Optimized frequent pattern mining algorithm based on Can Tree

Ning ShiYong, Zhao Xiao
Harbin University of Commerce, Harbin, Hei Longjiang, China
email: 101102@hrbcu.edu.cn

Abstract. Due to the continuous dynamic changes of data in the current era, research on incremental association rules is necessary. Among them, frequent pattern mining has always been the subject of research. The research found that among the existing algorithms, Can Tree is very suitable for incremental mining because of its superior nature that it does not require adjustment, merging, and/or splitting of tree nodes during maintenance. In this paper, a new method of mining Can Tree is proposed to solve the problem of time consuming caused by repeatedly traversing paths when obtaining conditional mode basis. The path only needs to be traversed once to meet the requirements and verify it. Experimental results show that the performance of the algorithm is better than the traditional Can Tree algorithm, reducing time consumption to a certain extent.

1. Introduction
Since the databases are being updated all the time, association rules also need to be updated according to the changes made in the database. Cheung et al. put forward the idea of incremental mining for the first time developed to handle incremental databases. After that incremental mining has been the theme in dynamic mining. Researchers continue to study the association rules order to get updated rules on the basis of the original mining, using the minimum cost. Guided by this idea, Cheung and li put forward FUP[1] and FUP2[2] algorithms based on Apriori[3] algorithms. Koh and Shieh expanded the idea of FP-tree[4,5] by adjusting the old tree via the bubble sort, proposed the AFPIM[6] algorithm designed to produce the FP-tree of the updated database.

However, the traditional Apriori-based algorithm and the FP-Growth algorithm need to rescan the entire updated database for incremental data mining, and rebuild the tree, causing a lot of overhead. Among many optimization algorithms, the improvement of FP-Growth algorithm is better than the improvement of other algorithms. Therefore, the study found that the a tree-based framework better meets the needs of incremental database to adjust the data, and is more suitable for incremental data mining.

Cheung and Za`iane [7] proposed the FELINE algorithm with the CATS tree, a novel data structure, improve storage compression and allow frequent pattern mining without generation of candidate itemsets. This algorithm keeps the original mining tree compactly, so that incremental data can continue to be mined on the original tree, and for the first time it realizes "build once, mine many". Leung improved the transaction sequence of the CATS tree in 2005, and proposed the Can[8,9] Tree. The Can Tree requires that the items in the transaction are sorted in a certain order, and then operations such as tree building and mining are started. On the basis of the CATS tree, it reduces the time consumption of continuous adjustment and sorting of transactions to improve mining efficiency.
A key contribution of Can Tree is to provide the user with a simple, but yet powerful, tree structure for efficient incremental mining, but a clear method has not been expressed for extraction frequent patterns from the tree and it has only been suggested that the mining method would be similar to FP-Growth. However, FP-Growth scans the database before building the tree, to obtain a frequent item sets in the order of transaction support, and then conducts tree-building mining. Because the Can Tree saves all infrequent item sets, and the tree is too large. The method of FP-Growth repeatedly traversing the path along the header table is also not suitable for Can Tree. To solve this problem, this paper proposes an improved Can Tree mining method. Experiments show that this method greatly reduces time consumption.

2. Can Tree
Can Tree mining frequent itemsets is mainly divided into two steps: the first step is to construct the Can Tree, and the second step is to mine frequent itemsets from the tree structure.

Can Tree construction: Can Tree sorts the data items according to the pre-specified order, and can be constructed by scanning the database once, and the insertion that is unaffected by changes in item frequency. The specified order can be dictionary order, alphabetical order, or user-defined order that the user is interested in. As long as the order is determined, the Can Tree is unique. Since support does not participate in the establishment of the tree during the construction process, it will not be affected by the minimum support. Therefore, when mining incremental association rules, the new data set can be directly merged into the original tree for mining, without the need to rebuild the tree also will not have any impact on the original tree, thus avoiding the time consumption caused by scanning the data set again. Improve the efficiency of incremental association rule mining. The shortcomings of the Can Tree are also brought about by the advantages, because it saves all the original information and takes up too much memory. The node structure of the Can Tree follows the structure of the FP-Tree, which will not be repeated here, Figure 1 is used to illustrate the idea of Can tree building.

Table 1. Examples of transaction sets

| TID | Contents    |
|-----|-------------|
| t1  | {a,b,d}     |
| t2  | {a,c,d,e}   |
| t3  | {c,d,e}     |
| t4  | {a,d,e}     |
| t5  | {c,d,e}     |
| t6  | {d,e}       |

Figure 1. Can Tree corresponding to transaction instance
Can Tree mining: During the mining step, because the Can Tree structure contains all the information, the tree is huge, which leads to too long time to build the tree; and the path contains not only frequent itemsets but also infrequent itemsets. Therefore, the traditional branch reduction strategy is not applicable. If mining is carried out according to the traditional FP-Growth mining method, the path in the tree is repeatedly scanned in the order of the head table with increasing frequency, which will increase a lot of time consumption (because there are only frequent itemsets in the FP tree, and the Can Tree Contains infrequent itemsets, the processing of infrequent itemsets will also cause time consumption), resulting in too long mining time.

At present, the improvement of the Can Tree is mainly centered on making the search and positioning faster. Zou Likun [9] used the child to point to the parent node pointer instead of the original parent to child node pointer in 2008. In this way, the conditional pattern tree is quickly generated and the algorithm efficiency is improved. Chen Gang et al. proposed a fast construction algorithm based on Can Tree (FCCAN) [11] in 2014. The algorithm also uses the child parent node pointer instead of the original parent child node pointer, and add an auxiliary storage structure based on the hash table to reduce the search time of the project. This article continues the idea of adding pointers from child nodes to parent nodes, and combined with the nature of Can Tree. We hope to traverse the path only once and extract all the conditional pattern bases of this path, reducing unnecessary multiple scans and saving follow the time of repeated traversal in the header table. Therefore, this paper proposes a mining optimization idea based on conditional pattern base.

3. Can Tree optimization

Conditional pattern base: The conditional pattern base of an item is the sub-pattern base under the transaction condition where the item exists.

A transaction item may exist in multiple paths, and the multiple paths that contain the item together form the conditional pattern base. Under a path, all its parent nodes are a child conditional pattern base. For example, e exists in two paths, and the conditional pattern base of e is {{(a, c, d): 1}, {(c, d): 2}}, where {{a, c, d}: 1} and {{c, d}: 2} are the sub-conditional pattern bases respectively.

Suppose we are seeking the conditional pattern base for term e, which appears in two paths. For the path {(a, c, d, e):1}, when the original algorithm traverses the header table in increasing order of frequency, when it encounters e: the first time the path is traversed, the sub-condition pattern base of e is obtained {(a,c,d):1}; when encountering d: the second traversal path, the sub-condition pattern base of d is obtained {(a,c):1}; when c is encountered: the third traversal path, the sub-condition pattern base of c is {a:1}. In fact, this path has already been traversed when seeking the conditional pattern base of e, and the mining results of other nodes on this path must be a subset of the conditional pattern base of e, so there is no need to traverse multiple times.

3.1 Optimization ideas

Traversing the same path multiple times is obviously undesirable on a large data set. For a path in this paper, when traversing upwards along the projection, the sub-conditional pattern bases of all items on the path are obtained and saved separately, and processed. The path will be skipped next time you encounter this path. It realizes that a path only needs to be traversed once, which greatly reduces the running time.

Since all infrequent itemsets are stored in the header table, in order to better "item skip" these infrequent items during mining. This article adds a hash table “frequentF1”, which stores all the frequent item sets whose support is greater than or equal to the minimum support, the key is the transaction data item, and the value is the support of the transaction item.

In order to break the defect in the traditional Can Tree that only the child pointer which from parent node points to the child node, it can only start from the parent node to find the child nodes layer by layer, which affects the execution efficiency. The parent pointer from the child node to the parent node is added, so that it is more convenient to find the parent node.
The header table continues the previous structure, consisting information on all database items along with their frequency, the node pointer “nextHomonym” that points to the next data item in the transaction tree.

3.2 Algorithm flow
When traversing the projection from the bottom to the top, all the conditional pattern bases under this path can be found, and the nodes on the current path are marked with the processed flags. Then traverse other items in the header table in increasing order, first check whether the item has been processed. If it is processed, it will be skipped and the next node with the same name will be traversed. If it has not been processed, it will further recursively obtain all the conditional pattern bases under the path and mark it.

Among them, since the path contains infrequent item sets. When obtaining the conditional pattern base for a certain path upwards, For infrequent itemsets, sub-conditional pattern bases should not be added, nor should a separate space be opened to obtain the conditional pattern bases for infrequent itemsets. This method is called "item skip". For example, the path {a, c, d, e}: 1, transactions a, c, e are frequent, and d is an infrequent item set. While finding the sub-conditional pattern base {a, c}: 1 of e, obtain the sub-conditional pattern base {a:1} of c, and add the processed mark isdone. Item d is skipped. After when processing c in the header table, if this path is encountered, it will first determine whether the path has been processed, if it has been processed, skip it, if not, find the sub-condition pattern base of c.

```java
for item in headerTable()
    if item.count >= min_sup{
        if (nextHomony != null){
            curr = item.nextHomony;
            if (curr.isdone = false){
                cpbs = paths(curr);
            } else { curr = curr.nextHomony; }
        }
    }
paths(curr) {
    if (frequentf1.contains curr.getname()){ List<String> prenodes;
        TreeNode parents = curr.getParent();
        if (parents.getName() != null ) {
            List<String> lujing = path(parents);
            if (curr.isdone == false){
                curr.isdone = true;
            } }
        prenodes.add(curr.getName());
    } else {
        curr = curr.getParent;
    }
```

**Algorithm 1.** new Can-Mining Algorithm

4. Evaluation and experimental results
The experimental environment is Intel® CoreTM i7-8750H CPU @ 2.20GHz 8GByte memory, the operating system is Windows 10 Ultimate Edition, and all algorithms are implemented using Eclipse (version=3.7.0 & Java 1.8.0) programming.

Experiment one: Conduct frequent pattern mining experiments on two public data sets of Mushroom and Accidents, set different thresholds, and compare the time for different algorithms to mine frequent patterns.
Although the algorithm avoids multiple scans of the same path and reduces the running time to a certain extent, the algorithm has the following shortcomings: the algorithm adds a hash table and a reverse pointer. Each node needs to add a flag field to indicate whether it has been processed, which increases time and memory consumption. It can be seen from the experimental results in Figure 2 and Figure 3. For a fixed data set and a fixed minimum support, the data mining time of the optimized Can Tree algorithm proposed in this paper is shorter than that of the traditional Can Tree mining. Since the data has not changed, the tree shape must be the same, and the difference in total time is the difference in algorithm mining time.

Experiment two: For incremental situations, use the T10I4D100K data set. By continuously inserting new data in the database to simulate an incremental database. The initial data volume is set to 40,000, and each time 20,000 data volume is used for testing, the minimum support is 3000. The experimental results are shown in Figure 4. It can be seen that the time spent by FP-Growth is less than the time of Can Tree mining at the beginning, but after the second increment, the time of FP-Growth algorithm is greater than the time of Can Tree mining. This is because the Can Tree includes all the original information from the beginning, so the initial mining time is not as good as the FP-Growth algorithm that has removed infrequent itemsets. But with the continuous addition of incremental data, the Can Tree can directly execute the algorithm on the original tree, while the FP-Growth algorithm needs to rescan the database. The drawbacks begin to appear and the running time is not as good as the Can Tree.

5. Conclusion and future works
This paper proposes a conditional pattern-based optimization method for the incremental association rule mining algorithm based on Can Tree, which avoids repeated backtracking of the same path. The algorithm adds a hash table-based storage structure to store infrequent itemsets. Although it will increase a part of the overhead, it is necessary to better "item skip" for infrequent itemsets in the mining process; Add a child node to point to the parent node pointer can quickly generate the conditional pattern base; Avoid repeated scanning of the header table and reduce mining time.
Experiments show that this algorithm can improve mining efficiency. In the future, this algorithm can also be applied in distributed or other environments. How to compress the Can Tree more effectively, make the tree shape smaller, and reduce the memory usage, remains to be solved.

References

[1] CHEUNG D W, HAN J, NG V T, et al. (1996) Maintenance of discovered association rules in large databases: an incremental updating technique. In: Twelfth International Conference on Data Engineering. Washington DC, pp. 106-144.

[2] CHEUNG W L, LEE S D, KAO B. (1997). A General Incremental Technique for Maintaining Discovered Association Rules In: International Conference on Database Systems for Advanced Applications. Singapore, pp. 185-194.

[3] R. Agrawal and R. Srikant. (1994). Fast algorithms for mining association rules. In: International Conference on Very Large Data Bases. San Francisco, pp. 487-499.

[4] J. Han and J. Pei. (2000) Mining frequent patterns by pattern-growth: methodology and implications. ACM SIGKDD explorations newsletter, vol. 2, pp. 14-20.

[5] J. Han, J. Pei, Y. Yin, and R. Mao. (2004) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data mining and knowledge discovery, vol. 8, pp. 533-87.

[6] J.-L. Koh and S.-F. Shieh. (2004) An efficient approach for maintaining association rules based on adjusting FP-tree structures, in Database Systems for Advanced Applications, pp. 417-424.

[7] W. Cheung and O. R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In Proc. IDEAS 2003, pp. 111–116.

[8] C. K. S. Leung, Q. I. Khan, and T. Hoque. (2005) CanTree: a tree structure for efficient incremental mining of frequent patterns, pp. 8.

[9] C. K. S. Leung, Q. I. Khan, Z. Li, and T. Hoque, "CanTree: a canonical-order tree for incremental frequent-pattern mining," Knowledge and Information Systems, vol. 11, pp. 287-311, 2007.

[10] ZOU L K, ZHANG Q S. (2008) Efficient Incremental Association Rules Mining Algorithm Based on CAN-tree[J]. Computer Engineering, 34(3): 29-31.

[11] CHEN G, YAN Y Z, LIU B Q. (2014) A Fast Construction Algorithm Based on CAN-tree[J]. Microelectronics & Computer, pp. 76-82.

[12] C. Borgelt, (2005) An Implementation of the FP-growth Algorithm," in Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, pp. 1-5.

[13] M. H. Dunham, Y. Xiao, L. Gruenwald, and Z. Hossain, "A survey of association rules," Teknik Rapor, Southern Methodist University, Department of Computer Science, TR, 2000.

[14] C. Hidber. Online association rule mining. In Proc. SIGMOD 1999, pp. 145–156.

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