On Computing, Storing and Querying Frequent Patterns*

Guimei Liu Hongjun Lu Wenwu Lou
The Hong Kong Univ. of Science & Technology
Hong Kong, China
{cslgm, luhj, wwlou}@cs.ust.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

Extensive efforts have been devoted to developing efficient algorithms for mining frequent patterns. However, frequent pattern mining remains a time-consuming process, especially for very large datasets. It is therefore desirable to adopt a "mining once and using many times" strategy. Unfortunately, there has been little work reported on managing and organizing a large set of patterns for future use. In this paper, we propose a disk-based data structure, CFP-tree (Condensed Frequent Pattern Tree), for organizing frequent patterns discovered from transactional databases. In addition to an efficient algorithm for CFP-tree construction, we also developed algorithms to efficiently support two important types of queries, namely queries with minimum support constraints and queries with item constraints, against the stored patterns, as these two types of queries are basic building blocks for complex frequent pattern related mining tasks. Comprehensive experimental study has been conducted to demonstrate the effectiveness of CFP-tree and efficiency of related algorithms.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining

Keywords

data mining and data warehousing, frequent pattern mining

1. INTRODUCTION

Mining frequent patterns is an important problem in data mining area. During the past decade, extensive efforts have been devoted to developing efficient algorithms for mining frequent patterns [1, 7, 12, 2, 16, 3, 11, 9, 5, 4]. Despite all these efforts, frequent pattern mining remains a time-consuming process. Therefore it is desirable to be conducted in a "mining once and using many times" fashion. However, there is a lack of studies in managing and organizing a large set of patterns on disk for future use. For this purpose, we propose a disk-based data structure, called Condensed Frequent Pattern Tree, or CFP-tree for short, to organize frequent patterns on disk, and develop efficient algorithms to retrieve patterns from it. Another motivation of our work is that frequent pattern mining requires a predefined minimum support threshold. In real applications, there are often no guidelines for choosing the proper minimum support, which makes frequent pattern mining a repeated process to find appropriate threshold for a given database and application. Furthermore, for the same database, different applications may require different thresholds. It is preferable to materialize the frequent patterns with a sufficiently low minimum support such that most, if not all, of the user requests can be answered by querying the materialized pattern set.

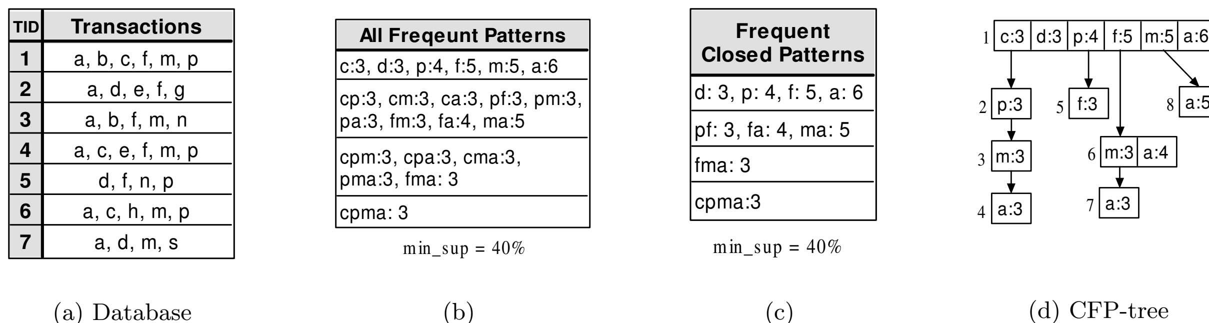
Given a minimum support threshold, the complete set of frequent patterns may be undesirably large, especially when long patterns exist. Recently, researchers have proposed to mine only the frequent closed patterns to reduce the output size [8, 15, 10, 14]. A pattern is closed if none of its proper supersets has the same support as it has. The set of frequent closed patterns is the most concise representation of the whole frequent pattern set without information loss, and it could be significantly smaller than the complete set of frequent patterns. Inspired by this, only frequent closed patterns are included in the CFP-tree structure. Furthermore, different patterns in the CFP-tree structure can share the storage of their prefixes, which makes CFP-tree a very compact data structure.

A CFP-tree constructed with minimum support threshold *min_sup* can efficiently support two types of important queries related to frequent pattern mining: (1) *query with minimum support constraint*, for example, "find all the patterns with support higher than s%", where $s \geq min_sup$; and (2) *query with item constraint*, e.g. "find all the frequent patterns containing items in I'", where I' is the set of items a user is interested in. These two types of queries are very common in practice, and are also essential for efficiently evaluating more complex queries.

The main contributions of our work can be summarized as follows. (1) We proposed a compact and efficient data structure, CFP-tree, for storing and querying frequent patterns on disk. (2) We proposed a set of CFP-tree related algorithms, including a fast tree construction algorithm, ef-

*This work was partly supported by the Research Grant Council of the Hong Kong SAR, China (CUHK4229/01E, Grants DAG01/02.EG14) and National 973 Fundamental Research Program of China (G1998030414).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGKDD'03, August24-27, 2003, Washington, DC, USA
Copyright 2003 ACM 1-58113-737-0/03/0008 ...$5.00.



(a) Database (b) (c) (d) CFP-tree

Figure 1: A CFP-tree Example

ficient algorithms for finding all frequent patterns w.r.t. a minimum support threshold and efficient algorithms for finding all frequent patterns containing a given set of items. (3) Comprehensive experimental study has been conducted to demonstrate the effectiveness of CFP-tree and efficiency of related algorithms.

The rest of the paper is organized as follows. Section 2 describes the CFP-tree structure and its properties; query processing algorithms are presented in Section 3; Section 4 presents the CFP-tree construction algorithm; Section 5 shows experiment results; some related works are introduced in Section 6; finally, Section 7 concludes the paper.

2. THE CFP-TREE STRUCTURE

In this section, we use an example to illustrate the CFP-tree structure and describe its properties.

2.1 An Example

Given a transaction database D as shown in Figure 1(a) and minimum support 40%, the complete set of frequent patterns and the frequent closed patterns are shown in Figure 1(b) and Figure 1(c) respectively. For brevity, an itemset $\{i_1, i_2, \cdots, i_m\}$ with support s is represented as $i_1 i_2 \cdots i_m : s$. In this example the set of frequent closed patterns is substantially smaller than the complete pattern set. In particular, all the patterns containing c can be represented by a single frequent closed pattern *cpma*. Such case is quite common in real datasets.

The CFP-tree constructed from D with minimum support 40% is shown in Figure 1(d). The nodes in the tree are numbered according to their creation time. Each node in a CFP-tree is represented as a variable-length array, and all the items in a node are stored in ascending order of their frequencies. Both types of queries can benefit from this ordering method as explained later in next subsection. A path in the tree starting from an entry in the root node represents a frequent pattern. All the entries in the same node share the same prefix. For an entry E in a node, suppose the pattern represented by the path from root to E is p, the entry E stores four pieces of information: (1) the last item of p, (2) the support of p, (3) a pointer pointing to the root of the sub CFP-tree that stores all the patterns having p as prefix, and (4) a hash bitmap which is not shown in Figure 1(d). In the rest of this paper, for an entry E in a CFP-tree node, we will use $E.item$ to denote the item stored in E, $E.support$ to denote its support, $E.child$ to denote the pointer to the root of its sub CFP-tree, and $E.bitmap$ to denote its hash bitmap.

2.2 Properties

The CFP-tree structure is called condensed, not only because it stores merely frequent closed patterns, but also because patterns can share the storage of their prefixes in the tree. It can lead to great space saving because item sharing is quite common among patterns. The CFP-tree structure has two important properties, which can be utilized in query processing.

Apriori Property. For an entry E in a CFP-tree node, the support of any pattern in the subtree pointed by E cannot be greater than $E.support$. We can take advantage of this property when processing queries with minimum support constraints. If the support of an entry does not satisfy the minimum support constraint specified in the query, then there is no need to access the subtree pointed by the entry.

Left Containment Property. For any entry E, $E.item$ can only appear in the subtrees pointed by the entries before E or in E itself. This property can be utilized when answering queries with item constraint. To find all the patterns containing an item $E.item$, only the subtrees pointed by the entries before E and E itself needs to be accessed.

Now we show how the item ascending order can help query processing. The Left Containment Property implies that the subtrees pointed by entries at the beginning of a node have a higher chance to be visited than the subtrees pointed by entries at the end of a node. An entry with lower minimum support will very possibly point to a smaller subtree. All the entries in a node are sorted in ascending order of their frequencies, so every time only small trees are to be traversed. It results in great saving when processing queries with item constraints. For queries with minimum support constraints, the most infrequent item of every pattern is first checked based on the ascending ordering method, so those patterns that contain an item dissatisfying the minimum support constraint can be quickly discarded.

To further avoid unnecessary traversal cost when processing queries with item constraints, we maintain a hash bitmap at each entry. Given an entry E in a CFP-tree node, for every item i in the subtree pointed by E, the j-th bit of $E.bitmap$ is set to 1, where j=i mod N and N is the length of the bitmap. Thus before performing search on the subtree pointed by $E.child$, we can first check $E.bitmap$ to see whether all the items being searched are in the sub CFP-tree. If the hash bitmap indicates that all the items are in the sub CFP-tree, we need to continue the search on that subtree, otherwise the search on that subtree can be terminated. Other hash functions can be used here as well, as long as the hash function does not introduce false dismissal.

The length of the hash bitmap is a trade-off between the size of the CFP-tree and the search time. In our experiments, the length of the bitmap is set to 32.

3. QUERY PROCESSING ON CFP-TREE

In this section, we show how to utilize the two properties, the ascending frequency order and the hash bitmap to process queries. We consider two basic types of queries: (1) query with minimum support constraint, and (2) query with item constraint. At the end of this section, we briefly discuss how to process queries with both constraints.

3.1 Query with Minimum Support Constraint

A query with minimum support constraint is to output all frequent patterns w.r.t. a user specified minimum support *min_sup*, where *min_sup* has to be no less than the constructing support of the CFP-tree. According to the Apriori property of the CFP-tree, for such queries only the subtrees pointed by an entry with support no less than *min_sup* should be searched. At each node items are sorted according to their frequencies, so a binary search can be performed to find the first entry whose support is no less than *min_sup*. Suppose this entry is E, then only the subtrees pointed by entries after E and by E itself need to be accessed. Algorithm 1 shows the pseudo-code for the search algorithm. BinarySearch(*cnode*, *min_sup*) procedure returns the first entry in *cnode* whose support is no less than *min_sup*.

Algorithm 1 Search_Minsup Algorithm

Input:
- p is a frequent pattern
- *cnode* the cfp-tree node pointed by p
- *min_sup* is the minimum support threshold

Description:

1: **if** $p \neq \emptyset$ AND (p has more than one children OR *support*(p) $>$ the support of p's only child) **then**
2:   output p and its support;
3: **end if**
4: E_1 = BinarySearch(*cnode*, *min_sup*);
5: **for all** entry $E \in cnode$, $E = E_1$ or E after E_1 **do**
6:   $s = p \bigcup \{E.item\}$;
7:   **if** $E.child \neq$ NULL **then**
8:     Search_Minsup(s, $E.child$, *min_sup*);
9:   **else**
10:     output pattern s and its support $E.support$;
11:   **end if**
12: **end for**

CFP-tree stores some patterns which are prefixes of some closed patterns, but themselves are not closed, e.g. pattern *cp* and *cpm* in Figure 1(d). To output only the closed patterns, we need to check whether a pattern has a greater support than its children before output it (line 1-3). Consider an example: find all the frequent patterns w.r.t 50%. In the CFP-tree shown in Figure 1(d), a binary search is performed on node 1, and p is found to be the first item with support no less than 50%. All the entries before p can be ignored. The node pointed by p (node 5) is visited, and no item has support greater than 50%. Next the node pointed by f (node 6) is accessed. Again a binary search is performed and item a is found to be frequent. Finally, patterns *ma* and a are found to be frequent and closed.

3.2 Query with Item Constraint

Based on the Left Containment property, the item of an entry E appears only in subtrees pointed by the entries before E or in E itself. A subtree pointed by an entry before E may not actually contain $E.item$. The hash bitmap maintained at each entry can be utilized to reduce unnecessary searching cost. Before a subtree pointed by an entry E is searched, the bitmap of E is first checked (line 6-7). If and only if it indicates that all the items being searched appear in that subtree, search on that subtree should be continued. Algorithm 2 shows the pseudo-code for evaluating queries with item constraint. In Algorithm 2, if $I = \emptyset$, then all the subtrees under *cnode* should be accessed.

Algorithm 2 Search_Item Algorithm

Input:
- p is a frequent pattern
- *cnode* is the sub CFP-tree pointed by p
- I is the set of items that must be contained in patterns

Description:

1: **if** $I = \emptyset$ AND (p has more than one children OR *support*(p) $>$ the support of p's only child) **then**
2:   output p and its support;
3: **end if**
4: E_1 = the first entry of *cnode* that satisfies $E_1.item \in I$;
5: **for all** entry $E \in cnode$, E before E_1 or E=E_1 **do**
6:   check $E.bitmap$;
7:   **if** all the items in $I - \{E.item\}$ are in $E.child$ **then**
8:     $s = p \bigcup E.item$;
9:     **if** $E.child \neq$ NULL **then**
10:       Search_Item(s, $E.child$, $I - \{E.item\}$);
11:     **else if** $I - \{E.item\}$ is empty **then**
12:       output s and its support $E.support$;
13:     **end if**
14:   **end if**
15: **end for**

To process query "find all the patterns containing item p and f" on the CFP-tree shown in Figure 1(d), only the subtrees pointed by entry c, d and p at node 1 need to be accessed. We start from entry c, and examine the hash bitmap of entry c, suppose it indicates that both items appear in the subtree rooted at node 2 (the hash bitmap may introduce false alarm, but no false dismissal), then the search on node 2 should be continued. Node 2 contains only one entry p, we check its bitmap to see whether f exists in the subtree rooted at node 2. Suppose the corresponding bit of f in the bitmap is 0, the search at node 2 is stopped. Then we continue our search on entry d at node 1. It points to an empty subtree. The next entry of node 1 is p, and its bitmap indicates that f exists in node 5 since there indeed exists a f in the subtree. Then node 5 is searched and we find f. Node 5 points to an empty tree, the search is finished.

3.3 Query with Both Constraints

Another type of query is to have both minimum support constraint and item constraint. The algorithm for evaluating such queries can also combine the pruning power of the Apriori property, the Left Containment property and the hash bitmap. Let us consider a query "find all the patterns with minimum support 50% and containing item p and f" on the database in Figure 1(a). According to the item constraint, we only need to check the entries c, d and p at node

1. And according to the minimum support constraint, only entry p and the entries after p need to be considered. Therefore, only the subtree pointed by entry p in node 1 needs to be accessed.

4. CFP-TREE CONSTRUCTION

In this section, we briefly present the algorithm for constructing the CFP-tree, which is similar to the AFOPT algorithm [4] except that we adopted several techniques to remove redundant patterns.

4.1 Construction Algorithm

Given a transactional database D and a minimum support threshold, the CFP-tree structure can be constructed with only two database scans by adopting the pattern growth approach. In the first database scan, all the frequent items in the database are mined, and they are sorted in ascending order of their frequencies. Then a CFP-tree node is created, which contains all the frequent items and their supports. Let the set of frequent items be $F = \{i_1, i_2, \cdots, i_m\}$. We perform another database scan, and construct a *conditional database* for each $i_j \in F$, denoted by D_{i_j}. During the second scan of the database, infrequent items in each transaction t are removed and the remaining items are sorted according to their orders in F. Transaction t is put into D_{i_j} if the first item of t is i_j. The conditional databases contain the complete information for mining frequent patterns. Once they are built, the remaining mining will be performed on them. There is no need to access the original database.

We first perform mining on D_{i_1} to mine all the patterns containing i_1. Mining on individual conditional database follows the same process as mining on the original database. After the mining on D_{i_1} is finished, D_{i_1} can be discarded. Since it also contains other items, the transactions in it will be inserted into the remaining conditional databases. Given a transaction t in D_{i_1}, suppose the next item after i_1 in t is i_j, then t will be inserted into D_{i_j}. This step is called push-right. Sorting the items in ascending order of their frequencies ensures that every time, a small conditional database is pushed right: i_1 is the most infrequent item, and D_{i_1} contains fewest transactions, with the increasing of j, the number of transactions in D_{i_j} increases, but the number of distinct items in D_{i_j} shrinks and the transaction length decreases.

The pseudo-code of the construction algorithm is shown in Algorithm 3. Algorithm 3 is independent of the representation of the conditional databases. We choose to use the prefix-tree structure, in which different transactions can share their prefixes. We chose this structure not only because it is compact, but also because it allows quick removal of redundant patterns, e.g the identification of single child can be very easy. Further optimizations can be made on the prefix-tree structure, e.g. using arrays to store single branches. We do not discuss further details here.

4.2 Removing Redundant Patterns

Algorithm 3 generates all the frequent patterns and stores them in CFP-tree. In this subsection, we describe how to remove redundant patterns during mining process. A pattern is called redundant if it is not closed. We have the following two lemmas.

Lemma 1. *In Algorithm 3, a pattern p is closed if and only if two conditions hold: (1) there is no previously mined pattern which is a superset of p and has the same support as p; (2) all the items in D_p have a lower support than p.*

Algorithm 3 CFP-Construct Algorithm

Input:
p is a frequent pattern
D_p is the conditional database of p
E_p is the entry of p in CFP-tree
min_sup is the minimum support threshold;
Description:
1: Scan D_p count frequent items, let F denotes them;
2: Sort items in F in ascending order of their frequencies;
3: Create a new CFP-tree node *cnode*, put items in F and their supports in *cnode*;
4: $E_p.child = cnode$;
5: **for all** item $i \in F$ **do**
6: set (i mod N)-th bit of $E_p.bitmap$ to 1;
7: $D_{p \bigcup \{i\}} = \emptyset$;
8: **end for**
9: **for all** transaction $t \in D_p$ **do**
10: remove infrequent items from t, and sort remaining items according to their orders in F;
11: let i be the first item of t, insert t into $D_{p \bigcup \{i\}}$.
12: **end for**
13: **for all** item $i \in F$ **do**
14: $s = p \bigcup \{i\}$;
15: $E_s =$ the entry of i in *cnode*;
16: CFP-Construct(s, D_s, E_s, *min_sup*);
17: PushRight(D_s);
18: **end for**

Proof: If all the items in D_p has a lower support than p, then all the patterns mined from D_p must have a lower support than p according to the Apriori property. Suppose the last item of p is i, then all the conditional databases after D_p cannot contain i according to the construction of the conditional databases, so any pattern mined from them cannot be a superset of p. That means only patterns mined from previous conditional databases can be a superset of p, if no such pattern exists, then p is closed.

Lemma 2. *In Algorithm 3, if a pattern p is not closed because condition (1) in Lemma 1 does not hold, then none of the patterns mined from D_p can be closed.*

Proof: Pattern p is not closed because condition (1) does not hold, then there exists a previously mined pattern q, $p \subset q$ and p, q have the same support. For every pattern s mined from D_p, the pattern $t = (s - p) \bigcup q$ has the same support as s and $s \subset t$, so s is not closed.

Based on Lemma 1, there are two pruning conditions for a redundant pattern p: (1) Examine whether there exists a previously mined pattern q, which is a superset of p and has the same support as q. This checking can be done before the mining on a pattern's conditional database starts. If such q exists, then there is no need to perform mining on D_p based on Lemma 2. Thus the identification of a redundant pattern not only reduces the size of the tree, but also avoids unnecessary mining cost. (2) Check whether there exists an item i, which appears in every transaction of D_p. If such i exists, then there is no need to consider the patterns that do not contain i when mining D_p. In other words, we can directly perform mining on $D_{p \bigcup \{i\}}$ instead of D_p. The efforts for mining $D_{p \bigcup \{j\}}$, $j \neq i$ are saved.

Data Sets	Size	#Trans	#Items	AvgTL
BMS-POS	19.20M	515,597	1657	6.53
Pumsb	14.75MB	49,046	2113	74.00
T20I10D1000k	89.57MB	987,139	8876	20.23

Table 1: Datasets

To incorporate the above two pruning techniques, the items that have the same support as p are removed from F before the new CFP-tree node for F is created (line 3). For each of such item i, a new CFP-tree node $node_i$ is created, which contains only item i itself. E_p or the most recently created CFP-tree node points to $node_i$. At line 14, before performing mining on D_s, condition (1) in Lemma 1 is checked. If it does not hold, then the mining on D_s can be skipped.

4.3 Closed Pattern Checking

When we do condition (1) pruning, p should be compared with previously mined frequent closed patterns. Since the CFP-tree constructed during the mining process stores all the frequent closed patterns mined so far, we can use the search algorithms proposed in Section 3 to do the closeness checking. For a pattern p, we call Search_Both($\emptyset$, $root$, $support(p)$, p) to find some previously mined pattern q such that $p \subset q$ and $support(p)$=$support(q)$. If such q exists, then we can safely discard D_p. Previous algorithms for mining frequent closed patterns require that all the frequent closed patterns must be in memory to do this checking. Our closeness checking technique does not have this requirement since our search algorithm is very efficient on disk. Moreover, the CFP-tree structure is a compact representation of the patterns, so it has a higher chance to be held in memory than the flat representation of the patterns.

5. PERFORMANCE STUDY

We conducted a set of experiments to demonstrate the efficiency of the CFP-tree structure. All the experiments were conducted on a 2.24Ghz Pentium IV with 512MB memory running Microsoft Windows XP. All codes were complied using Microsoft Visual C++ 6.0.

Table 1 shows the several datasets used for performance study. BMS-POS [17] is a large and sparse dataset containing click-stream data. Pumsb is a dense dataset obtained from UCI machine learning repository. T20I10D1000k is a large synthetic dataset generated by IBM Quest Synthetic Data Generation Code. Table 1 lists some statistical information about the datasets, including the size of the dataset on disk, the number of transactions, the number of distinct items and the average transaction length.

5.1 Query Processing

We built a CFP-tree for the three datasets respectively with minimum support 0.02%, 0.05% and 50%. The size of the CFP-tree for three datasets is 80.7MB, 199.5MB and 142.1MB respectively. We issued a set of queries with minimum support constraint or item constraint to see the efficiency of our query processing algorithms. We compared three algorithms: (1) the query processing algorithms proposed in this paper, denoted by "CFP", (2) the sequential scan algorithm, denoted by "SCAN", and (3) mining from scratch, denoted by "MINE". For sequential scan algorithm, we stored all the closed patterns in a flat file. For every pattern, we kept its length, support and the set of items in it.

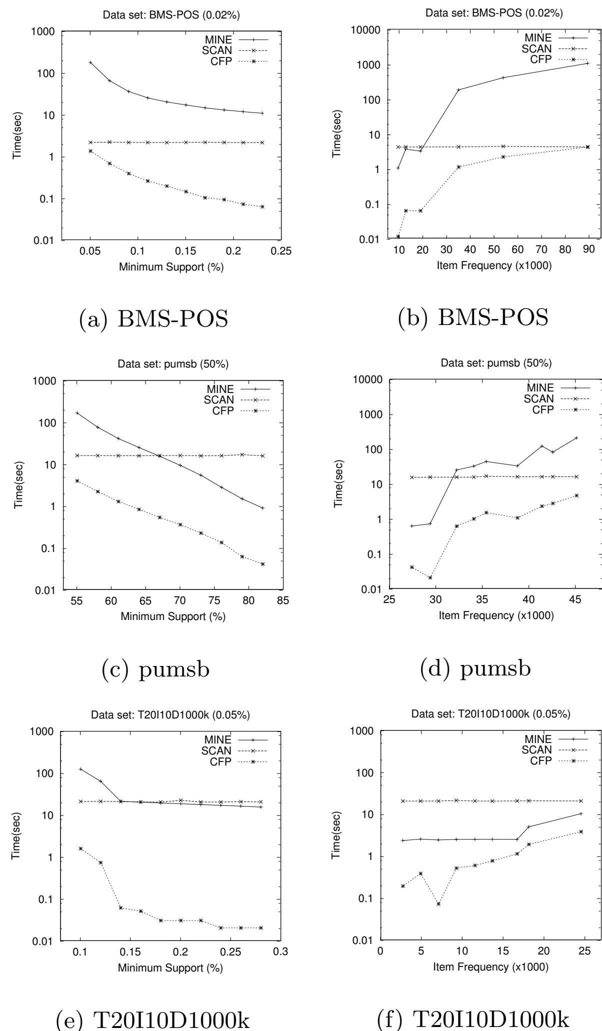


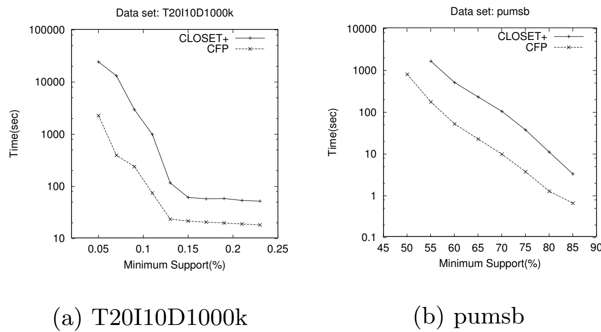
Figure 2: Query Processing Time

Then we sequentially scan the file to find all the patterns that satisfy the minimum support constraint or item constraint. The y-axis in all figures is logarithmic. For queries with item constraint, we used only one item as the constraint. The selection of the item is based on its frequency. The less frequent of an item, the more pruning power of it in searching the CFP-tree structure and also in the mining process. The x-axis in Figure 2(b), 2(d) and 2(f) is the frequency of the selected item. For "MINE" algorithm, we set the minimum support to the building minimum support of the CFP-tree. Figure 2 shows the time for processing two types of queries on three datasets. In most cases, searching patterns from pre-computed results needs less time than mining from scratch. The time for sequential scan does not vary with the minimum support and the frequency of the items. The time for retrieving patterns from CFP-tree increases with the minimum support threshold and the frequency of the items, but can hardly exceeds the time for sequential scan.

5.2 Construction Time and CFP-tree Size

The CFP-tree can be constructed quickly using our proposed algorithm. We compared our construction algorithm with CLOSET+[14], which is shown to consistently defeat

other frequent closed pattern mining algorithms. The two algorithms show comparable performance on BMS-POS. Figure 3 shows the running time on the other two datasets.



(a) T20I10D1000k (b) pumsb

Figure 3: CFP-tree Construction Time

We also compared the size of the CFP-tree structure with the size of the closed pattern set stored in flat files. The size of the CFP-tree structure can be much smaller than the flat representation, especially when dataset is dense. For example, on pumsb dataset with minimum support 50%, the size of the CFP-tree is almost one third of the size of flat representation. Due to the limitation of the space, detailed experiment results are not shown here.

6. RELATED WORK

During the past decade, extensive efforts have been devoted to developing efficient algorithms for mining frequent patterns. They can be classified into two categories: the Apriori family algorithms[1, 7, 12, 2] and the pattern-growth based algorithms[16, 3, 11, 9, 5, 4]. The pattern growth based algorithms differ mainly in the representation of the conditional databases. There were also works on mining only frequent closed patterns [8, 15, 10, 14].

However, there is a lack of studies on how to store and retrieve frequent patterns. Morzy et al. proposed a group bitmap index structure for retrieving association rules stored in a relation database [6]. This technique is similar to our hash bitmap. Tuzhilin et al. proposed a rule query language Rule-QL for querying multiple rulebases and a number of efficient query evaluation techniques for Rule-QL [13]. They use separated indexes for support/confidence constraint and item constraint. More specifically, they use B+ trees to index the support and the confidence of the rules, and use inverted lists for subset matching. We believe that the CFP-tree structure can be an alternative for indexing association rules.

7. DISCUSSION AND CONCLUSIONS

In this paper, we proposed a compact and efficient data structure, CFP-tree, for storing and querying frequent patterns on disk. With CFP-tree, frequent pattern mining can be conducted in "mining once and using many times" fashion by querying the stored CFP-tree. Two types of important queries related to frequent itemset mining can be answered quickly using our proposed query processing algorithms, namely the query with minimum support constraint and the query with item constraint. We have also observed consistent results concerning queries with both constraints. In fact, the benefit gained by the CFP-tree structure was more significant because we can exploit the pruning powers of the two constraints at the same time. This part of experiment results are not shown in this paper due to the limitation of space.

In this paper we did not discuss how to update the CFP-tree structure when underlying database changes. A simple but costly solution is to reconstruct the tree periodically. A more efficient approach is to adopt the idea of the incremental mining algorithms to minimize the scanning cost of the original database.

8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, 1997.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[4] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *DASFAA*, 2003.

[5] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *SIGKDD*, 2002.

[6] T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In *SIGKDD*, 1998.

[7] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *SIGMOD*, 1995.

[8] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[9] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, 2001.

[10] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *DMKD*, 2000.

[11] R.C.Agarwal, C.C.Aggarwal, and V.V.V.Prasad. A tree projection algorithm for finding frequent itemsets. *Journal on Parallel and Distributed Computing*, 61(3):350–371, 2001.

[12] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, 1995.

[13] A. Tuzhilin and B. Liu. Querying multiple sets of discovered rules. In *SIGKDD*, 2002.

[14] J. Wang, J. Pei, and J. Han. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD*, 2003.

[15] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. 2002.

[16] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *SIGKDD*, 1997.

[17] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *SIGKDD*, 2001.