Incremental Mining of Frequent Patterns Without Candidate Generation or Support Constraint

William Cheung and Osmar R. Zaïane
University of Alberta, Edmonton, Canada
{wcheung, zaiane}@cs.ualberta.ca

Abstract

In this paper, we propose a novel data structure called CATS Tree. CATS Tree extends the idea of FP-Tree to improve storage compression and allow frequent pattern mining without generation of candidate itemsets. The proposed algorithms enable frequent pattern mining with different supports without rebuilding the tree structure. Furthermore, the algorithms allow mining with a single pass over the database as well as efficient insertion or deletion of transactions at any time.

1. Introduction

One major function of association rules is to analyze large amounts of market basket transactions [1,2,3,4]. Association rules have been applied to many areas including outlier detection, classification, clustering etc [5,6,7,8,9]. The mining process can be broken down into the mining of underlying frequent itemsets and the generation of association rules. Association rule mining is an iterative process [10], thus, in practice, multiple frequent pattern mining processes with different supports are often required to obtain satisfactory results.

This paper introduces Compressed and Arranged Transaction Sequences Tree or CATS Tree and CATS Tree algorithms. Once CATS Tree is built, it can be used for multiple frequent pattern mining with different supports. Furthermore, CATS Tree and CATS Tree algorithms allow single pass frequent pattern mining and transaction stream mining. In addition, transactions can be added to or removed from the tree at any time.

It is assumed that there is no limitation on the main memory. The assumption is realistic for a reasonably large database due to the following reasons: 1) the current trend of modern computing moves towards computers with large amounts of main memory (gigabytes sized); 2) memory management techniques and data compression technique in the CATS Tree reduce memory footprint; 3) the same assumption has been used in other publications [11,12,13,14,15]. In addition, CATS Tree allows removal of transactions concurrently. Even a huge database can be processed by CATS Tree if out-of-date transactions are removed concurrently.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 introduces the CATS Tree structure and the algorithm to build it. CATS Tree based frequent pattern mining algorithm is introduced in Section 4. Section 5 presents some experimental results. Conclusions are given in Section 6.

2. Previous Work

2.1. Apriori-based Algorithms

The very first published and efficient frequent patterns mining algorithm is Apriori [2]. A number of Apriori-based algorithms [1,4,9] have been proposed to improve the performance of Apriori by addressing issues related to the I/O cost.

2.2. Pattern Growth Algorithms

Han et al. propose a data structure, frequent pattern tree or FP-Tree, and an algorithm called FP-growth that allows mining of frequent itemsets without generating candidate itemsets [3]. The construction of FP-Tree requires two data scans.

As pointed out by the designers of FP-Tree, no algorithm works in all situations. A new data structure called H-struct was introduced in [14] to deal with sparse data solely.

3. CATS Tree

In the present study, we have developed a novel data structure, CATS Tree, an extension of FP-Tree[3]. Researchers have proposed to use tree structure in data mining [3,16,17]. However, they are not suitable for interactive frequent pattern mining.

CATS Tree is a prefix tree and it contains all elements of FP-Tree including the header, the item links etc. Paths from the root to the leaves in CATS Tree represent sets of transactions. We use the database in Table 1 to illustrate the construction of a CATS Tree.

Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS'03)
1098-8068/03 $17.00 © 2003 **IEEE**

Initially, the CATS Tree is empty. Transaction 1 (F, A,C, D, G, I, M, P) is added as it is. As shown in Figure 1, Transaction 2 (A, B, C, F, L, M, O) is added, common items, F, A, C, are extracted from Transaction 2 and are merged with the existing tree. Although item D is not contained in Transaction 2, common items could be found underneath node D. Item M is found to be common. However, Transaction 2 cannot be merged directly at node M because it would violate the structure of CATS tree that the frequency of a parent node must be greater than the sum of its children's frequencies. Node M of CATS Tree is swapped in front of node D as shown in Figure 1 and it is merged with the transaction. After that, there is no more common item. The remaining portion of Transaction 2 is added to node M.

TID	Original Transactions	Projected transactions for FP-Tree
1	F, A, C, D, G, I, M, P	F, C, A, M, P
2	A, B, C, F, L, M, O	F, C, A, B, M
3	B, F, H, J, O	F, B
4	B, C, K, S, P	C, B, P
5	A, F, C, E, L, P, M, N	F, C, A, M, P

Table 1. Sample database

In Figure 1, Transaction 3 (B, F, H, J, O) is added. Item F of Transaction 3 is merged. Since the frequency of node A is the same as that of node F, the search for other possible merge nodes continues along the branch. It passes through node A, C, and M and finally, reaches node B. Even though Transaction 3 also contains an item B, but the frequency of node B is smaller than that of node M, the remaining of the transaction is inserted as a new branch at node F.

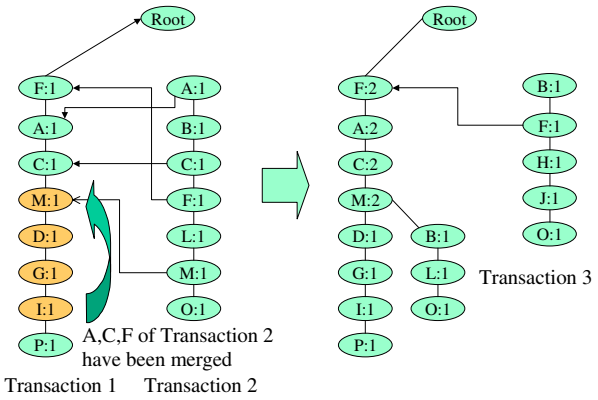

Figure 1. Insertion of Transaction 1, 2 & 3

When Transaction 4 (B, C, K, S, P) is added, there is no common item. Transaction 4 is added as it is. In Figure 2, Transaction 5 (A, F, C, E, L, P, M, N) is added; F, A, C, and M are merged. The search for common items continues along the path. Item P is common in both the tree path and Transaction 5. This triggers swapping of node P to the front of node D. After item P is merged, there is no more common item. The remainders of Transaction 5 are inserted as a new branch at node P.

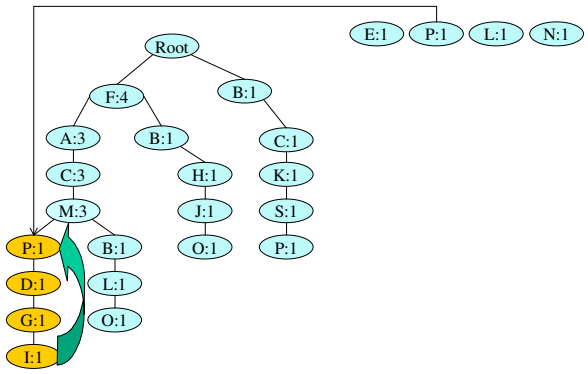

Figure 2. Insertion of Transaction 5

CATS Tree	FP-Tree
Contains all items in every transaction	Contains only frequent items
Sub-trees are locally optimized to improve compression	Sub-trees are not locally optimized
Ordering of items within paths from the root to leaves are ordered by local support	Ordering of items within paths from the root to leaves are ordered by global support
CATS nodes of the same parent are sorted in descending order according to local frequencies	Children of a node are not sorted

Table 2. CATS Tree versus FP-Tree.

All CATS Trees have the following properties:

1) The compactness of CATS Tree measures how many transactions are sharing a node. Compactness decreases as it is getting away from the root. This is the result of branches being arranged in descending order.

2) No item of the same kind could appear on the lower right hand side of another item. If there were items of the same kind on the right hand side, they should have been merged with the node on the left to increase compression. Any items on the lower right hand side can be switched to the same level as the item, split nodes as required if switching nodes violates the structure of CATS Tree. After that they can be merged with the item on the left. Because of the above properties, a vertical downward boundary is formed beside each node and a horizontal rightward boundary is formed at the top of each node. The vertical and horizontal boundaries combine to form a step-like individual boundary [18]. As shown in Figure 3, boundaries of multiple items can be

joined together to form a refined boundary for a particular item. Items of the same kind can only exist on the refined boundary. A few major differences between CATS Tree and FP-Tree are listed in Table 2.

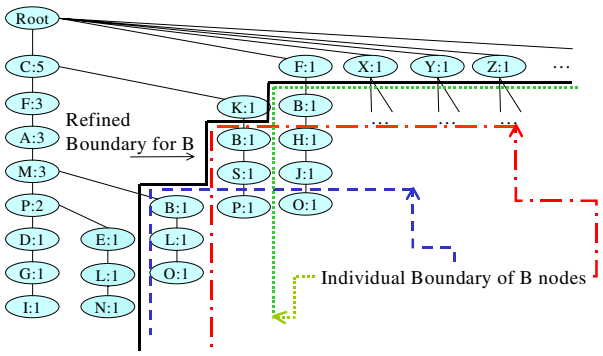

Figure 3. Item boundary in CATS Tree

3.1. CATS Tree Builder

CATS Tree contains all information of a dataset. Its construction requires only a single data scan. Thus, it is not optimal since there is no preliminary analysis before this single data scan. New transactions are added at the root level. At each level, items of the transaction are compared with those of children nodes. If the same items exist in both the new transaction and that of the children nodes, the transaction is merged with the node at the highest frequency. The frequency of the node is incremented. The remainder of the transaction is added to the merged nodes and the process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last merged node. Furthermore, CATS Tree Builder has to consider not only the immediate items of that level, but also all possible descendants. The frequency of a descendant node can become larger than that of its ancestor, once the frequency of the new transaction is added. If the frequency becomes larger, the descendant has to swap in front of its previous ancestor to maintain the structural integrity of CATS Tree. The CATS Tree Builder algorithm cannot afford to search blindly to locate common item. There are few properties of CATS Tree that can be used to prune the search space. 1) Inherited from FP-Tree, the sum of frequencies of all children nodes can only be smaller or equal to that of their parent. 2) Children of a node are sorted. Based on these properties, if a node cannot have local frequency greater than that of its parent, none of its sibling after it or any of its children can. As soon as an invalid node is found, CATS Tree Builder can abort the search and pursue other paths or insert the new transaction as a new branch. Since the frequency of the new transaction is 1, this implies that the frequency of descendant node must be equal to that of its ancestor. There can only be one node if we need to search downward. If the ordering of sibling node becomes out of order after merging, the offending node is repositioned to maintain the structural integrity of the tree. The algorithm CATS Tree Builder is listed as the following:

Algorithm: CATS Tree Builder
Input: set of transactions
Output: CATS Tree

```
1.  PROCEDURE CATSTreeBuilder(input_set S)
2.      for all transactions t ∈ S
3.          for all i ∈ t
4.          i.(frequency in header)++;
5.      root.add(t);
6.  PROCEDURE add(transaction t)
7.      if (this.children ∩ t ≠ ∅)
8.          child node.merge(t);
9.      else if (this.descendant ∩ t ≠ ∅))
10.         swap descendant node and split child
            node if necessary;
11.         descendant.merge(t);
12.     else
13.         this.children ← t;
14.     Reposition the merged node if necessary;
15. PROCEDURE merge(transaction t)
16.     this.frequency++;
17.     remove this.item from t;
18.     node.add(t);
```

Pseudo Code 1. CATS Tree Builder

In general, it is impossible to build a CATS Tree with maximal compression and without prior knowledge of the data. Therefore the compression of a CATS Tree is sensitive to both ordering of transaction and items within the transactions. However based on experiments, the size difference between maximal compressed CATS Tree and a CATS Tree produced by heuristic search is about 5 - 10%. A maximal compressed CATS Tree is an optimal tree where further loss-less compression is not possible. Different CATS Trees from the same database can be converted into a maximal compressed CATS Tree by recursively extracting the most compact item sequentially at each node. Thus, the ordering issues become irrelevant since the maximal compressed CATS Tree is insensitive to the order of input. CATS Tree based frequent patterns mining algorithm, FELINE, produces an identical set of frequent patterns as long as the underlying database remains the same.

4. FrEquent/Large patterns mINing with CATS trEe (FELINE)

Unlike FP-tree, once the CATS Tree is built, it can be mined repeatedly for frequent patterns with different support thresholds without the need to rebuild the tree. Like FP-growth [3], FELINE employs divide and conquer, fragment growth method to generate frequent patterns without generating candidate itemsets. FELINE partitions the dataset based on what patterns transactions have. For a pattern called *p*, a *p*'s conditional CATS Tree is a tree built from all transactions that contain pattern *p*. Transactions contained in a conditional CATS Tree can be easily gathered by traversing the item links of pattern *p*. A conditional condensed CATS Tree is one in which all infrequent items are removed and it is different enough from a conditional FP-Tree that FP-growth cannot be applied. By traversing upward only like FP-growth, the algorithm cannot guarantee that all frequent patterns in a conditional condensed CATS Tree are gathered. In order to ensure all frequent patterns are captured by FELINE, FELINE has to traverse both up and down to include all frequent items. However, this may cause duplications of frequent patterns in different conditional condensed CATS Trees because the same frequent pattern could appear in all trees of the frequent pattern's constituents. While building conditional condensed CATS Tree, items are excluded if the items are infrequent or the items have been mined. A detailed example of FELINE's execution can be found in [18]. The pseudo code for FELINE is given as follows:

Algorithm: FELINE
Input: a CATS Tree and required support
Output: a set of frequent pattern

```
1.  PROCEDURE FELINE(required support ε)
2.      sort(header.frequent items α);
3.      for each frequent item α
4.          build αTree = α's conditional condensed
            CATS Tree;
5.          mineCATSTree(αTree, ε, null)
6.  PROCEDURE mineCATSTree(αTree, ε, stack ps)
7.      if (αtree's support > ε)
8.          ps ← α;
9.          frequent pattern FP ← ps;
10.         FP's support = αtree's support;
11.         frequent itemsets ← FP;
12.         processed set s ← ∅; // prevent duplication
13.         if (αtree.children ≠∅)
14.             for all item i ∈ αtree ∧ i ∉ ps ∧ i ∉ s
15.                 s ← i;
16.                 build iTree = i's conditional
                    condensed CATS Tree;
17.                 mineCATSTree(iTree, ε, ps);
18.                 pop ps; // keep only the path to root
```

Pseudo Code 2. FELINE

CATS trees can be used with incremental updates of the transactional database. Indeed transactions could be added or deleted on the fly while the mining is still possible without having to rebuild the whole structure. Algorithms that add or remove a set of transactions from an already built CATS Tree, or merge trees can be found in [18].

5. Experiments and Results

The goal of the experiments is to find out the extent of different dataset properties that could affect the performance of CATS Tree algorithms and the relative performance compares with other algorithms. Datasets are generated with the data generator by IBM QUEST. To avoid implementation bias, external Apriori implementation, by Christian Borgelt [19], is used; FP-growth is provided by the original authors. To allow a fair comparison of algorithms, Apriori is also run in cached mode where all transactions are loaded into the main memory. Experiments are performed on a Pentium 4 1.6GHz PC with 512Mb RAM running on Window 2000 server. All programs are complied with the same compiler. They all yield the same patterns with the same dataset and the same parameters. All data files are generated with default parameters: 1 million transactions; average pattern length is 4; average transaction length is 10; number of unique items is 23,890 and the minimum support is 0.15%, unless stated otherwise.

Most of the previous published literature deals with database sized around 100k [3,4,12,14,16,20,21]. In our experiments, our database size is over a million transactions, which is a reasonable size for a respectable department store-like transactional database.

5.1. Scalability

The first experiment measures scalability of CATS Tree algorithms with respect to the number of transactions.

As shown in Figure 4, both CATS Tree Builder and FELINE scale linearly to the number of transactions. FELINE is very efficient while building the CATS Tree may seem expensive. However, the cost of building the CATS tree is quickly amortized in an ad-hoc interactive association rule mining context, since the tree needs only be built once. This matches the design goal: building once and mining multiple times with low overhead.

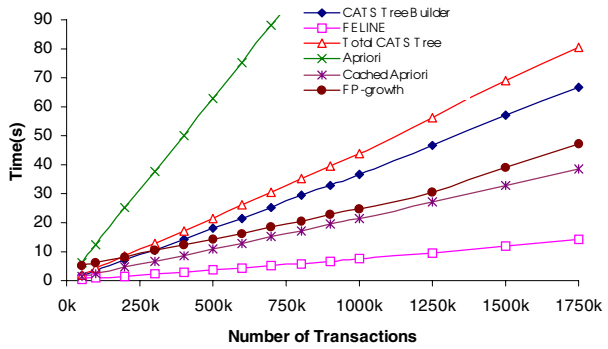

Figure 4. Scalability of CATS Tree with respect to number of Transactions with single run

The goal of the second experiment is to examine the effect of support on CATS Tree algorithms. In addition, the unique characteristic of CATS Tree, that "build once, mine many", is put to the test. A single CATS Tree is built from the data file. Frequent pattern mining iterations with different supports are performed on the same CATS Tree. In Figure 5, for comparison purposes, time required to build CATS Tree is added to the time for Total CATS Tree. In Figure 6, cumulated time from the adding of the first transaction until completion of frequent pattern mining at each data point is calculated. In other words, seven experiments with different supports were done on the same dataset.

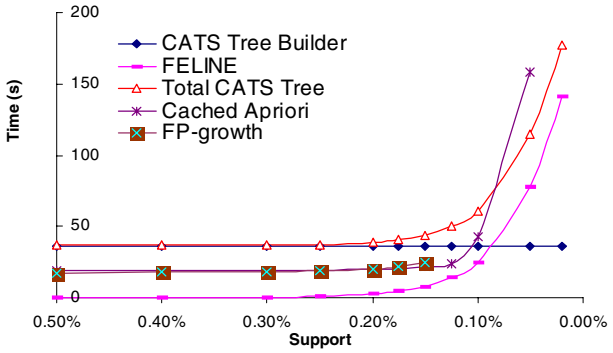

Figure 5. Scalability of CATS Tree with respect to support with single run

Time required by all algorithms increases as the support decreases. However, the rate of increase in Apriori is much faster than that of FELINE. Eventually, CATS Tree algorithms become faster than cached Apriori because FELINE does not generate candidate itemsets. Other than performance, the memory requirement for CATS Tree is smaller than other algorithms when support is low. Apriori and FP-tree runs out of memory at when the support is 0.02% and 0.15% respectively.

Unlike other frequent pattern mining algorithms, CATS Tree algorithms do not require to be started from scratch when the minimum support is decreased. The same CATS Tree can be used to mine frequent patterns with different supports; only FELINE needs to be rerun. As shown in Figure 6, the benefits of CATS Tree algorithms increase as the number of frequent pattern mining increases.

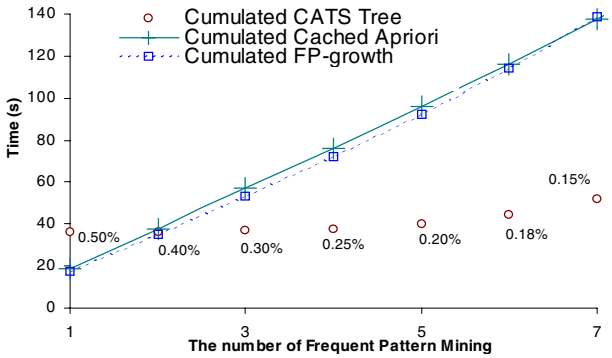

Figure 6. Build once, mine many with CATS Tree: scalability with multiple runs

5.2. Memory Usage

In this experiment, the memory usages of different algorithms are compared. The amounts of memory usage are the peak memory usage reported by the process monitor. In FP-Tree, the theoretical number of nodes is used because the source code of FP growth is not available. Only the executable code was provided to us.

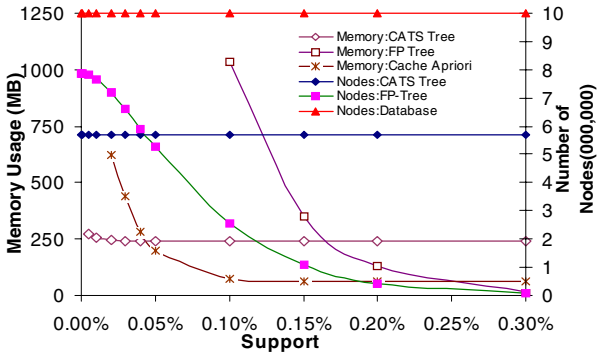

Figure 7. Memory Comparison

As shown in Figure 7, the memory usage of CATS Tree is relatively insensitive to the support while both FP-Tree and Cached Apriori are very sensitive to the support. As the support decreases, the memory consumption of FP-Tree increases exponentially and over takes that of CATS Tree at around 0.16% support.

From the theoretical aspect, CATS Tree is smaller than FP-Tree when the support is low. Because of the local memory management technique, CATS Tree will always consume less memory than a FP-Tree with 0% support. Furthermore, the structure of FP-Tree is based

on the frequency list of the items. As soon as a transaction is added, the frequency list could be changed and the FP-Tree may require a significant rearrangement of nodes to maintain the structure.

6. Conclusion

We propose a novel data structure, CATS Tree and an algorithm to build it. The algorithm FELINE is also proposed to mine frequent patterns from CATS Trees.

There are many advantages of CATS Tree algorithms over the existing algorithms. 1) Once a CATS Tree is built, frequent pattern mining with different supports can be performed without rebuilding the tree. The benefit of "build once, mine many" increases with the number of frequent patterns mining performed, i.e., interactive mining with different supports; the cost of CATS Tree construction is amortized over multiple frequent patterns mining. 2) CATS Tree allows single pass frequent pattern mining. 3) CATS Tree algorithms allow addition and deletion of transactions in the finest granularity, i.e., a single transaction. See [18] for more details. Currently, there is no known and published algorithm that can provide the same functionalities efficiently. This makes CATS Tree algorithm suitable for real time transactional frequent pattern mining where modifications and frequent patterns mining are common. Moreover, with the addition and deletion capability, CATS trees algorithms are appropriate to mine transaction streams since one single scan of the data suffices [18].

We have implemented CATS Tree algorithms and compared our approach with other algorithms. CATS Tree algorithms are shown to be efficient and scalable to large amount of transactions and outperform other algorithms in interactive setting.

7. Acknowledgement

We would like to thank Dr. Jian Pei for providing us with the executable code of the FP-growth program that was used in our experiments for comparison purposes. This work was partially supported by the National Science and Engineering Research Council of Canada .

8. References

[1] Savasere, A., Omiecinski, E., and Navathe, S. An Efficient Algorithm for Mining Association Rules in Large Databases. Proceedings of the VLDB Conference. 1995.

[2] Agrawal, R. and Srikant, R. Fast algorithms for mining association rules. VLDB, 487-499. 1994.

[3] Han, J., Pei, J., and Yin, Y. Mining Frequent Patterns without Candidate Generation. SIGMOD, 1-12. 2000.

[4] Brin, S., Motwani, R., Ullman Jeffrey D., and Tsur Shalom. Dynamic itemset counting and implication rules for market basket data. SIGMOD. 1997.

[5] Brin, S., Motwani, R., and Silverstein, C. Beyond market baskets: Generalizing association rules to correlations. SIGMOD 26[2], 265-276. 1997.

[6] Antonie, M.-L. and Zaïane, O. R., Text Document Categorization by Term Association , IEEE ICDM'2002, pp 19-26, Maebashi City, Japan, December 9 - 12, 2002

[7] Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., and Hsu, M.-C. FreeSpan: Frequent pattern-projected sequential pattern mining. ACM SIGKDD, 2000.

[8] Beil, F., Ester, M., Xu, X., Frequent Term-Based Text Clustering, ACM SIGKDD, 2002

[9] Orlando, S., Palmerini, P., and Perego, R. Enhancing the Apriori Algorithm for Frequent Set Counting. Proceedings of 3rd International Conference on Data Warehousing and Knowledge Discovery. 2001.

[10] Piatetsky-Shapiro, G., Fayyad, U., and Smith, P., "From Data Mining to Knowledge Discovery: An Overview," in Fayyad, U., Piatetsky-Shapiro, G., Smith, P., and Uthurusamy, R. (eds.) *Advances in Knowledge Discovery and Data Mining* AAAI/MIT Press, 1996, pp. 1-35.

[11] Huang, H., Wu, X., and Relue, R. Association Analysis with One Scan of Databases. Proceedings of the 2002 IEEE International Conference on Data Mining. 2002.

[12] Wang, K., Tang, L., Han, J., and Liu, J. Top down FP-Growth for Association Rule Mining. Proc.Pacific-Asia Conference, PAKDD 2002, 334-340. 2002.

[13] Zaki, M. J. and Hsiao, C.-J. CHARM: An Efficient Algorithm for Closed Itemset Mining. SIAM International Conference on Data Mining. 2002.

[14] Pei, J., Han, J., Nishio, S., Tang, S., and Yang, D. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. Proc.2001 Int.Conf.on Data Mining. 2001.

[15] Pei, J., Han, J., and Mao, R. CLOSET: An efficient algorithm for mining frequent closed itemsets. SIGMOD. 2000.

[16] Agrawal, R., Aggarwal, C. C., and Prasad, V. V. V. A Tree Projection Algorithm For Generation of Frequent Itemsets. Journal on Parallel and Distributed Computing[(Special Issue on High Performance Data mining)]. 2001.

[17] Goulbourne, G., Coenen, F., and Leng, P. H. Computing association rule using partial totals. In Proceedings of the 5th European Conference on Principles and Practice of Knowledge Discovery in Databases, 54-66. 2001.

[18] Cheung, W., "Frequent Pattern Mining without Candidate generation or Support Constraint." Master's Thesis, University of Alberta, 2002.

[19] Borgelt, C. Apriori. [2.11]. 2001.

[20] Lin, J. L. and Dunham, M. H. Mining association rules: Anti-skew algorithms. The 1998 14th International Conference on Data Engineering, 486-493. 1998.

[21] Zaki, M. J., Parthsarathy, S., Ogihara, M., and Li, W. New Algorithms for Fast Discovery of Association Rules. KDD, 283-286. 1997.