CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets[*]

Jianyong Wang
Department of Computer Science
University of Illinois at Urbana-Champaign
wangj@cs.uiuc.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
hanj@cs.uiuc.edu

Jian Pei
Department of Computer Science and Engineering
State University of New York at Buffalo
jianpei@cse.buffalo.edu

ABSTRACT

Mining frequent closed itemsets provides complete and non-redundant results for frequent pattern analysis. Extensive studies have proposed various strategies for efficient frequent closed itemset mining, such as depth-first search vs. breadth-first search, vertical formats vs. horizontal formats, tree-structure vs. other data structures, top-down vs. bottom-up traversal, pseudo projection vs. physical projection of conditional database, etc. It is the right time to ask "*what are the pros and cons of the strategies?*" and "*what and how can we pick and integrate the best strategies to achieve higher performance in general cases?*"

In this study, we answer the above questions by a systematic study of the search strategies and develop a winning algorithm CLOSET+. CLOSET+ integrates the advantages of the previously proposed effective strategies as well as some ones newly developed here. A thorough performance study on synthetic and real data sets has shown the advantages of the strategies and the improvement of CLOSET+ over existing mining algorithms, including CLOSET, CHARM and OP, in terms of runtime, memory usage and scalability.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications—*Data Mining*

General Terms

Mining methods and algorithms

Keywords

Frequent closed itemsets, association rules

[*]The work was supported in part by U.S. National Science Foundation NSF IIS-02-09199, University of Illinois, Microsoft Research, and IBM Faculty Award.

1. INTRODUCTION

Since the introduction of association rule mining [1], there have been extensive studies on efficient frequent itemset mining methods, such as [2, 11, 16, 6, 4, 3, 8, 7, 18, 10]. Most of the well studied frequent pattern mining algorithms, including Apriori [2], FP-growth [8], H-mine [13], and OP [10], mine the complete set of frequent itemsets. These algorithms may have good performance when the support threshold is high and the pattern space is sparse. However, when the support threshold drops low, the number of frequent itemsets goes up dramatically, and the performance of these algorithms deteriorates quickly because of the generation of a huge number of patterns. Moreover, the effectiveness of the mining of the complete set degrades because it generates numerous redundant patterns. A simple example is that, in a database having only one transaction of length 100, it will generate $2^{100}-1$ frequent itemsets if the absolute minimum support threshold is set to 1.

The closed itemset mining, initially proposed in [12], mines only those frequent itemsets having no proper superset with the same support. Mining closed itemsets, as shown in [17], can lead to orders of magnitude smaller result set (than mining frequent itemsets) while retaining the completeness, i.e., from this concise result set, it is straightforward to generate all the frequent itemsets with accurate support counts.

In the last several years, extensive studies have proposed fast algorithms for mining frequent closed itemsets, such as A-close [12], CLOSET [14], MAFIA (it has an option to generate closed itemsets) [5], and CHARM [18]. Various search strategies have been developed, such as depth-first search vs. breadth-first search, vertical formats vs. horizontal formats, tree-structure vs. other data structures, top-down vs. bottom-up traversal, pseudo projection vs. physical projection of conditional database, etc. However, two critical things are missing: (1) there is no systematic study on comparing the strategies and evaluate their pros and cons objectively; and (2) there is no thorough discussion on how to integrate the winning strategies and achieve an even better algorithm. With the research proceeded so far, it is the right time to ask "*what are the pros and cons of the strategies?*" and "*what and how can we pick and integrate the best strategies to achieve higher performance in general cases?*"

In this study, we answer the above questions by a systematic study on the search strategies and develop a winning algorithm CLOSET+. CLOSET+ integrates the advantages of the previously proposed effective strategies as well as some ones newly developed here. A thorough performance study on synthetic and real data sets has shown the advantages of the strategies and the improvement of CLOSET+ over existing mining algorithms, including CLOSET, CHARM and OP, in terms of runtime, memory usage and scalability.

The remaining of the paper is organized as follows. In Section 2, we briefly revisit the problem definition of frequent closed itemset mining and the related work. In Section 3, we present an overview of the principal search strategies developed before and analyze their pros and cons. In Section 4, we devise algorithm CLOSET+ by integrating some winning strategies as well as some novel ones developed here. A thorough performance study of CLOSET+ in comparison with several recently developed efficient algorithms is reported in Section 5. We conclude this study in Section 6.

2. PROBLEM DEFINITION AND RELATED WORK

A *transaction database* TDB is a set of transactions, where each transaction, denoted as a tuple $\langle tid, X\rangle$, contains a set of items (i.e., X) and is associated with a unique transaction identity *tid*. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the complete set of distinct items appearing in TDB. An *itemset* Y is a non-empty subset of I and is called an *l-itemset* if it contains l items. An itemset $\{x_1, \ldots, x_l\}$ is also denoted as $x_1 \cdots x_l$. A transaction $\langle tid, X\rangle$ is said to *contain* itemset Y if $Y \subset X$. The number of transactions in TDB containing itemset Y is called the *support* of itemset Y, denoted as $sup(Y)$. Given a minimum support threshold, *min_sup*, an itemset Y is *frequent* if $sup(Y) \geq min\_sup$.

Definition 1 (Frequent closed itemset). An itemset Y is a **frequent closed itemset** if it is frequent and there exists no proper superset $Y' \supset Y$ such that $sup(Y') = sup(Y)$. ■

Example 1. The first two columns in Table 1 show the transaction database TDB in our running example. Suppose $min\_sup = 2$, we can find and sort the list of frequent items in support descending order. The sorted item list is called *f_list*. In this example *f_list* = ⟨f:4, c:4, a:3, b:3, m:3, p:3⟩. The frequent items in each transaction are sorted according to *f_list* and shown in the third column of Table 1. Itemset fc is a frequent 2-itemset with support 3, but it is not closed, because it has a superset $fcam$ whose support is also 3. $facm$ is a frequent closed itemset. ■

Tid	Set of items	ordered frequent item list
100	a, c, f, m, p	f, c, a, m, p
200	a, c, d, f, m, p	f, c, a, m, p
300	a, b, c, f, g, m	f, c, a, b, m
400	b, f, i	f, b
500	b, c, n, p	c, b, p

Table 1: A transaction database TDB.

Related work Popular algorithms for mining frequent closed itemsets include A-close [12], CLOSET [14], MAFIA [5] and CHARM [18]. A-close uses a breadth-first search to find the frequent closed patterns. In dense datasets or datasets with long patterns, breadth-first searches may encounter difficulties since there could be many candidates and the searches need to scan the database many times. This is shown in several performance studies (e.g., [14, 18]).

CLOSET[14], is an extension of the FP-growth algorithm [8], which constructs a frequent pattern tree FP-tree and recursively builds conditional FP-trees in a bottom-up tree-search manner. Although CLOSET uses several optimization techniques to enhance the mining performance, its performance still suffers in sparse datasets or when the support threshold is low.

Both MAFIA [5] and CHARM [18] use a vertical representation of the datasets. MAFIA is mainly designed for mining maximal itemsets, but it has an option to mine closed itemsets. One of its main features is the compressed vertical bitmap structure. CHARM enumerates closed itemsets using a dual itemset-tidset search tree and adopts the *Diffset* technique to reduce the size of the intermediate tidsets. The most costly operation for the algorithms using vertical format is the intersection on tidsets. CHARM shows better performance than A-close, Pascal, MAFIA, and CLOSET in many dense datasets.

3. STRATEGIES FOR FREQUENT CLOSED ITEMSET MINING

Various strategies for mining frequent closed itemsets are proposed in the previous studies. In this section, we present a systematic overview of these strategies, and analyze their pros and cons.

One essential principle for frequent itemset mining is the Apriori property [2]: "*every subset of a frequent itemset must be frequent*". Accordingly, every frequent itemset consists of only frequent items.

Given a set of frequent items, F, the complete set of itemsets is a lattice over 2^F. It can be shown that the complete set of closed itemsets forms a sub-lattice of 2^F. Fig. 1(a) draws the portion of frequent closed itemsets in our running example.

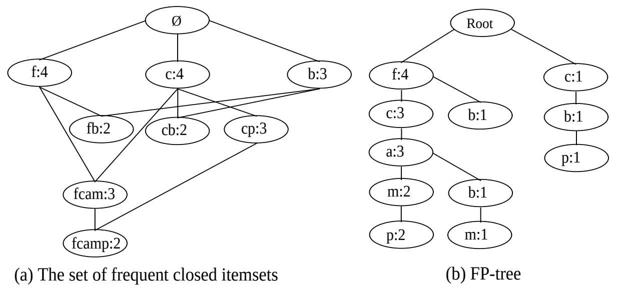

Figure 1: The set of frequent closed itemsets and the FP-tree in the running example.

The problem of searching for the complete set of frequent closed itemsets is to find the complete set of frequent itemsets in the lattice of closed itemsets. Most methods start from the top of the lattice (i.e., the frequent items). The strategies can be divided into several orthogonal categories.

Breadth-first vs. depth-first search. The breadth-first search approaches search the lattice level-by-level: it uses the frequent itemsets at the current level with length k to generate the candidates at the next level with length $(k+1)$, and a new database scan is needed to count the supports of length-$(k+1)$ candidates. Due to its too many database scans, it is not suitable for mining long patterns. In contrast, a depth-first search method traverses the lattice in depth-first order, and the subtree of an itemset is searched only if the itemset is frequent. Moreover, when the itemsets becomes longer, depth-first search shrinks search space quickly. As a result, the depth-first search method is usually a winner for mining long patterns. Some previous studies (e.g., [14, 5, 18]) clearly elaborate that the depth-first search methods are usually more efficient than the breadth-first search methods like A-close.

Horizontal vs. vertical data formats. The transaction information can be recorded in two alternative formats. The horizontal format is an intuitive bookkeeping of the transactions. Each transaction is recorded as a list of items. In the vertical format, instead of recording the transactions explicitly, a *tid-list* is kept for every item, where the identities of the transactions containing the item are listed. A-close and CLOSET use the horizontal data format, while CHARM and MAFIA use the vertical one.

A vertical format-based method needs to maintain a tidset for each frequent itemset. When the database is big, each tidset is on average big, and many such intermediate results will consume a lot of memory. In contrast, if we properly choose a compressed structure like FP-tree, a horizontal format-based method will not cause too much space usage, because the itemsets can share common path if they share common prefix, and each of their tidlists is represented by a count. Moreover, for a vertical format-based algorithm, one intersection operation can only find one frequent itemset. For a horizontal format-based method like CLOSET, one scan of a projected database can find many local frequent items which can be used to grow the prefix itemset to generate frequent itemsets. In this paper, we will compare CLOSET+, a horizontal format-based algorithm, with CHARM, a vertical format-based one, in terms of scalability and efficiency in both runtime and space usage.

Data compression techniques. A transaction database is usually huge. If a database can be compressed and only the information related to the mining is kept, the mining can be efficient. Recently, some data compression methods have been devised. FP-tree and *Diffset* are two typical examples.

An FP-tree [8] of a transaction database is a prefix tree of the lists of frequent items in the transactions. The idea can be illustrated in the following example.

Example 2. The FP-tree of our running example is constructed as follows: Scan the database once to find the set of frequent items and sort them in the support descending order to get the *f_list* (see Example 1). To insert a transaction into the FP-tree, infrequent items are removed and the remaining items in the transaction are sorted according to the item ordering in *f_list*, i.e., the least frequent item at the leaf, and the items with higher global support at a higher level in the FP-tree. Fig. 1(b) shows the FP-tree. ■

The FP-tree structure has several advantages in mining frequent itemsets. First, FP-tree often has a high compression ratio in representing the dataset because (1) infrequent items identified in the first database scan will not be used in the tree construction, and (2) a set of transactions sharing the same subset of items may share common prefix paths from the root in an FP-tree. According to our experience, for some dense datasets, its compression ratio can reach several thousand. Even for sparse datasets, it is still quite effective in compressing original datasets, especially when database is large (e.g., many real retail databases contain billions of transactions), since many transactions may share some common subsets of items. Second, the high compression ratio leads to efficient frequency counting at iterative scanning of FP-tree. Third, efficient depth-first search becomes straightforward using FP-tree. More importantly, FP-tree contains all the necessary information for mining frequent itemsets, its completeness can assure the correctness of an FP-tree based algorithm.

Diffset is an efficient compression of the tid-set for methods adopting the vertical data format. For a vertical format-based algorithm like CHARM, computing the supports requires intersections on tidsets, when the tidset cardinality becomes large, not only will the tidsets consume much memory, but also the tidset intersection gets costly. To overcome this, CHARM develops a *Diffset* technique to keep track of only the differences in the tids of a candidate pattern from its parent pattern. Experiments in [18] showed *Diffset* can reduce the space usage by orders of magnitude.

Pruning techniques for closed itemset mining. In the previous studies of *depth-first search* approaches for mining frequent closed (or maximal) itemsets, mainly two search space pruning techniques have been proposed as the following two lemmas. These techniques have been used in Max-Miner [3], CLOSET [14], MAFIA [5] and CHARM [18].

Lemma 3.1. ***(item merging)*** *Let X be a frequent itemset.* If every transaction containing itemset X also contains itemset Y but not any proper superset of Y, then $X \cup Y$ forms a frequent closed itemset and there is no need to search any itemset containing X but no Y.

Example *3.* In our running example shown in Table 1, the projected conditional database for prefix itemset fc:3 is $\{(a,m,p), (a,m,p), (a,b,m)\}$ (items d and g are infrequent and removed), from which we can see each of its transaction contains itemset am but no proper superset of am. Itemset am can be merged with fc to form a closed itemset $fcam$:3, and we do not need to mine closed itemsets containing fc but no am. ■

Lemma 3.2. ***(sub-itemset pruning)*** *Let X be the frequent itemset currently under consideration.* If X is a proper subset of an already found frequent closed itemset Y and $sup(X) = sup(Y)$, then X and all of X's descendants in the set enumeration tree [15] cannot be frequent closed itemsets and thus can be pruned.

Example *4.* Many frequent pattern mining algorithms follow the *divide-and-conquer* paradigm. In our running example shown in Fig. 1(b), a top-down *divide-and-conquer* paradigm follows the *f_list* order shown in Example 1 (in contrast, a bottom-up *divide-and-conquer* paradigm will follow the inverse *f_list* order): (1) first mine the patterns containing item f, (2) mine the patterns containing item c but no f, (3) mine the patterns containing item a but no f nor c, ..., and finally mine the patterns containing only p. At some point when we want to mine the patterns with prefix itemset ca:3, we will find that ca:3 is a proper subset of an already found closed itemset $fcam$:3 with the same support, we can safely stop mining the closed patterns with prefix ca:3. ■

4. CLOSET+: AN EFFICIENT METHOD FOR CLOSED ITEMSET MINING

In this section, we devise a new frequent closed itemset mining algorithm, CLOSET+, by integrating some winning search strategies and developing some novel ones.

4.1 Overview of CLOSET+

CLOSET+ follows the popular *divide-and-conquer* paradigm which has been shown one possible instance in Example 4 and the *depth-first search* strategy which has been verified a winner for mining long patterns by several efficient frequent pattern mining algorithms. It uses FP-tree as the compression technique. A depth-first search and horizontal format-based method like CLOSET+ will compute the local frequent items of a certain prefix by building and scanning its projected database. Therefor, a *hybrid tree-projection* method will be introduced to improve the space efficiency.

Unlike frequent itemset mining, during the closed itemset mining process there may exist some prefix itemsets that are unpromising to be used to grow closed itemsets. We should detect and remove such unpromising prefix itemsets as quickly as possible. Besides adopting the above mentioned *item merging* and *sub-itemset pruning* methods, we also proposed the *item skipping* technique to further prune search space and speed up mining.

Previous algorithms need to maintain all the frequent closed itemsets mined so far in memory in order to check if a newly found closed itemset candidate is really closed. If there exist a lot of frequent closed patterns, such kind of closure checking will be costly in both memory usage and runtime. We have also designed an efficient *subset-checking* scheme: the combination of the 2-level hash-indexed *result tree* based method and the *pseudo-projection based upward checking* method, which can be used to save memory usage and accelerate the closure-checking significantly. In the following, the above mentioned mining techniques and the CLOSET+ algorithm are developed step by step.

4.2 The Hybrid Tree Projection Method

Most previously developed FP-tree-based methods, such as FP-growth and CLOSET, grow patterns by projection of conditional databases in a bottom-up manner [8, 14]. However, such a search order may not always lead to the best performance for different kinds of data sets. In CLOSET+, a *hybrid tree-projection* method is developed, which builds conditional projected databases in two ways: *bottom-up physical tree-projection for dense datasets* and *top-down pseudo tree-projection for sparse datasets.*

4.2.1 Bottom-up physical tree-projection

For dense datasets, their FP-trees can be hundreds (or even thousands) times smaller than their corresponding original datasets due to compression. Its conditional projected FP-trees are usually very compact as well. Each projected FP-tree is much smaller than the original FP-tree, and mining on such a compact structure will also be efficient. As a result, for dense datasets CLOSET+ still physically builds projected FP-trees and it is done recursively in a bottom-up manner (i.e., in support ascending order).

To assist the physical FP-tree projection, there is a header table for each FP-tree, which records each item's ID, count, and a side-link pointer that links all the nodes with the same itemID as the labels. The global FP-tree in our example is shown in Fig. 2(a). To build conditional FP-tree for prefix item p:3, we will first find the conditional database containing p (denoted as $TDB|_{p:3}$) by following item p's side-link pointers. A *prefix path* from a node, N_p, which has an itemID p and a count C_p, upward to the root node represents an *amalgamative transaction* with support C_p for prefix item p. Here $TDB|_{p:3}$ consists of two amalgamative transactions: $\langle fcam : 2\rangle$ and $\langle cb : 1\rangle$, from which the projected FP-tree for prefix p:3 is built as Fig. 2(b).

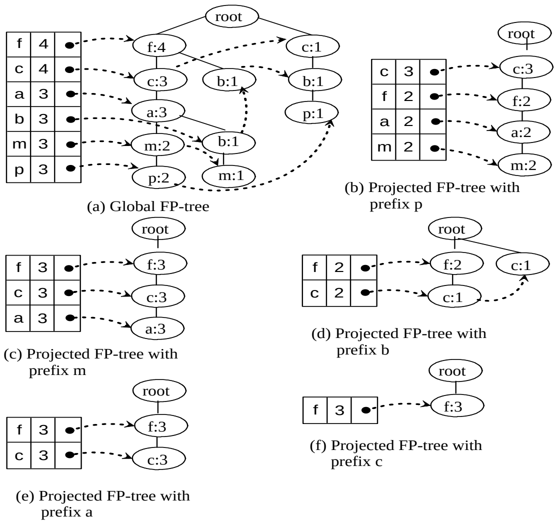
root
f 4
c 4
a 3
b 3
m 3
p 3
f:4
c:3
a:3
m:2
p:2
b:1
b:1
m:1
c:1
b:1
p:1
(a) Global FP-tree
c 3
f 2
a 2
m 2
c:3
f:2
a:2
m:2
(b) Projected FP-tree with prefix p
f 3
c 3
a 3
f:3
c:3
a:3
(c) Projected FP-tree with prefix m
f 2
c 2
f:2
c:1
c:1
(d) Projected FP-tree with prefix b
f 3
c 3
f:3
c:3
(e) Projected FP-tree with prefix a
f 3
f:3
(f) Projected FP-tree with prefix c

Figure 2: Bottom-up physical tree-projection.

After the projected FP-tree for prefix p:3 has been constructed, we will mine the frequent closed itemsets with prefix p:3 from it. (1) First, we mine the closed itemsets with prefix pm:2. By following item m's side-link pointer in Fig. 2(b), we build prefix pm:2's conditional database, as $TDB|_{pm:2} = \langle cfa : 2\rangle$. According to the *item merging* technique, prefix pm:2 can be merged with itemset cfa:2 to form a frequent closed itemset, $pmcfa$:2, and we do not need to mine closed itemsets with prefix pmc:2, pmf:2, or pma:2. (2) Second, we'll mine closed itemsets with prefix pa:2. Because pa:2 is a proper subset of an already mined itemset $pmafc$:2 with the same support, according to the *sub-itemset pruning* method, there is no need to mine closed itemsets with prefix pa:2. (3) Similarly, prefix pf:2 cannot be used as a start point to grow any closed itemsets. (4) Finally, we will mine closed itemsets with prefix pc:3, by following item c's side-link pointer in Fig. 2(b), we find its conditional database is empty, we only need to output prefix pc:3 as a frequent closed itemset candidate. Until now all the frequent closed itemsets for prefix p:3 have been mined.

Similarly, we can build physically projected FP-trees from the global FP-tree and mine frequent closed itemsets from them in a recursive way for prefixes m:3, b:3, a:3, c:4, and f:4 respectively, these FP-trees are shown in Fig. 2(c)-(f) (The FP-tree for prefix f:4 is empty and is not shown).

4.2.2 Top-down pseudo tree-projection

Physically projecting FP-trees will introduce some overhead both in space usage and runtime (due to allocating and freeing memory for projected FP-trees), especially if the dataset is sparse, the projected FP-tree will not be very

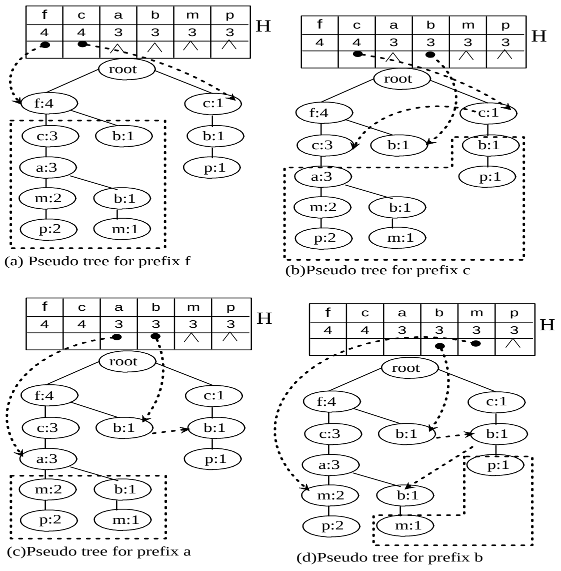
(a) Pseudo tree for prefix f
(b)Pseudo tree for prefix c
(c)Pseudo tree for prefix a
(d)Pseudo tree for prefix b

Figure 3: Top-down pseudo tree-projection.

compact and does not shrink quickly. Instead of physically building projected FP-trees, a new method is developed for sparse datasets: *top-down pseudo projection of* FP-tree. Unlike bottom-up physical projection of FP-trees, the pseudo projection is done in the *f_list* order (i.e., in support descending order).

Similar to bottom-up physical projection, a header table is also used to record enough information such as local frequent items, their counts and side-link pointers to FP-tree nodes in order to locate the subtrees for a certain prefix itemset. Based on our running example, the top-down pseudo-projection method is illustrated as follows. Fig. 3(a) shows the global FP-tree and the status of global header table. Initially only the child nodes (e.g., nodes f:4 and c:1) directly under the root node are linked from the global header table. Because we build FP-tree according to the *f_list* order, all the projected transactions containing item f can be found from the subtree under the node with label f:4 (i.e., the dashed polygon in Fig. 3(a)).

By following the side-link pointer of item f in the global header table of Fig. 3(a), we can locate the subtree under node f:4. The local frequent items can be found by scanning this subtree and used to build the header table for prefix itemset f:4, as shown in Fig. 4(a). Here only the child nodes directly under node f:4 are linked from the header table $H_{f:4}$. Based on the header table $H_{f:4}$, we can mine the frequent itemsets with prefix f. (1) First, we'll mine closed itemsets containing fc:3. By scanning the subtree under node c:3 in Fig. 4(a), we can find the local frequent items and build the header table for prefix itemset fc:3, as shown in Fig. 4(b). Items a and m have the same support as that of prefix fc:3, according to the *item merging* technique, they can be merged with fc:3 to form a new prefix $fcam$:3 (and it is also a closed itemset) and we will not need to mine closed itemsets with prefix fca:3 or fcm:3. Although fca:3 cannot be used as a prefix to grow closed itemsets, we still need to follow item a's side-link pointer to find all the child nodes directly under node a:3 and make them linked from header table $H_{fc:3}$. Because item b is infrequent in $H_{fc:3}$, the node b:1 under node a:3 does not need to be linked from $H_{fc:3}$, instead, its child node m:1 will be linked from $H_{fc:3}$. The new header table $H_{fc:3}$ with adjusted side-link pointers is shown in Fig. 4(c). Similarly, we do not need to mine closed itemsets with prefix fcm:3, but the child nodes under nodes m:2 and m:1 should be linked from $H_{fc:3}$, as shown in Fig. 4(d). When we mine the closed itemsets with prefix $fcamp$:2, we find the subtree under node p:2 is empty, we'll output $fcamp$:2 as a frequent closed itemset candidate and stop mining closed itemsets with prefix fc:3. (2) Second, after the child node a:3 of node c:3 (see Fig. 4(a)) has been linked from header table $H_{f:4}$, we can mine closed itemsets with prefix fa:3 but no c. (3) In a similar way, we can mine closed itemsets with prefix fb but no c nor a, with fm but no c nor a and nor b, and those only with fp, respectively.

As illustrated in the above example, we need to do two kinds of things in the process of mining closed itemsets for a certain prefix: (1) find its subtrees by following its side-link pointers and recursively mine these subtrees to find all its frequent closed itemsets; and (2) after that, do side-link pointer adjustment, i.e., all its child nodes should be linked from the head table. Fig. 3(b), Fig. 3(c), and Fig. 3(d) show the side-link adjustment of the global header table H after closed itemsets with prefix f:4, c:4, and a:3 have been mined respectively, and the dashed polygons in those figures represent the projected FP-trees for prefix c:4, a:3 and b:3, respectively.

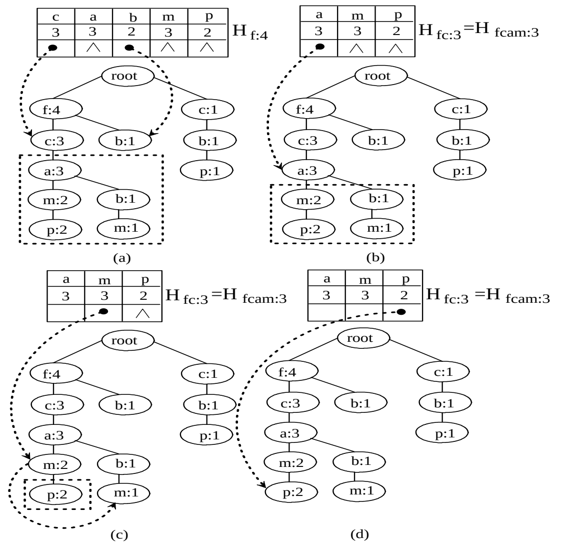

Figure 4: Top-down pseudo tree-projection for f:4.

4.3 The Item Skipping Technique

Since CLOSET+ adopts depth-first search, at each level, there will be a prefix itemset X associated with a header table and a projected FP-tree. According to the *Apriori* property, a local frequent item in X's header table must also appear in the higher-level header tables. If a local frequent item has the same support in more than one header table at different levels, we can use Lemma 4.1 to prune search space.

LEMMA 4.1. ***(Item skipping)*** *If a local frequent item has the same support in several header tables at different levels,* *one can safely prune it from the header tables at higher levels.*

Proof. *Assume item x at level l is a local frequent item of prefix itemset X_l and has the same support in level k's header table of prefix itemset X_k, where $(0 \le k < l) \wedge (X_k \subset X_l)$. Let $X'_k = X_k \cup x$ and $X'_l = X_l \cup x$, it is obvious that $(X'_k \subset X'_l) \wedge (sup(X'_k) = sup(X'_l))$, which means any frequent itemset grown from X'_k can be subsumed by a corresponding frequent itemset grown from X'_l. As a result it is non-closed and item x can be pruned from header table at level k.* ■

This pruning method can be used in both bottom-up physical and top-down pseudo tree-projection paradigms. For example, in Fig. 2(c), there is a local frequent item a with support 3 for prefix itemset m:3. We also find that item a appears in the global header table (see Fig. 2(a)) with the same support, item a can be safely pruned from the global header table. Similarly, from Fig. 4, we know that items a, m, and p in $H_{fc:3}$ have the same supports as those in $H_{f:4}$, these items can be safely removed from $H_{f:4}$.

4.4 Efficient Subset Checking

The search space pruning methods can only be used to remove some prefix itemsets that are unpromising to be used as a start point to grow closed itemsets, but they cannot assure that a frequent prefix itemset is closed. When we get a new frequent prefix itemset, we need to do two kinds of closure checking: the *superset-checking* checks if this new frequent itemset is a superset of some already found closed itemset candidates with the same support, while the *subset-checking* checks if the newly found itemset is a subset of an already found closed itemset candidate with the same support. Because both bottom-up physical projection and top-down pseudo projection work under the *divide-and-conquer* and *depth-first-search* framework, the following theorem states that CLOSET+ only needs to do subset-checking in order to assure a newly found itemset is closed.

Theorem 4.1. ***(Subset checking)*** *Under the framework of divide-and-conquer and using the item merging pruning method introduced in Lemma 3.1, a frequent itemset found by* CLOSET+ *must be closed if it cannot be subsumed by any other already found frequent closed itemset.*

Proof. *Let the list of items in which order* CLOSET+ *mines frequent closed itemset be m_list = $\langle I_1, I_2, \ldots, I_n \rangle$ and the current frequent itemset* CLOSET+ *has just found is $S_c = I_{c1}I_{c2}\ldots I_{cx}$. Also, we define the relationship between two items I_m and I_n as $I_m < I_n$ if item I_m is located before item I_n in m_list. For any two itemsets $S_1 = I_{11}I_{12}\ldots I_{1i}$ and $S_2 = I_{21}I_{22}\ldots I_{2j}$ where $i > 0$ and $j > 0$, we define $S_1 < S_2$ if there exists an integer k ($k \ge 1$), $I_{1k} < I_{2k}$ and $I_{1l} = I_{2l}$ (for all $l < k$) hold. Following we will prove itemset $I_{c1}I_{c2}\ldots I_{cx}$ cannot be subsumed by a later found frequent closed itemset, which also means $I_{c1}I_{c2}\ldots I_{cx}$ cannot subsume any other already found closed itemsets.*

First, for any frequent itemset $S_l = I_{l1}I_{l2}\ldots I_{ly}$ which is mined later than S_c, we can classify it into one of the two categories: (1) S_l is generated by growing S_c; (2) S_l is not generated by growing prefix S_c. If S_l belongs to the first category, S_l is a superset of S_c, but because we have applied the item merging technique, which means all the local items with the same support as S_c's must have been included in S_c, $sup(S_l)$ must be smaller than $sup(S_c)$. As a consequence, S_c cannot be subsumed by S_l. If S_l belongs to the second category, based on the nature of our divide-and-conquer framework, we know $S_c < S_l$, and there must exist an integer k, $I_{ck} < I_{lk}$ holds, which means S_l does not contain item I_{ck} and as a result S_l cannot be a superset of S_c. ■

From Theorem 4.1, we know that a newly found frequent itemset cannot be subsumed by any later found frequent itemset. That is, if it cannot be subsumed by any already found closed itemset, it must be closed. To assist the subset-checking, we have designed two efficient techniques.

Two-level hash-indexed result tree. The first method maintains in a compressed way the set of closed itemsets mined so far in memory. Inspired by the FP-tree [8] implementation, we can store all the frequent closed itemsets in a compressed *result tree* which has been used in a slightly different way to do both *superset and subset checking* [9]. Here we will use it to perform subset-checking in order to assure a newly found itemset is closed and only if it is closed can we insert it into the *result tree*.

Now we need to consider how to design efficient subset-checking based on the *result-tree*. In CLOSET+, we try to exploit some features of closed itemsets to reduce the search space. If the current itemset S_c can be subsumed by another already found closed itemset S_a, they must have the following relationships: (1) S_c and S_a have the same support; (2) length of S_c is smaller than that of S_a; and (3) all the items in S_c should be contained in S_a.

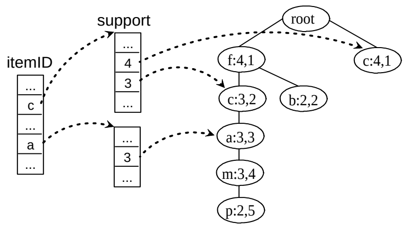

Figure 5: Two-level hash indexed *result tree*.

Using these heuristics, we can improve the structure of *result tree*. First, we introduce 2-level hash indices into *result tree*: one level uses ID of the last item in S_c as hash key, another uses support of S_c as hash key, and the *result tree* nodes falling into the same bucket will be linked together. Second, we insert each closed itemset into the *result tree* according to the *f_list* order, and at each node we also record its length of the path from this node to the root node. Following we use our running example to illustrate the maintenance of *result tree* and how to do subset checking.

Example 5. Under depth-first search paradigm, the set of closed itemsets shown in Fig.1(a) will be mined and inserted into the *result tree* in the following order: f:4, $fcam$:3, $fcamp$:2, fb:2, c:4, cb:2, cp:3, and b:3. Fig.5 depicts the status of the *result tree* after inserting closed itemset c:4 (Here we only show part of the index structure due to limited space). In each node, we record the itemID, support, and the length (relative to root node) respectively. Differently from the FP-tree structure, when several closed itemsets share some common prefixes, the support of a node in the common prefix will be the maximum one among the supports of itemsets sharing the common prefix instead of the sum of supports of itemsets sharing the common prefix.

At the status shown in Fig. 5, we may later get a frequent itemset *ca*:3. By using itemID *a* and support 3 as hash keys and following the corresponding hash link, we will find the node labeled as *a*:3,3 with a length greater than 2, we then check if *ca*:3 can be absorbed by the path from node *a*:3,3 to the root. Unfortunately it cannot pass the checking and will not be inserted into the *result tree*. ■

Pseudo-projection based upward checking. Although the *result tree* can compress the set of closed itemsets a lot, it still consumes much memory and is not very space-efficient for sparse datasets. Can we totally remove the requirement of maintaining the set of closed itemsets in memory for subset-checking? As we know, the global FP-tree contains the complete information about the whole set of frequent closed itemsets, thus we can use the global FP-tree to check if a newly found frequent itemset is closed. In such a way, we do not need any additional memory for storing the set of already mined closed itemsets and once a newly found itemset has passed the checking, it will be directly stored in an output file, F.

The problem becomes how to do *subset-checking* based on the global FP-tree. As we know, in the top-down pseudo projection method, all the tree nodes and their corresponding *prefix path*s w.r.t. a prefix itemset, X, can be traced by following its side-link pointer recorded in its header table. We can use the following lemma 4.2 to judge whether a newly found frequent itemset is closed.

LEMMA 4.2. *For a certain prefix itemset, X, as long as we can find any item which (1) appears in each prefix path w.r.t. prefix itemset, X, and (2)does not belong to X, any itemset with prefix X will be non-closed, otherwise, if there's no such item, the union of X and the complete set of its local frequent items which have the same support as $sup(X)$ will form a closed itemset.*

Proof. The first part is easy to prove: If we can find an item, i_x, which (1)appears in each *prefix path* w.r.t. prefix itemset, X, and (2) does not belong to X, this will mean $(X \subset (X \cup \{i_x\}))$ and $(sup(X) = sup(X \cup \{i_x\}))$ hold, as a result, X or any itemset with prefix X will be non-closed.

For the second part, because we cannot find any item which (1)appears in each *prefix path* w.r.t. prefix itemset, X, and (2) does not belong to X, and any other possible items that always appear together with X can only belong to the set of X's local frequent items, as a result, the union of X and the complete set of its local frequent items which have the same support as $sup(X)$ must form a closed itemset. ■

Here we'll use some examples to illustrate the *upward subset-checking*. Assume the prefix X=*c*:4, we can locate nodes *c*:1 and *c*:3 by following the side-link pointer of item *c* in Fig 3(b) and find there is only one item, *f*, which appears in prefix itemset *c*:4's prefix paths to the root, and it only co-occurs 3 times with prefix *c*:4. In addition, there's no local frequent item of prefix *c*:4 with support 4, thus *c*:4 is closed and will be stored in output file F. Using this method, we can easily figure out that prefix *am*:3 is not closed, because in the prefix paths of nodes *m*:2 and *m*:1, there are two other items, *f* and *c*, which appear with *am* 3 times.

4.5 The Algorithm

By integration of the techniques discussed above, we derive the CLOSET+ algorithm as follows.

ALGORITHM 1: *Closed itemset mining with* CLOSET+
INPUT: (1)*A transaction database TDB*, and (2) *support threshold min_sup.*
OUTPUT: *The complete set of frequent closed itemsets.*
METHOD:

1. *Scan TDB once to find the global frequent items and sort them in support descending order. The sorted frequent item list forms the f_list.*
2. *Scan TDB and build* FP-tree *using the f_list.*

   Note: In the tree building process, compute the average count of an FP-tree *node. After the tree has been built, we will judge whether the dataset is dense or sparse according to the average count of an* FP-tree *node: for dense dataset, choose bottom-up physical tree-projection method; whereas for sparse dataset, use top-down pseudo tree-projection method. According to the chosen tree projection method, initialize the global header table.*
3. *With the divide-and-conquer and depth-first searching paradigm, mine* FP-tree *for frequent closed itemsets in a top-down manner for sparse datasets or bottom-up manner for dense datasets. During the mining process, use the item merging, item skipping, and sub-itemset pruning methods to prune search space. For each candidate frequent closed itemset, use the two-level hash indexed result tree method for dense datasets or pseudo-projection based upward checking method for sparse datasets to do closure checking.*
4. *Stop when all the items in the global header table have been mined. The complete set of frequent closed itemsets can be found either from the result tree or the output file F.* ■

5. PERFORMANCE EVALUATION

5.1 Test environment and datasets

In this section we will evaluate CLOSET+ in comparison with three algorithms, OP, CHARM and CLOSET. All the experiments were performed on an IBM ThinkPad R31 with 384 MB memory and Windows XP installed. We used an improved implementation of CLOSET with better performance than that claimed in [14]. The source code of CHARM and the executable of OP were provided by their authors. We ran the four algorithms on the same Cygwin environment and with '-e 1 -d' options turned on for CHARM. Because our performance study showed that both OP and CLOSET cannot compete with CLOSET+, we only compared the peak memory usage between CLOSET+ and CHARM.

We used six real datasets to evaluate the performance and memory usage, and some synthetic datasets to test the scalability by varying database size and the number of distinct items. The characteristics of the datasets are shown in Table 2 (the last column shows the average and maximal transaction length).

Real datasets: Among the six real datasets three are dense and three are sparse (see the distribution of the number of frequent closed itemsets by support threshold in Table 3). The *connect* dataset contains game state information, *mushroom* dataset contains characteristics of various species of mushrooms, *pumsb** contains census data. The *gazelle* dataset contains click-stream data from Gazelle.com. These

Dataset	# Tuples	# Items	A.(M.) t. l.
connect	67557	150	43(43)
pumsb*	49046	2089	50.5(63)
mushroom	8124	120	23(23)
gazelle	59601	498	2.5(267)
retail-chain	106632	22036	2.98(61)
big-market	838466	38336	3.12(90)
T10I4DxP1k	200k-1400k	978	10(31)
T10I4D100kPx	100k	4333-29169	10(31)

Table 2: Dataset Characteristics.

have been used in the previous performance studies [18, 19]. Two other datasets, *retail-chain* and *big-market* are different retail transaction datasets.

Synthetic datasets: The synthetic datasets were generated from IBM dataset generator, with an average transaction length 10 and average frequent itemset length 4. To test the scalability against base size, we generated dataset series T10I4DxP1k by varying the number of transactions from 200K to 1400K and fixing the number of unique items at 1k. To test scalability in terms of number of unique items, we generated dataset series T10I4D100kPx by fixing number of transactions at 100k and setting the number of distinct items at 4333, 13845, 24550, 29169 (generated by setting *nitems* parameter at 5k, 25k, 125k, 625k respectively).

Dataset	Num. of F.C.I. (rel. sup.(%))
connect	24346(75), 94916(55), 328344(35)
pumsb*	2610(40), 16154(30), 122315(20)
mushroom	1197(20), 12855(5), 76199(0.5)
gazelle	808(0.2), 5706(0.09), 20447(0.07)
retail-chain	2788(0.025), 5147(0.015), 17399(0.005)
big-market	14881(0.005), 24478(0.003), 92251(0.001)
T10I4D100k	46993(0.05), 71265(0.03), 283397(0.01)

Table 3: Number of frequent closed itemsets vs. relative support threshold.

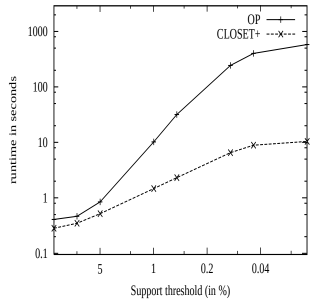

Fig. 6. Runtime (*mushroom*).

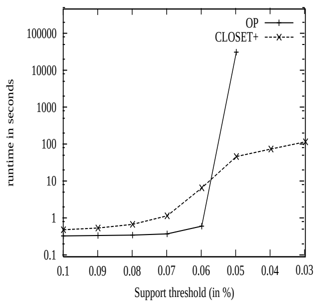

Fig. 7. Runtime (*gazelle*).

5.2 Experimental results

Comparison with OP. Our experiments show that due to generating a huge number of frequent itemsets, even the best frequent itemset mining algorithm like OP cannot compete with CLOSET+. Fig. 6 and Fig. 7 show the experimental results for *mushroom* and *gazelle* datasets.

As we can see in Fig. 6, for dense datasets like *mushroom* CLOSET+ always outperforms OP and when the support threshold is low, CLOSET+ is more than one order of magnitude faster than OP. For sparse datasets like *gazelle* (see Fig. 7), when the support is high there will not be too many frequent itemsets, and due to overhead incurred by closure checking in CLOSET+, OP is a little faster than CLOSET+. But once the support threshold is lowered to a certain point, there will be explosive increase in the number of frequent itemsets (e.g., with support 0.05%, a not too low support threshold for a sparse dataset like *gazelle*, the longest frequent closed itemset has a length 45, from which $2^{45} - 1$ frequent itemsets can be generated), the pruning methods adopted by CLOSET+ will make CLOSET+ orders of magnitude faster than OP.

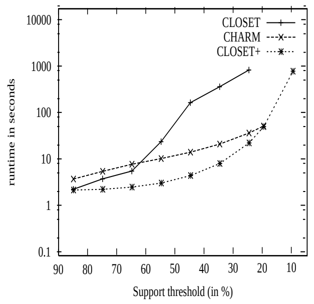

Fig. 8. Runtime performance (*connect*).

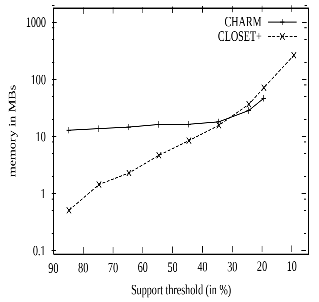

Fig. 9. Memory usage (*connect*).

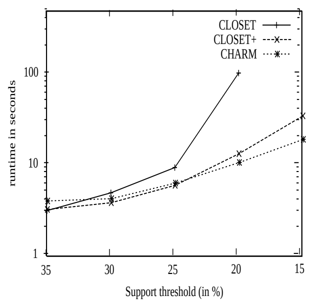

Fig. 10. Runtime performance (*pumsb**).

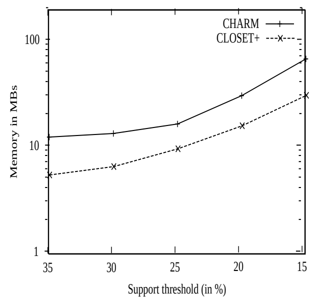

Fig. 11. Memory usage (*pumsb**).

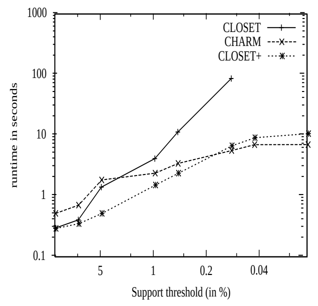

Fig. 12. Runtime performance (*mushroom*).

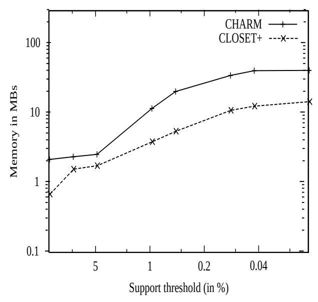

Fig. 13. Memory usage (*mushroom*).

Comparison with CHARM and CLOSET. We used all the 6 real datasets to test CLOSET+'s performance and memory usage in comparison with two other closed itemset mining algorithms: CHARM and CLOSET. For dense dataset *connect*, Fig. 8 and Fig. 9 show the results. Fig. 8 shows CLOSET+ can be orders of magnitude faster than CLOSET. When the support is not too low (i.e., higher than 20%), CLOSET+ is several times faster than CHARM. When support threshold is further lowered, they will have similar performance, but at support 10%, CHARM cannot run by reporting an error ′REALLOC: Not enough core′. From Fig. 9 we know overall CLOSET+ uses less memory than CHARM. For example, at support 85%, CLOSET+ consumes about 1MB while CHARM consumes about 15MB.

*Pumsb** is another dense dataset. Fig. 10 and Fig. 11 depict the results. Both CLOSET+ and CHARM have significantly better performance than CLOSET and once the support is lower than 20%, CLOSET just cannot finish running. Overall CLOSET+ and CHARM have very similar performance when the support threshold is not too low. At low support threshold like 15%, CHARM will outperform CLOSET+. Fig. 11 shows that CLOSET+ uses much less memory than CHARM.

Fig. 12 and Fig. 13 demonstrate the results for *mushroom* dataset. We can see CLOSET can be orders of magnitude slower than CLOSET+ and CHARM, and CLOSET even cannot finish running once the support is less than 0.1%. But there is no clear winner between CLOSET+ and CHARM: At high support threshold, CLOSET+ is several times faster than CHARM; and at very low support threshold, CHARM is a little better than CLOSET+. But CLOSET+ always beats CHARM in terms of memory usage.

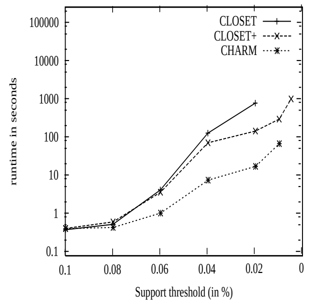

Fig. 14. Runtime performance (*gazelle*).

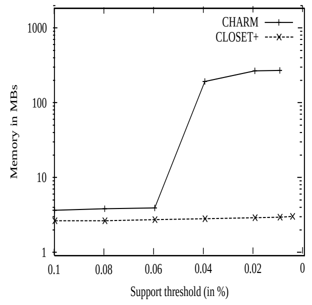

Fig. 15. Memory usage (*gazelle*).

Fig. 14 and Fig. 15 present the evaluation results for sparse dataset *gazelle*. Fig. 14 shows that CLOSET+ and CHARM are faster than CLOSET. At high support CLOSET+ and CHARM have similar performance, and at a little lower support, CHARM is several times faster than CLOSET+, but once we continued lowering the support threshold to 0.005%, CHARM could not run by reporting an error ′REALLOC: Not enough core′. From Fig. 15, we see that CHARM consumes about two orders of magnitude more memory than CLOSET+ at low support.

Fig. 16 and Fig. 17 demonstrate the results for *retail-chain* dataset. CLOSET+ runs the fastest among the three algorithms and uses less memory than CHARM: When support threshold is set to 0.005%, CLOSET+ runs almost 5 times faster than CHARM, but uses only 1/9 of the memory that

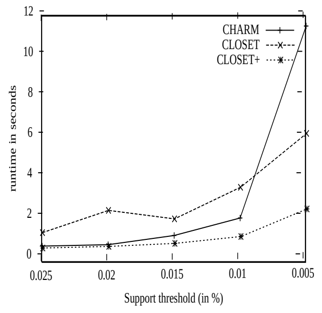

Fig. 16. Runtime performance (*retail-chain*).

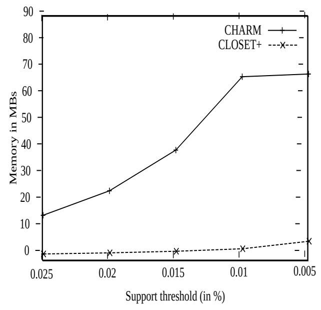

Fig. 17. Memory usage (*retail-chain*).

CHARM consumes.

Fig. 18 and Fig. 19 show the results for the *big-market* real dataset. We can see that CLOSET+ is also the fastest among the three and uses less memory than CHARM. It runs several times faster than CHARM but uses less memory.

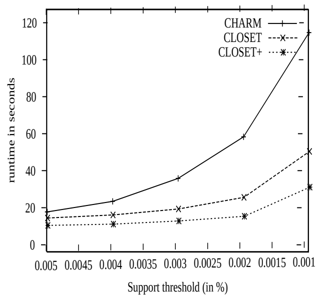

Fig. 18. Runtime performance (*big-market*).

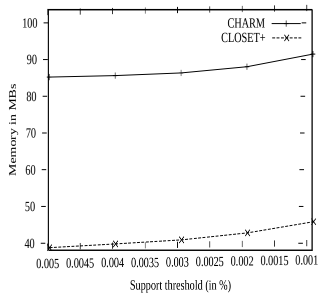

Fig. 19. Memory usage (*big-market*).

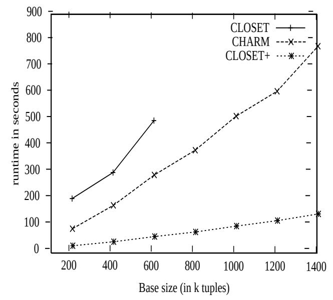

Fig. 20. Scalability test(T10I4DxP1k).

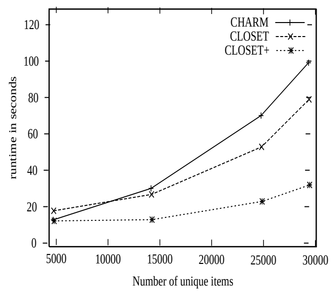

Fig. 21. Scalability test(T10I4D100kPx).

Scalability test. We used the IBM synthetic datasets to test the scalability of CLOSET+ and compared it with both CHARM and CLOSET. We first tested the scalability in terms of database size using the dataset series T10I4DxP1k with base size from 200k tuples to 1400k tuples and support threshold set at 0.005%. From Fig. 20 we can see that, CLOSET has the poorest scalability, and it even cannot run when the dataset contains more than 600K tuples. In comparison with CHARM, CLOSET+ not only runs much faster, it also has much better scalability in terms of base size:

the slope ratio for CHARM is much higher than that for CLOSET+.

We also tested the scalability of CLOSET+ in terms of number of distinct items using T10I4D100KPx series with number of distinct items set at 4333, 13845, 24550 and 29169, respectively, and minimum support set at 0.005%. From Fig. 21, we can see that initially these three algorithms have very similar performance when the number of distinct items is small, but once the number of distinct items increases, the runtime of CHARM and CLOSET will have a much bigger jump than CLOSET+, which means CLOSET+ also has better scalability than both CHARM and CLOSET in terms of the number of distinct items.

The above experimental results show that: (1) Although CHARM adopts the *Diffset* technique which can reduce space usage significantly [18], it still consumes more memory than CLOSET+, and in some cases it can use over an order of magnitude more memory than CLOSET+. (2) Due to the new techniques developed here, such as the *hybrid tree-projection* mining strategy, the *item-skipping* pruning method, and the subset-checking techniques(i.e., the two-level hash-indexed *result-tree* and pseudo-projection based *upward checking*), CLOSET+ can be orders of magnitude faster than CLOSET, and is very efficient with low support even in the case CLOSET and CHARM cannot run. (3) CLOSET+ has linear scalability and is more scalable than CHARM and CLOSET in terms of both base size and the number of distinct items.

6. CONCLUSIONS

Frequent pattern mining has been studied extensively in data mining research. In this study, we have re-examined some previously proposed methodologies, and mainly focused on the new techniques developed for CLOSET+, a highly scalable and both runtime and space efficient algorithm for dense and sparse datasets, on different data distributions and support thresholds.

The thorough performance evaluation in this study reveals that: (1) For mining frequent patterns, one should work on mining *closed patterns* instead of *all patterns* because the former has the same expressive power as the latter but leads to more compact and meaningful results and likely better efficiency. (2) There is a popular myth: Algorithms based on the vertical-format are better than those based on the horizontal-format. Our study shows that an algorithm based on the vertical format, due to its necessity to identify tids (even using the *Diffset* compression technique) will likely take more memory than an FP-tree-based algorithm and is less scalable if the latter is implemented nicely. (3) Multiple, integrated optimization techniques for database projection, search space pruning, and pattern closure-checking are needed for high performance pattern mining. Often, different data characteristics may require different mining methodologies, e.g., in CLOSET+, we use the *top-down pseudo tree-projection* and *upward subset-checking* for sparse datasets, whereas for dense datasets, the *bottom-up physical tree-projection* and a compressed *result-tree* have been adopted.

Currently CLOSET+ has been successfully employed to mine non-redundant association rules. In the future, we will explore more applications, including association-based classification, clustering, and dependency/linkage analysis in large databases.

7. ACKNOWLEDGEMENTS

We are grateful to Dr. Mohammed Zaki for providing us the source code of CHARM and Junqiang Liu for providing us the executable code of the OP algorithm.

8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD'93*, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, Sept. 1994.

[3] R.J. Bayardo. Efficiently Mining long patterns from databases. *SIGMOD'98*, June 1998.

[4] S. Brin, R. Motwani, J.D. Ullman, S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *SIGMOD'97*, May 1997.

[5] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *ICDE'01*, April 2001.

[6] D. Gunopulos, H. Mannila, and S.Saluja. Discovering All Most Specific Sentences by Randomized Algorithms. In *ICDT'97*, Jan. 1997.

[7] E. Han, G. Karypis, V. Kumar. Scalable Parallel Data Mining for Association Rules. In *TKDE 12(2)*, 2000.

[8] J. Han, J. Pei, Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD'00*, May 2000.

[9] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining Top-k frequent closed patterns without minimum support. In *ICDM'02*, Dec. 2002.

[10] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *SIGKDD'02*, July 2002.

[11] J. Park, M. Chen, P.S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *SIGMOD'95*, May 1995.

[12] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99*, Jan. 1999.

[13] 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'01*, Nov. 2001.

[14] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *DMKD'00*, May 2000.

[15] R. Rymon. Search through Systematic Set Enumeration. In *Proc. of 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, 1992.

[16] H. Toivonen. Sampling Large Databases for Association Rules. In *VLDB'96*, Sept. 1996.

[17] M. Zaki. Generating non-redundant association rules. In *SIGKDD'00*, Aug. 2000.

[18] M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SDM'02*, April 2002.

[19] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *SIGKDD'01*, Aug. 2001.