used to pre-process the dataset. The pre-processing process can simplify both the construction and the mining procedure of the GCT method, thus improving the efficiency of mining large item sets.

Figure 3 illustrates the pre-processing procedure. First, all of the transactions in the dataset are read to count the frequencies of items. Upon finishing the counting, the items are sorted in ascending order of their frequencies. Then, the items are mapped into an ascending sequence of n-integers, which are shown as the index row in the Index Table. Finally, the transferred transactions can be created with the sorted indexes.

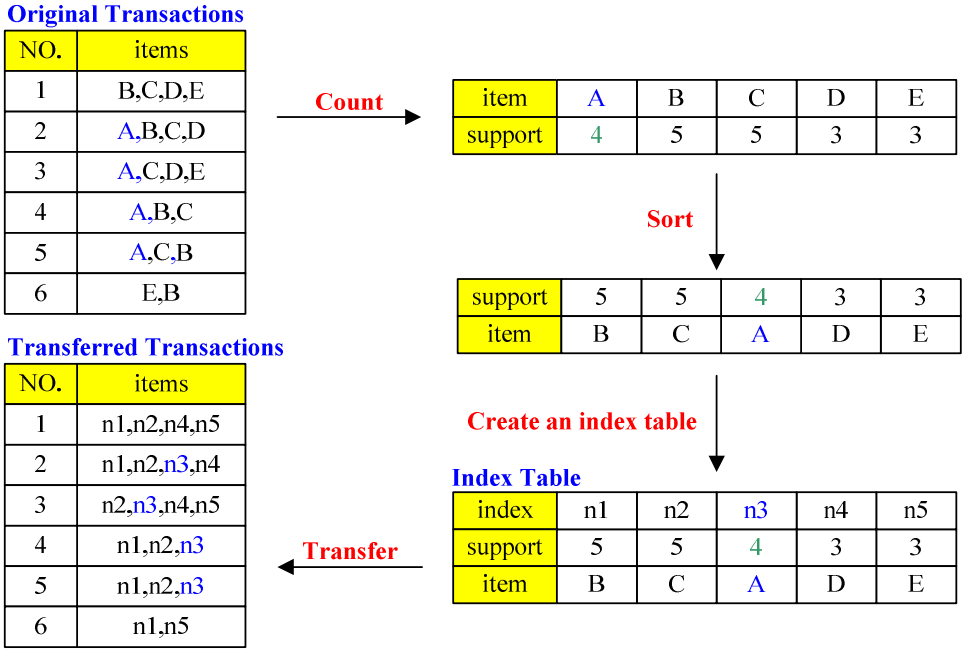

FIGURE 3. Example of the pre-processing process

3.3. **Construction.** After pre-processing, a GCT can be constructed with transferred transactions. The GCT method compresses a very large amount of data to reduce memory usage, and the data can be stored in memory, which decreases the time for scanning the database and increases the efficiency. The operation of the GCT construction is illustrated by the following example.

The authors have used the transferred transactions data shown in Figure 3 to illustrate the construction of a GCT. In step 1, all of the indexes in the Index Table of Figure 3 are read to construct a left skew tree in descending order of the supports. As mentioned earlier, this procedure avoids the duplication of item sets in the tree. In Figure 3, there are five indexes in the Index Table. The descending sequence of supports is $\{n1, n2, n3, n4, n5\}$. The left skew tree can be constructed as shown in Figure 4. Each node represents an index and has a corresponding record to keep its frequent information. The dotted rectangles show the records in Figure 4 as an illustration (they do not need to be created at the beginning).

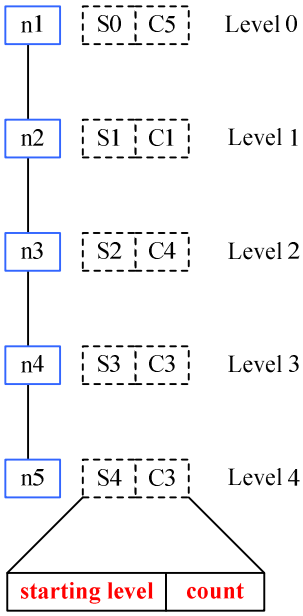

FIGURE 4. An example diagram of the left skew tree

In step 2, all of the transferred transactions are read to construct the GCT. Each transaction is inserted into the tree using the mapping indexes in the transferred index and is represented by one and only one path from its starting level. It is important to note that the starting level of each transaction must be from one of the left skew nodes. The ordering of items labeling nodes along any path follows the imposed ordering, which is the order of the indexes in the Index Table of Figure 3. The operations of the GCT construction are illustrated in Figure 5 and the algorithm is given in Figure 6. For example, initially, the GCT is a left skew tree that contains only five nodes. Transaction 1, $T1\{n1, n2, n4, n5\}$, is performed first. As shown in Figure 5(a), the existence of the item set, in other words, the path in the tree, is checked first. If the path is present in the tree, then the count of the path is increased; otherwise, a new path must be created. As described in Section 3.1, each path of the tree has an additional record that maintains a count of the transactions. In Figure 5(a), the record $(0, 1)$ is attached to the last node $\{n5\}$. This record represents that the starting level is 0 and the count is 1. After adding the transaction 3, $T3\{n2, n3, n4, n5\}$, the common items, $\{n2, n3, n4\}$, can be found.

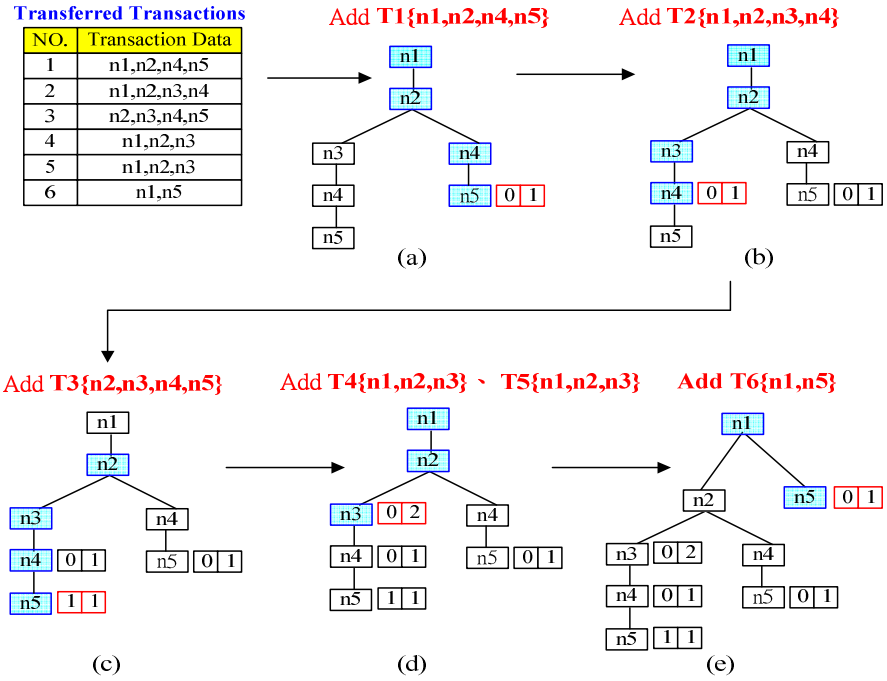

FIGURE 5. An illustration of the construction processes of the GCT

The GC Tee construction algorithm is as follows：

```
(1)   input：transferred transactions
(2)   output：GCT
(3)   BuildGCT(transferred_transactions trans[]){
(4)   If(GCT = = null)
(5)     Create a left skew tree；
(6)   Else{
(7)     For(int i = 0；i<trans.length；i++)
(8)       AddTran(trans[i])；
(9)     }
(10)  return GCT；
(11) }
(12) AddTran(item_index tran[]){
(13)    For(int i = 0；i<tran.length；i++){
(14)      If(i = = 0){
(15)        current_node = the node which index the same with i
(16)          s = current_node's level； //start level
(17)      }
(18)        else{
(19)      if(tran.[i] = = current_node.child's index label)
(20)        current_node = current.node.child；
(21)        else{
(22)        create a new node；
(23)          new node's index label = tran[i]；
(24)        current_node.link = new node；
(25)        current_node = new node；
(26)        }
(27)      }
(28)        if(i = = tran.length - 1)
(29)    modify current node's record table
(30) }
```

FIGURE 6. The algorithm of GCT construction

Then, the common items can be grouped together as shown in Figure 5(c). The last node, $\{n5\}$, corresponds to the record $(1, 1)$, which means that the item starts at level 1.

Figure 5(e) shows the result of step 2 for the sample database. From this figure, there are only eight nodes in the GCT, whereas the original item sets have 20 items.

Explanation: In the algorithm shown in Figure 6, the (1)st row indicates that the input data is the transferred transaction data; the (2)nd row indicates that the last output data is a GCT. First off, in the (3)rd row we read the transferred transaction data: represented by the trans. The (4)th to (5)th rows are mainly for checking whether GCT is null; if it is null, the action in step 1 should be done (to create the left skew tree). If the GCT is not null, it means the step 1 is already completed and in the (6)th to (9)th rows we will add data one by one into the GCT. The (8)th row indicates that when the function AddTran() will be called for subsequent procession when a transaction data is read. In the end, when all the above are constructed the (10)th row will output the complete GCT. Next, let us take a closer look at the processed content of function AddTran(). In the (12)th row we start with reading a transaction data, and this data may have more than one index number; therefore, we need the arrays to receive the index numbers in the transaction data. The (14)th to (17)th rows are mainly for finding the initial class of the read transaction data, and the (19)th and (20)th rows are for judging whether the remained index numbers in the transaction data are located in the subsidiary trees of initial class nodes; if not, the (21)th to (26)th rows should be operated to search for the unfound node and concatenate them in the *current_node*. Lastly, we should judge whether we should execute till the last number of the transaction data; if so, we will modify the node record table that corresponds to this number, which is the action we do in the (28)th to (29)th rows.

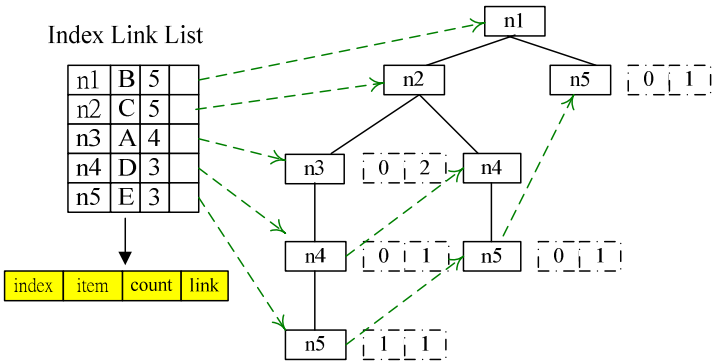

FIGURE 7. The GCT with an ILL table

In step 3, by traversing the tree, the Index Linked List (ILL) linked to the GCT is developed. The ILL table contains all of the individual index records. Each record consists of four fields: the index, the corresponding item, the support count and a link to the first occurrence of that item node in the GCT. Figure 7 shows the result for the final GCT in Figure 5(e). As shown in the figure, the same indexes in different transactions can be connected by the ILL table. In other words, there is a link that connects all of the same items in the GCT. In the later mining stage, the links can facilitate filtering the items that lower the frequency of the MS, thus improving the mining performance. Moreover, both the mapping indexes and the original items are stored in the table, which saves the time that would otherwise be needed to convert the indexes to the items when presented with large item sets. The mining process is described explicitly in the next section.

3.4. **GC-mining algorithm.** Similar to the CATS Tree [9], once the GCT is built, it can be mined repeatedly for frequent patterns with different support thresholds without the

need to rebuild the tree. The "build once, mine many" property can increase the mining performance. However, similar to existing methods [8], the CATS Tree algorithm mines frequent itemsets by repeatedly constructing a conditional tree. There are two problems that should be considered. One problem is the CPU time consumption because of the recurrent construction; the other problem is the need for extra spaces in the conditional tree. In this study, a new algorithm called GC-mining is proposed to mine large item sets; this algorithm adopts the pseudo projection technique [20]. This technique involves projection of a subtree of node x from the original GCT; this subtree shares nodes with the projection subtree. Hence, this algorithm can be CPU time efficient and can save memory. The key points in the mining processes are discussed next.

3.5. **Upward counting.** Unlike the FP tree or the CATS tree, the records in the GCT are attached only to the last node of each pattern. Hence, the GC-mining algorithm must accomplish the process of upward counting. It searches every extreme left node in the GCT from the bottom to the top by an index linked list. Supporting the search node is cn (current node), and the count of cn is added to its pn (parent node) recurrently until the starting level of cn is larger than the level of pn. For example, in Figure 8(a), the process of upward counting begins at the bottom node, $\{n5\}$. The mining algorithm traverses the entire node $\{n5\}$ in the GCT, following the linked list and adding the count of $\{n5\}$ to its parent node. Figure 8(d) shows the final result.

The process of upward counting can be built either as temporal or permanent according to the memory availability. This process is arranged at the last step of construction to save the mining time while the memory is sufficient, whereas the process is postponed to the mining stage for reducing the memory usage.

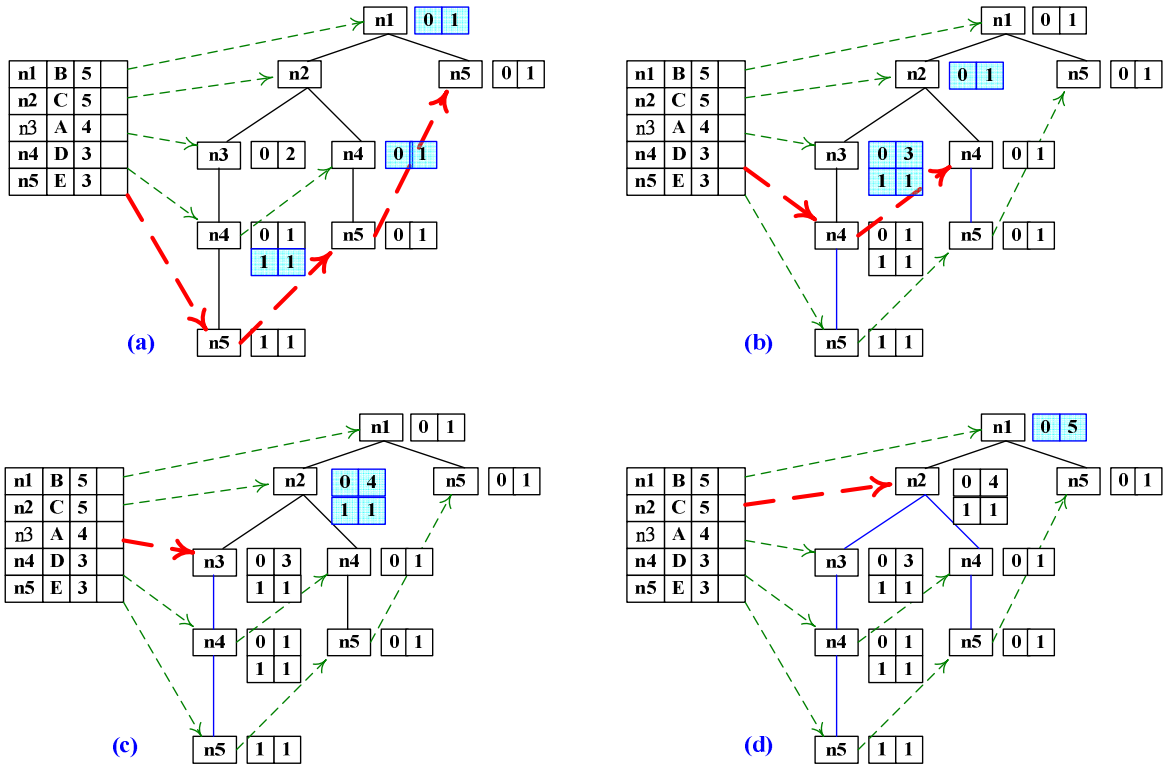

FIGURE 8. The process of upward counting

3.6. **Bottom-up mining.** In the mining step, all of the frequent item sets are mined by using the GC-mining algorithm with a recursive function and a bottom-up projection strategy. For a pattern r, the projection tree of r is a tree projected from transactions that contain pattern r. Transactions contained in a projection tree can be easily gathered by traversing the indexed links of pattern r. For example, suppose that the user wants to obtain frequent item sets with an MS 3. The operation of the projection is illustrated in Figures 9 through 11. Each item in the ILL table is used as a starting project point to mine all of the longer frequent item sets. Using bottom-up as a projection strategy, the bottom item $\{n5\}$ is explored first, and the topmost item $\{n1\}$ is explored last. Starting with item $\{n5\}$, the link is followed to obtain all of the other items that occur together with item $\{n5\}$, and the count of each item is accumulated. As shown in Figure 9, the item $\{n5(3)\}$ is frequent where the support of each pattern is given in parenthesis. Because there are no subnodes under item $\{n5\}$, item $\{n4\}$ is mined next. From Figure 10, one can see that item $\{n4(3)\}$ is frequent. The projection tree of item $\{n4\}$ is then built because item $\{n4\}$ has children nodes. In Figure 10, the item set $\{n4, n5(2)\}$ is discarded because the support is smaller than the MS.

Generating frequent patterns in GC-mining is a recursive procedure. After generating the frequent item sets of items $\{n5\}$ and $\{n4\}$, GC-mining recursively generates frequent sets of items $\{n3\}$, $\{n2\}$, and so on. In the end, the recursive process calls item $\{n1\}$. Figure 11 shows the final result of node $\{n1\}$. First, a frequent item $\{n1(5)\}$ is obtained. Second, projecting $\{n1\}$, there are two patterns that are larger than the MS: $\{n1, n2(4)\}$ and $\{n1, n3(3)\}$. Then, the projection tree of the two patterns are created and a large item set $\{n1, n2, n3(3)\}$ is obtained. The algorithm of GC-mining is given in Figure 12.

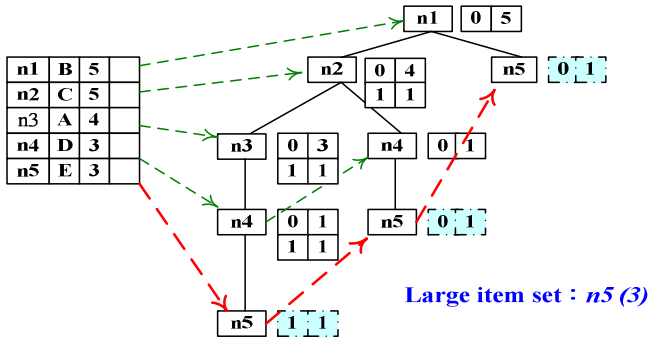

FIGURE 9. The projection of index $n5$

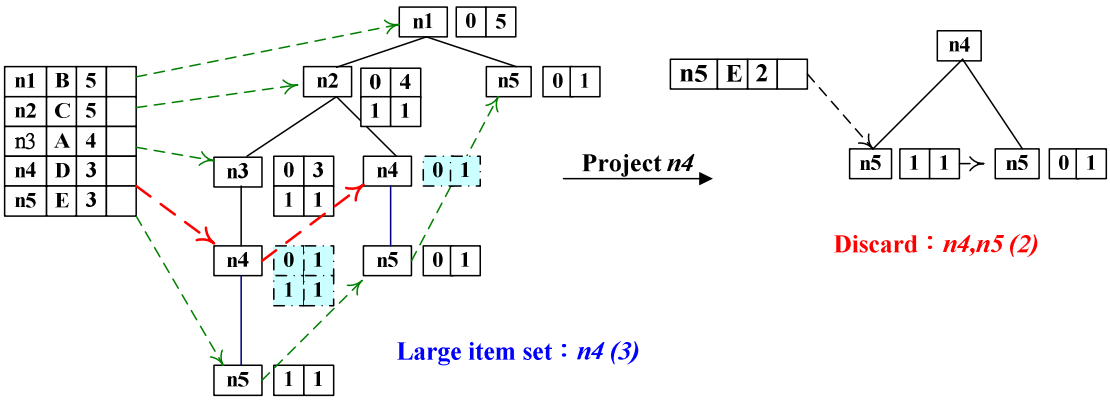

FIGURE 10. The projection of index $n4$

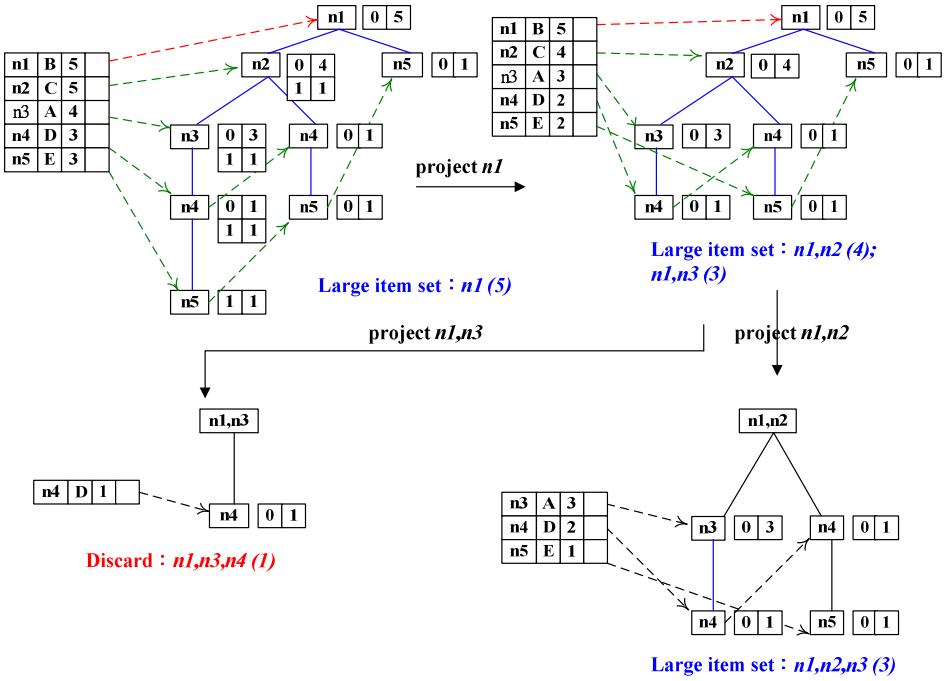

FIGURE 11. The projection of index $n1$

```
(1) input：GCT, index linked list, support threshold
(2) output：large item sets
(3) MineLISets(GCT gt , index linked list ind , support_threshold MS){
(4)     For(int i=ind.length-1;i>=0;i--){
(5)       if(ind[i].count >= MS){
(6)         add ind[i].item to large item sets and support = ind[i].count;
(7)           GCT pTree = projectTree(ind[i].index);
(8)           index_link_list new_ind = create_index_link_list(pTree);
(9)           MiningProjectionTree(pTree,new_ind,ind[i].item);
(10)    }
(11)  }
(12) }
(13) MineProjectTree(GCT pt , index_link_list ind , LargeItemSet L){
(14)  For(int i=ind.length-1;i>=0;i--){
(15)    if(ind[i].count >= MS){
(16)          LargeItemSets L1 = L ∪ ind[i].item;
(17)          add L1 to large item sets and support = ind[i].count;
(18)          GCT pTree = projectTree(ind[i].index);
(19)          new_ind = create_index_link_list(pTree);
(20)          MiningProjectionTree(pTree , new_ind , L1);
(21)        }
(22)   }
(23) }
```

FIGURE 12. The algorithm of GC-mining

Explanation: In the algorithm, the (1)st row indicates that the input data is GCT, index linked list, and MS set by the user. The (2)nd row indicates that the data is output

as large item sets after the handling. At first the function MineLISets in (3)rd row is operated, in which the three data initially input would be received. Then the (4)th to (11)th rows will do bottom-up mining, in which the (4)th row will start operation from the last data in the index linked list, and the (5)th row will judge whether it meets the MS limitation. If the action in (6)th to (10)th row is completed, the (6)th row will add the product items larger than MS into large item sets and record the support size; the (7)th row will project the index numbers that comply with MS and require the new GCT, then the (8)th row will construct the index linked list in connection with the projected new tree; lastly the (9)th row will do further mining in the projected new tree and call the function MineProjectTree. The (13)th to (23)rd rows are the handling actions of the function MineProjectTree; at first in the (13)th row the three parameters which include the projected new tree, the trees' index linked list, and the current large item set L are received; then in the (14)th to (22)nd rows the bottom-up mining is performed in the received new tree. Different from the (6)th to (10)th rows, the (16)th row will unite the data items that meet MS setting and the received large item set L by function MineProjectTree. Repeatedly, eventually we are able to receive all the large item sets that meet MS setting.

Observably, one can find the advantages of the projection technique. First, there is no need for extra spaces. Second, the projection for one specific node is traversed by the index linked list. The procedure is concise and efficient. Third, this approach can speed up the mining time.

4. **Performance Evaluation.** In this section, a performance comparison of the GCT algorithm with the recently proposed efficient method, the CATS Tree algorithm, is presented using several artificial datasets and two real-world datasets. All of the experiments are performed on an AMD Athlon XP 2500 PC machine with 512 megabytes of main memory, running on Microsoft Windows XP. All of the programs are written in Java. The algorithm of the CATS Tree has been implemented to the best of the authors' knowledge based on the published paper. A comparison has been made in the same running environment. To make the time measurements more reliable, no other application was running on the machine while the experiments were running.

4.1. **Datasets.** The datasets that were used in these experiments are described in this section. The synthetic datasets that were used for these experiments were generated with the data generator by IBM QUEST [21]. All of the data files were generated with the following parameters: the average length of transactions was 10; the average length of large item sets was 4; and the number of distinct items was 500. Four values were chosen for a number of transactions: 50k, 100k, 150k, and 200k. The two real-world datasets, BMS-WebView-1, and BMS-WebView-2, were used in the KDD-Cup 2000 competition [22]. The BMS-WebView-1 and BMS-WebView-2 datasets contain several months' worth of click stream data from two e-commerce web sites. Each transaction in these datasets is a web session that consists of all of the product detail pages viewed in that session [23]. The basic features of the real-world datasets are listed in Table 1.

Table 1. Characteristics of the datasets

	Trans. Number	Distinct Items	Max Trans. size	Aver. Trans. Size
BMS-WebView-1 (D1)	59,602	497	267	2.5
BMS-WebView-2 (D2)	77,512	3,340	161	5.0

4.2. **Experiments on synthetic datasets.** The scalability of the GCT and CATS Tree algorithms were first studied using synthetically generated datasets. As mentioned earlier, the tree builders of the two methods require only one run. Once the trees have been built, they can be used for multiple mining efforts with different supports. To test the building time scalability with the number of transactions, experiments on several datasets are used. The results are presented in Figure 13.

Both the GCT and the CATS Tree algorithms show linear scalability, with the number of transactions set from 50K to 200K. However, the building time of the GCT is less than the CATS Tree in all of the transactions. This relationship occurs because the construction process of GCT is static and has no adjustment or swap actions. In contrast, the CATS Tree method adopts complicated dynamic constructions. Because there is no preprocessing stage in the CATS Tree method, this method must combine nodes that have the same frequency continuously; furthermore, it must adjust nodes upwardly and exchange sibling nodes in its construction.

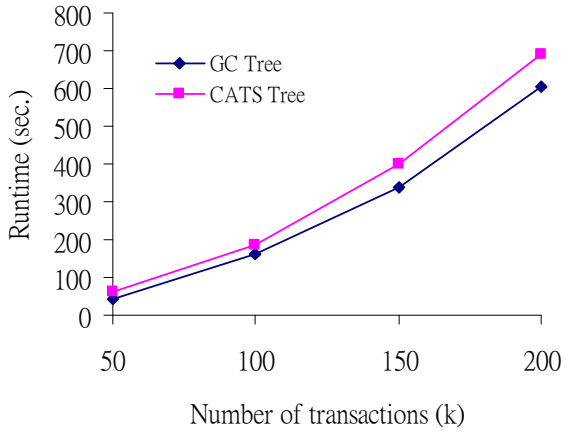

FIGURE 13. Scalability with the number of transactions

The second experiment is for a performance measure that evaluates the mining time (seconds) of the two algorithms on one dataset, with the MS threshold decreasing from 1.25% to 0.25%. Figure 14 shows performance curves of the two algorithms. The time required by the two algorithms increases as the support decreases. However, the GCT algorithm scales much better than the CATS Tree algorithm at all of the support thresholds. This improvement occurs because the items that have different patterns in the GCT are arranged in the same order because of the pre-processing process. The GC-mining can simply mine the frequent item sets using a one-way process, whereas the CATS Tree method traverses the tree both up and down to ensure that all of the frequent patterns are captured. Although the CATS Tree construction needs only one database scan, its mining performance is limited to the items order of the dataset. The CATS Tree method is more complex than GC-mining and hence needs a longer run time than the GCT. Furthermore, GC-mining uses a projection technique instead of the recurrent construction of the conditional tree, thus reducing the mining complexity and saving running time. In Figure 14, the dataset to be used in the experiment contains 100k transactions. The other three datasets (50k, 150k, and 200k) show similar results. These figures have been omitted for space reasons.

Other than performance, the memory requirements for the two algorithms are compared. From Figure 15, one can see that the memory usage of the CATS Tree is smaller

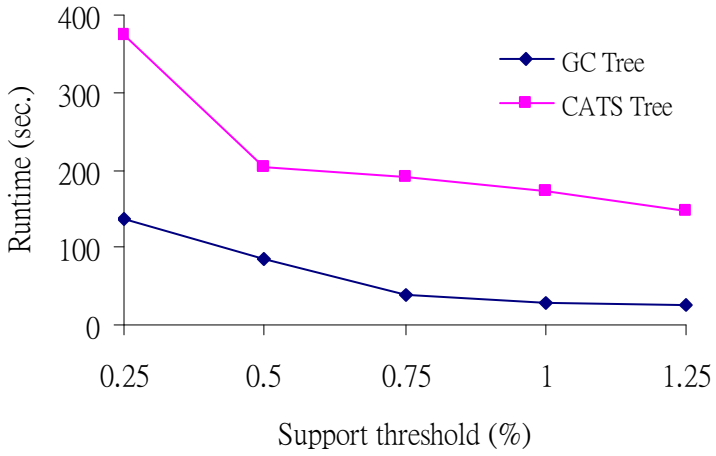

FIGURE 14. Scalability with the GCT and the CATS Tree with different support thresholds

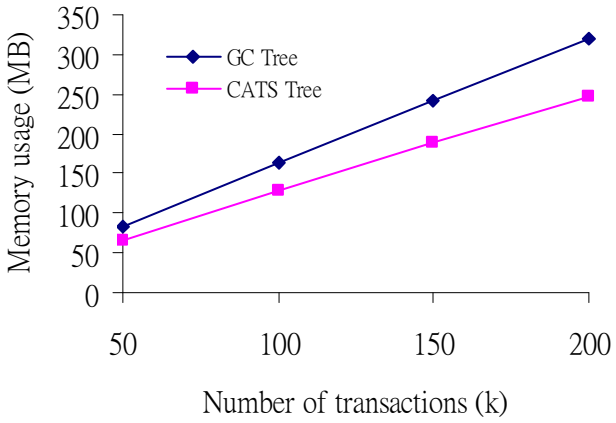

FIGURE 15. Memory comparison of the GCT and the CATS Tree, with different transactions

than that for the GCT. Because the CATS Tree adopts a dynamic construction, it can perform further adjustment and compression of the nodes. In the GCT, although the construction process uses a group compression technique to reduce the needed space of the nodes, it does not include an advanced combination and adjustment of the nodes. Therefore, the GCT consumes 25% more memory than the CATS Tree. However, the synthetic datasets are generated with random functions by IBM QUEST. The probabilities of the patterns are equal, which reduces the grouping effect of the GCT. In the next section, further analysis and discussion of the GCT performance are presented from using the real-world datasets of the KDD-Cup 2000 competition.

4.3. **Experiments on real-world datasets.** To further test the peformance of the two algorithms, experiments on the real world datasets BMS-WebView-1 (rD1) and BMS-WebView-2 (rD2), which were used in the KDD-Cup 2000 competition, were used. The support thresholds were from 0.125% to 1.25%, and the results are presented in Figures

16(a) and 16(b). According to the experimental results, the GCT runs much faster than the CATS Tree. However, in the rD2 dataset, the two methods perform almost equally when the support threshold is more than 1%, but the GCT is better than the CATS tree.

The main cost of the CATS Tree involves constructing conditional trees and mining bi-directionally. In a database with a large number of frequent items, the conditional tree can become quite large, and the computational cost can become high. However, the GCT method uses a projection technique and is mined one way, which saves nontrivial costs. This scenario explains why the GCT has distinct advantages.

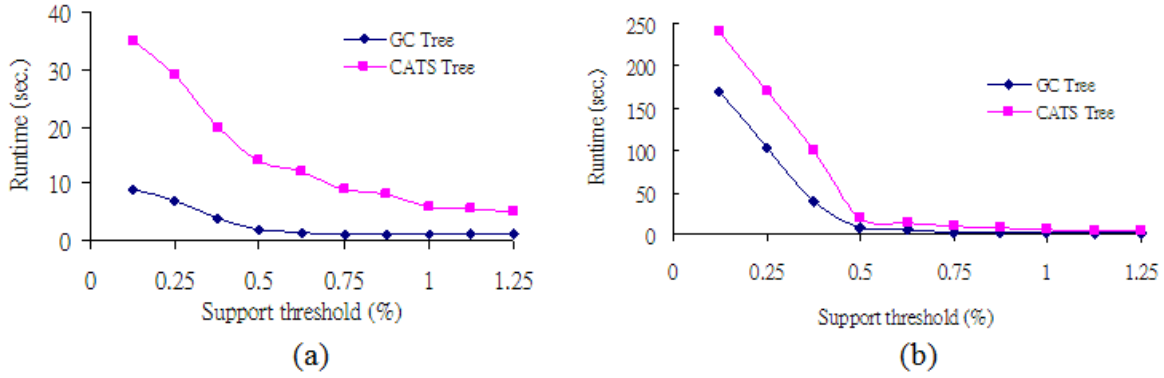

FIGURE 16. Scalability of the GCT and the CATS Tree with different support thresholds, using (a) rD1 and (b) rD2

Furthermore, in experiments that use real-world datasets, the compression efficiency of the GCT was significant. Figure 17 shows the memory usage comparisons of the GCT and CATS Tree with four artificial datasets and two real-world datasets. Obviously, from this figure it can be seen that the memory usage of the GCT is more than the CATS Tree by approximately $20\% \sim 30\%$ on the synthetic datasets ($50\text{K} \sim 200\text{K}$). However, the ratios reduce to $0.6\% \sim 0.75\%$ in real-world datasets (rD1 and rD2). These results prove the inferences that were mentioned earlier. Because the synthetic datasets are generated randomly, the probabilities of all of the patterns are equal, which reduces the grouping effect of the GCT. It is very clear that the artificial dataset has a very different grouping effect when compared with the real-world datasets, whereas the distributions of the two real-world datasets are similar.

Although the GCT expends more memory during the construction stage, it does not require extra memory during the mining process because it adopts the projection technique, whereas the CATS Tree recurrently constructs conditional trees that require extra space to store. The total memory consumption of the two methods is compared, including the construction space and the mining space. Figures 18(a) and 18(b) show the memory usage for the two algorithms on two real-world datasets.

Unlike other traditional frequent pattern mining algorithms, the GCT and the CATS Tree can be built once and can create only one tree. As shown in Figures 18(a) and 18(b), the memory usage of construction Trees is unrelated to the support, whereas the CATS Tree mining process is sensitive to the support and requires extra memory. For the purpose of comparison, in the two figures, the memory usage that is required for mining using the CATS Tree is added to the memory usage for the CATS Tree Total. In Figure 18(a), the GCT consumes less memory than a CATS Tree with all of the supports. In another case, the GCT could spend more space than the CATS Tree, an example of which appears in Figure 18(b).

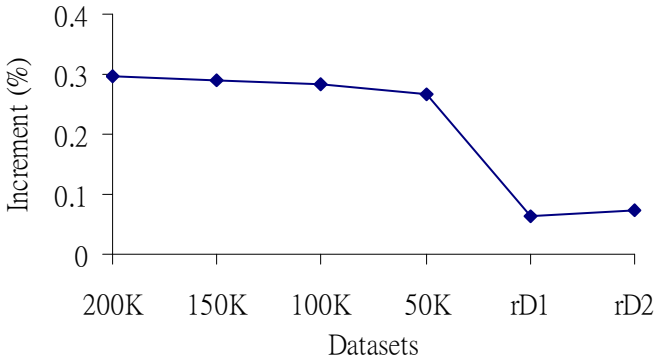

FIGURE 17. The increasing ratios of memory usage for the GCT over the CATS Tree, using four synthetic datasets and two real-world datasets

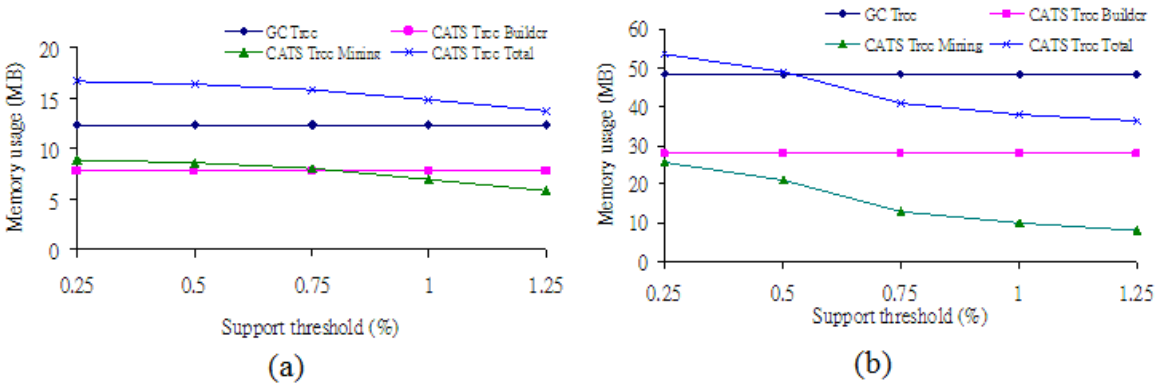

FIGURE 18. Memory usage comparison with the GCT and the CATS Tree, with different support thresholds and using (a) rD1 and (b) rD2

4.4. **The experiments for adding real-time dynamic data.** This section compares the processing efficiency of the algorithms when adding new data. How to rapidly output sale strategies and whether the algorithm is applicable are relatively important in actual use such as the sales in a marketplace where customers constantly shop and the data keeps increasing in the database. This experiment conducted real data BMS-WebView-1.dat, to take $x\%$ of the original data to construct GCT and CATS Tree, the rest $(100 - x)\%$ was added in; then the space and mining time were compared. The experiment results are shown as Table 2.

From Table 2, we are able to see that the adding action in GCT and CATS Tree did not show huge difference spatially, and GCT is also better than CATS Tree. In the process of the adding action, it may cause the index to show the items which appear less frequently instead of the items that appear the most frequently; however, since GCT uses the grouping compression technique, no matter what the index an item was allocated with, the grouped nodes can still save the required node space by grouping compression. That is why the adding action does not affect too much spatially. Besides, since the CATS Tree uses dynamic compression technique the adding action does not affect the space too much, either. However, the GCT still performs better than CATS Tree in the adding process. According to the results, the GCT processes $3.2 \sim 3.6$ times faster than CATS Tree which makes it very applicable in the rapid data process area.

TABLE 2. The experiment data using the real data BMS-WebView-1.dat and under MS = 0.25%

Adding Ratio	GCT		CATS Tree	
$(100-x)\%$	Node Number	Mining Time (s)	Node Number	Mining Time (s)
10%	55,219	8	52,221	29
30%	55,376	8	52,267	29
50%	55,794	8	52,293	29
70%	56,324	9	52,337	29
90%	56,543	9	52,374	29

4.5. **The comparison between GCT and CATS Tree.** GCT and CATS Tree are both the data structures built in connection with the original transaction data; however, CATS Tree still show insufficient performance in the process of constructing and mining efficiency. For example, CATS Tree is not applicable in the areas of rapid mining demand. Therefore, this paper promotes a new structure, GCT, to better the procession. From Table 3 we can see the difference between GCT and CATS Tree. It is obvious that the GCT is mainly for simplifying the methods of CATS Tree and speeding up the mining process for large item sets.

TABLE 3. The comparison between GCT and CATS Tree

GCT	CATS Tree
Twice of database scanning is needed.	Once of database scanning is needed.
The data is pre-processed and sorted, and the nodes in GC Tree appear in order and make it easier to search.	The data is not pre-processed and sorted, so that the products under different paths in CATS Tree appear randomly and slow down the search.
GCT uses the grouping compression technique to further compress the prefix trees.	CATS Tree uses dynamic adjusting to achieve partial maximization of sub trees.
The nodes in GCT represent the index numbers of products, so the data size is smaller.	The nodes in CATS Tree represent a product, so the data size is larger.
There is no need to adjust nodes in the constructing process which makes it easier.	The constructing process is more complex since the nodes should be adjusted in the process.
In GCT, the mining is faster because it only considers one-way upper path.	The mining in CATS Tree is slower since the mining considers both upper and lower paths.
The repeat projection on GCT is used for mining large item sets.	The repeat construction of conditional CATS Tree to mine the large item sets.
When repeatedly project to construct a certain tree of index number, only one extra node space and index link list are needed.	In the process of repeatedly constructing the conditional CATS Tree, the system needs to spend extra time and space to build new conditional CATS Tree and its header link list.
In the process of GCT mining, the mined large item sets must be only.	Repeat large item sets may appear in the mining process of CATS Tree. So the judgement is required in the algorithm to ensure the large item sets are the only ones.

The data processing speed of GCT is much better than the current algorithms, but it is meaningful only when the system needs to tolerate a longer period of pre-processing time. Many corporations such as banks and wholesale stores in reality are in need of this solution under their business models. Most of them have a period of closing time in a day, which can be effectively used to process the original transaction data more efficiently

and save the future processing time. Take the wholesale stores for example, the closing time at night can be used to preprocess the customer transactions of the day, including the purchase time, purchased products, purchased amount, the sales relationship between products and prices, customer age range and other data pre-processions. On the next day, the store is able to dynamically mine better sales strategies according to the market sales situation, customer numbers and the length of shopping time. Further according to the strategies, the store can immediately adjust the positioning of product, promote dynamic discount policy for increasing sales amount, or suggest products to customers through rapid mining to stimulate their desire to purchase and to improve customer satisfaction. The same approach is applicable to the E-commerce stores, where they can recommend merchandise and promote consumption through rapid analysis. It can also be applied to other applications such as the information security firewall, for with it it can rapidly analyze the invasion rules and user network access behaviors, automatically mine the firewall rules, and strengthen the anti-hacker ability. The advantage of GCT is its data processing speed, which can be widely used in the markets with rapid data processing needs, such as stock market prediction or weather prediction.

5. **Conclusions.** The authors of this paper have proposed a novel data structure, Grouping Compression Tree (GCT), and have developed a specific building method. For the efficient mining of frequent patterns from the GCT, the algorithm, GC-mining, was also proposed. Unlike traditional methods, once a GCT is built, frequent pattern mining with different supports can be performed without rebuilding the tree. The benefit is very remarkable when the user must repeatedly run the algorithm with different MSs to obtain the appropriately large item sets. The CATS Tree method can provide an analogous benefit. However, there are several advantages of the GCT approach over the CATS Tree approach. First, the data pre-processing stage guarantees a sequence of items, thus reducing the complexity of GC-mining. Second, the static construction of the GCT speeds up the building and mining time. Although the GCT requires slightly more memory to maintain the source item sets in reality, many enterprises nevertheless have sufficient storage to conserve the data. They indeed need only a short execution time. Third, GC-mining adopts the tree-based pseudo projection technique, which avoids the costly generation of conditional trees. In this context, the total memory usage of the GCT can be less than that of the CATS Tree.

REFERENCES

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Elsevier, San Francisco, 2006.
[2] S. Y. Chen and X. Liu, The contribution of data mining to information science, *Journal of Information Science*, vol.30, no.6, pp.550-558, 2004.
[3] C. Ordonez, Association rule discovery with the train and test approach for heart disease prediction, *IEEE Transactions on Information Technology in Biomedicine*, vol.10, no.2, pp.334-343, 2006.
[4] V. Estivill-Castro and A. HajYasien, Fast private association rule mining by a protocol for securely sharing distributed data, *Proc. of the IEEE Intelligence and Security Informatics*, New Brunswick, USA, pp.324-330, 2007.
[5] M. Kantardzic, *Data Mining-Concepts, Models, Methods, and Algorithms*, IEEE Press, New York, 2002.
[6] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *Proc. of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, pp.487-499, 1994.
[7] S. Brin, R. Motwani, D. Jeffrey and T. Shalom, Dynamic itemset counting and implication rules for market basket data, *Proc. of the ACM SIGMOD International Conference on Management of Data*, Tucson, USA, pp.255-264, 1997.
[8] J. Han, J. Pei and Y. Yin, Mining frequent patterns without Candidate generation, *Proc. of the ACM SIGMOD International Conference on Management of Data*, Dallas, USA, pp.1-12, 2000.

[9] W. Cheung and R. Zaiane, Incremental mining of frequent patterns without Candidate generation or support constraint, *Proc. of the 7th International Database Engineering and Applications Symposium*, HongKong, China, pp.111-116, 2003.
[10] Y. Woon, W. Ng and A. Das, Fast online dynamic association rule mining, *Proc. of the 2nd International Conference on Web Information Systems Engineering*, Kyoto, Japan, pp.278-287, 2001.
[11] M. Lin and S. Lee, Improving the efficiency of interactive sequential pattern mining by incremental pattern discovery, *Proc. of the 36th Hawaii international Conference on System Sciences*, Hawaii, USA, pp.68-75, 2002.
[12] R. Agrawal, T. Imieliski and A. Swami, Mining association rules between sets of items in large databases, *Proc. of the ACM SIGMOD International Conference on Management of Data*, Washion DC, USA, pp.207-216, 1993.
[13] Y. Sucahyo and P. Gopalan, CT-ITL: Efficient frequent item set mining using a compressed prefix tree with pattern growth, *Proc. of the 14th Australasian Database Conference*, Adelaide, Australia, pp.95-104, 2003.
[14] F. C. Tseng and C. C. Hsu, Generating frequent patterns with the frequent pattern list, *Proc. of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, HongKong, China, pp.376-386, 2001.
[15] G. Florez, S. A. Bridges and R. B. Vaughn, An improved algorithm for fuzzy data mining for intrusion detection, *Proc. of the North American Fuzzy Information Processing Society*, New Orleans, USA, pp.457-462, 2002.
[16] D. Sun, S. Teng, W. Zhang and H. Zhu, An algorithm to improve the effectiveness of apriori, *Proc. of the 6th IEEE International Conference on Cognitive Informatics*, Sydney, Australia, pp.385-390, 2007.
[17] J. Han and J. Pei, Mining frequent patterns by pattern-growth: Methodology and implications, *ACM SIGKDD Explorations Newsletter*, vol.2, no.2, pp.14-20, 2000.
[18] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, H-Mine: Hyper-structure mining of frequent patterns in large databases, *Proc. of the IEEE International Conference on Data Mining*, San Jose, USA, pp.441-448, 2001.
[19] R. C. Agarwal, C. C. Aggarwal and V. V. V. Prasad, A tree projection algorithm for generation of frequent itemsets, *Journal of Parallel and Distributed Computing*, vol.61, no.3, pp.350-371, 2001.
[20] J. Liu, Y. Pan, K. Wang and J. Han, Mining frequent item sets by opportunistic projection, *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Alberta, Canada, pp.229-238, 2002.
[21] IBM Almaden Research Center, *Quest Synthetic Data Generation Code*, http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html, 2007.
[22] R. Kohavi, C. Brodley, B. Frasca, L. Mason and Z. Zheng, KDD-Cup 2000 organizers' report: Peeling the onion, *SIGKDD Explorations*, vol.2, no.2, pp.86-98, 2000.
[23] Z. Zheng, R. Kohavi and L. Mason, Real world performance of association rule algorithms, *Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.401-406, 2001.