Fast and Memory Efficient Mining of Frequent Closed Itemsets

Claudio Lucchese[1,2], Salvatore Orlando[1], Raffaele Perego[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Venezia, Italy, `orlando@dsi.unive.it`

[2] ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy, {`r.perego,c.lucchese`}`@isti.cnr.it`

Technical Report CS-2004-9, Nov 2004

Abstract

This paper presents a new scalable algorithm for discovering closed frequent itemsets, which are a lossless, condensed representation of all the frequent itemsets and associated supports that can be mined from a transactional database.

Our algorithm exploits a depth-first visit of the search space, and a bitwise vertical representation of the database. It adopts several innovative optimizations aimed to save both space and time in computing itemset closures and their supports. Moreover, since one of the main problems raising up in this type of algorithms is the multiple generation of the same closed itemset, we have also devised a general technique for promptly detecting and discarding generators of duplicated closed itemsets. Our technique is memory-efficient, since, unlike other previously proposed ones, does not need to keep in the main memory the whole set of closed patterns mined so far.

The tests conducted on a large number of publicly available datasets show that our algorithm outperforms other state-of-the-art algorithms like Closet+ and FP-Close, in some cases by more than one order of magnitude. More importantly, the performance improvements become more and more significant as the support threshold is decreased.

Keywords: Data Mining, Association rules, Frequent itemsets, Equivalence classes, Closed itemsets.

1 Introduction

Frequent Itemsets Mining (FIM) is a demanding task common to several important data mining applications that looks for interesting patterns within databases (e.g., association rules, correlations, sequences, episodes, classifiers, clusters). The problem can be stated as follows. Let $\mathcal{I} = \{a_1, ..., a_M\}$ be a finite set of items, and $\mathcal{D}$ a dataset containing a finite set of transactions, where each transaction $t \in \mathcal{D}$ is a list of *distinct* items $t = \{i_0, i_1, ..., i_T\}$, $i_j \in \mathcal{I}$. We call k-itemset a sequence of k *distinct* items $I = \{i_0, i_1, ..., i_k\} \mid i_j \in \mathcal{I}$. Given a k-itemset I, let $supp(I)$ be its *support*, defined as the number of transactions in $\mathcal{D}$ that include I. Mining all the frequent itemsets from $\mathcal{D}$ requires to discover all the itemsets having a support higher than (or equal to) a given threshold *min_supp*. This requires to browse the huge search space given by the power set of $\mathcal{I}$.

The FIM 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 [10, 5, 13, 18, 2, 16, 6, 8, 3]. The Apriori *downward closure* property makes it possible to effectively mine *sparse* datasets, also for very low *min_supp* thresholds. Sparse datasets contain in fact transactions with weak correlations, and the search space can be effectively pruned by exploiting the antimonotonicity of the support constraint. On the other hand, *dense* datasets contain strongly correlated transactions, and are much harder to mine since pruning is less effective, and the number of frequent itemsets grows very quickly as the minimum support threshold is decreased. As a consequence, the complexity of the mining task becomes rapidly intractable by using traditional mining algorithms. Moreover, the huge size of the output makes hard the task of the analyst, who has to extract useful knowledge from a very large amount of frequent patterns.

Closed itemsets are a solution to these problems: when a dataset contains highly correlated transactions, closed itemsets are orders of magnitude fewer than frequent itemsets, since they implicitly benefit from data correlations. Furthermore, they concisely represent exactly the same knowledge. From closed itemsets it is in fact trivial to generate all the frequent itemsets along with their supports. More importantly, association rules extracted from closed sets have been proved to be more meaningful for analysts, because all redundancies are discarded [17]. Some efficient algorithms for mining closed itemsets have been recently proposed [14, 15, 4, 12, 19, 17]. Unfortunately, no algorithm able to directly mine closed itemsets only has been yet devised, but all of them perform unuseful or redundant computations in order to determine whether a given frequent itemset is closed or not, and, if so, whether it was already previously discovered or not. In most cases these algorithms are not memory-efficient, since they require to maintain all the closed itemsets mined so far in the main memory in order to avoid generating *duplicates*, i.e., to check whether the closure of a given itemset yields an already mined closed itemset.

In this paper we investigate in depth the problem of the generation of duplicated closed itemsets. We claim that this problem is a consequence of the strategy adopted by current algorithms to browse the lattice of frequent itemsets, and we propose an innovative strategy, based on a lexicographically ordered visit of the lattice, which entails a general technique for promptly detecting and discarding duplicates. Our technique is highly efficient, and does not require to keep in the main memory the whole set of closed patterns mined so far. It can be exploited with substantial performance benefits by all algorithms using a vertical representation of the dataset.

We implemented our technique within DCI_Closed, a new algorithm which exploits a depth-first visit of the search space, and adopts a vertical bitmap representation of the dataset. DCI_Closed inherits from DCI [8, 7] – an efficient algorithm to mine frequent itemsets previously proposed – the in-core vertical bitwise representation of the dataset, and several optimization heuristics. In addition we devised innovative techniques specifically for DCI_Closed, aimed at saving both space and time in computing itemset closures and their supports. In particular, since the basic operations to perform closures, support counts, and duplicate detections, are intersections of *tidlists*, i.e., lists of identifiers of the transactions which contain a given item, we particularly optimized this operation, and, when possible, we reused previously computed intersections to avoid redundant computations.

The experimental evaluation demonstrates that DCI_Closed remarkably outperforms other state-of-the-art algorithms such as Closet+ and FP-Close, and that the performance advantage – up to one or two orders of magnitude – becomes more and more significant as the support threshold decreases.

The paper is organized as follows. In Section 2 we introduce closed itemsets and 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 describes the implementation of our closed itemset mining algorithm, while Section 6 discusses the experimental results obtained. Follow some concluding remarks.

2 Closed itemsets

Let T and I, $T \subseteq \mathcal{D}$ and $I \subseteq \mathcal{I}$, be subsets of all the transactions and items appearing in $\mathcal{D}$, respectively. The concept of closed itemset is based on the two following functions f and g:

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

Function f returns the set of itemsets included in all the transactions belonging to T, while function g returns the set of transactions supporting a given itemset I.

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

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

where the composite function $c = f \circ g$ is called Galois operator *or* closure operator*.*

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class *iff* they have the same closure, i.e. they are supported by the same set of transactions. We can also show that an itemset I is closed if no superset of I with the same support exists. Therefore mining the *maximal* elements of all the equivalence classes corresponds to mining all the closed itemsets.

Figure 1.(a) shows the lattice of frequent itemsets derived from the simple dataset reported in Fig. 1.(b), 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 closed itemsets (six) are remarkably less than frequent itemsets (sixteen).

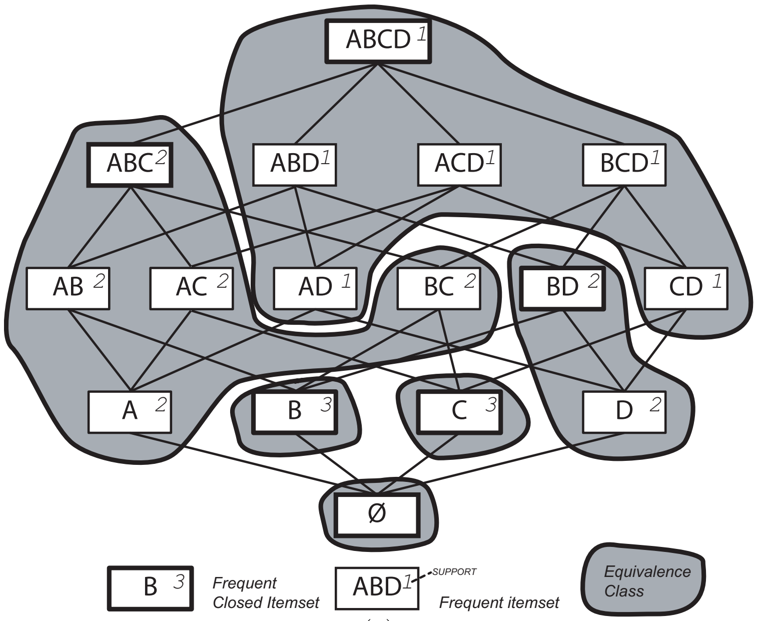
All the algorithms for mining frequent closed itemsets adopt a strategy based on two main steps: *Search space browsing*, and *Closure computation*. In fact, they *browse* the search space by traversing the lattice of frequent itemsets from an equivalence class to another, and compute the *closure* of the frequent itemsets visited in order to determine the maximal elements (closed itemsets) of the corresponding equivalence classes. Let us analyze in some depth these two phases.

Browsing the search space.

The goal of an effective browsing strategy should be to devise a particular visit of the lattice of frequent itemsets, able to exactly touch a single itemset for each equivalence class. We could in fact mine all the closed itemsets by computing the closure of just a representative itemset for each equivalence class, without generating any duplicates. Let us call these representative itemsets *closure generators*.

Some algorithms choose the minimal elements (or *key patterns*) of each equivalence class as generators. Key patterns form a lattice, and this lattice can be easily traversed with a simple apriori-like algorithm [16]. Unfortunately, an equivalence class can have more than one minimal element leading to the same closed itemset. For example, the closed itemset $\{ABCD\}$ of Fig. 1 could be mined twice, since it can be obtained as closure of the two minimal elements of its equivalence class, namely $\{AD\}$ and $\{CD\}$.

Other algorithms use a different technique, which we call *closure climbing*. As soon as a generator is devised, its closure is computed, and new generators are built as supersets of the closed itemset discovered so far. Since closed itemsets are maximal elements of their own equivalence classes, this strategy always guarantees to jump from an equivalence class to another. Unfortunately, it does not ensure that a new generator belong to an equivalence class that was never visited. Hence, similarly to the approach based on key patterns, we can visit multiple generators of the same closed itemset. For example, $\{AB\}$ and $\{AC\}$ are



(a)

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

(b)

Figure 1: (a) Lattice of frequent itemsets with closed itemsets and equivalence classes given by the dataset (b).

both generators of the closed itemset $\{ABC\}$, since they can be obtained as supersets of the closed itemsets $\{B\}$ and $\{C\}$, respectively.

Hence, regardless of the strategy adopted, we need to introduce a duplicate check to avoid generating multiple times the same closed itemset. A naive approach to check for duplicates is to search for each generated closed itemset among all the others mined so far. Indeed, in order to avoid to perform a lot of expensive closure operations, several algorithms exploit the following lemma:

Lemma 1 *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 $g(Y) = g(X)$. $g(X) = g(Y) \Rightarrow f(g(X)) = f(g(Y)) \Rightarrow c(X) = c(Y)$.
□

Therefore, given a generator X, if we find an already mined closed itemsets Y that set-includes X, where the supports of Y and X are identical, we can conclude that $c(X) = c(Y)$. Hence we can prune the generator X without computing its closure. Unfortunately this may become expensive, both in time and space. In time because it requires the possibly huge set of closed itemsets mined so far to be searched for the inclusion of each generator. In space because to perform efficiently set-inclusion checks all the closed itemsets have to be kept in the main memory. To reduce such costs, closed sets can be stored in compact prefix tree structures, indexed by one or more levels of hashing.

Computing Closures.

To compute the closure of a generator 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 obtain this closure is suggested by the following lemma:

Lemma 2 *Given an itemset X and an item $i \in \mathcal{I}$, $g(X) \subseteq g(i) \Leftrightarrow i \in c(X)$.*

Proof.
$(g(X) \subseteq g(i) \Rightarrow i \in c(X))$:

Since $g(X \cup i)^1 = g(X) \cap g(i)$, $g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X)$. Therefore, if $g(X \cup i) = g(X)$ then $f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X)$.

($i \in c(X) \Rightarrow g(X) \subseteq g(i)$):

If $i \in c(X)$, then $g(X) = g(X \cup i)$. Since $g(X \cup i) = g(X) \cap g(i)$, $g(X) \cap g(i) = g(X)$ holds too. Thus, we can deduce that $g(X) \subseteq g(i)$.

$\square$

From the above lemma, we have that if $g(X) \subseteq g(i)$, then $i \in c(X)$. Therefore, by performing this inclusion check for all the items in $\mathcal{I}$ not included in X, we can *incrementally* compute $c(X)$. Note that the set $g(i)$ can be represented by a list of *transaction identifiers*, i.e., the *tidlist* associated with i. This suggests the adoption of a vertical format for the input dataset in order to efficiently implement the inclusion check: $g(X) \subseteq g(i)$.

The closure computation can be performed *off-line* or *on-line*. In the former case we firstly retrieve the complete set of generators, and then compute their closures. In the latter case, as soon as a new generator is discovered, its closure is computed on-the-fly. The algorithms that compute closures on-line are generally more efficient than those that adopt an off-line approach, since the latter ones usually exploit key patterns as generators. Key patterns are the minimal itemsets of the equivalence class, and thus are the shortest possible generators. Conversely, the on-line algorithm usually adopt the *closure climbing* strategy, according to which new generators are created recursively from closed itemsets. These generators are likely longer than key patterns. Obviously, the longer a generator is, the fewer checks (on further items to add) are needed to get its closure.

3 Related Works

The first algorithm proposed for mining closed itemsets was A-Close [11] (N. Pasquier, et al.). A-Close first browses level-wise the frequent itemsets lattice by means of an Apriori-like strategy, in order to mine the generators of all the closed itemsets. In particular, in this first step the generators extracted by A-Close are all the key-patterns, i.e. the minimal itemsets of all equivalence classes. Since a k-itemset is a key pattern if and only if no one of its $(k-1)$-subsets has the same support [16], minimal elements can only be discovered with an intensive subset checking. In its second step, A-Close computes 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 suffers from the high cost of the off-line closure computation and the huge number of subset searches.

The authors of FP-Growth [5] (J. Han, et al.) proposed Closet [14] and Closet+ [15]. 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 transactions, pruned from infrequent items, 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, and by incrementally growing up frequent closed itemsets with items having the same support in the conditional dataset. Duplicates are discovered with subset checking by exploiting Lemma 2. Thus, all closed sets previously discovered are kept in a two level hash table stored in the main memory. Closet+ is similar to Closet, but exploits an adaptive behaviour in order to fit both sparse and dense datasets. As regards the duplicate problem, Closet+ introduces a new detection technique for sparse datasets named *upward checking*. This technique consists in intersecting every path of the initial FP-Tree leading to a candidate closed itemset X. If such intersection is empty then X is actually a closed itemset. The rationale for using this technique only for mining sparse dataset is that the transactions are in this case generally quite short, and thus the intersections can be performed quickly. Note that with dense dataset, where the transactions are usually longer, closed itemsets equivalence classes are large and the number of duplicates is high, such technique is not used because

[1]For the sake of readability, we will drop parentheses around singleton itemsets, i.e. we will write $X \cup i$ instead of $X \cup \{i\}$, where single items are represented by lowercase characters.

of its inefficiency, and Closet+ adopts the same duplicate detection strategy of Closet, i.e. the one based on keeping every mined closed itemset in the main memory.

FP-Close [4] (G. Grahne and J. Zhu), which is a variant of Closet+, resulted to be the best algorithm for closed itemsets mining presented at the 2003 Frequent Itemset Mining Implementations Workshop (`http://fimi.cs.helsinki.fi`).

Charm [19, 17] (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. If one tid-list includes another one, the associated nodes are merged since both 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 [18]. Since different paths can however lead to the same closed itemset, also in this case a duplicates detection and 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 single level hash.

According to the framework introduced in Section 2, A-Close exploits a key pattern browsing strategy and performs off-line closure computation, while Charm, Closet+ and FP-Close are different implementations of the same closure climbing strategy with on-line incremental closure computation.

4 Memory-efficient duplicate detection and pruning

In this Section we propose a particular visit of the lattice of frequent sets that efficiently identifies *unique generators* for each equivalence class, and allow all the closed patterns to be mined through the minimum number of closure computations.

We assume that a *closure climbing* strategy is adopted to browse the search space and find out new generators. As soon as a closed itemset Y has been identified, new generators are built as proper supersets of Y, i.e., generators of the form $gen = Y \cup i$, where $i \in \mathcal{I}$, $i \notin Y$. It is straightforward to show that for each closed itemset Y', $Y' \neq c(\emptyset)$, there must exists at least a generator of the form $gen = Y \cup i$, where Y, $Y \subset Y'$, is a closed itemset, $i \notin Y$, and $Y' = c(gen)$.

By looking at Figure 1.(a), we can see that it is possible to discover multiple generators of the form $gen = Y \cup i$ for the same closed itemset. For example, we have four generators, $\{A\}$, $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$, whose closure is the closed itemsets $\{A, B, C\}$. Note that all these generators have the form $Y \cup i$, since they can be obtained by adding a single item to a smaller closed itemset, namely $\emptyset$, $\{B\}$ and $\{C\}$.

Our strategy exploits a total lexicographic order $\prec$ between itemsets, in turn based on a total order relationship between single item literals, according to which each k-itemset I can be considered, without loss of generality, an *increasingly sorted set* of k *distinct* items $\{i_0, i_1, ..., i_k\}$. In the following we will assume that all the k-itemsets are increasingly sorted sets, and that a relation $\prec$ always exists between each pair I, I' of itemsets (i.e., if $I \neq I'$, then either $I \prec I'$ or $I \prec I'$).

In the following we will formally show that, for each closed itemset, it is possible to devise *one* and *only one* sequence of generators that respect a specific property, i.e., whose closures can be carried out according to the *total lexicographic order* relation $\prec$ between itemsets (see Definition 2). In other words, if we force the computation of generator closures to be carried out in the order established by $\prec$, no duplicates will ever occur.

Definition 2 *A generator $X = Y \cup i$, where Y is a closed itemset and $i \notin Y$, is said to be* order preserving *iff $i \prec (c(X) \setminus X)$.*

Theorem 1 below shows that, for any closed itemset Y, there exists a sequence of *order preserving* generators that allow to climb a sequence of closed itemsets and reach Y. Corollary 1 shows instead the uniqueness of this sequence.

Hence, the goal of an algorithm that mines closed itemsets by avoiding redundances is to compute the closure of all order preserving generators, and prune the other generators that do not respect the order preserving property.

Theorem 1 *For each closed itemset $Y \neq c(\emptyset)$, a sequence of n, $n \geq 1$, items $i_0 \prec i_1 \prec \ldots \prec i_{n-1}$ exists such that*

$$\{gen_0, gen_1, \ldots, gen_{n-1}\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_{n-1} \cup i_{n-1}\}$$

where the various gen_i are order preserving generators, *with $Y_0 = c(\emptyset), \forall j \in [0, n-1]$, $Y_{j+1} = c(Y_j \cup i_j)$, and $Y_n = Y$.*

Proof. First we have to show that, for all generators gen_i, if $gen_i \subseteq Y$, then $c(gen_i) \subseteq Y$. Note that $g(Y) \subseteq g(gen_i)$ because $gen_i \subseteq Y$. Moreover, Lemma 2 states that if $j \in c(gen_i)$, then $g(gen_i) \subseteq g(j)$. Thus, since $g(Y) \subseteq g(gen_i)$, then $g(Y) \subseteq g(j)$ holds too, and from Lemma 2 it also follows that $j \in c(Y)$. So, if $j \notin Y$ held, Y would not be a closed itemset because $j \in c(Y)$, and this is in contradiction with the hypothesis.

As regards the proof of the Theorem, we show it by constructing a sequence of closed itemsets and associated generators having the properties stated above.

Since $Y_0 = c(\emptyset)$, Y_0 is included in all the transactions of the dataset, then, by definition of closure, all the items in Y_0 must be also included in Y, i.e. $Y_0 \subseteq Y$.

Moreover, since $Y_0 \neq Y$ by definition, in order to create the first order preserving generator $\{Y_0 \cup i_0\}$, we choose $i_0 = \min_{\prec}(Y \setminus Y_0)$, i.e. i_0 is the smallest item in $\{Y \setminus Y_0\}$ with respect to the lexicographic order $\prec$. Afterwards, we compute $Y_1 = c(Y_0 \cup i_0) = c(gen_0)$. If $Y_1 = Y$ we can stop.

Otherwise, in order to build the next *order preserving* generator $gen_1 = Y_1 \cup i_1$, we choose $i_1 = \min_{\prec}(Y \setminus Y_1)$, where $i_0 \prec i_1$ by construction, and we compute $Y_2 = c(Y_1 \cup i_1) = c(gen_1)$.

Again, if $Y_2 = Y$ we can stop, otherwise we iterate the process by choosing $i_2 = \min_{\prec}(Y \setminus Y_2)$, and so on.

Note that each generator $gen_j = \{Y_j \cup i_j\}$ is *order preserving*, because $c(\{Y_j \cup i_j\}) = Y_{j+1} \subseteq Y$ and i_j is the minimum item in $\{Y \setminus Y_j\}$ by construction, i.e. $i_j \prec \{Y_{j+1} \setminus \{Y_j \cup i_j\}\}$.
□

Corollary 1 *For each closed itemset $Y \neq c(\emptyset)$, the sequence of* order preserving generators *of Theorem 1 is unique.*

Proof. Suppose that, during the construction of the sequence of generators, we choose $i_j \neq \min_{\prec}(Y \setminus Y_j)$ to build generator gen_j. Since gen_j and all the following generators must be *order preserving*, it should be impossible to obtain Y, since we could no longer consider the item $i = \min_{\prec}(Y \setminus Y_j) \in Y$ in any other generator or closure in order to respect the *order preserving* property.
□

Looking at Figure 1.(a), for each closed itemset we can easily identify the unique sequences of order preserving generators. For example, for the the closed itemset $Y = \{A, B, C, D\}$, we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup \{A\}$, $Y_1 = c(gen_0) = \{A, B, C\}$, $gen_1 = \{A, B, C\} \cup \{D\}$, and, finally, $Y = c(gen_1)$. Another example regards the closed itemset $Y = \{B, D\}$, where we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup \{B\}$, $Y_1 = c(gen_0) = \{B\}$, $gen_1 = \{B\} \cup \{D\}$, and, finally, $Y = c(gen_1)$.

4.1 Detecting order preserving generators

In order to exploit the results of Theorem 1, we need to devise an efficient method to check whether a given generator is order preserving or not. Let us introduce the following Definition:

Definition 3 *Given a generator $gen = Y \cup i$, where Y is a closed itemset and $i \notin Y$, we define pre-set(gen) as follows:*

$$\textit{pre-set}(gen) = \{j \mid j \in \mathcal{I},\ j \notin gen,\ \textit{and}\ j \prec i\}.$$

The following Lemma gives a way to check the order preserving property of *gen* by considering the tidlists $g(j)$, for all $j \in$ *pre-set*(gen).

Lemma 3 *Let $gen = Y \cup i$ be a generator where Y is a closed itemset and $i \notin Y$. If $\exists j \in$ pre-set(gen), such that $g(gen) \subseteq g(j)$, then gen is* not order preserving.

Proof. If $g(gen) \subseteq g(j)$, then $j \in c(gen)$. Since by hypothesis $j \notin gen$, we have that $j \in (c(gen) \setminus gen)$. Since $j \prec i$ because $j \in$ *pre-set*(gen), $i \not\prec (c(gen) \setminus gen)$ and thus, according to Definition 2, *gen* is *not order preserving.*
□

We have thus contributed a deep study on the the problem of duplicates in mining frequent closed itemsets. We have introduced the concept of order preserving generators, i.e. specific generators whose closures can be computed by browsing the itemset lattice according to a total lexicographic order. We have shown that duplicate generators are the ones that do not respect the order preserving property. So, the duplication check can be carried out by simply using the *tidlists* associated with single items, instead of using the set of closed itemsets already mined. Our technique is not resource demanding, because frequent closed itemsets need not to be stored in the main memory during the computation. Moreover, once introduced suitable optimization techniques (see Section 5.1.2), our order preserving check becomes less time demanding, and results cheaper than searching the set of closed itemsets mined so far. Note that CLOSET+ needs the initial FP-tree as an additional requirement to the current FP-tree in use, and moreover does not use its upward checking technique with dense datasets.

5 The DCI_CLOSED algorithm.

The DCI_CLOSED algorithm adopts two different strategies to deal with *dense* and *sparse* datasets. It is well known that frequent closed itemsets are a lossless condensed representation of frequent itemsets. However, in sparse datasets the number of closed itemsets is nearly equal to the number of frequent ones. Thus, due to the high number of closure computations, mining closed itemsets in sparse datasets may become more expensive than extracting all the frequent itemsets. The profitability of mining closed frequent itemsets rather than frequent itemsets, roughly depends on the ratio between the number of frequent closed itemsets and the corresponding total number of frequent itemsets. Unfortunately, this ratio is not known till the end the mining process. We found however that a simple statistical measure of the density of a dataset [9], allows to effectively discriminate between datasets and adopt the better strategy.

The two different strategies for *dense* and *sparse* datasets are implemented within two different procedures: DCI_CLOSED$_s$(), suitable for *sparse* datasets, and DCI_CLOSED$_d$(), suitable for *dense* datasets. Before starting the actual mining process, the dataset $\mathcal{D}$ is scanned to determine the frequent single items $\mathcal{F}_1 \subseteq \mathcal{I}$, and to build the bitwise vertical dataset $\mathcal{VD}$ containing the various *tidlists* $g(i)$, $\forall i \in \mathcal{F}_1$.

In Section 5.1 we will detail the most interesting part of the algorithm, i.e. the techniques and optimizations used in DCI_CLOSED$_d$(). Here we only sketch the strategy adopted by the DCI_CLOSED$_s$() procedure, used to mine sparse datasets.

While DCI_CLOSED$_d$(), used to mine dense datasets, adopts a *depth-first* exploration of the lattice of frequent itemsets, DCI_CLOSED$_s$() adopts a traditional *level-wise* visit. The reason is that in mining sparse datasets we can effectively exploit the anti-monotone Apriori property to prune *candidates.* DCI_CLOSED$_s$() is thus based on a slightly modified version of our level-wise DCI algorithm for mining frequent itemsets [8, 7], with a simple additional *closedness test* over the frequent itemsets discovered. Since a frequent k-itemset I can be identified as closed if no superset of I with the same support exists, we delay the output of the frequent k-itemsets (level k) until all the frequent $(k+1)$-itemsets (level $k+1$) have been discovered. Then, for each frequent $(k+1)$-itemset I', we mark as *non closed* all subsumed k-itemsets I ($I \subset I'$) with exactly the same support as I' ($supp(I) = supp(I')$). At the end, we can finally identify as closed ones all the frequent k-itemsets which result to be not marked.

5.1 Mining dense datasets.

The pseudo-code of the recursive procedure DCI_CLOSED$_d$() is shown in Algorithm 1. The procedure has three input parameters: a closed itemset CLOSED_SET, and two sets of items, PRE_SET and POST_SET.

It outputs all the closed itemsets that properly contain CLOSED_SET by analyzing the *valid generators* obtained by extending CLOSED_SET with the items in POST_SET.

As previously discussed, the dataset $\mathcal{D}$ is firstly scanned to determine the frequent single items $\mathcal{F}_1 \subseteq \mathcal{I}$, and to build the bitwise vertical dataset $\mathcal{VD}$ containing the various *tidlists* $g(i)$, $\forall i \in \mathcal{F}_1$. The procedure DCI_CLOSED$_d$() is then called by passing as arguments: CLOSED_SET $= c(\emptyset)$ [2], PRE_SET $= \emptyset$, and POST_SET $= \mathcal{F}_1 \setminus c(\emptyset)$.

The procedure builds all the possible *generators*, by extending CLOSED_SET with the various items in POST_SET (lines 2–5). Both *infrequent* and *not order preserving* generators are promptly discarded (lines 6–7) without computing their closure. Note that the items $i \in POST\_SET$ used to obtain these invalid generators will no longer be considered in the following recursive calls. The closures of valid generators are then computed (lines 8–17). It is worth noting that each generator $new\_gen \leftarrow$ CLOSED_SET $\cup\, i$ is strictly extended according to the order preserving property, i.e. by using all items $j \in$ POST_SET such that $i \prec j$ (lines 3–4, and line 10). Moreover, all the items j, $i \prec j$, that do not belong to $c(new\_gen)$ are included in the new POST_SET (line 14) to be used for the next recursive call. At the end of this process, a new closed set (CLOSED_SET$_{New} \leftarrow c(new\_gen)$) is obtained (line 17). From this new closed set, new generators and corresponding closed sets can be built, by recursively calling the procedure DCI_CLOSED$_d$() (line 18).

Algorithm 1 DCI_CLOSED pseudocode

```
1:  procedure DCI_CLOSED_d(CLOSED_SET, PRE_SET, POST_SET)
2:      while POST_SET ≠ ∅ do
3:          $i \leftarrow \min_{\prec}$(POST_SET)
4:          POST_SET ← POST_SET \ $i$
5:          $new\_gen$ ← CLOSED_SET ∪ $i$                              ▷ Build a new generator
6:          if $supp(new\_gen) \geq min\_supp$ then                     ▷ $new\_gen$ is frequent
7:              if $is\_dup(new\_gen$, PRE_SET) = FALSE then   ▷ $new\_gen$ is an order preserving generator
8:                  CLOSED_SET$_{New}$ ← $new\_gen$
9:                  POST_SET$_{New}$ ← ∅
10:                 for all $j$ ∈ POST_SET do                       ▷ Compute closure of $new\_gen$
11:                     if $g(new\_gen) \subseteq g(j)$ then
12:                         CLOSED_SET$_{New}$ ← CLOSED_SET$_{New}$ ∪ $j$
13:                     else
14:                         POST_SET$_{New}$ ← POST_SET$_{New}$ ∪ $j$
15:                     end if
16:                 end for
17:                 Write out CLOSED_SET$_{New}$ and its support
18:                 DCI_CLOSED_d(CLOSED_SET$_{New}$, PRE_SET, POST_SET$_{New}$)
19:                 PRE_SET ← PRE_SET ∪ $i$
20:             end if
21:         end if
22:     end while
23: end procedure
24:
25:
26: function $is\_dup(new\_gen$, PRE_SET)
27:     for all $j$ ∈ PRE_SET do                                    ▷ Duplicate check
28:         if $g(new\_gen) \subseteq g(j)$ then
29:             return TRUE                                         ▷ $new\_gen$ is not order preserving
30:         end if
31:     end for
32:     return FALSE
33: end function
```

[2]The closed itemset $c(\emptyset)$ contains, if any, the items that occur in all the transactions of the dataset $\mathcal{D}$.

Before recursively calling the procedure, it is however necessary to prepare the suitable PRE_SET and POST_SET. Note that when the recursive level is deepened, the size of CLOSED_SET (CLOSED_SET$_{New}$) monotonically increases, while the size of POST_SET (POST_SET$_{New}$) monotonically decreases, and PRE_SET does not change. On the other hand, PRE_SET size is monotonically increased each time a recursive call returns (line 19): the new PRE_SET is then used to explore in depth the next valid generator (*new_gen*) obtained by extending the current CLOSED_SET.

Regarding the construction of POST_SET$_{New}$ made before the recursive call of the procedure, assume that the closed set X=CLOSED_SET$_{new}$ passed to the procedure (line 16) has been obtained by computing the closure of a generator $new\_gen = Y \cup i$ (i.e., $X = c(new\_gen)$), where Y=CLOSED_SET and $i \in$ POST_SET. Note that, since i has been chosen as $i \leftarrow \min_{\prec}$(POST_SET) (line 3), and then removed from POST_SET (line 4), we have that $i \prec j$ for all j used for computing the closure of *new_gen* (line 10). POST_SET$_{new}$ is thus built as the set of all the items that follow i in the lexicographic order, but that have not been already included in X. More formally, POST_SET$_{new}$ $= \{j \in POST\_SET \mid i \prec j \text{ and } j \notin X\}$. This condition allows the recursive call of the procedure to only build new generators $X \cup j$, where $i \prec j$, according to the hypothesis of Theorem 1.

The composition of PRE_SET instead depends on the sequence of *valid* generators, i.e. the ones that have passed the frequency and duplicate tests, that precede $new\_gen = Y \cup i$ in the lexicographic order. If all the generators were valid, it would simply be composed of all the items j that precede i in the lexicographic order, and $j \notin X = c(new\_gen)$. In other words, PRE_SET would be the complement set of $X \cup$ POST_SET$_{new}$. Conversely, if invalid generators are discovered, this permits us to prune PRE_SET, as discussed later in Section 5.1.1.

While the composition of POST_SET guarantees that the various generators will be produced according to the lexicographic order $\prec$, the composition of PRE_SET guarantees that duplicate generators will be correctly pruned by function *is_dup*() (lines 26–33).

Since we have shown that for each closed itemset Y one and only one sequence of *order preserving* generators exists, and since our algorithm clearly explores every possible *order preserving* generator from every closed itemset, we have that the algorithm is *complete* and does not produce any duplicate.

5.1.1 Pruning PRE_SET

We have discussed above that the PRE_SET passed to the recursive call of DCI_CLOSED$_d$() is monotonically increased. Each time a generator *new_gen* is built by extending CLOSED_SET, PRE_SET should be modified accordingly, in order to obtain *pre-set*(*new_gen*) (see Definition 3) for the duplication check.

However, PRE_SET $\subseteq$ *pre-set*(*new_gen*), since the items added to PRE_SET are only those that have been already used to build valid generators (line 19). In other words, the items that have been used to build *invalid* generators are pruned from *pre-set*(*new_gen*), thus obtaining the pruned PRE_SET exploited by our algorithm. The following lemmas show that we can safely prune these items, since the duplication check still works properly. While Lemma 4 regards the pruning of items used to build not order preserving generators, Lemma 5 is concerned with those used to produce infrequent generators.

Lemma 4 *Let* $gen = Y \cup i$ *be a* not order preserving *generator, where* Y *is a closed itemset and* $i \notin Y$*. Thus there must exist* $h \in$ *pre-set*(gen)*, i.e.* $h \prec i$*, such that* $g(Y \cup i) \subseteq g(h)$*. Also suppose that there exists another closed itemset* Y'*,* $Y \subset Y'$*, from which we obtain a generator* $gen' = Y' \cup i'$*, where both* $h, i \in$ *pre-set*(gen')*. Then, in order to check the order preserving property of* gen' *(see Lemma 3), we can avoid to check whether* $g(gen') \subseteq g(i)$*, because* $g(gen') \not\subseteq g(h) \Rightarrow g(gen') \not\subseteq g(i)$*.*

Proof. We prove by contrapositive. So, in order to show that $g(gen') \not\subseteq g(h) \Rightarrow g(gen') \not\subseteq g(i)$, we equivalently show that $g(gen') \subseteq g(i) \Rightarrow g(gen') \subseteq g(h)$. If $g(gen') \subseteq g(i)$, then $g(gen') \equiv g(gen' \cup i)$. Since we can also express gen' as $gen' = Y \cup Z \cup i'$, where $Z = Y' \setminus Y$, then $g(Y \cup Z \cup i') \equiv g(Y \cup Z \cup i' \cup i)$. Since if $A \subseteq B \Rightarrow g(B) \subseteq g(A)$, then we have that $g(Y \cup Z \cup i' \cup i) \subseteq g(Y \cup i)$. Thus we have that $g(gen') \equiv g(gen' \cup i) = g(Y \cup Z \cup i' \cup i) \subseteq g(Y \cup i)$. Since, by hypothesis, $g(Y \cup i) \subseteq g(h)$, then we can deduce that $g(gen') \subseteq g(h)$ holds.
□

Lemma 4 can be exploited as follows. We know that both i and h should belong to *pre-set*(gen'), but we can remove i from this set. Consider in fact, that, once checked whether $g(gen') \subseteq g(h)$, if the inequality holds then we can deduce that gen' is not order preserving and no further checks are needed. Otherwise, if the inequality does not hold, it is not needed to also check whether $g(gen') \subseteq g(i)$, because this inequality surely will not hold too.

Lemma 5 *Let* $gen = Y \cup i$ *be a* not frequent *generator (i.e.,* $|g(gen)| < min\_supp$*), where* Y *is a closed itemset and* $i \notin Y$*. Suppose that there exists a closed itemset* Y'*,* $Y \subset Y'$*,* $|g(Y')| \geq min\_supp$*, from which we obtain a* frequent *generator* $gen' = Y' \cup i'$*, i.e.* $|g(gen')| \geq min\_supp$*. Then, in order to check the order preserving property of* gen' *(see Lemma 3), we can avoid checking whether* $g(gen') \subseteq g(i)$*, because* $g(gen') \nsubseteq g(i)$*.*

Proof. We prove the Lemma by contradiction. Assume that $g(gen') \subseteq g(i)$. Then, $g(gen') \equiv g(gen' \cup i)$, and thus $|g(gen')| = |g(gen' \cup i)|$. Note that, since $Y \subset Y'$, then we can write $gen' = Y \cup Z \cup i'$, where $Z = Y' \setminus Y$. Since if $A \subseteq B \Rightarrow g(B) \subseteq g(A)$, then we have that $g(gen' \cup i) = g(Y \cup Z \cup i' \cup i) \subseteq g(Y \cup i)$ because $Y \cup i \subseteq gen' \cup i$. Thus $|g(Y \cup Z \cup i' \cup i)| \leq |g(Y \cup i)|$. Since by hypothesis $|g(Y \cup i)| < min\_supp$, then also $|g(gen')| \equiv |g(Y \cup Z \cup i' \cup i)| < min\_supp$. This is in contradiction with the hypothesis that gen' is frequent.
□

5.1.2 Optimizations to save bitwise intersection work

We adopted a large amount of optimizations to reduce the amount of bitwise *AND* intersections performed by our algorithm. These intersections are needed for duplication checking and closure computations (line 10 and 34), and also for computing the tidlist ($g(new\_gen)$) and the support ($supp(new\_gen)$) of each new generators (line 6). For the sake of simplicity, these optimizations were not reported in the pseudo-code shown in Algorithm 1, but are described in the following paragraphs.

Dataset projection. Let us consider a closed set X and its tidlist $g(X)$, i.e. the set of all the transactions that set-include X. All the closed itemsets that are proper supersets of X, i.e. those that are discovered by recursively calling DCI_CLOSED$_d(X, \_, \_)$, will be supported by subsets of $g(X)$. Thus, once X is found, we can save work in the subsequent recursive calls of the procedure by projecting $\mathcal{VD}$. This is carried out by deleting from $\mathcal{VD}$ all the columns corresponding to $T \setminus g(X)$. Since this bit-wise projection is quite expensive, we limit it to generators of the first level of recursion only, i.e., those obtained from the order preserving generators obtained by extending $c(\emptyset)$. In Section 6 we will evaluate the benefits of this optimization technique, and will refer to it as the **projection** optimization.

Highly correlated datasets. DCI_CLOSED inherits the internal representation of our previous works DCI[8] and kDCI[7]. The dataset is stored in the main memory using a vertical bitmap representation. With two successive scans of the dataset, a bitmap matrix $\mathcal{VD}_{M \times N}$ is stored in the main memory. The $\mathcal{VD}(i, j)$ 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 tidlist associated with item i.

The columns of $\mathcal{VD}$ are then reordered to profit of data correlation, which entails high similarity between the rows of the matrix when we mine dense datasets. As in [7][8], columns are reordered to create a submatrix $\mathcal{VE}$ of $\mathcal{VD}$ having all its rows identical. Every operation (e.g. intersection ones) involving rows in the submatrix $\mathcal{VE}$ will be performed only once, thus gaining strong performance improvements.

In Section 6 we will evaluate the benefits of this optimization technique, and will refer to it as the **section eq** optimization.

Reusing results of previous bitwise intersections. Besides the above optimizations, we exploited another technique made possible by the depth-first visit of the lattice of itemsets, and by the repeated scans of the same tidlists for performing bitwise *AND* operations.

In order to determine that an itemset X is closed, the tidlist $g(X)$ must be compared with all the tidlists $g(j)$, of items j contained in the PRE_LIST (POST_LIST) of X, i.e. the items that precede (follows) all items included in X according to the lexicographic order. The tidlists of items in PRE_SET are accessed for *checking duplicate generators*, while those of POST_SET for *computing the closure*. In particular, for all $j \in$ PRE_SET $\cup$ POST_SET, we already know that $g(X) \nsubseteq g(j)$, otherwise those items j should have been included in X. Therefore, we can save some important information regarding each comparison between each $g(j)$ and $g(X)$. Such information can profitably be reused to reduce the cost of the following computations involving $g(j)$, i.e., when each $g(j)$ is exploited to look for further closed itemsets.

In particular, even if, for all j, it is true that $g(X) \nsubseteq g(j)$, we may know that large sections of the bitwise tidlists $g(X)$ are however strictly included in $g(j)$. Let $g_h(X)$ be the section of $g(X)$, composed of h words, strictly included in the corresponding section $g_h(j)$ of $g(j)$. Hence, since $g_h(X) \subseteq g_h(j)$, it is straightforward to show that $g_h(X \cup Y) \subseteq g_h(j)$ holds, for every itemset Y, because $g(X \cup Y) \subseteq g(X)$. So, when we extend X to obtain a new generator, we can limit the inclusion check of the various $g(j)$ to the *complementary* portions of $g_h(j)$. It is worth noting that, as soon as our visit of the itemset lattice gets deeper, the closed itemset X we deal with becomes larger, while the portions $g(X)$ strictly included in the corresponding portion of $g(j)$ gets larger, thus making possible to save a lot of work related to inclusion check.

In Section 6 we will evaluate the benefits of this optimization technique, and will refer to it as the **included** optimization.

5.1.3 Space complexity.

The size of the output is actually a lower bound on the space complexity of those algorithms that require to keep in the main memory the whole set of closed itemsets to perform duplicate check. Conversely, the memory size required by an implementation based on our duplicate check is almost independent of the size of the output. To some extent, its memory occupation depends on those data structures that also need to be maintained in memory by other algorithms that visit depth-first the lattice and exploit tidlist intersections.

The main information that DCI_CLOSED has to maintain in the main memory is the tidlist of each generator in the current path of the lattice explored by the algorithm, and the tidlist of every frequent single item. In this way we are able to browse the search space intersecting nodes with tidlists of single items, and also to discard duplicates checking the order preserving property.

The worst case in memory occupation occurs when the number of generators and the length of the longest closed itemsets are maximal: this occurs when $c(\emptyset) = \emptyset$, and every itemset is frequent and closed. If N is the number of frequent single items, the deepest path along the lattice is composed of N nodes, each associated with a distinct tidlist. Therefore, also considering the tidlists of the original vertical dataset, the total number of tidlists to be kept in the main memory is $2N - 1$. Since the length of a tidlist is equal to the number of transactions T in the dataset, the worst space complexity of our algorithm is

$$O\left((2N - 1) \times T\right).$$

Note that if this worst case occurred, the total number of closed itemsets should be $O(2^N)$, so that, from the point of view of space complexity, it should be always better to store tidlists rather than storing the closed itemsets already mined in order to detect duplicates.

An important remark regards the features of dense datasets, which are the subject of this space complexity comparison. Fortunately the worst case above should not actually occur with such datasets, since frequent itemsets are usually orders of magnitude greater than closed itemsets, and equivalence classes are thus composed of several frequent itemsets. The number of frequent itemsets is instead comparable to the number of closed ones and associated generators in sparse datasets, but the length of the largest closed frequent itemset that can be extracted from them is usually much less than N. Note that DCI_CLOSED uses a different method for mining sparse datasets, based on a level-wise visit of the lattice and the same in-core vertical bitwise dataset. Independently of the algorithm adopted, mining sparse datasets should not be a big issue from the point of view of memory occupation, because number and length of frequent itemsets do not explode even for very low support thresholds.

6 Performance comparisons

We tested our implementation on a suite of publicly available datasets: `chess`, `connect`, `pumsb`, `pumsb*`, `retail`, `T40I10D100K`. They are all real datasets, except for the last one, which is a synthetic dataset available from IBM Almaden. The first four datasets are dense and produce large numbers of frequent itemsets also for large support thresholds, while the last two are sparse.

We compared the performances of DCI_CLOSED with those of two well known state of the art algorithms: FP-CLOSE [4], and CLOSET+ [15]. FP-CLOSE is publicly available from the FIMI repository `http://fimi.cs.helsinki.fi/fimi03/`, while the Windows binary executable of CLOSET+ was kindly provided us from the authors. We did not included CHARM in our tests, because FP-CLOSE was already proved to be faster. The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory. The FP-CLOSE and DCI_CLOSED algorithms were compiled with the gcc compiler available in the cygwin environment.

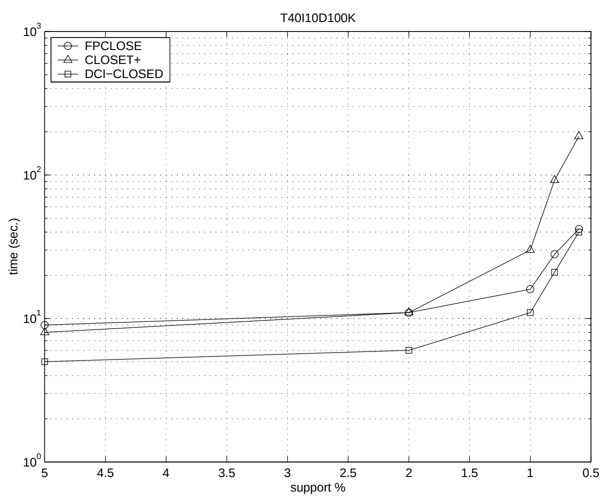
As shown in Figure 2.(a-f), DCI_CLOSED outperforms both competitors in all the tests conducted. CLOSET+ performs quite well on the `connect` dataset with relatively high supports, but in all other cases it is about two orders of magnitude slower. FP-CLOSE is effective in `pumsb*`, where its performance is close to that of DCI_CLOSED, but it is one order of magnitude slower in all the other tests.

As stated in Section 5.1.2, we adopted a large amount of optimizations to reduce the number of operations performed by DCI_CLOSED on dense datesets. It is worth recalling that, for each generator $X = Y \cup i$, we need to: (1) compute the associated tidlist $g(X)$ and thus the support of X, i.e. $|g(X)|$; (2) perform the duplication checks; (3) compute its closure. For all these operations we need to execute bitwise *AND* operations. In case (1), we perform *AND* operations to compute $g(X) = g(Y) \cap g(i)$, in order to derive cardinality $|g(X)|$, i.e. the number of 1's contained in the resulting bitvector $g(X)$. In both cases (2) and (3), we have to check for the inclusion of $g(X)$ in the various tidlists $g(j)$ associated with each single item j that either precedes or follows i in the lexicographic order. All these inclusion checks are carried out by intersecting $g(X) \cap g(j)$, and stopping at the first resulting word which is not equal to the corresponding word in $g(X)$. The two plots in Figure 3.(a) and 3.(c) show to the number of bitwise *AND* operations relative to case (1). They plot the number of operations actually executed by DCI_CLOSED to mine `chess` and `connect`, as a function of the support threshold. On the other hand, the two plots in Figure 3.(b) and 3.(d) refer to the number of bitwise *AND* operations relative to cases (2) and (3). Similarly to the above case, they show the number of operations actually executed to mine `chess` and `connect`, as a function of the support threshold.

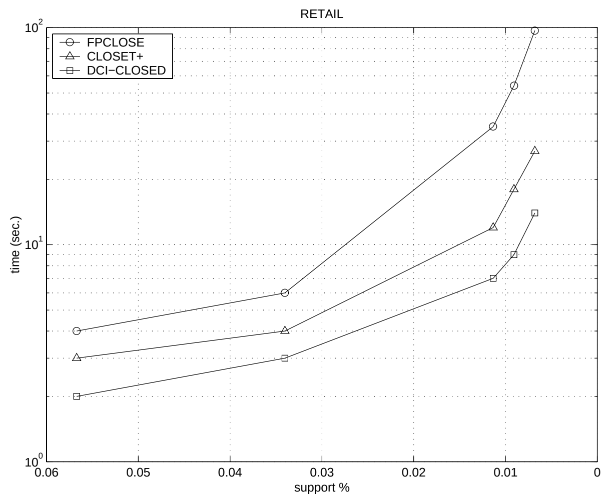
In all the plots of Figure 3, the top curves represent the baseline case, when no optimizations are exploited, so that the bitwise tidlists associated with single items are the original ones, i.e. the rows in the vertical dataset $\mathcal{VD}$. From the top curves to the bottom ones, we incrementally introduced the three optimization techniques discussed in Section 5.1.2, namely **projection**, **section eq**, and **included**. Therefore, the bottom curves correspond to the number of operations actually performed when all the optimizations are activated. We can see that the exploitation of all the three techniques allowed us to reduce the total number of *AND* operations up to an order of magnitude.

Another interesting remark regards the comparison between the amounts of operations actually executed to compute the tidlist $g(X)$ associated with each generators X (Figures 3.(a,c)), and the amounts of operations carried out for the various inclusion checks (Figures 3.(b,d)). The two amounts, given a dataset and a support threshold, appear to be similar. This result might surprise a careful reader. One could think that the operations needed for the inclusion checks are the majority, since, for each frequent generator X, we have to check the inclusion of $g(X)$ with almost all the tidlists $g(j)$. Indeed, while $g(X)$ must be generated for all generators X, only those that turn out to be *frequent* are furtherly processed for inclusion checks. Moreover, whereas lists must be completely scanned in order to produce $g(X)$, the same does not hold for inclusion checks relative to *order preserving check* or *closure computation*: when a single particular word of $g(X)$ is not included in the corresponding word of $g(j)$, we can stop since the whole inclusion check surely fails. Finally, if a single inclusion check fails during the *order preserving test*, then we discard X and not continue with further inclusions checks.

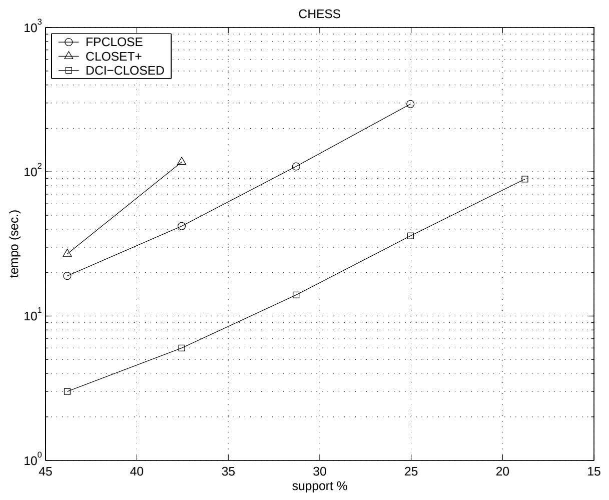
In order to compare our memory efficient duplication check technique with a more traditional one, based on the presence of a data structure that stores all the closed itemsets mined so far, we instrumented the



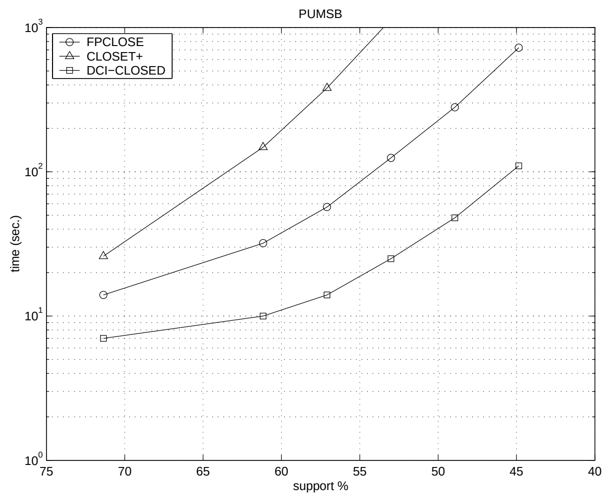
(a)



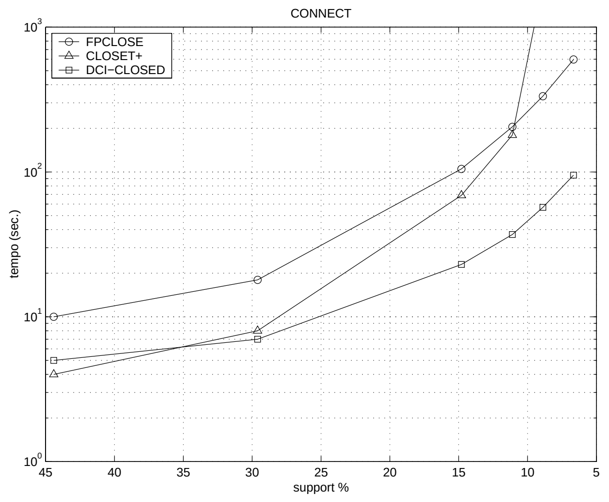
(b)



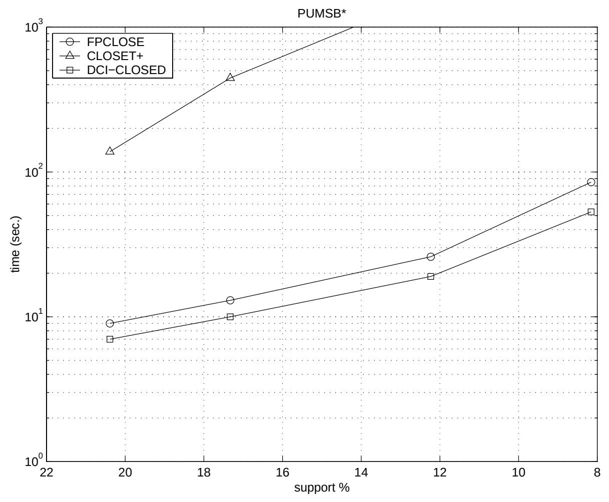
(c)



(d)

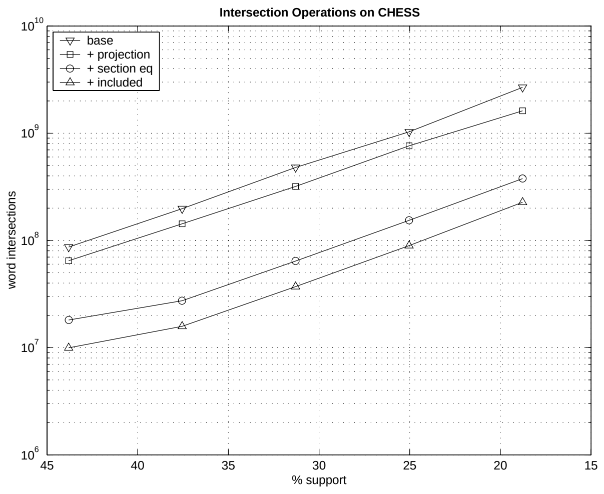


(e)

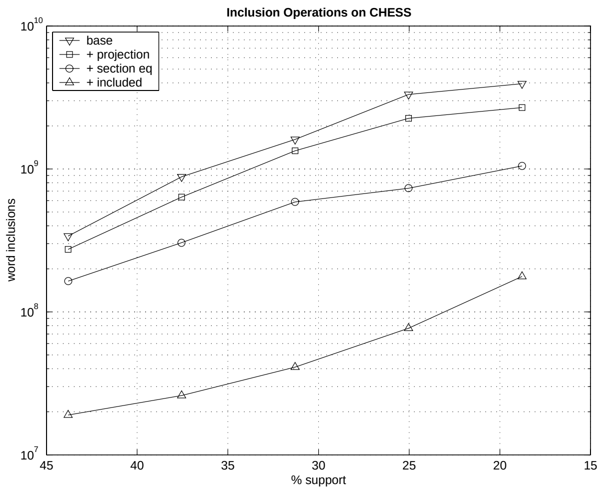


(f)

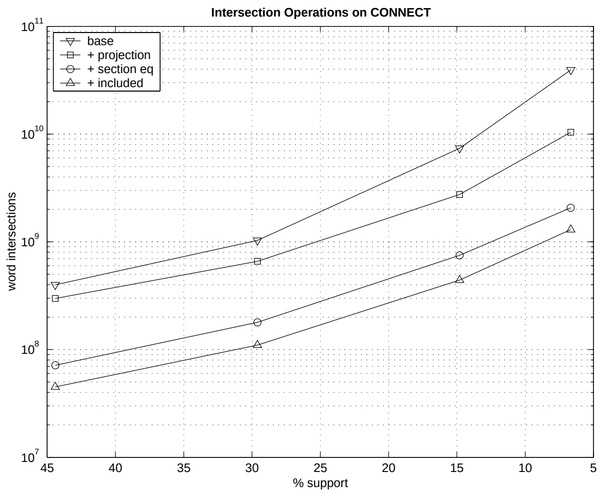
Figure 2: Execution times of FP-Close, Closet+, and DCI_Closed as a function of the minimum support threshold on various publicly available datasets.



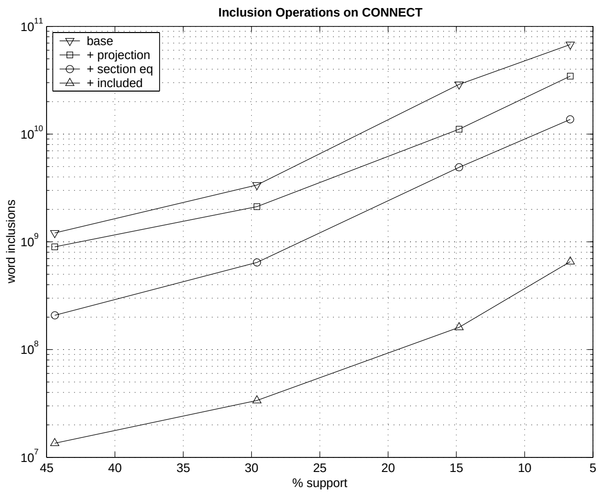
(a)



(b)



(c)



(d)

Figure 3: Number of intersections *(a-c)*, and of inclusions *(b-d)*, operations actually performed by the DCI_CLOSED$_d$() procedure when the various optimization techniques discussed in Section 5.1.2 are exploited or not. The plots refers to the `chess` *(a-b)* and `connect` datasets *(c-d)*. The number of operations is plotted as a function of the support threshold.

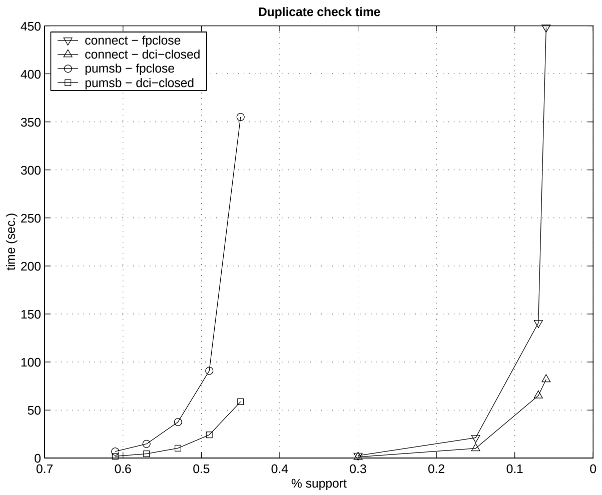


Figure 4: Absolute times spent for the duplication check by DCI_Closed and FP-Close two mine two dense datasets, `chess` and `connect`, as a function of the support threshold.

publicly available FP-Close code and our DCI_Closed code. Figure 4 shows the absolute times spent for the duplication check while mining two dense datasets, `chess` and `connect`, as a function of the support threshold. For small values of the support threshold our technique resulted the best, with a speedup up to six.

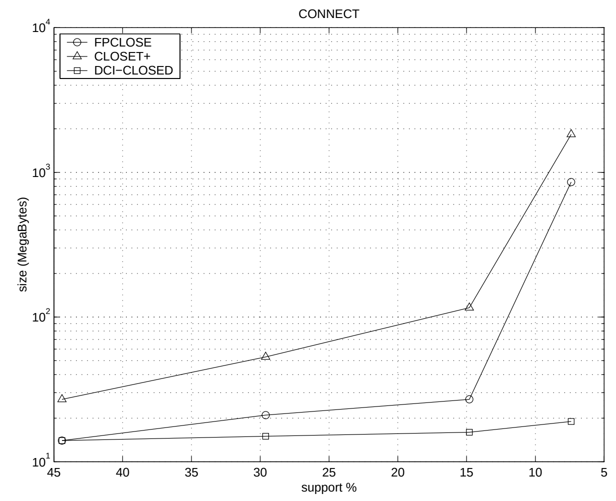
Another way to observe the efficiency of our method for duplication check is to measure main memory usage. Figure 5(a) plots memory occupation of FP-Close, Closet+ and our algorithm DCI_Closed when mining the `connect` dataset as a function of the support threshold. Note that the amount of main memory used by Closet+ and FP-Close grows exponentially when the support threshold is decreased due to the huge number of closed itemsets generated that have to be stored in the main memory. Conversely, the memory used by DCI_Closed, which does not need to maintain the frequent closed itemsets in memory for duplication check, continues to be nearly constant. In Figure 5(b) we plotted the results of the same test using the sparse dataset `T40I10D100K`, where Closet+ is supposed to use its upward checking technique. Also in this last test, DCI_Closed outperforms every other algorithm using ten times less memory.

7 Conclusions

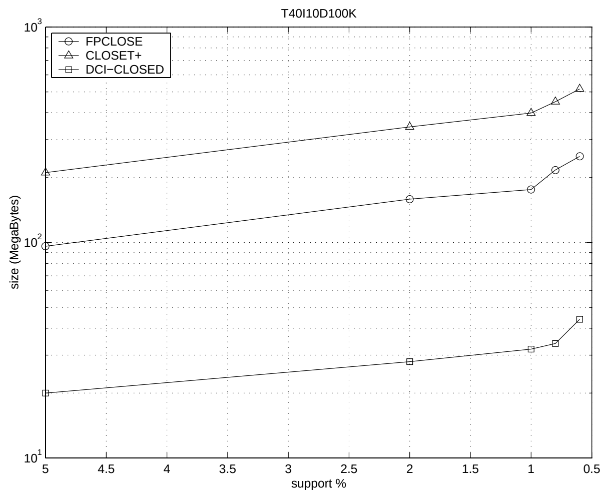
In this paper we have investigated in depth the problem of efficiency in mining closed frequent itemsets from transactional datasets. We claimed that state-of-the-art algorithms like Charm, Closet+, and FP-Close, are not memory efficient since they require to keep in the main memory the whole set of closed patterns mined so far in order to detect duplicated closed itemsets. We asserted that the duplicates generation problem is a consequence of the strategy adopted by current algorithms to browse the itemset lattice, and thus we proposed a new strategy for browsing the lattice which allows to effectively detect and discard duplicates without storing the closed patterns in the main memory. The proposed strategy has been formally proved to be valid, and can be exploited with substantial performance benefits by algorithms using a vertical representation of the dataset.

Moreover, we implemented our technique within DCI_Closed, a new algorithm which exploits a depth-first visit of the search space, and adopts a vertical bitmap representation of the dataset. DCI_Closed also exploits several innovative optimizations aimed to save both space and time in computing itemset closures and their supports. Since the basic operation to perform closures, support counts, and duplicate detections, is bitwise list intersection, we particularly optimized this operation, and, when possible, we reused previously computed intersections to avoid redundant computations.

As a result of the efficient strategies and optimizations introduced, DCI_Closed outperforms other state-



(a)



(b)

Figure 5: Memory used by DCI_Closed, Closet+, and FP-Close when a dense (a), and a sparse (b) dataset are mined, as a function of the support threshold.

of-the-art algorithms and requires orders of magnitude less memory when dense datasets are mined with low support thresholds. The in depth experimental evaluation conducted, demonstrates the effectiveness of our optimizations, and shows that the performance improvement over competitor algorithms – up to one order of magnitude – becomes more and more significant as the support threshold decreases.

References

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

[2] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[3] Bart Goethals and Mohammed J. Zaki. Advances in Frequent Itemset Mining Implementations: Report on FIMI'03. *SIGKDD Explor. Newsl.*, 6(1):109–117, 2004.

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

[5] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD '00*, pages 1–12, 2000.

[6] J. Liu, Y. Pan, K. Wang, and J. Han. Mining Frequent Item Sets by Opportunistic Projection. In *Proc. 2002 Int. Conf. on Knowledge Discovery in Databases (KDD'02), Edmonton, Canada*, 2002.

[7] Claudio Lucchese, Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[8] 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 (ICDM '02)*, pages 338–345, 2002.

[9] P. Palmerini, S. Orlando, and R. Perego. Statistical properties of transactional databases. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 515–519. ACM Press, 2004.

[10] J. S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 1995.

[11] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT '99*, pages 398–416, 1999.

[12] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.

[13] J. Pei, J. Han, H. Lu, S. Nishio, and D. Tang, S. amd Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *Proc. The 2001 IEEE International Conference on Data Mining (ICDM'01)*, San Jose, CA, USA, 2000.

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

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

[16] Rafik Taouil, Nicolas Pasquier, Yves Bastide, Lotfi Lajhal, and Gerd Stumme. Mining freqent patterns with counting inference. *SIGKDD Explorations*, 2(2):66–75, December 2000.

[17] Mohammed J. Zaki. Mining non-redundant association rules. *Data Min. Knowl. Discov.*, 9(3):223–248, 2004.

[18] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM Press, 2003.

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