Mining Frequent Closed Itemsets without Duplicates Generation.

Claudio Lucchese
ISTI "A. Faedo", Consiglio Nazionale delle Ricerche (CNR)
Via Moruzzi, 1
56100 Pisa, Italy
claudio.lucchese@isti.cnr.it

Salvatore Orlando
Dipartimento di Informatica, Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia, Italy
orlando@dsi.unive.it

Raffaele Perego
ISTI "A. Faedo", Consiglio Nazionale delle Ricerche (CNR)
Via Moruzzi, 1
56100 Pisa, Italy
raffaele.perego@isti.cnr.it

ABSTRACT

Closed itemsets are semantically equivalent to frequent itemsets but are orders of magnitude fewer, thus allowing the knowledge extracted from a transactional database to be represented very concisely. Unfortunately, no algorithm has been yet devised which allows to mine closed patterns directly. All existing algorithms may in fact generate the same closed itemset multiple times, and need to maintain the closed itemsets mined so far in the main memory in order to check if the current element has been already discovered or not.

In this paper we propose a general technique for promptly detecting and discarding duplicate closed itemsets without the need of keeping in the main memory the whole set of closed patterns. Our approach can be exploited with substantial performance benefits by any depth first algorithms for mining frequent closed itemsets. As a case of study, we implemented our technique within a new mining algorithm which adopts a vertical bitmap representation of the dataset. The experimental evaluation demonstrates that our algorithm outperforms other state of the art algorithms such as CHARM, CLOSET+ and FPCLOSE.

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

1. INTRODUCTION

Frequent itemsets mining is the most important and demanding task in many data mining applications. Let $\mathcal{I} = \{a_1, ..., a_M\}$ be a finite set of items and $\mathcal{D}$ a finite set of transactions (the dataset) where each transaction $t \in \mathcal{D}$ is a list of *distinct* items $t = \{x_0, x_1, ..., x_T\}$, $x_i \in \mathcal{I}$. An ordered sequence of n *distinct* items $I = \{i_0, i_1, ..., i_n\} \mid i_j \in \mathcal{I}$ is called itemset of length n, or n-itemset. The number of transactions in the dataset including an itemset I is defined as the *support* of I (or $supp(I)$). Given a threshold min_supp, an itemset is said to be *frequent* if its support is greater than (or equal to) min_supp, *infrequent* otherwise.

Mining frequent itemsets requires to discover all the itemsets in a given dataset $\mathcal{D}$ which have a support higher (or equal) than a given minimum frequency threshold min_supp. This problem has been extensively studied in the last years. Several variations to the original Apriori algorithm [1], as well as completely different approaches, have been proposed. All the algorithms proposed browse bottom-up the huge solution search space, i.e. the lattice of $2^{\|\mathcal{I}\|}$ itemsets, by exploiting various strategies for pruning it. Among these strategies, the most effective regards the exploitation of the *downward closure* property: if a k-itemset is frequent, then all of its k-1-subsets have to be frequent as well. On the other hand, if a k-itemset is discovered to be infrequent, then all its supersets will not be frequent as well. Thus, when during browsing we discover an infrequent itemset I, we can safely prune I and all its supersets. Moreover, transactions which do not contain at least a frequent itemset can be deleted from the datasets since they cannot contribute to the support of any frequent itemset.

This property makes it possible to effectively mine *sparse* datasets, also with very low min_supp thresholds. Sparse datasets in fact contain transactions with weak correlations, and the downward closure property allows to strongly reduce the search space and the dataset itself as the execution progresses. On the other hand, *dense* datasets contain strongly related transactions, and are much harder to mine since only a few itemsets can be in this case pruned, and the number of frequent itemsets grows very quickly while decreasing of the minimum support threshold. As a consequence, the mining task becomes rapidly intractable by traditional mining algorithms, which try to extract all the frequent itemsets.

Closed itemsets mining is a solution to this problem. Closed itemsets are a small subset of frequent itemsets, but they

represent exactly the same knowledge in a more succinct way. From the set of closed itemsets it is in fact straightforward to derive both the identities and supports of all frequent itemsets. Mining the closed itemsets is thus semantically equivalent to mine all frequent itemsets, but with the great advantage that closed itemsets are orders of magnitude fewer than frequent ones. Using closed itemsets we implicitly benefit from data correlations which allow to strongly reduce problem complexity. Moreover, association rules extracted from closed sets have been proved to be more concise and meaningful, because all redundancies are discarded.

Several algorithms for mining closed itemsets have been proposed in the last years. Unfortunately, no algorithm has been yet devised which allows to mine closed patterns directly. In fact, while a candidate frequent itemset can be always generated as the union of two other frequent itemsets, thus making easy to devise a strategy to visit each node of the frequent itemset lattice just once, the same does not hold for closed frequent itemsets. All existing algorithms may generate duplicate closed itemsets, and need to maintain all the closed itemsets mined so far in the main memory in order to check if the current candidate closed itemset has been already discovered or not.

In this paper we investigate the problem of duplicates. We assert that the duplicates generation problem is strictly related to the strategy adopted to browse the itemset lattice. By reformulating the problem of mining closed itemset as the problem of building a suitable tree over the lattice of frequent itemsets, we propose a general technique for promptly detecting and discarding duplicate closed itemsets without the need of keeping in the main memory the whole set of closed patterns. Our approach can be exploited with substantial performance benefits by all depth-first algorithms for mining frequent closed itemsets. As a case of study, we implemented our technique within an algorithm which adopts a vertical bitmap representation of the dataset. The experimental evaluation demonstrates that our algorithm outperforms other state of the art algorithms such as CHARM,CLOSET+ and FPCLOSE.

The paper is organized as follows. In Section 2 we introduce closed itemsets and we describe a framework for mining them. This framework is shared by all the algorithms surveyed in Section 3. In Section 4 we formalize the problem of duplicates and propose our technique. Section 5 proposes an implementation of our technique and discusses the experimental results obtained. Follows some concluding remarks.

2. CLOSED ITEMSETS

Given the functions:

$$f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\},$$

which returns all the itemset included in the set of transactions T, and

$$g(I) = \{t \in \mathcal{T} \mid \forall i \in I, i \in t\}$$

which returns the set of transactions supporting a given itemset I (its *tid-list*), the composite function $f \circ g$ is called *Galois operator* or *closure operator*.

We have the following definition:

Definition 1. An itemset I is said to be closed if and only if

$$c(I) = f(g(I)) = f \circ g(I) = I$$

Roughly speaking we can say that an itemset I is closed if and only if no superset with the same support exists. It is intuitive that the closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belongs to the same equivalence class if and only if they have the same closure, i.e. their support is the same and is given by the same set of transactions.

From the above definitions, the relationship between equivalence classes and closed itemsets is clear: the maximal itemsets of all equivalence classes are closed itemsets. Mining all these maximal elements means mining all closed itemsets.

Figure 1 shows the lattice of frequent itemsets derived from the simple dataset reported in Table 1, mined with $min\_supp = 1$. We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that the number of closed itemsets (five) is much lesser than the number of frequent itemsets (fifteen).

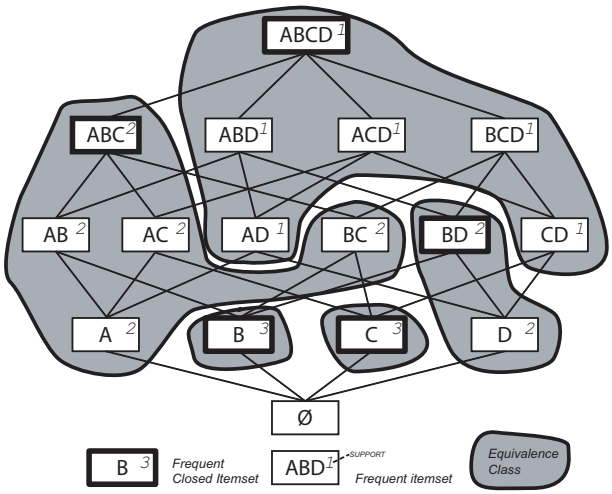

Figure 1: Lattice of frequent itemsets with closed itemsets and equivalence classes

Table 1: Simple Dataset

TID	items			
1	B	D		
2	A	B	C	D
3	A	B	C	
4	C			

Corollary 1. Given an itemset X and an item i, $g(X) \subseteq g(i) \Rightarrow i \in c(X)$.

PROOF. Since $g(X \cup i) = g(X) \cap g(i)$, $g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X)$. Therefore, if $g(X \cup i) = g(X)$ then $f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X)$ □

Given an itemset X, by exploiting Corollary 1 we can determine whether an item i belongs to $c(X)$ or not. This corollary is used by most algorithms to calculate closures incrementally. Another important corollary is the following:

Corollary 2. Given two itemsets X and Y, if $X \subset Y$ and $supp(X) = supp(Y)$ (i.e. $|g(X)| = |g(Y)|$), then $c(X) = c(Y)$.

PROOF. If $X \subset Y$, then $g(Y) \subseteq g(X)$. Since $|g(Y)| = |g(X)|$ then $g(Y) = g(X)$. $g(X) = g(Y) \Rightarrow f(g(X)) = f(g(Y)) \Rightarrow c(X) = c(Y)$. □

We will see that one of the problem arising during the mining of closed itemsets is the detection of *duplicates*, where we define as duplicate any itemset belonging to the equivalence class of an already discovered closed itemset. Corollary 2 can thus be used to check if itemset X is a duplicate of some already mined closed itemset Y.

The algorithms for mining frequent itemsets adopt two different strategies for their bottom-up visit of the itemset lattice: depth-first and breadth-first. When a depth-first strategy is adopted, given a frequent k-itemset X, candidate *(k+1)*-itemsets are generated by adding an item $i, i \notin X$, to X. If also $\{X \cup i\}$ is discovered to be frequent, the process is recursively repeated on $\{X \cup i\}$. Conversely, by adopting a breadth-first (or level-wise) strategy, all candidate k-itemsets are generated with set unions of two frequent *(k-1)*-itemsets.

When the mining process regards the extraction of closed itemsets only, similar generation and check strategies cannot be directly applied. in fact, it is not possible to generate candidate closed itemset starting from previously computed closed itemsets. For example, given a closed itemset X, it may happen that no one of its next supersets obtained by extending X with one single item, is closed. From Figure 1 we can see that given the closed itemset $\{BD\}$, both $\{ABD\}$ and $\{BCD\}$ are not closed. On the other hand, from the above definitions, we are sure that each superset of a closed itemset belongs to another equivalence class.

Since closed candidates cannot be generated directly, all the algorithms for mining frequent closed itemsets are based on two steps. They *browse* the search space by traversing the lattice of frequent itemsets from an equivalence class to another, and they calculate the *closure* of the frequent itemsets visited in order to determine closed itemsets of the corresponding equivalence class.

2.1 Browsing

The goal of an effective browsing strategy is to find a tree over the lattice of frequent itemsets, visiting exactly a single itemset in every equivalence class. Let us call the itemsets visited closure *generators*. We can thus identify all the closed itemsets by calculating the closure of each generator.

Some algorithms choose the minimal elements (or *key patterns*) of each equivalence class as closure generator. Key patterns form a lattice, and this lattice can be traversed with a simple apriori-like algorithm.

Other algorithms use instead a different technique that we call *closure climbing*. Once a generator is found, we calculate its closure, and then create new generators from the closed itemset discovered. In this way the generators are not necessarily minimal elements, and the computation of their closure is faster.

Regardless of the strategy adopted, some kind of duplicate check has to be introduced. Both the key patterns and the closure climbing strategies visit more than one generator for each equivalence class. For example, considering the key pattern strategy, the closed itemset $\{ABCD\}$ of Figure 1 will be mined twice, i.e., when the closures of both the minimal elements of its equivalence class ($\{AD\}$ and $\{CD\}$) are computed.

2.2 Closure

To calculate the closure of an itemset X, we have to apply the Galois operator c. Applying c requires to intersect all the transactions of the dataset including X. Another way to calculate the same closure is finding *incrementally* all the items in $\mathcal{I}$ belonging to $c(X)$. From Corollary 1, we know that if $g(X) \subseteq g(i)$, then $i \in c(X)$. Therefore, by performing the inclusion check on all the items in $\mathcal{I}$ we can incrementally compute $c(X)$.

The closure calculation can be performed off-line or on-line. In the off-line case we can imagine to firstly retrieve the complete set of generators, and then to calculate their closure. In the second case we can calculate closures during the browsing: each time a new generator is mined, its closure is immediately computed.

The algorithms which compute closures on-line are generally more efficient than off-line ones. This is because they can adopt the closure climbing strategy, where new generators are created recursively from closed itemsets (see Section 2.1). These generators are likely to be longer than key patterns. Obviously the longer the generator is, the fewer checks (on further items) are needed to get its closure.

2.3 Duplicates

In the absence of an optimal browsing strategy, different paths can lead to the same closed itemset. Each time a generator X is built, instead of computing its closure and checking if the resulting closed itemset has been already discovered or not, most algorithms look for a closed itemsets Y which includes the generator, and has exactly the same support. If such closed itemset Y is found, from Corollary 2 we can say that $c(X) = c(Y)$, and thus we can prune the generator. Duplicate check is however very expensive in both time and space. In time because we may need to search among a huge number closed itemsets, in space because to perform searches we need to keep them in the main memory. To reduce such costs, closed sets are usually stored in compact prefix tree structures indexed by one or more levels of hashing. In this paper, we propose an effective and efficient technique for detecting duplicates which does not require to keep the closed itemsets mined in the main memory, thus reducing the cost both in space and time.

3. RELATED WORKS

A-CLOSE [6] (N. Pasquier et al.) was the first algorithm for closed itemset mining. During its first step, A-CLOSE browses level-wise the frequent itemsets lattice by means of Apriori-like strategy, and prunes the non-minimal elements of each equivalence class. Since a k-itemset X is a key pattern if and only if no one of its $(k-1)$-subsets has the same support, such key pattern itemsets are detected with intensive subset checking. In the second step, A-CLOSE calculates the closure of all the minimal generators previously found. Since a single equivalence class may have more than one minimal itemsets, redundant closures may be computed. A-CLOSE performance suffer from the high cost of the off-line closure calculation and of the huge number of subset searches.

The authors of FP-Growth [3] (J. Han et al.) proposed CLOSET [7] and CLOSET+ [8]. These two algorithms inherit from FP-Growth the compact FP-Tree data structure and the exploration technique based on recursive conditional projections of the FP-Tree. Frequent single items are detected after a first scan of the dataset, and with another scan the pruned transactions are inserted in the FP-Tree stored in the main memory. With a depth first browsing of the FP-Tree and recursive conditional FP-Tree projections, CLOSET mines closed itemsets by closure climbing, growing up frequent closed itemsets with items having the same support in the conditional dataset. Duplicates are discovered with subset checking by exploiting Corollary 2. All closed sets previously discovered have thus to be kept in the main memory. To allow fast retrieval of closed itemsets, they are indexed by a two level hash. CLOSET+ is similar to CLOSET, but exploits an adaptive behavior in the tree exploration in order to fit both sparse and dense datasets.

CHARM [10] (M. Zaki et al.) performs a bottom-up depth-first browsing of a prefix tree of frequent itemsets built incrementally. As soon as a frequent itemset is generated, its tid-list is compared with those of the other itemsets having the same parent. When two tid-lists are equal, or one includes the other, the associated nodes are merged since the itemsets surely belong to the same equivalence class. Itemset tid-lists are stored in each node of the tree by using the diff-set technique [9]. Since different paths can however lead to the same closed itemset, also in this case a duplicates pruning strategy is implemented. CHARM adopts a technique similar to that of CLOSET, by storing in the main memory the closed itemsets indexed by a hash table.

According to our classification, A-CLOSE exploits a key pattern browsing and off-line closure calculation, meanwhile CHARM and CLOSET+ are different implementations of the same closure climbing browsing and incremental closure algorithm.

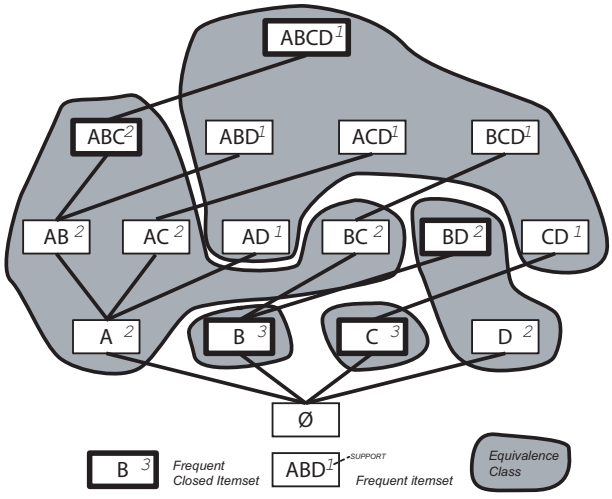

Figure 2: Spanning tree over the lattice of frequent itemsets

4. DETECTING DUPLICATES BY VISITING A SPANNING TREE

We have seen that the strategy adopted to browse the lattice of frequent itemsets is very important for the effectiveness of the algorithms for mining frequent closed itemsets. In fact, since generic equivalence classes members instead of closed ones may be mined, the browsing strategy adopted can visit for each equivalence class multiple generators which have to be pruned in order to avoid to perform unuseful closure computations. Moreover, the pruning of the generators, as performed by CHARM and CLOSET+, is very expensive in both time and space.

As Figure 2 shows, we can easily devise a spanning tree over the lattice of frequent itemsets inducing a lexicographic order on frequent itemsets with a left-to-right depth-first visit. By considering the lexicographic order $\prec$, we can see that such spanning tree exactly corresponds to a prefix tree, since from each itemset X we can reach only all the itemsets having X as a prefix.

When frequent itemsets are mined, the ordering defined by such visiting strategy guarantees that each candidate is generated only once. Unfortunately, the same does not hold when closed itemsets have to be extracted. By looking at the example in Figure 2, we can see that both A and B subtrees cross the same equivalence class given by the closed itemset $\{ABCD\}$. Moreover, duplicates are generated whenever the traversal order is violated by itemset closures. For example in Figure 2 we have that $c(AC) = \{ABC\}$, but $c(AC) \not\prec \{ABC\}$, and similarly $c(BCD) = \{ABCD\}$, but $c(CBD) \not\prec \{ABCD\}$. Therefore, each time we calculate the closure of the currently reached itemset, we may come back to some already mined branch of the tree. Then we can assert that:

Claim 1. The closure $c(X)$ of an itemset X does not respect the ordering $\prec$, i.e. there may exists an itemset X such that $c(X) \prec X$, where $\prec$ is the lexicographical ordering among the literals of the itemsets.

This claim explains why and where we generate duplicates. During our traversal of the lattice, we generate duplicates because the closure operator c is not order preserving with respect to the ordering $\prec$. For this reason, each time we apply the closure operator, we can break the visiting order and reach nodes of the lattice we have already visited. If the closure operator was order preserving, we would not have duplicates.

Let us introduce the following definition:

Definition 2. Given the lexicographical ordering $\prec$ between the literals of the itemsets in a lattice of frequent itemsets, an itemset X is *order preserving* iff $X \prec c(X)$.

The algorithms adopting a left-to-right depth-first visit of the lattice, such as CHARM and CLOSET+, thus generate a duplicate each time a *not order preserving* generator is reached during the visit. If we could easily check if an itemsets X is order preserving or not, we could easily discriminate between *bad* generators leading to duplicates, and *good* generators leading to new closed itemsets. In some sense CHARM and CLOSET+ actually perform this check by searching for a superset of X having the same support among all the mined closed itemsets: if this check is successful then the itemsets is clearly not order preserving. Unfortunately this duplicate checking strategy requires to maintain in the main memory the whole set of closed itemsets mined so far.

We can however demonstrate the following:

Theorem 1. Given an itemset X, if an item i, $i \in c(X)$, exists such as $\{X \cup i\} \prec X$, then X is *not order preserving.*

PROOF. We adopt a left-to-right depth-first visit of the lattice of frequent itemsets. Thus we can assume that when X is reached, all the itemsets Y such that $Y \prec X$ have been already visited. Since $\{X \cup i\} \prec X$, all closed itemsets including $\{X \cup i\}$ have been already mined, and therefore also $c(X) \supseteq \{X \cup i\}$. By finding such i, we can say that $X \neq c(X)$ and that $c(X)$ has already been mined. □

We have thus demonstrated that the order preserving check for a generator X can be performed by comparing the tid-list of the single item i with the tid-list of X, and by applying Corollary 1. We can say that if $g(X) \subseteq g(i)$ then $i \in c(X)$. If during the traversal a not order preserving itemset is reached, then it can be safely discarded, because its closure was surely previously mined.

By exploiting this simple technique, in the example reported in Figure 2, we detect $\{AC\}$ as a duplicate because item B is such that $\{\{AC\} \cup \{B\}\} = \{ABC\} \prec \{AC\}$ and $B \in c(AC) = \{ABC\}$. The same holds for BCD, since the item A is such that $\{\{BCD\} \cup \{A\}\} = \{ABCD\} \prec \{BCD\}$ and $A \in c(BCD) = \{ABCD\}$.

The above Theorem provides a new mean to face the problem of duplicates in mining frequent closed itemsets. By reformulating the mining problem as a spanning tree problem, we have moved the dependencies of the order preserving check from the set of closed itemsets mined so far to the set of single items. In this way, duplicates detection and pruning is no more resource demanding, because frequent closed itemsets need not to be stored in the main memory during the computation, and it is less time demanding because the check of the order preservance property is cheaper than the search between closed itemsets already mined.

5. IMPLEMENTATION: DCI-CLOSED

In order to test the effectiveness of our duplicate discarding technique, we designed and implemented DCI-CLOSED, a new algorithm for mining closed itemsets. DCI-CLOSED inherits the internal representation of our previous works DCI[5] and kDCI[4].

5.1 Internal representation

The dataset is stored in the main memory by using a vertical bitmap representation [4][5]. After a first scan of the dataset, the frequent items F_1 are detected. During a second scan all transactions are pruned from infrequent items and stored in a bitmap matrix $D_{M \times N}$, where $M = |F_1|$ and N is the number of transactions. The $D(i, j)$ bit is set to 1 if and only if the j-th transaction contains the i-th frequent single item. Row i of the matrix thus represent the tid list of item i.

The columns of D are then reordered to profit of data correlations. As in [4][5] columns are reordered to create a submatrix E of D having all its rows identical. Every operation involving rows in the submatrix E will be performed only once, thus gaining strong performance improvements.

Tid lists of each itemset are calculated by intersecting the tid lists of the single items included in the itemset with bitwise AND operations. The depth-first exploration guarantees that only one intersection is performed for any new itemset. Once the itemset tid list is computed, its support is calculated by counting the 1s in the resulting tid list.

Also checking whether an item i belongs to some closure $c(X)$ or not, i.e. if $g(X) \subseteq g(i)$, can be easily checked. It is sufficient to compare, byte by byte, the inclusion of the tid list of the single item in the tid list of the itemset. As soon as the comparison fails we can stop since the item will be surely not contained in the closure.

5.2 Browsing and closures

The lattice of frequent itemsets is browsed left-to-right in a depth-first way by adopting the closure climbing strategy. The first closed itemset discovered is $c(\emptyset)$ at the bottom of the lattice. The calculation of $c(\emptyset)$ is straightforward because it only contains those items having support equal to the number of transactions in the dataset. Afterward, $c(\emptyset)$ is used as a starting point for the usual two-step mining recursive process.

First Step: Generation. Given a closed set X, a set of generators $\{X \cup i\}$ is created. Since the closed itemset X is the maximal element of its equivalence class, every itemset $\{X \cup i\}$ belongs to a different class from the class of X, thus allowing to discover new closed itemsets. At the first level of the lattice all the single frequent items not included in $c(\emptyset)$ can be used as generators. For all other levels, since X is in the form $c(\{X \cup i\})$ (see the second step), only item j,$i \prec j$ are used, according to the order constraints.

Second Step: Closure Climbing. To calculate the closure of the generator itemset $\{X \cup i\}$, the incremental closure technique is adopted. Its tid-list is compared with the tid-list of every single item j,$i \prec j$. If $g(\{X \cup i\}) \subseteq g(j)$ then, applying Corollary 1, we can conclude that $j \in c(\{X \cup i\})$. Once $c(\{X \cup i\})$ is eventually found, the first step is then applied recursively on $c(\{X \cup i\})$.

In this step an effective optimization is applied. Instead of using tid-list $g(j)$, associated with a single item j, in order to extend $\{X \cup i\}$, we equivalently exploit tid-list $g(\{X \cup j\})$. The tid-list $g(\{X \cup j\})$ is more sparse than $g(j)$, and therefore, the inclusion check has a greater chance to fail after a few operations. Note that $\{X \cup j\}$ is one of the generators built from X in the previous step, for which we already computed the associated tid-list $g(\{X \cup j\})$. Moreover, we avoid considering j to extend $\{X \cup i\}$ if the generator $\{X \cup j\}$ actually results to be infrequent.

To exploit our duplicate detection strategy, a pruning step is introduced between the two main steps to discard *not order preserving* generators.

Order Check Step. The tid list of $\{X \cup i\}$ is compared with the tid list of single items j, $j \prec i$. If $g(\{X \cup i\}) \subseteq g(j)$, then $\{X \cup i\}$ is pruned, and its closure is not calculated in the next step, because it surely leads to some already discovered closed itemset.

A working example of DCI-CLOSED. For this working example, we will use Figure 2 to illustrate step-by-step the recursive process of the algorithm aimed to find generators, prune them, and climbing the relative closures/closed itemsets. The main steps of our algorithm in this case are the following:

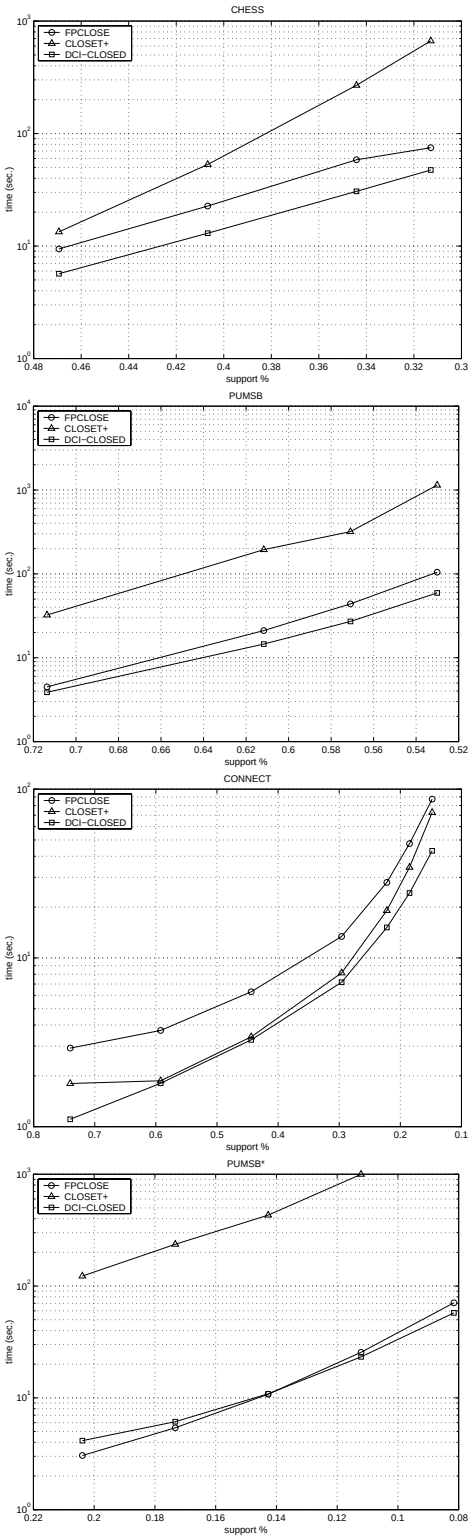

Figure 3: Execution times of FPCLOSE, CLOSET+, and DCI-CLOSET as a function of the support threshold on various publicly available datasets.

1. We first generate the four generators $\{A\}$, $\{B\}$, $\{C\}$, and $\{D\}$ from the empty set. Then the algorithm recursively explores the tree routed on them.

2. $\{A\}$ cannot of course be pruned, because no other smaller items exist according to the lexicographic order $\prec$. During the closure climbing step, its closure $\{\mathbf{ABC}\}$ is calculated.

3. From the closed itemset $\{ABC\}$, the only possible generator we can derive is $\{ABCD\}$. It cannot be pruned nor extended, since no further items exist in the database. Thus $\{\mathbf{ABCD}\}$ is also a closed itemset.

4. According to the recursive behavior, we go to consider the other generator $\{B\}$ found at step 1. $\{B\}$ is "order preserving", and thus is not pruned since $g(\{B\}) \not\subseteq g(\{A\})$. During closure climbing, $\{\mathbf{B}\}$ itself is recognized as a closed itemset. From $\{B\}$ we build the new generators $\{BC\}$ and $\{BD\}$.

5. The generator $\{BC\}$ is pruned, since $g(\{BC\}) \subseteq g(\{A\})$.

6. The generator $\{BD\}$ is not pruned, since $g(\{BD\}) \not\subseteq g(\{A\})$ and $g(\{BD\}) \not\subseteq g(\{C\})$. Moreover, during closure climbing, $\{\mathbf{BD}\}$ itself is recognized as a closed itemset, but can not be further extended according to the lexicographic order $\prec$ to build other generators.

7. According to the recursive behavior, we go to consider the other generator $\{C\}$ found at step 1. $\{C\}$ is "order preserving", and thus is not pruned since $g(\{C\}) \not\subseteq g(\{A\})$, and $g(\{C\}) \not\subseteq g(\{B\})$. During closure climbing, $\{\mathbf{C}\}$ itself is recognized as a closed itemset. From $\{C\}$ we can only build the new generator $\{CD\}$.

8. The generator $\{CD\}$ is pruned, since $g(\{CD\}) \subseteq g(\{A\})$.

9. According to the recursive behavior, we go to consider the last generator $\{D\}$ found at step 1. $\{D\}$ is "not order preserving" and is pruned, since $g(\{D\}) \subseteq g(\{B\})$.

10. Finally, the visit of the tree is completed.

5.3 Experimental results

We tested our implementation on a suite of publicly available dense datasets (chess, connect, pumsb, pumsb*), and compared the performances with those of two well known state of the art algorithms: FPCLOSE, and CLOSET+. FPCLOSE [2] (which is a variant of CLOSET+) resulted to be the best algorithm for closed itemsets mining presented at the ICDM 2003 Frequent Itemset Mining Implementations Workshop, while CLOSET+ was kindly provided us from the authors. FPCLOSE was already proved to outperform CHARM in every dataset, so CHARM was not used in the tests. The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory. The algorithms FPCLOSE and DCI-CLOSED were compiled in the cygwin environment, while the binary executable of CLOSET+ were provided us by the authors.

As shown in Figure 3, DCI-CLOSED outperforms both algorithms in all the tests conducted with low support thresholds. CLOSET+ performs quite well on the connect dataset, and FPCLOSE on pumsb*, but DCI-CLOSED is at least

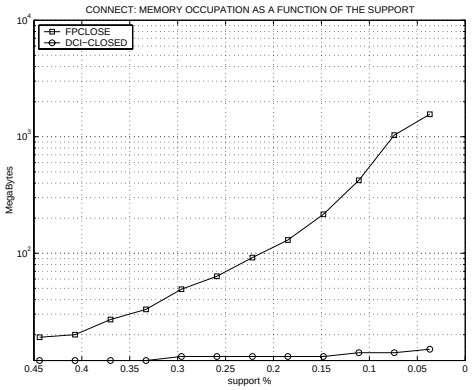

Figure 4: Memory occupation of FPCLOSE and DCI-CLOSED on the connect dataset as a function of the minimum support.

twice faster than its competitors when mining with the lowest support thresholds considered. It is worth noting that in order to fairly compare the implementations, the performance results reported here refer to tests conducted with support thresholds that allowed FPCLOSE, and CLOSET+ data structures to entirely fit in the main memory of the PC used.

In fact, the other important advantage of using our duplicate detection strategy is the low memory requirement. Figure 4 plots memory occupation of DCI-CLOSED and FPCLOSE when mining the connect dataset as a function of the minimum support threshold. As it can be seen, the amount of memory required by our implementation, and consequently by any other implementation based on our duplicate discarding technique, is constant and independent of the size of the output. On the other hand, the amount of memory required by FPCLOSE, and consequently by any other implementation which need to keep in the main memory the whole set of closed itemsets, depends on the size of the output and thus on the support threshold. The size of the output can be in fact considered a lower bound to the amount of memory required by these algorithms. Our technique does not require the set of closed itemsets to be kept in the main memory, and this makes the space complexity of our implementation independent of the size of the output.

The tid-lists associated with each generator itemset in the currently visited subtree, and the tid-lists of all the frequent single items are the only information kept in the main memory. The worst case is obtained when the number of generators and frequent single items is maximal. This occurs when $c(\emptyset) = \emptyset$, $min\_supp = 1$, and every frequent itemset is also a closed itemset. In this case, if N is the number of frequent items, we will have that the most crowded subtree is the leftmost one, i.e. the subtree rooted in $\emptyset$ with N child nodes, with the leftmost node in the first level having $N-1$, with the leftmost node in the second level having $N-2$ child nodes, and so on. The total number of nodes is thus $N^2/2$. Since the length of a tid list is equal to the number of transactions T, the space complexity of our implementation is:

$$S(DCI\_CLOSED) = O\left(\frac{N^2}{2} \times T\right)$$

6. CONCLUSIONS

We have proposed a new technique for promptly detecting and discarding duplicates in closed itemsets mining, without the need of keeping in the main memory the whole set of closed patterns, as done by state of the art algorithms. Our approach can be exploited with substantial performance benefits by any depth-first algorithm for mining frequent closed itemsets. To validate the claim we have implemented our technique within a new mining algorithm. The experimental evaluation demonstrated that our approach is very effective. The proposed implementation outperformed CHARM, FPCLOSE, and CLOSET+ in almost all the test conducted.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB '94*, pages 487–499, September 1994.

[2] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD '00*, pages 1–12, 2000.

[4] C. Lucchese, S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[5] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM02)*, page 338345, 2002.

[6] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT '99*, pages 398–416, 1999.

[7] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD International Workshop on Data Mining and Knowledge Discovery*, May 2000.

[8] J. Pei, J. Han, and J. Wang. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD '03*, August 2003.

[9] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Technical Report 01-1, Computer Science Dept., Rensselaer Polytechnic Institute*, March 2001.

[10] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemsets mining. In *2nd SIAM International Conference on Data Mining*, April 2002.