The DCP algorithm for Frequent Set Counting

Salvatore Orlando[1], Paolo Palmerini[1,2], Raffaele Perego[2]

[1]Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy.

[2]CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy.

Abstract

In this paper we review the *Apriori* class of Data Mining algorithms for solving the Frequent Set Counting problem, and propose **DCP**, a new algorithm which makes several improvements on the classic *Apriori*. Our goal was to optimize the most time consuming phases of *Apriori* algorithms for problems characterized by short or medium length frequent patterns. For these problems, the initial iterations of the algorithm, during which small sets of items (itemsets) are counted, are the most expensive. The main enhancements introduced by **DCP** are the use of an innovative method for storing candidate itemsets and counting their support, and the exploitation of effective pruning techniques which significantly reduce the size of the dataset as execution progresses. We implemented and engineered several algorithms belonging to the *Apriori* class, and conducted in-depth experimental evaluations to compare them, by taking into account not only execution time, but also virtual memory usage and I/O activity. When possible, locality of data and pointer dereferencing were optimized due to their importance with respect to developments in computer architectures. The results confirm that our new algorithm, **DCP**, significantly outperforms the others previously proposed. Our test bed was a Pentium-based Linux workstation, while the datasets used for the tests were synthetically generated.

1 Introduction

The *Frequent Set Counting* (FSC) [1, 2, 3, 4, 5, 7, 12, 13, 14, 15, 17, 19] problem has been extensively studied as a method of unsupervised Data Mining [9, 10, 16] for discovering all the subsets of items (or attributes) that frequently occur in the transactions of a given database. Knowledge of the frequent sets is generally used to extract *Association Rules* stating how a subset of items influences the presence of another itemset in the transaction database. The process of generating association rules (*Association Mining*) has historically been adopted for *market-basket analysis*, where transactions are records representing point-of-sale data, while items represent products.

In this paper we focus on the FSC problem, which is the most time-consuming phase of the *Association Mining process*. An *itemset* is *frequent* if it appears in at least $s\%$ of all the n transactions of the database $\mathcal{D}$. The set of these transactions constitutes the *support* of the itemset. In this case we say that the itemset has a *minimum support*, i.e. it appears in at least min_sup transactions, where $min\_sup = s\%\ n$. When $\mathcal{D}$ and the number of items included in the transactions are huge, and we are looking for itemsets with small support, the number of frequent itemsets becomes very large, and the FSC problem very expensive to solve, both in time and space.

Apriori [5] is one of the most popular FSC algorithms. Although a number of other solutions have been proposed, it is still the most commonly recognized reference to evaluate FSC algorithm performances. *Apriori* iteratively searches frequent itemsets: at each iteration k, F_k, the set of all the frequent itemsets of k items (k-itemsets), is identified. In order to generate F_k, a *candidate* set C_k of potentially frequent itemsets is first built. By construction, C_k is a superset of F_k, and thus to discover frequent k-itemsets the support of all candidate sets is computed by scanning the entire transaction database $\mathcal{D}$. All the candidates with minimum support are then included in F_k, and the next iteration started. The algorithm terminates when F_k becomes empty, i.e. when no frequent set of k or more items is present in the database.

It is worth considering that the computational cost of the k-th iteration of *Apriori* strictly depends on both the cardinality of C_k and the size of $\mathcal{D}$. In fact, the number of possible candidates is, in principle, exponential in the number m of items appearing in the various transactions of $\mathcal{D}$. *Apriori* considerably reduces the number of candidate sets on the basis of a simple but very effective observation: a k-itemset can be frequent only if all its subsets of $k-1$ items are frequent. C_k is thus built at each iteration as the set of all k-itemsets whose subsets of $k-1$ items are all included in F_{k-1}. Conversely, k-itemsets that contain at least one infrequent $(k-1)$-itemset are not included in C_k.

An important algorithmic problem addressed by *Apriori* is to efficiently count the support of the candidate itemsets. During iteration k, all the k-subsets of each transaction $t \in \mathcal{D}$ must be determined and their presence in C_k be checked. To reduce the complexity of this phase, *Apriori* stores the various candidate itemsets in the leaves of a *hash-tree*, while suitable hash tables are placed in the internal nodes of the tree to direct the search of k-itemsets within C_k. The performance, however, only improves if the hash-tree splits C_k into several small disjointed partitions stored in the leaves of the tree. Unfortunately this does not happen for small values of k since the depth of the tree and thus the number of its leaves depends on k. Depending on the particular instance of the problem, itemsets of cardinality lower than 4 can contribute to even more than 90% of the total execution time. In particular, this property is true for problems where the maximal frequent itemsets are not very long.

The new algorithm proposed in this paper is called **DCP** (Direct Count of candidates & Pruned transactions). The algorithm significantly enhances the *Apriori* family of algorithms, and is aimed at solving the issues stated above for frequent itemsets of limited length. **DCP** exploits an innovative method for storing candidate itemsets and counting their support. The method is a generalization of the *Direct Count* technique used by *Apriori* for counting the support of unary itemsets, and allows the cost of the initial iterations of the algorithm to be reduced considerably, both in time and space. Moreover, **DCP** adopts a simple and effective pruning of $\mathcal{D}$, without using the complex hash filter used by DHP [15]. As an example of the heuristics used to prune $\mathcal{D}$, consider that the items that are not present in any itemset of F_k are not useful for the subsequent steps of the algorithm, and can thus be removed from $\mathcal{D}$. Similarly, transactions with less than k items can also be removed from $\mathcal{D}$, since they cannot contain any k-itemset. In **DCP** a pruned dataset $\mathcal{D}_{k+1}$ is thus written to the disk at each iteration k of the algorithm, and employed at the next iteration.

DCP does not address the issues arising in databases from which very long patterns can be mined [2, 7]. In this case we have an explosion of the candidate number, since all the 2^l subsets of each long maximal frequent itemset of length l have to be produced. More importantly, in these problems the supports of short frequent k-itemsets are usually very large, thus making the counting process very expensive. In order to speed up the counting phase of the algorithm, several techniques can be used to deduce that an itemset is frequent without actually counting its support [2, 7]. This allows the candidate sets to be pruned. Unfortunately, the exact supports of all the frequent itemsets need to be known in order to correctly compute the confidence of the derived association rules.

To validate our algorithm, we conducted in-depth experimental evaluations by taking into account not only execution times, but also virtual memory usage, I/O activity and their effects on the elapsed time. When possible, locality of data and pointer dereferencing were optimized due to their importance with respect to the recent developments in computer architectures. The experimental results showed that our new algorithm, **DCP**, outperforms the others. Our test bed was a Pentium-based Linux workstation, while the datasets used for tests were generated synthetically.

The paper is organized as follows. Section 2 introduces **DCP**, and discusses its features in depth. Section 3 details the method used to generate the synthetic datasets used in the tests, and reports the promising results obtained with **DCP**. In Section 4 we review some of the most recent results in the FSC field, and compare the **DCP** approach with others. Finally, Section 5 draws some conclusions and outlines future work. In Table I we report the notations adopted throughout the paper.

$\mathcal{D}$	transaction database
n	number of transactions in $\mathcal{D}$
m	number of items appearing in the n transactions of $\mathcal{D}$
t	a generic transaction in $\mathcal{D}$
t_i	an item identifier appearing at position i in transaction t
F_k	set of the frequent k-itemsets
C_k	set of candidate k-itemsets
c	a generic candidate itemset belonging to C_k
c_i	an item appearing at position i in the candidate itemset c
$\mathcal{D}_k$	pruned transaction database read at iteration k ($\mathcal{D}_1 = \mathcal{D}$)
M_k	set of the significant items appearing in the transactions of $\mathcal{D}_k$
$\overline{m}_k$	cardinality of M_k, i.e. $\overline{m}_k = \|M_k\|$

Table I: Symbols used in the paper.

2 The DCP algorithm

In this section we will discuss our new algorithm, **DCP** (candidate Direct Count & transaction Pruning), for solving the FSC problem.

As with the other algorithms of the *Apriori* class, **DCP** uses a *level-wise*, *counting-based* approach, according to which at each iteration k all the transactions are counted against a set of candidate k-itemsets. The database $\mathcal{D}$ is horizontally stored, i.e. each record corresponds to a transaction containing a set of bought items. Records have variable lengths, since only the integer identifiers, associated with the items actually belonging to a transaction, are included in the corresponding record. Moreover, we assume that item identifiers are stored in sorted order in both transactions and itemsets. Finally, the sets C_k and F_k, $F_k \subseteq C_k$ are stored as lexicographically ordered vectors of k-itemsets.

As in *Apriori*, at each iteration k, the set C_k of candidate itemsets is built on the basis of F_{k-1}. In this construction we exploit the lexicographic order of F_{k-1} to find pairs of $(k-1)$-itemsets sharing a common $(k-2)$-prefix. Due to this order, the various pairs occur in close positions within the vector storing F_{k-1}. The union of each pair is a k-itemset that becomes a candidate $c \in C_k$ only if all its subsets are included in F_{k-1}. Also in this case we can exploit the lexicographic order of F_{k-1}, thus checking whether all the subsets of c are included in F_{k-1} in logarithmic time.

The main enhancements introduced by **DCP** regard the exploitation of database pruning techniques, and the use of an innovative method for storing candidate itemsets and counting their support:

Pruning. **DCP** prunes the dataset at each iteration. In particular, a pruned dataset $\mathcal{D}_{k+1}$ is written to the disk at each iteration k, and employed at the next iteration. Note that this pruning entails a reduction in I/O activity as the algorithm progresses, since the size of $\mathcal{D}_k$ is always smaller than the size of $\mathcal{D}_{k-1}$. However, the main benefits come from the reduced computation required for subset counting at each iteration k, due to the reduced number and size of transactions.

Counting. In **DCP** we do not use a hash tree data structure for counting frequent sets. Instead we base our algorithm on directly accessible data structures, thus avoiding complex and expensive pointer dereferencing. Finally, **DCP** exploits high spatial locality in accessing its counting data structures.

2.1 Pruning the dataset

Two different pruning techniques are exploited. *Dataset global pruning* which transforms a generic transaction t, read from $\mathcal{D}_k$ into a pruned transaction $\hat{t}$, and *Dataset local pruning* which further prunes the transaction, and transforms $\hat{t}$ into $\ddot{t}$ before writing it to $\mathcal{D}_{k+1}$. While the former technique is original, the latter has already been adopted by DHP.

Dataset global pruning. At each iteration k, $k > 1$, the *Dataset global pruning* technique is applied to each $t \in \mathcal{D}_k$ to generate $\hat{t}$. The technique is based on the following argument: t may contain a frequent

k-itemset I only if all the $(k-1)$-subsets of I belong to F_{k-1}. Since searching F_{k-1} for all the $(k-1)$-subsets of any $I \subseteq t$ may be very expensive, a simpler heuristic technique, whose pruning effect is smaller, was adopted. In this regard, note that the $(k-1)$-subsets of a given k-itemset $I \subseteq t$ are exactly k, but each item belonging to t should only appear in $k-1$ of these k itemsets. Therefore, we derive a necessary (but weaker) condition to keep a given item in t.

The item t_i is retained in $\hat{t}$ if it appears in at least $k-1$ frequent itemsets of F_{k-1}.

To check the condition above, we simply use a *global* vector $G_{k-1}[\,]$ that is updated on the basis of F_{k-1}. Each counter of $G_{k-1}[\,]$ is associated with one of the m items of $\mathcal{D}_k$. For each frequent $(k-1)$-itemset belonging to F_{k-1}, the global counters associated with the various items appearing in the itemset are incremented. After all $(k-1)$-itemsets have been scanned, $G_{k-1}[j] = x$ means that item j appears in x frequent itemsets of F_{k-1}.

Counters $G_{k-1}[\,]$ are thus used at iteration k as follows. An item $t_i \in t$ is copied to the pruned transaction $\hat{t}$ only if $G_{k-1}[t_i] \geq k-1$. Then, if $|\hat{t}| < k$, the transaction is skipped, because it cannot possibly contain any frequent k-itemset.

Dataset local pruning. The *Dataset local pruning* technique is applied to each transaction $\hat{t}$ during subset counting. The arguments this pruning technique is based on, are similar to those of its global counterpart. Transaction $\hat{t}$ may contain a frequent $(k+1)$-itemset I only if all the k-subsets of I belong to F_k. Unfortunately, F_k is not yet known when our *Dataset local pruning* technique should be applied. However, since C_k is a superset of F_k, we can check whether all the k-subsets of any $(k+1)$-itemset $I \subseteq \hat{t}$ belong to C_k. This check could be made *locally* during subset counting of transaction $\hat{t}$.

Note that to implement the check above we should have to maintain, for each transaction $\hat{t}$, information about the inclusion of all the k-subsets of $\hat{t}$ in C_k. Since storing this information may be expensive, we adopted the simpler technique already proposed in [15], whose pruning effect is however smaller:

The item $\hat{t}_i$ is retained in $\ddot{t}$ if it appears in at least k candidate itemsets of C_k.

To check the condition above, for each transaction $\hat{t} = \{\hat{t}_1, \ldots, \hat{t}_{|\hat{t}|}\}$ to be counted against C_k, we use an array of $|\hat{t}|$ counters $L_k[\,]$, where each $L_k[i]$ is associated with a distinct item $\hat{t}_i \in \hat{t}$. The counter $L_k[i]$ is incremented every time we find that $\hat{t}_i$ is contained in a k-itemset of $\hat{t}$ which also belongs to C_k. At the end of the counting phase for transaction $\hat{t}$, we obtain a pruned transaction $\ddot{t}$ by removing from $\hat{t}$ all the items $\hat{t}_i$ for which $L_k[i] < k$. Transaction $\ddot{t}$ is then written to $\mathcal{D}_{k+1}$ only if $|\ddot{t}| \geq k+1$.

This pruning technique works because the presence of counters greater than or equal to k represents a necessary condition for the existence of a $(k+1)$-subset $I \subseteq \hat{t}$ all of whose k-subsets belong to C_k. In this case, in fact, since all the possible k-subsets of I are exactly $k+1$, but each item belonging to I may only appear in k of these $k+1$ subsets, the counters associated with all the items of I should be at least k.

2.2 Direct count of frequent $k-$itemsets

As discussed above, for problems characterized by short or medium length patterns, most of the execution time of *Apriori* is spent on the first iterations, when the smallest frequent itemsets are searched for. While *Apriori* uses an effective direct count technique for $k = 1$, the hash-tree data structure, used to count candidate occurrence for the other iterations, is not efficient for small values of k. For example, for $k = 2$ or 3, candidate sets C_k are usually very large, and the hash tree used by *Apriori* splits them into only a few partitions, since the depth of the hash tree depends on k.

Taking into account these considerations, for $k \geq 2$ we used a *Direct Count* technique which is based on a generalization of the technique exploited for $k = 1$. The technique is different for $k = 2$ and for $k > 2$ so we will illustrate the two cases separately.

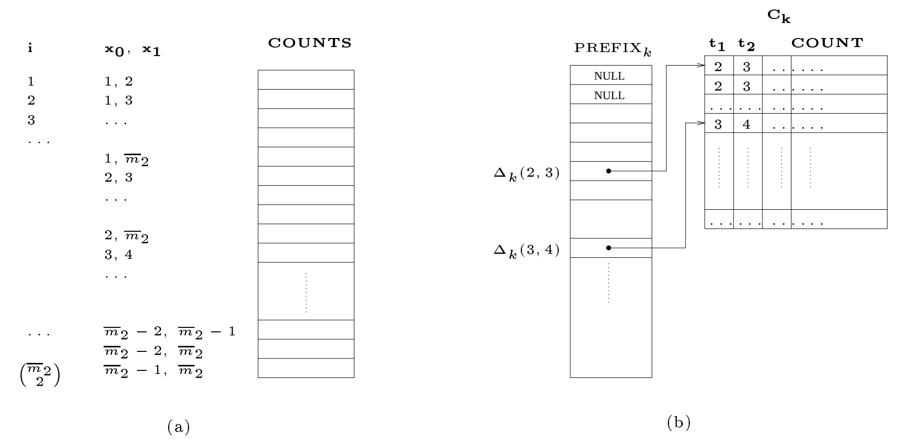


Figure 1: Data structures used to count (a) 2-itemsets and (b) $k > 2$-itemsets.

Counting frequent 2-itemsets. A trivial direct method for counting 2-itemsets can simply exploit a matrix of m^2 counters, where only the counters appearing in the upper triangular part of the matrix will actually be incremented [19]. Unfortunately, for large values of m, this simple technique may waste a lot of memory. In fact, since $|F_1|$ is usually less than m and $C_2 = F_1 \times F_1$, we have that $|C_2| = \binom{|F_1|}{2} << m^2$.

Before detailing the technique, note that at each iteration k we can simply identify M_k, the set that only contains the significant items that have not been pruned by the *Dataset global pruning* technique at iteration k. Let $\overline{m}_k = |M_k|$, where $\overline{m}_k < m$. In particular, for $k = 2$ we have that $M_2 = F_1$, so that $\overline{m}_2 = |F_1|$.

Our technique for counting frequent 2-itemsets is thus based upon the adoption of vector COUNTS[ ], which contains $|C_2| = \binom{\overline{m}_2}{2} = \binom{|F_1|}{2}$ counters (see Figure 1.(a)). The counters are used to accumulate the frequencies of all the possible itemsets in C_2 in order to obtain F_2.

It is possible to devise a perfect hash function to directly access the counters in COUNTS[ ]. Let $\mathcal{T}_2$ be a strictly monotonous increasing function $\mathcal{T}_2 : M_2 \to \{1, \ldots, \overline{m}_2\}$. A generic itemset $c \in C_2$, $c = \{c_1, c_2\}$, where $1 \le c_1 < c_2 \le m$, can thus be transformed into a pair $\{x_1, x_2\}$, where $x_1 = \mathcal{T}_2(c_1)$ and $x_2 = \mathcal{T}_2(c_2)$, so that $1 \le x_1 < x_2 \le \overline{m}_2$.

The entry of COUNTS[ ] corresponding to a generic candidate 2-itemset $c = \{c_1, c_2\}$ can thus be accessed *directly* by means of the following order preserving, minimal perfect hash function:

$$\Delta_2(c_1, c_2) = \mathcal{F}_2^{\overline{m}_2}(x_1, x_2) \;=\; \sum_{i=1}^{x_1-1} (\overline{m}_2 - i) \;+\; (x_2 - x_1) \;=\; \overline{m}_2(x_1 - 1) - \frac{x_1(x_1-1)}{2} + x_2 - x_1, \qquad (1)$$

where $x_1 = \mathcal{T}_2(c_1)$ and $x_2 = \mathcal{T}_2(c_2)$. Equation (1) can easily be derived by considering how the counters associated with the various 2-itemsets are stored in vector COUNTS[ ]. We assume, in fact, that the counters relative to the various pairs $\{1, x_2\}$, $2 \le x_2 \le \overline{m}_2$ are stored in the first $(\overline{m}_2 - 1)$ positions of vector COUNTS, while the counters corresponding to the various pairs $\{2, x_2\}$, $3 \le x_2 \le \overline{m}_2$, are stored in the next $(\overline{m}_2 - 2)$ positions, and so on. Moreover, the pair of counters relative to $\{x_1, x_2\}$ and $\{x_1, x_2 + 1\}$, where $1 \le x_1 < x_2 \le \overline{m}_2 - 1$, are stored in contiguous positions of COUNTS[ ].

Counting frequent k-itemsets. The technique above cannot be generalized to count the frequencies of k-itemsets when $k > 2$. In fact, although $\overline{m}_k$ decreases with k, the amount of memory needed to store $\binom{\overline{m}_k}{k}$ counters might be huge, since in general $\binom{\overline{m}_k}{k} >> |C_k|$.

Before detailing the technique exploited by **DCP** for $k > 2$, remember that, at step k, for every transaction t, we have to check whether any of its $\binom{|t|}{k}$ k-subsets belong to C_k. Adopting a naive approach, one could generate *all* the possible k-subsets of t and check each of them against *all* candidates in C_k. The hash tree used by *Apriori* is aimed at limiting this check to only a subset of all the candidates. A *prefix tree* is another data structure that can be used for the same purpose [14]. In **DCP** we adopted a limited and directly accessible *prefix tree* to select subsets of candidates sharing a given prefix, the first two items of the k-itemset. Note that, since the various k-itemsets of C_k are stored in a vector in lexicographic order, all the candidates sharing a common 2-item prefix are stored in a *contiguous section* of this vector. To efficiently implement our *prefix tree*, a directly accessible vector PREFIX$_k$[ ] of size $\binom{\overline{m}_k}{2}$ is thus allocated (see Figure 1.(b)). Each location in PREFIX$_k$[ ] is associated with a distinct 2-item prefix, and contains the pointer to the first candidate in C_k characterized by the prefix. More specifically, PREFIX$_k[\Delta_k(c_1, c_2)]$ contains the starting position in C_k of the segment of candidates whose prefix is $\{c_1, c_2\}$. As for the case $k = 2$, in order to specify $\Delta_k(c_1, c_2)$, we need to exploit a strictly monotonous increasing function $\mathcal{T}_k : M_k \rightarrow \{1, \ldots, \overline{m}_k\}$. $\Delta_k(c_1, c_2)$ can be thus defined as follows:

$$\Delta_k(c_1, c_2) \;=\; \mathcal{F}_2^{\overline{m}_k}(x_1, x_2)$$

where $x_1 = \mathcal{T}_k(c_1)$ and $x_2 = \mathcal{T}_k(c_2)$, while the hash function $\mathcal{F}_2^{\overline{m}_k}$ is that defined by Equation (1).

DCP exploits $PREFIX_k[\,]$ as follows. In order to count the support of the candidates in C_k, we select all the possible prefixes of length 2 of the various k-subsets of each transaction $t = \{t_1, \ldots, t_{|t|}\}$. Since items within transactions are ordered, once a prefix $\{t_{i_1}, t_{i_2}\}$, $t_{i_1} < t_{i_2}$, is selected, the possible completions of this prefix needed to build a k-subsets of t can only be found in $\{t_{i_2+1}, t_{i_2+2}, \ldots, t_{|t|}\}$. The contiguous section of C_k which must be visited is thus delimited by both PREFIX$_k[\Delta_k(t_{i_1}, t_{i_2})]$ and PREFIX$_k[\Delta_k(t_{i_1}, t_{i_2}) \;+\; 1]$. Note that this counting technique exploits high spatial locality. In fact, during subset counting relative to a given transaction, subsequent memory references are directed to contiguous addresses of the various sections of C_k.

We highly optimized the code to check whether each candidate itemset, selected through the prefix tree above, is included or not in $t = \{t_1, \ldots, t_{|t|}\}$. Our technique requires at most k comparisons. The algorithmic trick used is based on the knowledge of the number and the range of all the possible items appearing in each transaction t and in each candidate k-itemset c. In fact, this allows us to build a vector $POS[1 \ldots m]$, storing information about which items actually appear in t. More specifically, for each item t_i of t, $POS[t_i]$ stores the position of t_i in t, zero otherwise. The possible positions thus range from 1 to $|t|$. Therefore, given a candidate $c = \{c_1, \ldots, c_k\}$, c is not included in t if there exists at least one item c_i such that $POS[c_i] = 0$.

Moreover, since both c and t are ordered, we can deduce that c is not a subset of t without checking all the items occurring in c. In particular, given a candidate $c = \{c_1, \ldots, c_k\}$ to be checked against t, we can derive that $c \nsubseteq t$, even if c_i actually appears in t, i.e. $POS[c_i] \neq 0$. Suppose that the position of c_i within t, i.e. $POS[c_i]$, is such that

$$(|t| - POS[c_i]) \;<\; (k - i)$$

If the disequation above holds, then $c \nsubseteq t$ because c contains other $(k-i)$ items greater than c_i, but such items in t are only $(|t| - POS[c_i])$, $(|t| - POS[c_i]) \;<\; (k - i)$.

Remarks. Our technique is based on a directly accessible, limited prefix tree, and is particularly efficient for small values of k, where it really reduces the search space within C_k. Moreover, the technique adopted enhances spatial locality exploitation.

The counting technique is still effective for larger values of k. We have experimentally verified that, for each transaction t, most of the candidate k-itemsets that are scanned turn out to be set-included in t. Note that this effectiveness in selecting candidates to be checked against a t is also due to our pruning technique, which removes unimportant items from t. Finally, it is worth considering that our set inclusion check, which is based on the vector $POS[\,]$, permits the number of comparisons to be considerably reduced when a given candidate is not a subset of t.

2.3 Pseudo code of DCP.

The first iteration of **DCP**, during which the database is scanned to find frequent items, is the same as the *Apriori* one. F_1 is optimally built by counting all the occurrences of each item $i \in \{1, \ldots, m\}$ in every $t \in \mathcal{D}$.

```
1:  $global\_counter(G_1, F_1)$
2:  $k \leftarrow 2$
3:  $\overline{m}_2 \leftarrow |F_1|$
4:  **for all** $i \in [1, \overline{m}_2]$ **do**
5:      COUNTS[i] $\leftarrow 0$
6:  **end for**
7:  $\mathcal{D}_3 \leftarrow \emptyset$
8:  **for all** $t \in \mathcal{D}$ **do**
9:      $\hat{t} = global\_pruning(t, G_1, 2)$
10:     **if** $|\hat{t}| \geq 2$ **then**
11:         **for all** $\{\hat{t}_{i_1}, \hat{t}_{i_2}\} \subset \hat{t} \mid 1 \leq i_1 < i_2 \leq |\hat{t}|$ **do**
12:             $\Delta = \Delta_2(\hat{t}_{i_1}, \hat{t}_{i_2})$
13:             COUNTS[$\Delta$] $\leftarrow$ COUNTS[$\Delta$] + 1
14:         **end for**
15:     **end if**
16:     **if** $|\hat{t}| \geq 3$ **then**
17:         $\mathcal{D}_3 \leftarrow \mathcal{D}_3 \cup \hat{t}$
18:     **end if**
19: **end for**
20:
21: $F_2 = \{c_1, c_2 \in C_2 \mid COUNTS[\Delta_2(c_1, c_2)] \geq min\_sup\}$
22: $k \leftarrow 3$
```

Figure 2: Pseudo code of the second iteration of **DCP**.

Figure 2 shows the pseudo code of the second iteration of **DCP**, which exploits the direct count technique discussed in Section 2.2. We first update the counters used for the global pruning (line 1). The pseudo-code for subroutine *global_counter*(G_k, F_k) is not reported. The subroutine handles a vector of m counters $G_k[\,]$, and simply increments the counter $G_k[i]$ each time an item i is included in a frequent k-itemset of F_k. For each transaction read from the dataset, we prune all the items whose associated global counters are lower than 1 (line 9). Note that, since during the first iteration of the algorithm the dataset cannot be pruned, we still read transactions from $\mathcal{D}$. Then we generate all the 2-itemsets of the pruned transaction and increment the corresponding counters (lines 11-14). At step 2 it is not possible to apply the local pruning technique, since all the 2-itemsets of $\hat{t}$ are included in $C_2 = F_1 \times F_1$ by definition. Therefore we just add the transactions $\hat{t}$ to the pruned dataset $\mathcal{D}_3$ (line 17).

The pseudo-code for the following iterations $k \geq 3$ is shown in Figure 3. First we set the global counters on the basis of F_{k-1} (line 2). Then candidates are generated adopting the same procedure as in *Apriori* (line 3). Once the candidates have been generated, the limited prefix tree described in the above section is built (line 7). Then the various transactions are processed. After applying the global pruning technique (line 10), we start scanning the candidates to count how many of them are contained in any k-subset of $\hat{t}$ (lines 11-24). For this purpose, we generate all the possible prefixes of two items from the elements of $\hat{t}$, and we store the addresses of the first and the last candidates of C_k sharing this common prefix in variables *start* and *end* (lines 14-17). Then the subroutine *count_candidates*() (line 18) is invoked. It scans the contiguous section of C_k identified by *start* and *end* against $\hat{t}$. The set inclusion check of the various candidates against $\hat{t}$ employs the vector $POS[\,]$, which is initialized with the positions of all the items included in $\hat{t}$ (line 13). Note that subroutine *count_candidates*() also updates $L_k[\,]$, the per-transaction vector of counters exploited by the local pruning technique (line 20). $L_k[\,]$ is zeroed for each new transaction read from the dataset (line 12). Finally, Figure 4 shows the pseudo-code for subroutine *count_candidates*(), which exploits the technique previously discussed.

3 Performance evaluation

The results we present in this section were obtained running our implementations of *Apriori*, DHP, and **DCP**. Besides *Apriori*, we also compared **DCP** with DHP, since DHP is particularly efficient for problems characterized by short patterns, and it also adopts a database pruning technique. In addition, we also

```
while $F_{k-1} \neq \emptyset$ do
  $global\_counter(G_{k-1}, F_{k-1})$
  $C_k = apriori\_gen(F_{k-1})$
  if $C_k = \emptyset$ then
    return
  end if
  $\text{PREFIX}_k[\,] = init\_candidates(k, C_k)$
  $\mathcal{D}_{k+1} \leftarrow \emptyset$
  for all $t \in \mathcal{D}_k$ do
    $\hat{t} = global\_pruning(t, G_{k-1}, k)$
    if $|\hat{t}| \geq k$ then
      Initialize local counters $L_k[\,]$
      $POS[\,] = init\_positions(\hat{t})$
      for all $\{t_{i_1}, t_{i_2}\} \subseteq \hat{t} \mid 1 \leq i_1 < i_2 \leq |\hat{t}| - k + 2$ do
        $\Delta = \Delta_k(t_{i_1}, t_{i_2})$
        $start = \text{PREFIX}_k[\Delta]$
        $end = \text{PREFIX}_k[\Delta + 1]$ - 1
        $count\_candidates(|\hat{t}|, k, C_k, POS, start, end, L_k)$
      end for
      $\ddot{t} = local\_pruning(\hat{t}, L_k)$
      if $|\ddot{t}| \geq (k+1)$ then
        $\mathcal{D}_{k+1} \leftarrow \mathcal{D}_{k+1} \cup \ddot{t}$
      end if
    end if
  end for
  $F_k = \{c \in C_k \mid c.\text{COUNTS} \geq min\_supp\}$
  $k \leftarrow k + 1$
end while
```

Figure 3: Pseudo code of a generic iteration of **DCP** for $k \geq 3$.

tested a version of *Apriori*, called *Apriori*$_{DP}$, which enhances *Apriori* by employing the same dataset pruning technique introduced in **DCP**[*].

For the tests we used several synthetic datasets obtained with one of the most commonly adopted dataset generator [5]. The datasets we used in our experiments are characterized by the parameters reported in Table II, where T indicates the average transaction size, p the size of the maximal potentially frequent itemset, n the number of transactions, m the number of items, and L the number of maximal potentially frequent itemsets.

Database	T	p	n	m	L	Size (MB)
200k_t10_p4_m1k	20	4	200k	1k	2000	10
400k_t10_p8_m1k	10	8	400k	1k	2000	18
400k_t10_p8_m100k	10	8	400k	100k	2000	18
400k_t30_p8_m1k	30	8	400k	1k	2000	50
400k_t30_p8_m100k	30	8	400k	100k	2000	50
800k_t30_p8_m1k	30	8	800k	1k	2000	100
2000k_t20_p4_m1k	20	4	2000k	1k	2000	180
5000k_t20_p8_m1k	20	8	5000k	1k	2000	438

Table II: Values for parameters of the synthetic datasets used in the experiments

The test bed architecture used in our experiments was a Linux-based workstation, equipped with a Pentium III running at 450MHz, 512MB RAM, and an Ultra2 SCSI disk.

Pruning. We first compared our pruning technique with the one used by DHP for two different datasets (Table III.(a) and III.(b)). The fields *Number of transactions* and *Dataset size* appearing in a generic row k of the two tables both refer to the dataset written at iteration k, i.e. to the dataset $\mathcal{D}_{k+1}$ read at the next iteration. The dataset generated at each iteration by **DCP** is bigger than the one generated by

[*]In all the plots, the label identifying the classic *Apriori* will be AP, while the label identifying *Apriori*$_{DP}$ will be APdp.

```
Subroutine $count\_candidates(|\hat{t}|, k, C_k, POS, start, end, L_k)$
 1: for all $c = \{c_1, \ldots, c_k\}$ | $C_k[start] \le c \le C_k[end]$ do
 2:   // $c$ is included in the ordered segment of candidates
 3:   // comprised between $C_k[start]$ and $C_k[end]$
 4:   found $\leftarrow True$
 5:   $i \leftarrow 3$
 6:   while ( $(i \le k)$ AND found ) do
 7:     if ( $(POS[c_i] = 0)$ OR $(|\hat{t}| - POS[c_i] < k - i)$ ) then
 8:       found $\leftarrow False$
 9:     else
10:       $i \leftarrow i + 1$
11:     end if
12:   end while
13:   if found then
14:     $c$.COUNTS $\leftarrow c$.COUNTS $+ 1$
15:     for all $c_i \in c$ do
16:       $L_k[c_i] \leftarrow L_k[c_i] + 1$
17:     end for
18:   end if
19: end for
end Subroutine
```

Figure 4: Pseudo code of the subroutine *count_candidates*().

DHP. However, due to the *global dataset pruning* technique, **DCP** further reduces the size of the various transactions as soon as they are read from $\mathcal{D}_k$. Finally, the two tables highlight that, after a certain dimension of the candidate set ($k = 12$ in both cases), the effects of the two dataset pruning techniques are exactly the same.

I/O costs. Since in several instances of the FSC problem, input datasets are larger than main memory and are accessed repeatedly, these datasets must be maintained on disks and accessed in blocks by using an out-of-core technique. It is however possible to exploit modern OS features such as caching and prefetching [6], thus limiting I/O overheads. In particular, if a file is accessed sequentially, the OS prefetches the next block while the current one is being elaborated. Moreover, the OS stores blocks in the buffer cache, i.e. in the main memory, for possible future reuse.

To prove the benefits of I/O prefetching, we conducted some synthetic tests whose results are plotted in Figure 5. In these tests, we read a file of 256 MB in blocks of 4KB, and used an SCSI Linux workstation with 256 MB of RAM. Before running the tests, the buffer cache did not contain any blocks of the file. We artificially varied the per-block computation time, and measured the total elapsed time. The difference between the elapsed time and the CPU time actually used to elaborate all the blocks corresponds to the combination of CPU idle time and time spent on I/O activity. The plots in Figure 5.(a) correspond to tests where the file is read and elaborated only once, and the x-axis reports the total CPU time needed to elaborate all the blocks of the file. Note that an approximated measure of the I/O cost for reading the file can be deduced for null per-block CPU time ($x = 0$): in this case the measured I/O bandwidth is about 10MB/sec. As we increase the per-block CPU time, the total execution time does not increase proportionally, but remains constant up to a certain limit. After this limit, the computational grain of the program is large enough to allow the OS to overlap the computation and I/O.

This means that, when an application is compute-bound, there is a quasi complete overlapping between I/O activity and useful computation. In our case, an *Apriori* algorithm turns out to be compute-bound when counting candidates is very expensive. This often happens for instances of the FSC problem with small supports. Therefore, we argue that the performance problems observed in *Apriori* are often due to the extremely high computational cost of candidate counting, more than to the I/O cost of multiple dataset scans.

We repeated the experiment above by introducing disk writing, and also by iterating the elaboration performed on the dataset. Specifically, we only wrote half of the blocks read each time, thus reproducing

	N. Trans		D. Size (bytes)	
k	DCP	DHP	DCP	DHP
1	399979	399979	52401548	52401548
2	399979	399979	51358952	40981912
3	399973	399153	34139464	9055900
4	387094	200532	6310708	5860728
5	130528	128096	4934708	4934708
6	91845	104152	3241580	3241580
7	64403	64403	2643864	2643864
8	48031	50686	1548384	1548384
9	27332	27332	1224564	1224564
10	20760	20760	361984	361984
11	4381	4381	361984	361984
12	4381	4381	361984	361984
13	4381	4381	361792	361792
14	4378	4378	360500	360500
15	4359	4359	356036	356036
16	4297	4297	341444	341444
17	4105	4105	273924	273924
18	3261	3261	0	0

(a)

	N. Trans		D. Size (bytes)	
k	DCP	DHP	DCP	DHP
1	338603	338603	19695500	19695500
2	338388	338399	19432864	9484108
3	334452	229062	5792664	3198796
4	119390	76960	2748736	2464004
5	59367	57135	1943112	1943112
6	40790	43309	1341392	1341392
7	27037	27786	1049884	1049884
8	18651	20666	589696	589696
9	10648	10648	436924	436924
10	7486	7486	164120	164120
11	2259	2259	116800	116800
12	1414	1414	116680	116680
13	1412	1412	116616	116616
14	1411	1411	116208	116208
15	1405	1405	115200	115200
16	1391	1391	109804	109804
17	1320	1320	88284	88284
18	1051	1051	0	0

(b)

Table III: Pruning effect in **DCP** and DHP for (a) dataset 400k_t30_p8_m1k and $s = 0.75\%$, and for (b) dataset 400k_t10_p8_m1k and $s = 0.25\%$.

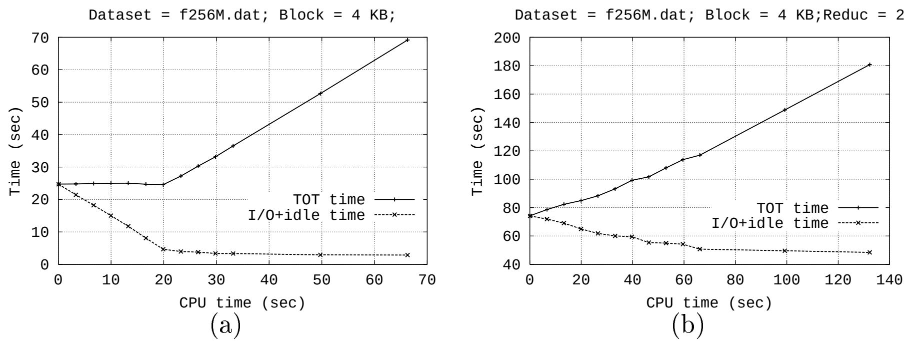


Figure 5: Total and I/O+idle time versus computational granularity. Dataset size is 256MB. In (a) the file is completely read and elaborated once. In (b) we have an iterated elaboration of the dataset, which is re-written at each step. Only half of the blocks read are however written back on disk and used at the next iteration.

the dataset pruning, as in DHP and **DCP**. The dataset read at each iteration is thus the one written at the previous iteration. Our test is iterated until the dataset becomes empty. Figure 5.(b)) refers to these tests. Also in this case, the x-axis reports the total CPU time needed to elaborate all the blocks read, and thus takes into account the iterated elaboration of the pruned dataset. Note that, due to write activities, the effect of I/O overlapping is less effective than in the previous test. However, when the written dataset becomes smaller than the main memory size, it can fit entirely into the buffer cache so that blocks can be read without actually accessing the disk. Finally, note that even if blocks are accessed in the buffer cache, I/O does not disappear, since blocks written to the cache must be synchronized with the disk.

Per-iteration Execution Times. From the analysis of the execution times for every step of the three algorithms studied in this work, we can observe that the behavior of the algorithms strictly depends on the dataset chosen. Besides the values of parameters which are known statically - such as the number of transactions, or the number of itemsets - also the internal correlations present in the transactions lead to significantly different behaviors.

The plots reported in Figure 6 show per-iteration execution times of **DCP**, $Apriori_{DP}$ and DHP. The two plots refer to tests conducted on the same dataset, for different values of the support s. First note that **DCP** always outperforms the other algorithms due to its more efficient counting technique. The performance improvement is impressive for small values of k. In particular from Figure 6.(a) we can see that the second iteration of **DCP** takes about 21 sec. with respect to 853 and 1321 sec. for DHP and $Apriori_{DP}$. Moreover, we can observe that DHP is effective only when the number of candidates can actually be reduced, otherwise the construction of its hash table (see Section 4) introduces unnecessary overheads. For a larger support ($s = 0.75\%$), in fact, DHP outperforms $Apriori_{DP}$ (see Figure 6.(a)), since it is able to prune more candidates than $Apriori_{DP}$. For a lower support ($s = 0.50\%$), since only a few transactions and items can be pruned, DHP only pays the overhead of constructing the hash table (see Figure 6.(b)). In other words, for low supports and small values of k, almost all the candidates selected by $Apriori_{DP}$ are found to be frequent.

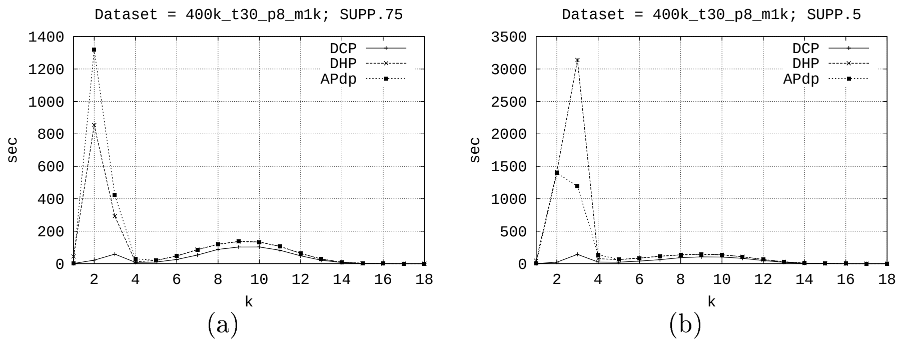


Figure 6: Per-iteration execution times of DHP, $Apriori_{DP}$, and **DCP** on dataset 400k_t30_p8_m1k with $s = 0.75\%$ (a) and $s = 0.50\%$ (b).

Total Execution Times. Figure 7 reports the total execution time obtained running *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on a dataset containing a small number n of transactions as a function of support s. Figures 7.(a) and (b) refer to datasets where the average transaction size is 10 and 20, with a fixed n. Changing the average transaction size has the effect of increasing the dataset size. In all the tests **DCP** performed better.

We also studied the influence of the total number m of items present in $\mathcal{D}$. In our synthetic datasets, as we increase m, and maintain constant sizes of database and transactions, we find smaller maximal frequent itemsets. In other words, the effect of increasing m is the reduction of the number of algorithm iterations. This is confirmed by our experiments, whose results are reported in Figure 8. In particular, for $m = 100k$, and $s = 1\%$ or 0.75%, we observed that $F_2 = \emptyset$. This is why DHP particularly underperforms in this case, since the additional cost of the hash table construction at the second iteration turns out to be useless.

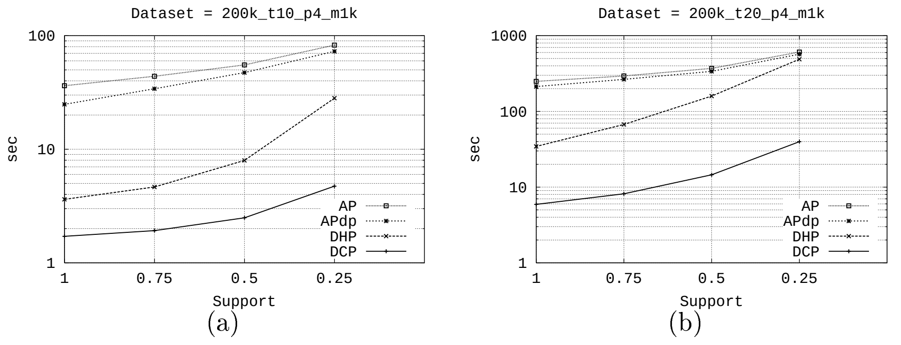
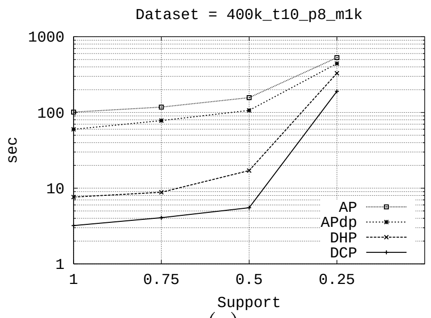
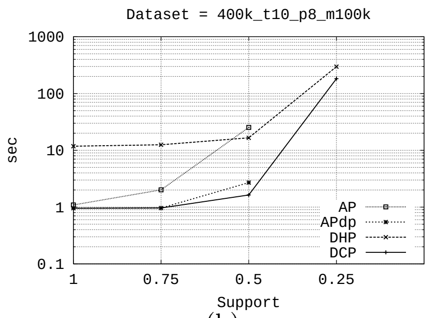


Figure 7: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on datasets 200k_t10_p4_m1k (a), 200k_t20_p4_m1k (b) for different supports.



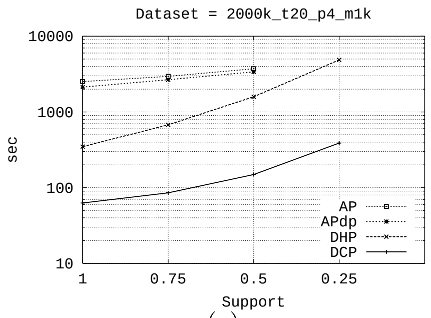
(a)



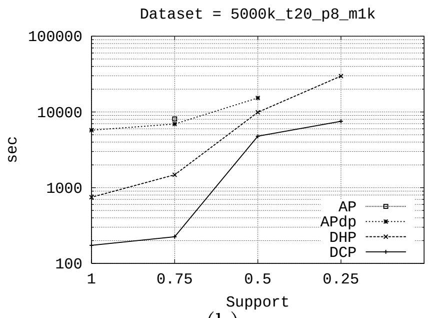
(b)

Figure 8: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on datasets 400k_t10_p8_m1k (a), 400k_t10_p8_m100k (b) for different supports.

Finally, we tested the algorithm behaviors on large datasets (see Figure 9). Specifically, dataset 5000k_t20_m1k is about as large as the total physical memory available on the workstation used. This test is important, since the disk buffer cache could not possibly contain the whole dataset. Thus we cannot take advantage of the presence in the buffer cache of blocks of the dataset read at previous iterations. In most of these tests, **DCP** execution times are better than the others by about one order of magnitude. Moreover, for very small supports (0.25%), some tests with the other algorithms were not able to allocate all the memory needed.



(a)



(b)

Figure 9: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on datasets 2000k_t20_p4_m1k (a), 5000k_t20_p8_m1k (b) for different supports.

DCP, on the other hand, requires less memory than its competitors, which exploit a hash tree for counting candidates. **DCP** is thus able to handle very low supports, without the *explosion* of the size of the data structures used. In this regard, Figure 10 plots the maximum amount of memory allocated by the various algorithms during the tests on two different datasets.

4 Related work

The algorithms of the *Apriori* class [4, 5, 11, 15] adopt a a level-wise behavior, which involves a number of dataset scans equal to the size of the largest frequent itemset. The frequent itemsets are identified by using a *counting-based* approach, according to which, for each level k, we count the candidate k-itemsets that are set-included in each transaction.

An alternative approach, used by several other algorithms, is the *intersection-based* one [8, 17, 19]. In this case the database is stored in a vertical layout, where each record is associated with a distinct item and is stored as a list of transaction identifiers (*tid-list*). *Partition*, an *intersection-based* algorithm that solves several FSC *local* problems on distinct partitions of the dataset is discussed in [17]. Dataset partitions are chosen which are small enough to fit in the main memory, so that all these local FSC

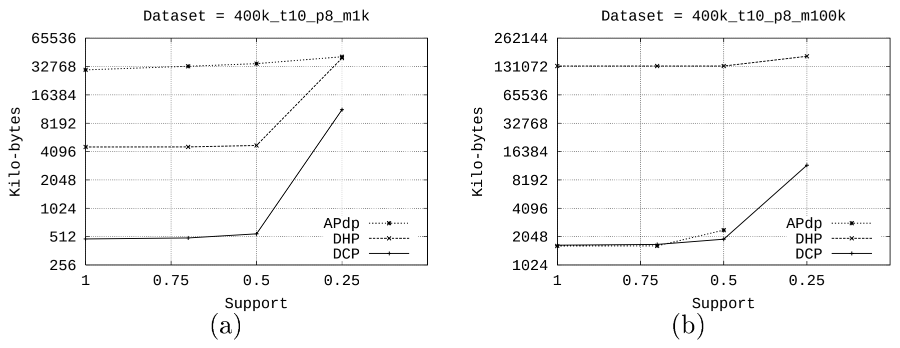


Figure 10: Maximal sizes of physical memory allocated during the execution for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on datasets 400k_t10_p8_m1k (a), 400k_t10_p8_m100k (b) for different supports.

problem can be solved with a single dataset scan. While, during the first scan, the algorithm identifies a superset of all the frequent itemsets, a second scan is needed to compute the actual global support of all the itemsets. This algorithm, despite the small I/O costs due to the reduced number of database scans, may generate a superset of all the frequent itemsets which is too large due to data skew, thus making the next iteration of the algorithm very expensive in terms of time and space. In [13] some methods to reduce the effects of this problem are discussed.

Dataset sampling [18, 20] as a method of reducing computation and I/O costs has also been proposed. Unfortunately, the FSC results obtained from a sampled dataset may not be accurate since data patterns are not precisely represented due to the sampling process.

Even though the DHP [15] algorithm is often cited only in terms of its ability to reduce the number of candidate sets, it also reduces I/O activity without modifying the level-wise behavior of *Apriori*. DHP, in fact, at each iteration re-writes a dataset which is smaller in size. The next iteration thus has to cope with a smaller input dataset than the previous one. The benefits are not only in terms of I/O, but also in terms of reduced computation for subset counting due to the reduced number and size of transactions. DHP prunes both the candidates and the dataset using a hash filtering technique. The hash filter of DHP requires the construction of a hash table H_{k+1} at each iteration k. H_{k+1} provides some approximate knowledge of the actual composition of F_{k+1}. For some datasets DHP is able to drastically reduce the difference between $|C_k|$ and $|F_k|$. This difference, however, is usually very high only for small values of k, e.g. $k = 2, 3$. Our **DCP** algorithm exploits a technique for database pruning similar to DHP, though it does not require the expensive construction of a hash table.

As regards the typical I/O behavior of algorithms for solving the FSC problem, in [6] we observed that when a dataset is sequentially and repeatedly scanned by reading fixed blocks of data, the OS *prefetching* and *buffering* turn out to be very effective. More specifically, OS *prefetching* is able to overlap computation and I/O activity provided that the computation granularity is large enough, while OS *buffering* is only useful for small datasets. As discussed in Section 3, for some datasets and small supports, the *Apriori* family of algorithms becomes *compute-bound*, so that techniques as those illustrated in [6] can be effectively exploited to overlap I/O time. Since algorithms that reduce dataset scans [13, 17, 18] increase the amount of work carried out at each iteration, we argue that further work has to be done to quantitatively analyze the advantages and disadvantages of adopting these algorithms rather than level-wise ones.

Recently, several new algorithms for solving the FSC problem have also been proposed [19]. Like *Partition*, they use an intersection-based approach by dynamically building a vertical layout database. While the *Partition* algorithm addresses these issues by relying on a blind subdivision of the dataset, Zaki's algorithms exploit clever dataset partitioning techniques that rely on a lattice-theoretic approach for decomposing the search space. For example, in the *Eclat* algorithm each subproblem is concerned with finding all the frequent itemsets which share a common prefix, which is in turn a frequent itemset. On the basis of the common prefix it is possible to determine a partition of the dataset, which will be composed of only those transactions which are included in the support of the prefix itemset. By recursively applying Eclat's search space decomposition we can thus obtain subproblems which can fit entirely into the main memory. However, Zaki obtains the best computational times with algorithms which mine only the maximal frequent itemsets (e.g. *MaxEclat, MaxClique*). While it is simple to derive all the frequent itemsets from the maximal ones, the same does not hold for their supports, which require a further counting step. Remember that the exact supports of all the frequent itemsets are needed to correctly compute association rule confidences.

In [7] the Max-Miner algorithm is presented, which aims to find maximal frequent sets by looking ahead throughout the search space. This algorithm is explicitly devised to work well for problems characterized by long patterns. When the algorithm is able to quickly find a long frequent itemset and its support, it prunes the search space and saves the work to count the support of all the subsets of this long frequent itemset. Moreover, it uses clever lower bound techniques to determine whether an itemset is frequent without accessing the database and actually counting its support. Hence, in this case too the method is not able to exactly determine the support of all frequent itemsets.

FP-growth [12], a very interesting algorithm able to find frequent itemsets in a database without candidate generation, has been recently presented. The idea is to build in memory a compact representation of the dataset where repeated patterns are represented once along with the associated repetition counters. The data structure used to store the dataset is called *frequent pattern tree*, or FP-tree for short. The tree is then explored without using the candidate generation method, which may explode for problems that have long itemsets. The algorithm is recursive. It starts by identifying paths on the original tree which share a common prefix. These paths are intersected by considering the associated counters. During its first steps, the algorithm determines long frequent patterns. Then it recursively identify the subsets of these long frequent patterns which share a common prefix. Moreover, the algorithm can also exactly compute the support of all the frequent patterns discovered. The authors only present results for datasets where the maximal frequent itemsets are long enough. We believe that for problems where the maximal frequent sets are not so long, the resulting FP-tree should not be so compact, and the cost of its construction would significantly affect the final execution time. A problem that has been recognized for FP-growth is the need to maintain the tree in memory. Solutions based on a partition of the database, in order to permit problem scalability, are also illustrated.

Finally we discuss Tree Projection [1], an algorithm which, like **DCP** prunes the transactions before counting the support of candidate itemsets, and also adopts a counting-based approach. The pruning technique exploited by **DCP** takes into account global properties regarding the frequent itemsets currently found, and produces a new pruned dataset at each iteration. Tree Projection, on the other hand, prunes (or projects) each transaction in a different way whenever the support of a specific group of candidates has to be counted. To this end candidates are subdivided into groups, where each group shares a common prefix. To determine the various projections of each transaction at level k, the transaction has to be compared with each frequent itemset at level $k-2$. These itemsets determine the common prefix of a given group of candidates of length k. This is the most expensive part of the algorithm. Once each projection has been determined, counting is very fast, since it only requires access to a small matrix of counters. The counting method used by **DCP** is different from Tree Projection. We need to access different groups of candidates (and associated counters) which share a common 2-item prefix. However, we do not need to scan all the possible 2-item prefixes. On the other hand, for each pair of items occurring in each transaction t we look up a directly accessible prefix table of C_k, and then scan the group of candidates to determine their inclusion in t. Note that the cost of this scan, due to the pruning operated on t, and the technique used to check the inclusion of each candidate in t, is very fast. Finally, we verified experimentally that a very large part of the candidates checked by **DCP** against t are actually included in t. In other words, **DCP** is able to optimally select the candidates to be checked against t.

5 Conclusions

In this paper we have reviewed the *Apriori* class of algorithms proposed for solving the FSC problem. These algorithms have often been criticized because of their level-wise behavior, which requires a number of scans of the database equal to the cardinality of the largest frequent itemset discovered. However we have shown that, in many practical cases, *Apriori*-like algoritms are not I/O-bound. When this occurs, the computational granularity is large enough to take advantage of the features of modern OSs, which allow computation and I/O to be overlapped. To speed up these compute-bound algorithms, the only solution is to make their most expensive computational part, i.e. the subset counting step, more efficient.

Moreover, as the DHP algorithm demonstrates, counting the number of dataset scans as a measure of the complexity of the *Apriori* algorithms does not take into account that very effective dataset pruning techniques can be devised. These pruning techniques can rapidly reduce the size of the dataset until it fits into the main memory. Nevertheless, our results showed that the efforts to reduce the size of the dataset and the number of candidates are of little use if the subset counting procedure is not efficient.

Our ideas to design **DCP**, a new algorithm for solving the FSC problem, originate from all the above considerations. **DCP** uses effective database pruning techniques which, differently from DHP, introduce only a limited overhead, and exploits an innovative method for storing candidate itemsets and counting their support. Our counting technique exploits spatial locality in accessing the data structures that store candidates and associated counters. It also avoids complex and expensive pointer dereferencing. A possible enhancement to **DCP** is to adopt a *blocking* technique [1] to improve temporal locality as well. In fact, for each transaction t, **DCP** explores specific sections of a vector storing C_k in order to count the support of candidates. Instead for single transactions, we could explore C_k for blocks of transactions, thus increasing the probability of repeated and close (in time) accesses to the same sections of candidates and associated counters.

As a result of its design, **DCP** significantly outperforms both DHP and *Apriori*. For many datasets the performance improvement is even more than one order of magnitude. More importantly, **DCP** exhibits better scalability. Consequently, due to its counting efficiency and low memory requirements, **DCP** can be used for finding low-support frequent itemsets within very large databases.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing*. To appear.

[2] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In *Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 108–118, Boston, MA, USA, 2000.

[3] R. Agrawal, T. Imielinski, and Swami A. Mining Associations between Sets of Items in Massive Databases. In *Proc. of the ACM-SIGMOD 1993 Int'l Conf. on Management of Data*, pages 207–216, Washington D.C., USA, 1993.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast Discovery of Association Rules in Large Databases. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[5] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[6] R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego. Implementation issues in the design of I/O intensive data mining applications on clusters of workstations. In *Proc. of the 3rd Workshop on High Performance Data Mining, in conjunction with IPDPS-2000, Cancun, Mexico*, pages 350–357. LNCS 1800 Spinger-Verlag, 2000.

[7] R. J. Bayardo Jr. Efficiently Mining Long Patterns from Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 85–93, Seattle, Washington, USA, 1998.

[8] Brian Dunkel and Nandit Soparkar. Data organization and access for efficient data mining. In *Proceedings of the 15th ICDE Int. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, 1999. IEEE Computer Society.

[9] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1998.

[10] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.

[11] E. H. Han, G. Karypis, and Kumar V. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):337–352, May/June 2000.

[12] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, USA, 2000.

[13] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proceedings of the 14-th Int. Conf. on Data Engineering*, pages 486–493, Orlando, Florida, USA, 1998. IEEE Computer Society.

[14] A. Mueller. Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison. Technical Report CS-TR-3515, Univ. of Maryland, College Park, 1995.

[15] 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 International Conference on Management of Data*, pages 175–186, San Jose, California, 1995.

[16] N. Ramakrishnan and A. Y. Grama. Data Mining: From Serendipity to Science. *IEEE Computer*, 32(8):34–37, 1999.

[17] A. Savasere, E. Omiecinski, and S. B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21th VLDB Conference*, pages 432–444, Zurich, Switzerland, 1995.

[18] H. Toivonen. Sampling Large Databases for Association Rules. In *Proceedings of the 22th VLDB Conference*, pages 134–145, Mumbai (Bombay), IndiaA, 1996.

[19] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, May/June 2000.

[20] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of Sampling for Data Mining of Association Rules. In *7th Int. Workshop on Research Issues in Data Engineering (RIDE)*, pages 42–50, Birmingham, UK, 1997.