Fast and Memory Efficient Mining of Frequent Closed Itemsets

Claudio Lucchese, Salvatore Orlando, Raffaele Perego

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

Our algorithm exploits a divide-and-conquer approach and a bitwise vertical representation of the database, and adopts a particular visit and partitioning strategy of the search space based on an original theoretical framework, which formalizes the problem of closed itemsets mining in detail. The algorithm adopts several optimizations aimed to save both space and time in computing itemset closures and their supports. In particular, since one of the main problems raising up in this type of algorithms is the multiple generation of the same closed itemset, we propose a new effective and memory-efficient pruning technique, which, unlike other previous proposals, does not require the whole set of closed patterns mined so far to be kept in the main memory. This technique also permits each visited partition of the search space to be mined independently in any order and thus also in parallel.

The tests conducted on many publicly available datasets show that our algorithm is scalable and 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.

***Index Terms*— Data Mining, Association rules, Frequent itemsets, Condensed representations, Closed itemsets, High performance algorithms.**

I. INTRODUCTION

FREQUENT Itemsets Mining (FIM) is a demanding task common to several important data mining applications that look 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 N transactions, where each transaction $t \in \mathcal{D}$ is a list of *distinct* items $t = \{i_1, ..., i_{|t|}\}$, $i_j \in \mathcal{I}$. Let I be a k-*itemset*, where $I = \{i_1, ..., i_k\}$ is a set of k *distinct* items $i_j \in \mathcal{I}$. 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 $\mathcal{I}$.

Manuscript received ... ...

Claudio Lucchese and Salvatore Orlando are with the Department of Computer Science, Ca' Foscari University of Venice, Italy. E-mail:{clucches,orlando}@dsi.unive.it

Raffaele Perego is with the High Performance Computing Lab at I.S.T.I.-C.N.R. of Pisa, Italy. E-mail:raffaele.perego@isti.cnr.it

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 [15], [8], [18], [23], [3], [21], [11], [14], [6]. Unfortunately, such algorithms may fail to extract all the frequent itemsets from *dense* datasets, which contain strongly correlated items and long frequent patterns. Such datasets are in fact very hard to mine since the Apriori *downward closure* property does not guarantee an effective pruning of candidates, while the number of frequent itemsets grows up 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 the task of analysts very complex, since they have to extract useful knowledge from a large amount of frequent patterns.

Closed itemsets are a solution to the problems described above. They are the unique maximal elements of the equivalence classes that can be defined over the lattice of all the frequent itemsets. Each class includes a distinct group of frequent itemsets, all supported by the same set of transactions. When a dataset is dense, frequent closed itemsets extracted can be orders of magnitude fewer than corresponding frequent itemsets, since they implicitly benefit from data correlations. Nevertheless, 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 itemsets have been proved to be more meaningful for analysts, because all redundancies are discarded [22]. Some efficient algorithms for mining closed itemsets have been recently proposed [19], [20], [7], [17], [24], [22]. These algorithms cannot mine closed itemsets directly, but all of them perform not useful or redundant computations in order to determine whether a given frequent itemset is closed or not, and, if so, whether it was already discovered or not. Moreover, in most cases these algorithms are not memory-efficient, since they require to maintain in the main memory all the closed itemsets mined during the execution in order to avoid generating *duplicates*, i.e., to check whether the closure of a given frequent itemset yields an already mined closed itemset or not.

A. Contributions

This paper presents a new scalable algorithm for closed frequent itemsets mining which exploits a divide-and-conquer approach and a bitwise vertical representation of the database. We introduced an innovative visit of the search space derived from an original theoretical framework which formalizes the problem of mining closed itemsets in detail. Furthermore, we investigated in depth the problem of identifying and pruning duplicates in closed itemsets generation. Differently from others, our algorithm does not need to mine closed itemsets according to a "strict" lexicographic order, and to keep previously mined closed itemsets in the main memory to perform duplicate checks. One of the positive effects of our visiting policy (and entailed duplicate checking technique) is that it permits to subdivide the search space and produce completely independent subproblems, which can be solved in whatever order, and thus also in parallel.

We called our algorithm DCI_CLOSED, since it inherits from DCI [14], [12] – a previously proposed algorithm to mine frequent itemsets – 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 bitwise *tidlists* – i.e., lists of identifiers of the transactions which contain a given item – we optimized this operation, and, whenever possible, reused results of 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 improvement – up to one or two orders of magnitude – becomes more remarkable as the support threshold decreases. Furthermore, due to its time and space efficiency, DCI_CLOSED completes successfully the mining tasks on input datasets and with support thresholds that cause all the other algorithms to fail.

B. Paper organization

The paper is organized as follows. In Section II we introduce closed itemsets and describe a framework for mining them. This framework is shared by all the algorithms surveyed in Section III. In Section IV we formalize the problem of duplicates, and devise the theoretical results at the basis of the particular visit of the search space followed by DCI_CLOSED. Section V describes the implementation of our closed itemset mining algorithm, while Section VI discusses the experimental results obtained. Some concluding remarks follows.

II. 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 items 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 *iff* no supersets of I with the same support exist. Therefore mining the *maximal* elements of all the equivalence classes corresponds to mine all the closed itemsets.

Fig. 1(b) shows the lattice of frequent itemsets derived from the simple dataset reported in Fig. 1(a), 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.

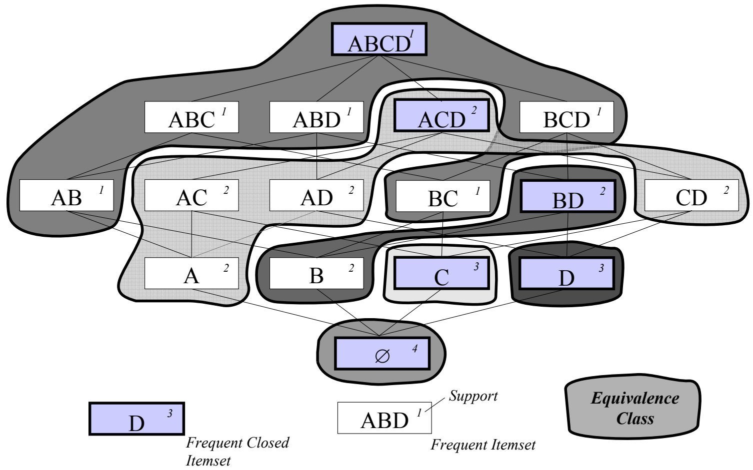
Browsing the search space: The goal of an effective browsing strategy should be to identify exactly a single itemset for each equivalence class. We could in fact mine all the closed itemsets by computing the closure of just this single representative itemset for each equivalence class, without generating any duplicate. Let us call 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 [21]. Unfortunately, an equivalence class can have more than one minimal element leading to the same closed itemset. For example, the closed itemset $\{A, B, C, D\}$ of Fig. 1(b) could be mined twice, since it can be obtained as the closure of two minimal elements of its equivalence class, namely $\{A, B\}$ and $\{B, C\}$.

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 the new generator belongs to an equivalence class that was not yet visited. Hence, similarly to the approach based on key patterns, we can visit multiple times the same equivalence class. For example, both $\{A, C\}$ and $\{C, D\}$ are generators of the same closed itemset $\{A, C, D\}$, and they can be obtained as

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

(a)



(b)

Fig. 1. (a) The input transactional dataset, represented in its horizontal form. (b) Lattice of all the frequent itemsets ($min\_supp = 1$), with closed itemsets and equivalence classes.

supersets of the closed itemsets $\{C\}$ and $\{D\}$, respectively.

Hence, regardless of the strategy adopted, we need to introduce some duplicate checking technique in order to avoid generating multiple times the same closed itemset. A naive approach is to compute the closure for each generator and look for duplicates among all the closed itemsets mined so far. Indeed, in order to avoid to perform useless 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 $c(X) = c(Y)$. In this case we also say that Y *subsumes* X. If this holds, we can safely prune the generator X without computing its closure. Otherwise, we have to compute $c(X)$ in order to obtain a new closed itemset.

Unfortunately, this technique 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, in order to efficiently perform set-inclusion checks, all the closed itemsets have to be kept in the main memory.

Several algorithms, like CHARM, CLOSET, and CLOSET+ [24], [19], [20], base their duplicate avoidance technique on the above Lemma. However, they might encounter problems in applying correctly the Lemma, because, given a whatever generator X, in some cases they are not able to compute its closure $c(X)$. This is because they implicitly perform recursive *projections* of the dataset. To guarantee a correct duplicate avoidance technique based on Lemma 1, they are thus forced to adopt a strict lexicographic visiting order of the search space. We illustrate this issue and the corresponding solution in the following.

As soon as the subtree of the lattice rooted in some item i has been exhaustively explored, such algorithms prune all occurrences of the item from every transaction (thus projecting the dataset), since all the closed itemsets containing i were surely already mined. On one hand, by progressively reducing the size of the dataset, projections make support and closure computations faster. On the other hand, by removing item i from the dataset, they unfortunately loose the possibility of checking whether i belongs to the closure of a generator. Therefore, once i has been deleted, it is no longer possible to mine closed itemsets containing i. For example, consider Fig. 1(b), and suppose that item A has been pruned from the projected dataset. When we reach generator $\{C, D\}$, we find out that it appears only in the two transactions $T_2 = \{B, C, D\}$, and $T_3 = \{C, D\}$ of the projected dataset (i.e., the one obtained by removing item A). In this situation, $\{C, D\}$ itself would be wrongly identified as a closed itemset. By applying the above lemma it should still be possible to identify $\{C, D\}$ as a duplicate generator that must be discarded, because it belongs to the same equivalence class of the closed itemset $\{A, C, D\}$. But, in order to verify that $\{A, C, D\}$ *subsumes* $\{C, D\}$ (i.e., $\{C, D\} \subseteq \{A, C, D\}$ and $supp(\{C, D\}) = supp(\{A, C, D\})$), the closed itemset $\{A, C, D\}$ must have already been mined. If the subtree rooted in C was mined after the one rooted in A, i.e. without respecting a strict lexicographic order, we would wrongly detect $\{C, D\}$ as a closed itemset, because we would not find any superset that subsumes $\{C, D\}$.

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 $g(X) \subseteq g(i)$.

■

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 $g(X) \subseteq g(i)$.

Closure computations can be performed *off-line* or *on-line*. In the former case we first 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 algorithms usually adopt the *closure climbing* strategy, according to which new generators are recursively created 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.

III. Related Work

As stated in Section I, closed itemsets are a lossless concise representation of all the frequent itemsets (and their supports) that can be extracted from a transactional dataset. Such concise representations are particularly important when datasets contain highly correlated items and very long frequent itemsets can be extracted. In this cases the size of a concise representation can be orders of magnitude smaller than the one of the ordinary uncompressed counterpart, i.e. the set of all the frequent itemsets.

Several concise representations [13] of frequent itemsets where proposed, such as closed itemsets (also known as free sets) [2], [16], disjunction-free sets [4], [9], generalized disjunction-free sets [10], non-derivable sets [5], and so on. This paper focuses on closed itemsets and related algorithmic issues.

The first algorithm proposed for mining closed itemsets was A-Close [16] (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 [21], minimal elements can be discovered with 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 [8] (J. Han, et al.) proposed Closet [19] and Closet+ [20]. 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 discovered 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 detected 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 intersections can be performed quickly. Note that with dense datasets, 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 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 [7] (G. Grahne and J. Zhu) is a variant of Closet+, and resulted to be the fastest algorithm for closed itemsets mining presented at the 2003 Frequent Itemset Mining Implementations Workshop (`http://fimi.cs.helsinki.fi`).

Charm [24], [22] (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 tidlist is compared with those of the other itemsets having the same parent. If one tidlist includes another one, the associated nodes are merged since both the itemsets surely belong to the same equivalence class. Itemset tidlists are stored in each node of the tree by using the diff-set technique [23]. Since different paths can however lead to the same closed itemset,

[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.

also in this case a duplicates detection and pruning strategy is implemented. To this purpose CHARM exploits a hash table to quickly individuate all the already mined closed itemsets Y that subsumes a given frequent itemset X. In more detail, the hash function is applied to $g(X)$, but, due to possible conflicts, after retrieving the matching closed itemsets, it is however needed to check subsumption, i.e., whether both $X \subset Y$ and $supp(X) = supp(Y)$ hold for each returned Y.

According to the framework introduced in Section II, 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.

IV. MEMORY-EFFICIENT DUPLICATE DETECTION AND PRUNING

In this Section we propose a particular visit of the lattice of frequent sets that allows *unique generators* for each equivalence class to be efficiently identified.

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 gen are built as proper supersets of Y having the form $gen = Y \cup \{i\}$, where $i \in \mathcal{I}$, $i \notin Y$. It is straightforward to show that for all closed itemsets Y', $Y' \neq c(\emptyset)$[2], at least one generator of the form $gen = Y \cup \{i\}$ exists, where Y, $Y \subset Y'$, is a closed itemset, $i \notin Y$, and $Y' = c(gen)$. From Fig. 1 we can see that it is possible to discover multiple generators of the above form for the same closed itemset. For example, the closed itemsets $\{A, C, D\}$ has four such generators, namely $\{A\}$, $\{A, C\}$, $\{A, D\}$ and $\{C, D\}$, obtained by adding a single item to the closed itemsets $\emptyset$, $\{C\}$, $\{D\}$, and $\{D\}$, respectively.

In the following, by introducing the *order-preserving* property of generators, we devise a general and memory-efficient technique to avoid duplicate generation.

Given any total order relation $\mathcal{R}$ defined among item literals, thereinafter we will always consider an itemset as an *ordered set*, in particular a *sequence* of *distinct* and increasingly sorted items. Thus, with $X \cup Y$, we will indicate the ordered sequence of items contained in X or Y. Moreover, let us denote with symbol $\prec$ the usual *lexicographic total order* between two ordered itemsets, in turn defined on the basis of $\mathcal{R}$.

For the sake of simplicity, in the following examples we will use the alphabetic order among item literals as total order relation $\mathcal{R}$, although any other total order could be actually used (e.g., a total order relationship induced by sorting items per increasing frequencies).

Definition 2: A generator of the form $X = Y \cup i$, where Y is a closed itemset and $i \notin Y$, is said to be *order-preserving* iff either c(X) = X or $i \prec (c(X) \setminus X)$.[3]

[2] The closed itemset $c(\emptyset)$ contains, if any, the items that occur in all the transactions of the dataset $\mathcal{D}$. If no such items exist, then $c(\emptyset) = \emptyset$.

[3] Also in this case, to simplify the notation we omit parenthesis when singleton itemsets must be represented. So, we will write $i \prec Y$ and $i \prec j$, instead of $\{i\} \prec Y$ and $\{i\} \prec \{j\}$, respectively.

Example 1: The above definition simply says that, if one or several items must be added to an *order-preserving* generator $X = Y \cup i$ in order to compute its closure $c(X)$, they have to follow i in the total order $\mathcal{R}$. Looking at the example of Fig. 1(b), we have that $\{A\} = \emptyset \cup \{A\}$ is an order-preserving generator of the closed itemset $\{A, C, D\}$, while $\{C, D\} = \{C\} \cup \{D\}$ is not an order-preserving generator for the same closed itemset. Note, however, that the definition does not impose that $Y \prec (Y \cup i)$. For example, $\{A, C, D\} \cup \{B\}$ is an order-preserving generator of $\{A, B, C, D\}$, even if $\{A, C, D\} \not\prec \{A, B, C, D\} = \{A, C, D\} \cup \{B\}$. $\square$

In order to mine all the closed itemsets by avoiding redundances, we compute the closure of order-preserving generators only, and prune the others. To show the soundness of our technique, we will formally show that, for each closed itemset, it is possible to devise *one* and *only one* sequence of order-preserving generators. In particular, Theorem 1 below shows that, for any closed itemset $\overline{Y}$, there exists a sequence of order-preserving generators that allow us to climb a sequence of closed itemsets and reach $\overline{Y}$, and Corollary 1 shows instead the uniqueness of this sequence.

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

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

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 $Y_n = \overline{Y}$.

Proof: First we have to show that, for all generators gen_i, if $gen_i \subseteq \overline{Y}$, then $c(gen_i) \subseteq \overline{Y}$. Note that $g(\overline{Y}) \subseteq g(gen_i)$ because $gen_i \subseteq \overline{Y}$. Moreover, Lemma 2 states that if $j \in c(gen_i)$, then $g(gen_i) \subseteq g(j)$. Thus, since $g(\overline{Y}) \subseteq g(gen_i)$, then $g(\overline{Y}) \subseteq g(j)$ holds too, and from Lemma 2 it also follows that $j \in c(\overline{Y})$. So, if $j \notin \overline{Y}$ holds, $\overline{Y}$ would not be a closed itemset because $j \in c(\overline{Y})$, 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 $\overline{Y}$, i.e. $Y_0 \subseteq \overline{Y}$.

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

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

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

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

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

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

Example 2: Looking at Fig. 1(b), for each closed itemset we can easily identify the unique sequence of order-preserving generators. For the closed itemset $\overline{Y} = \{A, B, C, D\}$, we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup \{A\}$, $Y_1 = c(gen_0) = \{A, C, D\}$, $gen_1 = \{A, C, D\} \cup \{B\}$, and, finally, $\overline{Y} = c(gen_1)$. Note that $Y_0 \prec Y_1 \not\prec \overline{Y}$. Thus closed itemsets are not necessarily generated following the lexicographic order. A simpler example regards the closed itemset $\overline{Y} = \{B, D\}$, where we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup \{B\}$, and $\overline{Y} = c(gen_0) = \{B, D\}$. □

A. 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:

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

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*. ■

B. Assessments

We have contributed a deep study on the the problem of duplicates in mining frequent closed itemsets. We have introduced the concept of sequence of order-preserving generators, and proved that this sequence always exists and is unique for each closed itemset. This sequence is obtained by always growing the closed itemsets encountered with single items respecting a given total order relationship. Generators that instead do not respect the order-preserving property can safely be pruned, since their closures would generate duplicate closed itemsets.

It is worth noting that, differently from other algorithms, our technique does not force any particular strict order in the visit of the lattice. Indeed, it entails a subdivision of the search space which produces completely independent subproblems, which can be solved in whatever order, and thus also in parallel.

Moreover, this duplicate avoidance technique is memory-efficient since it uses the tidlists associated with single items, instead of using the set of closed itemsets already mined. In the following we will show that its implementation is also time-efficient.

V. THE DCI_CLOSED ALGORITHM.

In the literature, datasets are identified as *dense* when transactions are long and contain strongly correlated items. Usually, very long frequent patterns can be extracted from dense datasets even if large support thresholds are used during the mining process. Conversely, datasets are defined as *sparse* when groups of items are not mutually correlated, and transaction are short. Even when low support thresholds are employed during the mining process, the resulting frequent patterns extracted from sparse dataset are not long.

Frequent closed itemsets are a lossless condensed representation of frequent itemsets, but in sparse datasets, the number of closed itemsets may be nearly equal to the number of frequent ones. Therefore, mining closed itemsets in sparse datasets may become remarkably more expensive than extracting all the frequent itemsets. To deal with this issue, DCI_CLOSED adopts different strategies to extract frequent closed itemsets from sparse and dense datasets. The two strategies are implemented within the procedures DCI_CLOSED$_s$(), suitable for sparse datasets, and DCI_CLOSED$_d$(), suitable for dense datasets. The original results discussed in Section IV are exploited within DCI_CLOSED$_d$().

In order to discriminate between sparse and dense datasets, DCI_CLOSED uses simple heuristics in the early phases of the mining process [12], [14], as detailed in the following.

DCI_CLOSED starts by scanning the input dataset $\mathcal{D}$ to determine the frequent single items $\mathcal{F}_1 \subseteq \mathcal{I}$, and builds the bitwise vertical dataset $\mathcal{VD}$ containing the various tidlists $g(i)$, $\forall i \in \mathcal{F}_1$. At the end of this step the items that have survived, i.e. those included in $\mathcal{F}_1$, are increasingly sorted with respect to their frequencies.

After this first step, DCI_CLOSED decides whether $\mathcal{VD}$ corresponds to either a dense or a sparse dataset. Since $\mathcal{VD}$ is bitwise, if the percentage of 1's is large, the dataset is soon classified as dense. Otherwise, DCI_CLOSED uses a heuristic method aimed at discovering if a large set of items are however strongly correlated, i.e. they appear together in a large set of transactions. To this end, it computes the percentage of the most frequent items which co-occur in the same transactions. If this percentage is greater than a fixed threshold, it considers the dataset as dense, sparse otherwise. It is worth considering that the last check also entails the reordering of the columns of $\mathcal{VD}$ which is at the basis of the optimization discussed in Section V-A.2.b.

In Section V-A we will detail the most interesting part of the DCI_CLOSED$_d$() algorithm, whereas in the following we only sketch the strategy adopted by the DCI_CLOSED$_s$() procedure, used to mine sparse datasets.

While DCI_CLOSED$_d$() adopts a *depth-first* exploration of the lattice of frequent itemsets, DCI_CLOSED$_s$() adopts a *level-wise* visit. One of the reasons 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 [14], [12], with a simple additional *closedness test* over the frequent itemsets discovered. Since a frequent k-itemset Y can be identified as closed if no superset of Y with the same support exists, i.e. if Y is not *subsumed* by any $(k+1)$-itemset, 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 Y', we mark as *non closed* all subsumed k-itemsets Y, i.e., such that $Y \subset Y'$ and $supp(Y) = supp(Y')$. At the end, we can finally identify as closed ones all the frequent k-itemsets Y that results to be *unmarked*.

A. 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 all the *valid generators* – i.e. both frequent and order-preserving – that can be 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 = $c(\emptyset)$, 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 $i \in$ POST_SET (lines 2–5). Note that, even if in the code items i are chosen according to order $\prec$ (line 3), this is not actually required since each possible generator originates an independent sub-problem. Both *infrequent* and *not order-preserving* generators, i.e. *invalid* generators, are promptly discarded (line 6), without computing their closures. Note that items $i \in$ POST_SET, which have been used to obtain these invalid generators, will no longer be considered in the following recursive calls. Then we compute the closures of valid generators only (lines 7–15). It is worth noting that each generator $new\_gen \leftarrow$ CLOSED_SET $\cup\ i$ is strictly extended by using all items $j \in$ POST_SET such that $i \prec j$, thus respecting the *order-preserving* property (see lines 3–4, and line 9). Moreover, all the items j, $i \prec j$, that are found to not belong to $c(new\_gen)$ are added to the POST_SET$_{New}$ (line 13), 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 16). From this new closed set, new generators and corresponding closed sets can be built, by recursively calling the procedure DCI_CLOSED$_d()$ (line 17).

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

Algorithm 1 DCI_CLOSED pseudocode

```
1: procedure DCI_CLOSED_d (CLOSED_SET, PRE_SET, POST_SET)
2:   while POST_SET ≠ ∅ do
3:     i ← min_≺(POST_SET)
4:     POST_SET ← POST_SET \ i
5:     new_gen ← CLOSED_SET ∪ i              ▷ Build a new generator
6:     if supp(new_gen) ≥ min_supp and
 :       ¬is_dup(new_gen, PRE_SET) then
 :                ▷ if new_gen is both frequent and order preserving
7:       CLOSED_SET_New ← new_gen
8:       POST_SET_New ← ∅
9:       for all j ∈ POST_SET do         ▷ Compute closure of new_gen
10:        if g(new_gen) ⊆ g(j) then
11:          CLOSED_SET_New ← CLOSED_SET_New ∪ j
12:        else
13:          POST_SET_New ← POST_SET_New ∪ j
14:        end if
15:      end for
16:      Write Out CLOSED_SET_New and its support
17:      DCI_CLOSED_d(CLOSED_SET_New, PRE_SET, POST_SET_New)
18:      PRE_SET ← PRE_SET ∪ i
19:    end if
20:  end while
21: end procedure
 :
 :
22: function is_dup(new_gen, PRE_SET)               ▷ Duplicate check
23:   for all j ∈PRE_SET do
24:     if g(new_gen) ⊆ g(j) then
25:       return TRUE             ▷ new_gen is not order preserving
26:     end if
27:   end for
28:   return FALSE
29: end function
```

CLOSED_SET (CLOSED_SET$_{New}$) monotonically increases, while the size of POST_SET (i.e., 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 17) 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 all $j \in$ OST_SET, used for computing $c(new\_gen)$, now follow i in the lexicographic order (line 9). This ensures that we will surely extend generator new_gen according to the *order-preserving* property. Note that while $c(new\_gen)$ is computed, we build POST_SET$_{new}$ as follow: POST_SET$_{new} = \{j \in POST_SET \mid i \prec j \ and\ j \notin X\}$. In other words, only items j that follow i in the lexicographic order, and that have not been already included in X, are added to POST_SET$_{new}$. This condition allows the recursive call of the procedure to only build new generators $X \cup j$, where the constraint $i \prec j$ surely holds, according to the hypothesis of Theorem 1.

The composition of PRE_SET depends instead on the sequence of the *valid* generators (i.e., the ones that have passed both frequency and duplicate tests) that are set included in

$new\_gen = Y \cup i$. If all these generators were valid, it would simply be composed of all the items j such that $j \prec i$ and $j \notin Y$. However, if invalid generators are discovered, this permits us to prune PRE_SET, as discussed later in Section V-A.1.

While the composition of POST_SET guarantees that the various generators will be produced according to the hypothesis of Theorem 1, the composition of PRE_SET guarantees that duplicate generators, i.e. *not order-preserving* ones, will be correctly pruned by function $is\_dup()$ (lines 22–29).

Since our algorithm clearly explores every possible *order-preserving* generator that can be built according to the hypothesis of Theorem 1, our algorithm is also *complete*, i.e. it is able to extract all the frequent closed itemsets without duplications.

Example 3 (Running example): We can show how DCI_CLOSED$_d()$ works by looking at the example of Fig. 1. The particular visit of the lattice is illustrated in Fig. 2, where we used the alphabetic order among item literals as the total order $\mathcal{R}$.

Once $c(\emptyset) = \emptyset$ is found, four generators can be constructed by adding a single item to $c(\emptyset)$, namely $\{A\}$, $\{B\}$, $\{C\}$, and $\{D\}$.

Suppose we first compute the closure of $gen = \emptyset \cup \{A\} = \{A\}$. Note that, since no items precede A in the lexicographic order, then its PRE_SET is empty, and thus we can conclude that gen is *order-preserving*. DCI_CLOSED$_d()$ checks if $g(A)$ is set-included in $g(j)$, $\forall j \in$ POST_SET (i.e., $g(B)$, $g(C)$, and $g(D)$), and discovers that $c(A) = \{A, C, D\}$.

DCI_CLOSED$_d()$ is then recursively called, with parameters CLOSED_SET = $\{A, C, D\}$, POST_SET = $\{B\}$, while PRE_SET is still empty. CLOSED_SET = $\{A, C, D\}$ is thus extended with B (its POST_SET), so obtaining a new generator $gen = \{A, C, D\} \cup \{B\} = \{A, B, C, D\}$. Since PRE_SET is empty, this generator is *order-preserving* by definition, but is also closed because POST_SET is now empty. Therefore, starting from generator $\{A\}$, DCI_CLOSED$_d()$ has carried out its first exploration of the search space, and has found two closed itemsets, first $\{A, C, D\}$ and then $\{A, B, C, D\}$. Note that these two closed itemsets have not been generated in order, since and $\{A, C, D\} \not\prec \{A, B, C, D\}$.

After this first recursive exploration, DCI_CLOSED$_d()$ starts solving another independent subproblem by exploring generator $gen = \emptyset \cup \{B\} = \{B\}$, where PRE_SET = $\{A\}$ and POST_SET = $\{C, D\}$. Since gen is *order-preserving* (this is checked by comparing $g(B)$ with its PRE_SET, i.e. with $g(A)$), its closure is computed. So we check if $g(B)$ is set-included in $g(C)$ and $g(D)$ (i.e., its POST_SET), and we have that $c(B) = \{B, D\}$. DCI_CLOSED$_d()$ is thus recursively called, with parameters CLOSED_SET = $\{B, D\}$, POST_SET = $\{C\}$, while PRE_SET is still equal to $\{A\}$. $\{B, D\}$ is thus extended with C, so obtaining a new generator $gen = \{B, D\} \cup \{C\} = \{B, C, D\}$. Since $g(\{B, C, D\}) \subset g(A)$, then gen is not *order-preserving*, and it is pruned with all its possible extensions.

Now DCI_CLOSED$_d()$ can start another exploration starting from generator $gen = \emptyset \cup \{C\} = \{C\}$, where PRE_SET = $\{A, B\}$ and POST_SET = $\{D\}$. Since generator $\{C\}$ is *order-preserving* (this is checked by comparing $g(C)$ with $g(A)$ and $g(B)$, i.e. with its PRE_SET), we try to compute its closure by comparing $g(C)$ with $g(D)$, i.e. its POST_SET. Since $\{C\}$ cannot be further extended, then it is closed ($c(\{C\} = \{C\})$. DCI_CLOSED$_d()$ is thus recursively called, with parameters CLOSED_SET = $\{C\}$, POST_SET = $\{D\}$, while PRE_SET is still equal to $\{A, B\}$. $\{C\}$ is thus extended with D, so obtaining a new generator $gen = \{C\} \cup \{D\} = \{C, D\}$. Since $g(\{C, D\}) \subset g(A)$, then this generator is not *order-preserving*, and can be pruned without considering its possible extensions.

Finally, DCI_CLOSED$_d()$ starts exploring the last generator $gen = \emptyset \cup \{D\} = \{D\}$, where PRE_SET = $\{A, B, C\}$ and POST_SET = $\emptyset$. Since gen is *order-preserving* (this is checked by comparing $g(D)$ with $g(A)$, $g(B)$, and $g(C)$, i.e. with its PRE_SET), it is not pruned. But we also can conclude that $\{D\}$ is also closed, since POST_SET = $\emptyset$. □

1) Pruning PRE_SET: We have seen that the cardinality of PRE_SET passed to the recursive call of DCI_CLOSED$_d()$ increases monotonically. Each time a generator new_gen is built by extending CLOSED_SET, PRE_SET is modified accordingly in order to obtain the set *pre-set*(new_gen) (see Definition 3).

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 18), i.e. generators that are not only frequent, but also order-preserving. In other words, the items that have been used to build *invalid* generators are evicted 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 $h, i \in$ *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 it by contrapositive: 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 $g(gen') \equiv g(gen' \cup i)$. We can also express gen' as $gen' = Y \cup Z \cup i'$, where $Z = Y' \setminus Y$. Hence we have that $g(Y \cup Z \cup i') \equiv g(Y \cup Z \cup i' \cup i)$. 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. Suppose to know that both i and h belong to *pre-set*(gen'), where $gen' = Y' \cup i'$.

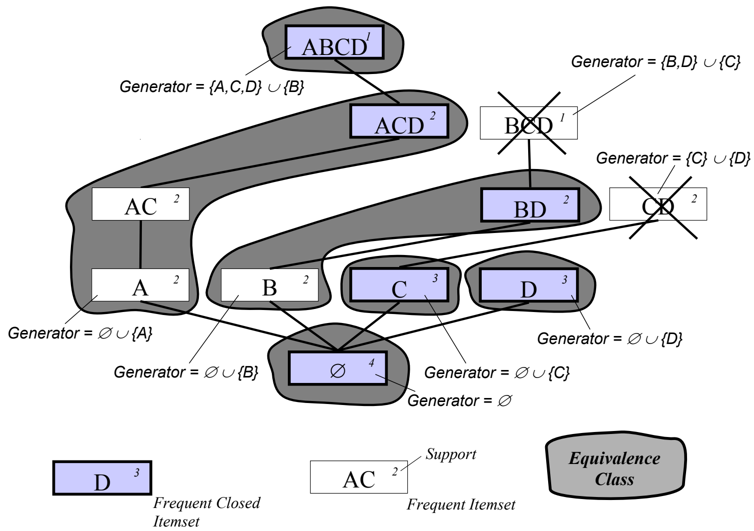


Fig. 2. The visit of the lattice performed by our algorithm to extract all the frequent closed itemsets ($min\_supp = 1$).

If we know that $g(Y \cup i) \subseteq g(h)$, where $Y \subset Y'$, we can safely remove i from *pre-set*(gen'). Consider in fact, that, if $g(gen') \subseteq g(h)$ holds, then we can deduce that gen' is not *order-preserving* and no further checks are needed. Otherwise, if $g(gen') \not\subseteq g(h)$, we know from the above lemma that $g(gen') \not\subseteq g(i)$ holds too. So, in order to check the *order-preserving* property of gen', in no cases it is needed to also check whether $g(gen') \subseteq g(i)$.

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 $g(gen') \not\subseteq g(i)$.

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. ■

2) Optimizations saving bitwise operations: DCI_CLOSED inherits the internal representation of our previous frequent set mining algorithms DCI [14] and kDCI [12]. The vertical dataset is represented by a bitmap matrix $\mathcal{VD}_{M \times N}$ 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 represents $g(i)$, the tidlist associated with item i.

Given a generator $X = Y \cup i$, the basic operations performed by DCI_CLOSED$_d$() are:

1) generating the tidlist $g(X)$, and computing its support $|g(X)|$;
2) performing duplication check;
3) computing its closure $c(X)$.

To efficiently perform all these operations, we exploit bitwise *AND* operations. In case (1), we exploit *AND* operations to compute $g(X) = g(Y) \cap g(i)$, in order to derive the tidlist of X and its 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 the inclusion of $g(X)$ in the various tidlists $g(j)$ associated with single items j that either precedes or follows i in the lexicographic order. All these inclusion checks are carried out by intersecting word-by-word $g(X) \cap g(j)$, and stopping at the first resulting word which is not equal to the corresponding word in $g(X)$.

We adopted many optimizations to reduce the number of bitwise operations actually performed by our algorithm. These optimizations, for the sake of simplicity, were not detailed in the pseudo-code of Algorithm 1.

The performance improvements resulting from each one of these optimizations are shown in Fig. 3, which plots, for several datasets, the actual number of bitwise *AND* operations performed DCI_CLOSED$_d$() as a function of the support threshold. In particular, Fig. 3(a,c) refer to the tidlist intersections to perform task (1), while Fig. 3(b,d) deal with the inclusion checks to complete tasks (2) and (3). In all the plots, the top curves represent the baseline case, when no optimizations are exploited at all. From the top curves to the bottom ones, we incrementally introduced the optimization techniques detailed in the following.

a) Dataset projection: Let us consider a closed set X and its tidlist $g(X)$. 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 mined, we can save work in the subsequent recursive calls of the procedure by projecting the vertical dataset $\mathcal{VD}$. This is carried out by deleting from $\mathcal{VD}$

all the columns corresponding to transactions not occurring in $g(X)$. Since this bit-wise projection is quite expensive, we limited 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 the plots of Fig. 3 we denoted this optimization with the label **projection**, and we compared its efficacy with the baseline case. We can see that the resulting number of bitwise operations performed is about halved with respect to the baseline case.

b) Datasets with highly correlated items: The columns of $\mathcal{VD}$ are reordered to profit of data correlation, which entails high similarity between the rows of the matrix when dense datasets are mined. As in [12][14], columns are reordered to maximize the size of a submatrix $\mathcal{VE}$ of $\mathcal{VD}$ having all its rows and columns identical. Every operation (e.g. intersection) involving tidlists within the submatrix $\mathcal{VE}$ is performed by exploiting the equality of the rows in $\mathcal{VE}$.

This is the most effective optimization regarding the intersection cost. In dense dataset submatrix $\mathcal{VE}$ is likely to be large w.r.t. to tidlist size, and it includes the most frequent items. This means that many frequent itemsets are mined within this submatrix, and we can thus strongly reduce the actual number of intersection operation performed. We labeled this optimization with **section eq** in the plots of Fig. 3. From the figure we can see that, exploiting this optimization, the number of operations performed further decreases of about one order of magnitude.

c) Reusing results of previous bitwise intersections: Besides the above ones, we introduced another optimization which exploits the depth-first visit of the lattice of itemsets.

Let us consider the case in which we have to check the closedness of some itemsets X. In order to determine that an itemset X is closed, the tidlist $g(X)$ must be compared with all the tidlists $g(j)$, for all items j contained in the PRE_LIST (and the POST_LIST) of X, i.e. the items that precede (or 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 the case X results to be closed, then for all j, we have that $g(X) \nsubseteq g(j)$. However, we might discover that a large section of the bitwise tidlist $g(X)$ is 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 too, 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.

The same holds in the case we have to perform intersections. If we know that $g_h(X) \subseteq g_h(j)$, then the first h words of the intersection $g(X) \cap g(j)$ will be the identical to the ones of $g(X)$.

In the plots of Fig. 3 we denoted this optimization with the label **included**. As it can be seen from these plots, this optimization is very effective in reducing the cost of inclusion operations of about one order of magnitude.

Another interesting remark regards the comparison between the number of operations actually executed to compute tidlists $g(X)$ (Fig. 3(a,c)), and the number of operations carried out for the various inclusion checks needed to check order-preserving properties and closures of generators (Fig. 3(b,d)). The two amounts, given a dataset and a support threshold, appear to be similar. This result might surprise the reader. One could think that the operations required for inclusion checks are the vast 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 tidlist $g(X)$ must be computed for all generators X, only those that survives the pruning steps described in Section V-A.1 are further processed for inclusion checks. Moreover, whereas tidlists must be completely scanned in order to produce $g(X)$, the same does not hold for inclusion checks: we can stop the check as soon as a single word of $g(X)$ results to be not included in the corresponding word of $g(j)$.

VI. Performance analysis

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. The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory.

A. Performance comparisons

We compared the performances of DCI_CLOSED with those achieved by two well known state-of-the-art algorithms: FP-CLOSE [7], and CLOSET+ [20]. FP-CLOSE is publicly available from the FIMI repository page `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 [6]. The FP-CLOSE and DCI_CLOSED algorithms were compiled with the gcc compiler available in the `cygwin` environment.

As shown in the plots reported in Fig. 4(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 than DCI_CLOSED. FP-CLOSE is effective in mining `pumsb*`, where its performance is close to that of DCI_CLOSED, but it is one order of magnitude slower in all the other tests. It is worth noting that in order to fairly compare the different implementations, the above plots refer to tests conducted with support thresholds that allowed FP-CLOSE and CLOSET+ never to run out of memory.

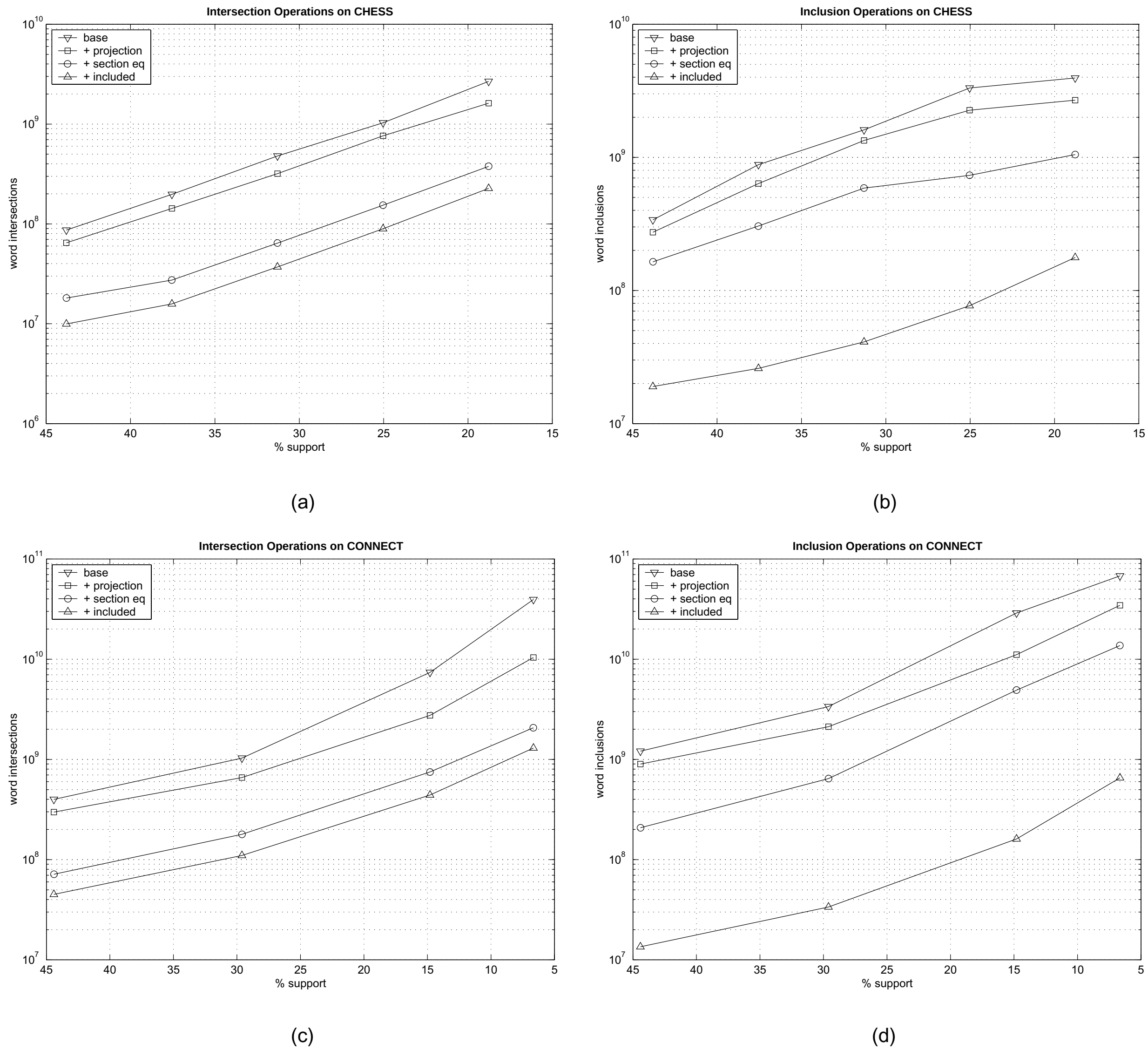


Fig. 3. Number of intersections *(a,c)*, and of inclusions operations *(b,d)*, actually performed by the DCI_CLOSED$_d()$ procedure when the various optimization techniques discussed in Section V-A.2 are exploited or not. The plots refers to the `chess` *(a,b)* and `connect` datasets *(c,d)*. The number of operations is plotted as a function of the support threshold.

TABLE I

RELATIVE PERCENTAGES OF TIME SPENT BY DCI_CLOSED AND FP-CLOSE CALCULATING SUPPORTS, CLOSURES AND IN CHECKING DUPLICATES ON VARIOUS DATASETS.

	DCI_CLOSED			FP-CLOSE		
dataset (min. supp.)	support	duplicates	closure	support	duplicates	closure
chess (25%)	26%	63%	11%	38%	60%	02%
connect (6%)	32%	60%	08%	27%	70%	03%
pumsb (45%)	41%	51%	08%	28%	70%	02%
pumsb* (8%)	56%	39%	05%	48%	47%	05%

B. Time efficiency of duplicate check

In order to compare the efficiency of our duplication check technique with respect to that adopted by competitor algorithms, we instrumented the publicly available FP-CLOSE code and our DCI_CLOSED code. Fig. 5 shows the absolute times spent checking for duplicates while mining two dense datasets, `chess` and `connect`, as a function of the support threshold. As we can see, our technique remarkably outperforms the one used by FP-CLOSE, with a speedup up to six when small support thresholds are used.

In Tab. I we report the relative percentage of time spent by DCI_CLOSED and FP-CLOSE performing each one of the three basic operations of the closed mining task: support count, duplication check, and closure computation. These percentages

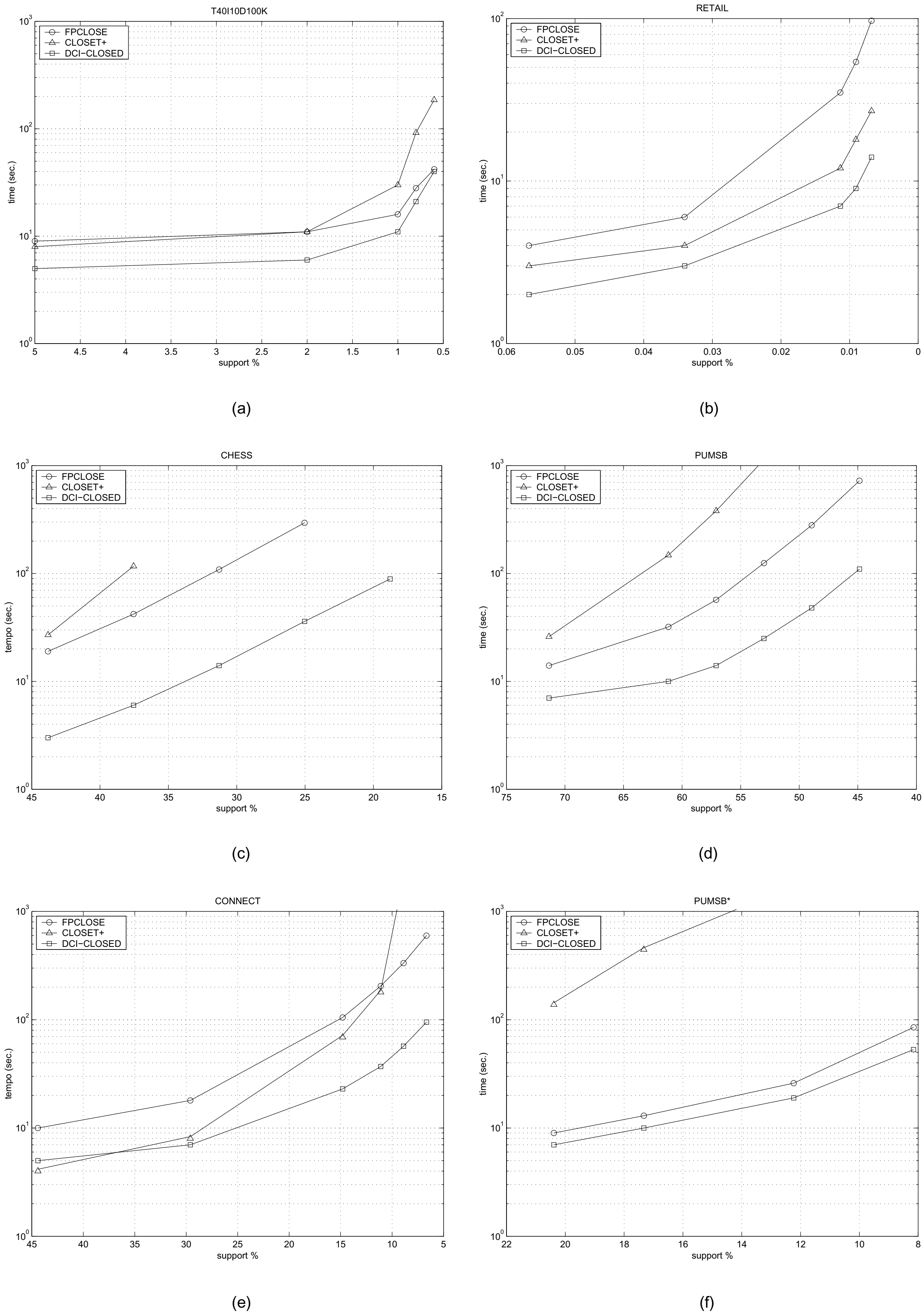


Fig. 4. Execution times (in seconds) required by FP-CLOSE, CLOSET+, and DCI_CLOSED to mine various publicly available datasets as a function of the minimum support threshold.

were computed by not considering the time spent by both implementations in other operations (e.g., input/output). In both algorithms, the time spent checking for duplicates is significant, but while in FP-CLOSE the computation cost is, in some cases, dominated by the duplication checking task, the same does not hold for DCI_CLOSED in which the time is more evenly distributed between duplication check and support computations. If we also consider that DCI_CLOSED remarkably outperforms FP-CLOSE in the total elapsed time, the advantage of using our duplicate checking technique is further highlighted.

The reasons of this good performance are discussed in depth in the following. The worst case of a duplicate check happens when a generator X is order-preserving. In such case, every tidlist in PRE_SET will be scanned without finding any of them to be included in $g(X)$. Let l be the number of words of each bitwise tidlist. Since PRE_SET is at most as big as $\mathcal{F}_1$, then the worst case cost of this inclusion check is proportional to $|\mathcal{F}_1| \times l$.

However, as soon as a generator is recognized as not order-preserving, it is pruned without considering any more the other tidlists associated with items in PRE_SET. Moreover, the inclusion check can stop the scan of each tidlist at the first word for which the inclusion check fails. Finally, thanks to its multiple optimizations (see Section V-A.2), DCI_CLOSED scans any tidlist only once along any path from the root to the leaf of the visit. This means that, even if, in principle, the cost of a *single* duplicate check operation involving a single tidlist in PRE_SET is directly proportional to l, it turns out to be considerably lower due to these optimizations. In practice, every duplicate check involves only a distinct and small portion of the tidlists associated with items in the PRE_SET of each generator.

To support this claim, in Tab. II we reported the average number of operations, i.e. the bitwise word intersections (Ops), needed to perform the duplicate check for each encountered generator as a function of the minimum support threshold. Note that the number of operations per generator grows linearly with the size of $\mathcal{F}_1$ as expected. However, the number of actual operations performed is considerably smaller than $|\mathcal{F}_1| \times l$, since it resulted to be much less even than l, i.e. the size of a single tidlist (1533 words, in this case).

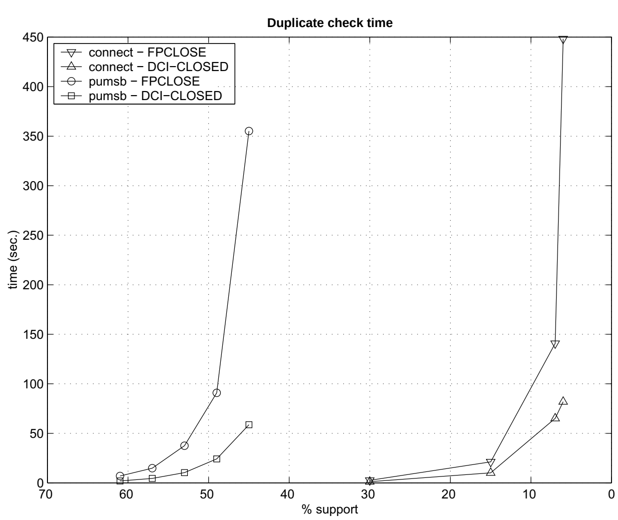


Fig. 5. Absolute times (in seconds) spent checking for duplicates by DCI_CLOSED and FP-CLOSE on datasets `chess` and `connect`, as a function of the support threshold.

TABLE II
PER GENERATOR AVERAGE NUMBER OF BITWISE WORD INTERSECTIONS (PUMSB DATASET).

min_supp	$\lvert\mathcal{F}_1\rvert$	l	$\lvert C\rvert$	Ops
40%	71	1533	44,450,000	54.2
50%	51	1533	7,120,000	38.5
60%	39	1533	1,150,000	26.5
70%	34	1533	240,000	23.8
80%	25	1533	2,000	16.9
90%	20	1533	1,400	16.5

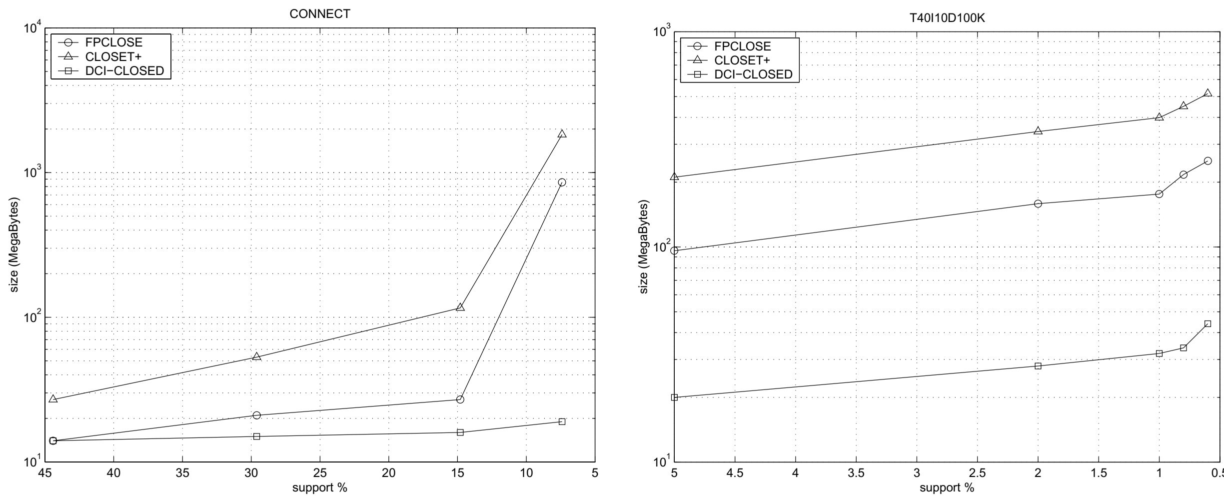
The table also shows, for each support threshold, the number of closed frequent itemsets extracted ($|C|$). Other algorithms hash the already mined closed itemset to detect duplicates, but since C grows exponentially, the hash performaces may degrade suddenly as confirmed in Fig. 5. Conversely, Tab. II shows that the number of operations performed by DCI_CLOSED is independent on the cardinality of C.

C. Space efficiency of duplicate check

Another issue related to the efficiency of our method for duplicate checking is main memory usage. Fig. 6(a) plots memory requirements of FP-CLOSE, CLOSET+ and our algorithm DCI_CLOSED when mining the `connect` dataset as a function of the support threshold. Note that the amounts of main memory used by CLOSET+ and FP-CLOSE rapidly grow when the support threshold is decreased, due to the huge number of closed itemsets generated that must be stored in the main memory. Conversely, the amount of memory used by DCI_CLOSED, which does not need to maintain the frequent closed itemsets mined in memory, is nearly constant. In Fig. 6(b) we plotted the results of the same test conducted on `T40I10D100K`, a sparse dataset where CLOSET+ is supposed to use its upward checking technique. Also in mining this sparse dataset, memory requirements of DCI_CLOSED are remarkably lower than those of its competitors. In this case, however, we can see that DCI_CLOSED memory requirements are not constant due to the need of maintaining in the main memory frequent k- and $(k+1)$-itemsets (see Section V).

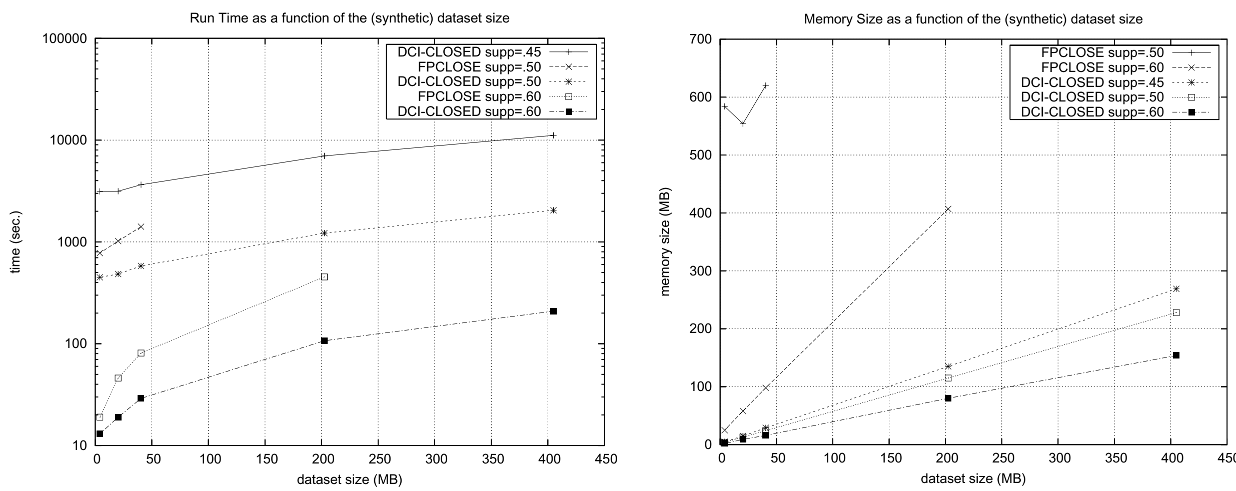
We can also analyze algorithm space requirement from a theoretical point of view. If we consider the algorithms that require to keep in the main memory the whole set of closed itemsets to perform duplicate check, then the size of the output is a lower bound on their space complexity. Conversely, the memory size required by an implementation based on our duplicate check technique 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



(a) (b)

Fig. 6. Amount of memory required by DCI_CLOSED, CLOSET+, and FP-CLOSE for mining a dense (a), and a sparse (b) dataset, as a function of the support threshold.



(a) (b)

Fig. 7. Execution times (in seconds) and memory requirements of DCI_CLOSED and FP-CLOSE for different support thresholds as a function of the size of the sample extracted from a large synthetic dataset.

current path of the lattice explored by the algorithm, and the tidlist of every frequent single item. With such information DCI_CLOSED can browse the search space by performing tidlists intersections, and check the order-preserving property of generators encountered.

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. Given $\mathcal{F}_1$, the deepest path along the lattice is composed of $|\mathcal{F}_1|$ 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 $2|\mathcal{F}_1|-1$. Since the length of a tidlist is equal to the number of transactions N in the dataset, the worst space complexity of our algorithm is

$$O\left((2|\mathcal{F}_1|-1)\times N\right).$$

Note that if this worst case occurred, the total number of closed itemsets should be $O(2^{|\mathcal{F}_1|})$, 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 more than closed itemsets, since each equivalence class contains many frequent itemsets. The number of frequent itemsets is instead comparable to the number of closed ones in sparse datasets, but the length of the largest closed frequent itemset that can be extracted from them is usually much smaller than $|\mathcal{F}_1|$. Note that DCI_CLOSED uses, however, 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 technique adopted, mining sparse datasets should not be a big issue from the point of view of memory occupation, because the number and length of frequent itemsets do not explode even for very low support thresholds.

D. Scalability

In this section we discuss results of tests conducted in order to verify the algorithm scalability, which is strongly related to the amount of main memory exploited by an algorithm. In other words, the demand of main memory is the main factor that may limit scalability of in-core closed itemset mining algorithms.

In order to verify the scalability of DCI_CLOSED, we ran the algorithm on random samples of different sizes, taken from a large dense dataset created with the IBM generator. The synthetic dataset occupies 405 MB, and contains $300,000$ transactions with an average length of 300. The number of distinct items is 1000, while the correlation factor 0.9.

The results of the tests conducted are reported in Fig. 7, which shows execution times (Fig. 7(a)) and memory requirements (Fig. 7(b)) of DCI_CLOSED and FP-CLOSE for different support thresholds, as a function of the size of the sample extracted from the synthetic dataset. Note that FP-CLOSE failed in most cases, since it ran rapidly out of memory when the support threshold was lowered. FP-CLOSE mined successfully only $1/10$ of the dataset with a minimum support threshold of 50%, and was not able to mine even $1/100$ of the dataset with a minimum support of 45%. On the other hand, DCI_CLOSEDcompleted successfully in all the tests. We can see from the plots that both execution times and memory requirements grow linearly with the size of the dataset. This means that DCI_CLOSED can mine any dataset as long as its vertical representation fits in the main memory, disregarding the number of closed itemsets extracted.

This is also confirmed by the results reported in Tab. III, which shows the number of closed itemsets discovered, and the execution times required by DCI_CLOSED to mine some dense datasets with an "absolute" support threshold of a single transaction. The tests with datasets `chess`, `connect` and `pumsb*` completed correctly after a huge, but still reasonable for the first two, amount of time, also using the lowest possible support threshold. The huge number of closed itemsets mined unequivocally demonstrates the limited scalability, with respect to the support threshold, of any other algorithms which, differently from DCI_CLOSED, need to store in the main memory the full set of closed itemsets.

VII. CONCLUSIONS

In this paper we have investigated 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 that allows duplicates to be effectively detected and discarded without needing to store the set of closed patterns in the main memory.

We have introduced a novel visit of the search space, based on the concept of sequence of order-preserving generators of frequent closed itemsets, and proved that this sequence always exists and is unique for each closed itemset. One of the positive effects of our visiting policy (and entailed duplicate checking technique) is that we are able to subdivide the search space thus producing completely independent subproblems, which can be solved in whatever order, and thus also in parallel.

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 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 the bitwise intersection of tidlists, 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-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. Finally, we accurately assessed DCI_CLOSED scalability, and we showed that it allows dense datasets to be effectively mined also with the lowest possible support threshold.

Acknowledgements

We acknowledge the financial support of the Project *Enhanced Content Delivery*, funded by the *Ministero Italiano dell'Università e della Ricerca*. Finally, we are grateful to the Associate Editor and the anonymous referees which allowed us to improve this paper with their helpful comments.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile*, Sept. 1994.

[2] J.-F. Boulicaut, A. Bykowski, and C. Rigotti, "Approximation of frequency queries by means of free-sets," in *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases, Lyon, France*, Sept. 2000.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, USA*, May 1997.

TABLE III

EXECUTION TIMES REQUIRED BY DCI_CLOSED, AND NUMBER OF CLOSED ITEMSETS EXTRACTED FROM SOME DENSE DATASETS MINED WITH THE LOWEST POSSIBLE SUPPORT THRESHOLD.

dataset	absolute min_supp	time	number of closed itemsets
chess	1 transaction	2,791 sec.	930,851,336
connect	1 transaction	9,486 sec.	1,414,804,579
pumsb*	1 transaction	45 days	8,998,642,796

[4] A. Bykowski and C. Rigotti, "A condensed representation to find frequent patterns." in *Proc. of the 20th ACM Symposium on Principles of Database Systems, Santa Barbara, California, USA*, May 2001.

[5] T. Calders and B. Goethals, "Mining all non-derivable frequent itemsets," in *Proc. of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases, Helsinki, Finland*, Aug. 2002.

[6] B. Goethals and M. J. Zaki, "Advances in frequent itemset mining implementations: report on fimi'03," *SIGKDD Explorations*, vol. 6, no. 1, pp. 109–117, 2004.

[7] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," in *Proc. of the ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA*, Dec. 2003.

[8] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA.*, May 2000.

[9] M. Kryszkiewicz, "Concise representation of frequent patterns based on disjunction-free generators." in *Proc. of the IEEE International Conference on Data Mining, San Jose, California, USA*, Nov. 2001.

[10] M. Kryszkiewicz and M. Gajek, "Concise representation of frequent patterns based on generalized disjunction-free generators," in *Proc. of the 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Taipei, Taiwan*, May 2002.

[11] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection," in *Proc. of the 9th ACM Int. Conf. on Knowledge Discovery and Data Mining, Edmonton, Canada*, 2002.

[12] C. Lucchese, S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "kdci: a multi-strategy algorithm for mining frequent sets," in *Proc. of the 2003 Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA*, Dec. 2003.

[13] H. Mannila and H. Toivonen, "Multiple uses of frequent sets and condensed representations," in *Proc. of ACM Int. Conf. on Knowledge Discovery and Data Mining, Portland, USA*, Aug. 1996.

[14] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and resource-aware mining of frequent sets," in *Proc. of the IEEE Int. Conference on Data Mining, Maebashi City, Japan*, Dec. 2002.

[15] J. S. Park, M.-S. Chen, and P. S. Yu, "An Effective Hash Based Algorithm for Mining Association Rules," in *Proc. of the ACM Int. Conference on Management of Data, San Jose, California*, May 1995.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *Proc. of 7th Int. Conf. on Database Theory, Jerusalem, Israel*, Jan. 1999.

[17] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Efficient mining of association rules using closed itemset lattices," *Information Systems*, vol. 24, no. 1, pp. 25–46, 1999.

[18] 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 IEEE Int. Conf. on Data Mining, San Jose, USA*, Nov. 2001.

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

[20] J. Pei, J. Han, and J. Wang, "Closet+: Searching for the best strategies for mining frequent closed itemsets," in *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington DC, USA*, August 2003.

[21] R. Taouil, N. Pasquier, Y. Bastide, L. Lajhal, and G. Stumme, "Mining frequent patterns with counting inference," *SIGKDD Explorations*, vol. 2, no. 2, December 2000.

[22] M. J. Zaki, "Mining non-redundant association rules," *Data Mining and Knowledge Discovery*, vol. 9, no. 3, pp. 223–248, 2004.

[23] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proc. of the 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Washington DC, USA*, Aug. 2003.

[24] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemsets mining," in *Proc. of the 2nd SIAM International Conference on Data Mining, Arlington, VA, USA*, Apr. 2002.

Claudio Lucchese received a summa cum laude master degree in Computer Science from the Ca' Foscari University of Venice in October 2003. In 2004 he held a fellowship at ISTI-CNR of Pisa, an Institute of the Italian National Research Council. He currently is a Ph.D. student in Computer Science at the the Ca' Foscari University of Venice, and a research associate at ISTI-CNR. He is mainly interested in data mining, privacy-preserving data mining and data mining techniques for information retrieval. Contact him at claudio.lucchese@dsi.unive.it; http://hpc.isti.cnr.it/∼claudio/.

Salvatore Orlando received a summa cum laude Laurea degree in Computer Science from the University of Pisa (Italy) in 1985, and a PhD in Computer Science from the same University in 1991. Then he was a post-doc fellow at the HP laboratories and the University of Pisa. In 1994 he joined as an assistant professor the Ca' Foscari University of Venice, where since 2000 he has been an associate professor. His research interests include parallel and distributed systems, Grid computing, parallel languages and programming environments, scalable algorithms and techniques for data and Web mining problems. Dr. Orlando has published numerous journal/conference papers and chapter books on these topics. He co-chaired the EuroPVM/MPI'03 Conference and the SIAM HPDM'05 Workshop, and served as a program committee member for many conferences concerning parallel and distributed computing, and data mining. Contact him at orlando@dsi.unive.it; http://www.dsi.unive.it/∼orlando/.

Raffaele Perego received his Laurea degree in Computer Science from the University of Pisa in 1985. He has been a contract professor of computer science at the University of Pisa since 1989. He is currently a researcher at ISTI-CNR, an Institute of the Italian National Research Council. His main research interests include high performance parallel and distributed computing, data mining, and information retrieval. Dr. Perego published many papers on these topics in several journals and proceedings of international conferences. Contact him at raffaele.perego@isti.cnr.it; http://raffaele.isti.cnr.it/.