Enhancing the *Apriori* algorithm for Frequent Set Counting*

Salvatore Orlando

Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy. E-mail: orlando@unive.it

Paolo Palmerini, Raffaele Perego

CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy. E-mail: {paolo.palmerini,raffaele.perego}@cnuce.cnr.it

Abstract

In this paper we review the *Apriori* class of Data Mining algorithms proposed for solving the Frequent Set Counting problem, and we propose **DCP**, a new algorithm which introduces several improvements to the classic *Apriori*. Our goal was the optimization of the most time consuming phases of Apriori algorithms, i.e. the initial iterations during which small itemsets are counted. The main enhancements regard the use of an innovative method for storing candidate itemsets and counting their support, and the exploitation of an effective pruning techniques which sensibly reduce the size of the dataset as execution progresses. We implemented and engineered several algorithms belonging to the *Apriori* class, and conducted accurate 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 accurately optimized due to their importance with respect to the recent developments in computer architectures. The experimental results confirm that our new algorithm, **DCP**, sensibly outperforms the others previously proposed. Our test bed was a Pentium-based Linux workstation, while the datasets used for tests were synthetically generated.

1 Introduction

The *Frequent Set Counting* (FSC) [1] problem has been extensively studied as a method of unsupervised Data Mining [6, 7, 12] for discovering all the subsets of items (or attributes) that frequently occurs in the transactions of a given database. Knowledge on 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 concentrate our attention 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 *min_sup* transactions of the database $\mathcal{D}$. In this case we say that the itemset has a *minimum support*, where the support of an itemset is the set of all the transactions in $\mathcal{D}$ which actually includes the itemset itself. When $\mathcal{D}$ and the number of items included in the transactions are huge, and we are looking for itemsets which are not very frequent (i.e., *min_sup* is small w.r.t. the number n of transactions in $\mathcal{D}$), the number of frequent itemsets becomes very large, and the FSC problem very expensive to solve both in time and space.

Apriori [3] is one of the most effective algorithms proposed for solving the FSC problem. *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 firstly 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 $\mathcal{D}$. Note that the number of possible candidates is, in

*__Paper ID 125__. Corresponding author: Raffaele Perego, CNUCE-CNR, Via V. Alfieri, 1, 56010 Ghezzano, Pisa, Italy. E-mail: raffaele.perego@cnuce.cnr.it. Tel.: (+39) 050 3152993. Fax: (+39) 050 3138091

principle, exponential in m, the number of items appearing in the various transactions of $\mathcal{D}$. *Apriori* strongly 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 at least contain an infrequent $(k-1)$-itemset are not included in C_k.

DHP [11], an algorithm presented as an enhancement of *Apriori*, not only tries to further reduce the size of C_k by pre-computing an approximated and larger F_k (through a hash table) during the previous algorithm iteration, but also recognizes the need to reduce the size of $\mathcal{D}$ as execution progresses. As an example of heuristics to prune $\mathcal{D}$, consider that items which are not present in any itemset of F_k are not useful for the subsequent steps of the algorithm, and can be thus removed from $\mathcal{D}$. Similarly, transactions with less than k items can be also removed from $\mathcal{D}$, since they cannot contain any k-itemset. In DHP 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.

An important algorithmic problem addressed by these algorithms is counting 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, both *Apriori* and DHP store 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 disjoint 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 problem instance, itemsets of cardinality lower than 4 can contribute to the total execution time even for more than the 90%.

In this paper we propose a new algorithm, **DCP** (Direct Count of candidates & Prune transactions), which introduces important enhancements aimed at solving the issues stated above. **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 both *Apriori* and DHP for counting the support of unary itemsets, and allows to strongly reduce both in time and space the cost of the initial iterations of the algorithm. Moreover, **DCP** adopts a simple and effective pruning of $\mathcal{D}$ without using the complex hash filter used by DHP.

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

The paper is organized as follows. In Section 2 we review some of the most recent results in the FSC field. Section 3 describes *Apriori* and DHP and discusses related implementation issues. Section 4 introduces **DCP**, and discusses in depth its peculiarities. Section 5 details the method used to generate the synthetic datasets used in the tests, and reports the promising results obtained with **DCP**. Finally, Section 6 draws some conclusions and outlines future work. In Table I we report the notation adopted throughout the paper.

$\mathcal{D}$	transaction database
t	generic transaction
m	number of items in $\mathcal{D}$
n	number of transactions in $\mathcal{D}$
F_k	frequent k-itemsets
C_k	candidate k-itemsets
H_k	hash table used by DHP at iteration k
$\mathcal{D}_k$	pruned transaction database read at iteration k ($\mathcal{D}_1 = \mathcal{D}$)
M_k	set of the significative items in $\mathcal{D}_k$
$\overline{m}_k$	cardinality of M_k

Table I: Symbols used in the paper.

2 Related work

The algorithms of the *Apriori* class [2, 3, 8, 11] are based on the same simple observation: if a given itemset in not frequent then none of its supersets can be frequent. They have a level-wise behavior: they start with $k = 1$ by evaluating singleton itemsets, and base the computations performed at step k on the results of the previous iteration $k - 1$. This level-wise behavior has been often criticized because of the consequent multiple scans of the dataset, one for each level. A lot of research has been thus devoted to minimize the number of dataset scans.

In [13] an algorithm that solves several FSC *local* problems on distinct partitions of the dataset is discussed. Partitions are chosen small enough to fit in main memory, so that all these local FSC problem can be solved with a single dataset scan. While during this first scan a superset of all the frequent itemsets is identified, a second scan is needed to compute the actual global support of all the itemsets. This algorithm may generate a too large superset of all the frequent itemsets due to data skew, thus making very expensive in time and space the next iteration of the algorithm. In [9] some methods to reduce the effects of this problem are discussed.

Dataset sampling as a method of reducing computation and I/O costs has also been proposed. Unfortunately, the FSC results obtained from a sampled dataset can not be accurate since data patterns are not precisely represented due to the sampling process. An algorithm has been proposed [14] that, although based on sampling, is able to exactly find the frequent sets in the original dataset. The main idea is to solve the FSC problem on the sampled dataset by using a lowered frequency threshold in order to obtain a superset of the frequent itemsets. Then this superset, suitably augmented with other itemsets, is checked against the rest of the original dataset to compute the exact frequency of itemsets. The method is however probabilistic, and the sampling may miss some frequent itemsets. In the last case, the method becomes more expensive.

Even though the DHP [11] algorithm is often cited only for its capacity of reducing the number of candidate sets, it reduces I/O activity without modifying the level-wise behavior of *Apriori*. DHP, in fact, at each iteration re-writes a dataset of smaller size. The next iteration has thus to cope with a smaller input dataset than the previous one. The benefits are not only in terms of I/O, but also of reduced work in subset counting due to the reduced number and size of transactions. In [4] we observed that, when a dataset is sequentially scanned by reading fixed blocks of data, one can take advantage of the OS *prefetching*. Overlapping between computation and I/O activity can occur, provided that the computation granularity is large enough. Moreover, OS buffer cache virtualizes I/O disk accesses, and, more importantly, small datasets can be completely contained in buffer cache, i.e. in main memory. In this case, subsequent scans of the dataset do not entail I/O disk accesses. In summary, if granularity of computation is large enough and dataset are sequentially scanned, prefetching and buffer cache are able to hide I/O time.

In the following, when we will discuss our experimental results, we will see that on modern architectures, level-wise algorithms belonging to the *Apriori* family are *compute* and not *I/O-bound*. We think that performance issues of level-wise algorithms for the FSC problem are only partially related to the multiple dataset scans involved. Since algorithms that reduce dataset scans [13, 9, 14] entail increasing the amount of work that is carried out at each iteration, we argue that further work has to be done to quantitatively analyze advantages and disadvantages in adopting these algorithms rather than level-wise ones.

Recently some new algorithms [15] for solving the FSC problem were also proposed. They are not based on *Apriori* and adopt a different format of the database, which is organized as a set of lists composed of transaction identifiers (*tid-list*). Each list is associated with a distinct item, and contains the identifiers of the transactions where this item appears. The idea here is to reduce the scans of the dataset, and to partition the dataset on the basis of a decomposition of the original search space into smaller parts. This allows these partitions to be processed independently in memory. The algorithm uses a lattice-theoretic approach to reason about search space decomposition. The new approach proposed seems truly interesting, but the experimental evaluations have only compared the new algorithms with the classic *Apriori* [2] and the *Partition* [13] algorithms, which are not very efficient.

3 *Apriori*-based algorithms

This section reviews the classic *Apriori* and DHP algorithms. Some of our implementation choices are also detailed. We have in fact implemented and engineered both these algorithms in order to make an effective comparison with **DCP**.

3.1 The *Apriori* algorithm

As stated above, *Apriori* iteratively looks for frequent itemsets. At each iteration k, F_k, the set of frequent k-itemsets is identified. In order to generate F_k, a *candidate* set C_k of potentially frequent k-itemsets is first built. The occurrences in $\mathcal{D}$ of all the itemsets of C_k are then counted. Finally, the itemsets of C_k having a minimum support are included into F_k.

In the following, we illustrate the pseudo code of the algorithm. We separated the first algorithm iteration, during which F_1 is generated, from the following iterations of *Apriori*. Moreover, some important subroutines of the algorithm are detailed: the generation of the *candidate* set C_k starting from F_{k-1}, and the exploitation of a hash-tree to count the support of the candidate itemsets in C_k.

Searching frequent 1-itemsets. The first iteration of *Apriori*, whose pseudo code is shown in Figure 1, is very simple. F_1 is optimally built by counting all the occurrences of each item $i \in \{1, 2, \ldots, m\}$ in every $t \in \mathcal{D}$. To this end, an array of m positions is used to store the item counters. Occurrences are counted by scanning $\mathcal{D}$, and the items having minimum support are included into F_1.

```
for all i | 1 ≤ i ≤ m do
    COUNTS[i] ← 0
end for
for all t ∈ D do
    for all i ∈ t do
        COUNTS[i] ← COUNTS[i] + 1
    end for
end for
F_1 = {i| 1 ≤ i ≤ m | COUNTS[i] ≥ min_sup}
```

Figure 1: First iteration of the *Apriori* algorithm.

Searching frequent k-itemsets. Subsequent iterations of *Apriori* are much more complex. Figure 2 shows the pseudo code of the main steps performed by the *Apriori* algorithm.

```
k ← 2
while F_{k-1} ≠ ∅ do
    C_k = apriori_gen(F_{k-1})
    if C_k = ∅ then
        return
    end if
    build_hash_tree(C_k)
    for all t ∈ D do
        subset_and_count(C_k, t)
    end for
    F_k = {c ∈ C_k | c.COUNT ≥ min_sup}
    k ← k + 1
end while
```

Figure 2: Main loop of the *Apriori* algorithm.

Most part of the total execution time is spent within subroutines *apriori_gen*() and *subset_and_count*(). The first one generates C_k from F_{k-1}. The second one, which is called for each transaction t, checks whether the subsets of k items of t belong to C_k. To this end, subroutine *subset_and_count*() exploits a hash tree, built over C_k by subroutine *build_hash_tree*(C_k).

Generating candidate sets. The subroutine *apriori_gen*(F_{k-1}) generates C_k from F_{k-1}. C_k is a superset of F_k, and is built by observing that a k-itemset can be frequent only if all its subsets of $k-1$ elements are frequent (i.e. belong to F_{k-1}).

The pseudo code of the subroutine is shown in Figure 3, where we assume that each itemset c is stored in lexicographic order in a vector, i.e. $c[i] < c[i+1], \forall i \in \{1, \ldots, k-1\}$. The subroutine takes advantage from the fact that also the frequent $(k-1)$-itemsets in F_{k-1} are lexicographically ordered. This order simplifies the selection of the pairs $c_p, c_q \in F_{k-1}$. For each c_p in F_{k-1}, in fact, we look for c_q in the tuples which immediately follow c_p in F_{k-1}, and we stop the search when we find a c_q whose first $k-2$ items are not equal to the first $k-2$ items of c_p. Only if we find a c_q such that $c_p[i] = c_q[i]\ \forall i \in \{1, \ldots, k-2\}$, then we can create the k-itemset $c = c_p \bigcup c_q = \{c_p[1], \ldots, c_p[k-2], c_p[k-1], c_q[k-1]\}$. Note that, due to the same ordering of the tuples in F_{k-1}, checking whether the subsets of c are included in F_{k-1} (line 4 in Figure 3) can be done in logarithmic time.

Subroutine *apriori_gen*(F_{k-1})

1: $C_k \leftarrow \emptyset$
2: **for all** $c_p, c_q \in F_{k-1} \mid c_p[i] = c_q[i],\ \forall i \in \{1, \ldots, k-2\}$ **do**
3: $\quad c = c_p \bigcup c_q = \{c_p[1], \ldots, c_p[k-2], c_p[k-1], c_q[k-1]\}$
4: $\quad$ **if** $\forall\ \bar{c} \subset c \mid |\bar{c}| = k-1,\ \bar{c} \in F_{k-1}$ **then**
5: $\qquad C_k \leftarrow C_k \bigcup c$
6: $\quad$ **end if**
7: **end for**

end Subroutine

Figure 3: Pseudo code of the subroutine *apriori_gen*().

Counting candidate sets with a hash tree. After the creation of C_k, subroutine *build_hash_tree*(C_k) (see Figure 2) is called to create the tree whose leaves will contain pointers to the various candidate itemsets and the associated counters. More specifically, these leaves reference to distinct partitions of C_k. The hash function used to direct both the insertion of candidate itemsets and the search in C_k of transaction subsets is very simple. It is a function $hash(i) = i \bmod H$, $H < m$, where m is the number of items.

Subroutine *subset_and_count*(C_k, t) (see Figure 2) recursively traverses the tree from the root to the leaves, with every item in $t = \{i_1, \ldots, i_d\}$ chosen as a possible starting item of a candidate itemset[*]. The recursive behavior of the subroutine allows us to apply the same technique at each level of the tree: for example, at the second level, if i_1 is the item in t chosen at the first level (root) of the tree, we go on with the visit of the tree by choosing every items in t which follows i_1 as possible next items of a candidate itemset. When a transaction t reaches a leaf of the tree, all candidate itemsets are checked against t and their counters updated accordingly.

The advantage of using a hash tree for FSC should be an effective partitioning of C_k resulting in a reduction of the number of candidates against which a transaction t has to be checked. However, we can achieve this goal only if each transaction t ends all its recursive visits of the tree by only reaching a limited number of leaves, and these leaves contain a small number of candidate itemsets.

Unfortunately, for small values of k, we have that:

- the tree can only have a few levels and thus a few leaves. Therefore we are not able, using such a tree, to split C_k into a large number of small partitions;

- by growing the branch degree of the tree, i.e. by increasing the constant H of our hash function, we obtain a larger number of C_k's partitions.
  Unfortunately, this may not always bring directly proportional advantages in the measured performance of the routine *subset_and_count*(C_k, t). In fact a transaction, during its tree traversing, might now reach much more leaves than for small values of H;

[*] More precisely, since the items in t are ordered, and we are looking for candidate itemsets whose size is k, the starting items of each itemset can only be chosen in $\{i_1, \ldots, i_{d-k+1}\} \subset t$.

- in general, since k is small with respect to the size of t, there are a lot of t's subsets composed of k elements. The presence of several subsets increases the probability that during the visit of the tree the transaction t traverses several paths and reaches several leaves.

3.2 The DHP algorithm

DHP [11] introduces several enhancements to *Apriori*. The basic idea of DHP is pruning both the candidates and the dataset by means of an effective hash filtering technique. Remember that, in the first steps of *Apriori*, the cardinality of C_k is of crucial importance for performance. Adopting its filtering technique, DHP is able to drastically reduce the difference observed in *Apriori* between $|C_k|$ and $|F_k|$. This difference, however, is usually very high only for small values of k, e.g. $k = 2, 3$. DHP also re-write $\mathcal{D}$ at each iteration, thus progressively reducing the size of the dataset. In this case the advantages are twofold. The reduced number and size of transactions, along with the reduced dimension of C_k, improves the performance of the more expensive steps of the *Apriori* algorithm. Moreover, since for large values of k the size of $\mathcal{D}$ becomes often very small, $\mathcal{D}$ could be completely contained in main memory. If this is the case, the original *out-of-core* algorithm could become *in-core*, with obvious performance improvements. Note that this transformation is automatically achieved by modern OSs, even if the dataset continues to be virtually accessed from the disk [4]. In fact, in modern OSs, the main memory left unused by the kernel and the processes is employed as a *buffer cache* for block devices such as disks [5].

The hash filter of DHP requires the construction of a hash table H_{k+1} at each iteration k. H_{k+1} furnishes an approximate knowledge on the actual composition of F_{k+1}. H_{k+1} is build during iteration k as follows: while reading each transaction t all the $(k+1)$-subsets of t are hashed to a bucket of H_{k+1} through a suitable hash function. The counter associated with the bucket is thus incremented. At the end of iteration k, we know that a generic $(k+1)$-itemset may be frequent only if it is hashed to a bucket of H_{k+1} which stores a counter greater than or equal to min_sup. Note that this is a necessary condition, since there might be several different $(k+1)$-itemsets which conflict on the same bucket of H_{k+1}.

At iteration k, $k > 1$, the hash table H_k built at the previous iteration is thus exploited to remove from C_k those candidate itemsets whose bucket in H_k contains a counter that is smaller than min_sup. In addition, H_k is used also to prune transactions in order to generate, the pruned dataset $\mathcal{D}_{k+1}$.

By looking at the buckets of H_k in fact, DHP removes from each transaction t all those items which cannot belong to frequent $(k+1)$-itemsets. This is done by checking whether all the k-subsets of each $(k+1)$-subset of t can be frequent or not[†].

Given a generic k-itemset $\{x_1, \ldots, x_k\}$, the hash function $h_k(x_1, \ldots, x_k)$ used by DHP to build H_k is the following:

$$h_k(x_1, \ldots, x_k) = (\sum_{i=1}^{k} x_i \cdot A^{k-i-1}) \bmod s \tag{1}$$

where the two constants s and A define the filter. Constant s gives the number of different buckets in H_k, while A is generally set to be equal to the number of items m. For the above technique to work well, it is necessary that s is large enough to avoid conflicts. In [11] s is chosen as a power of 2 as close as possible to $\binom{m}{k}$. In our experiments the maximum value used for s was $2^{\lfloor \log_2 \binom{m}{k} \rfloor + 1}$. Of course, there is a tradeoff in the adoption of such large tables, which may entail sensible performance degradations due to virtual memory swapping activity.

Finally, it is worth noting that the construction of the hash table can be very expensive, both in space and time. So, besides the benefits of pruning through hash filtering, we have to consider also the drawbacks of the construction of the hash table. In particular, the space depends on s and thus on m and k, while the time depends on the average transaction length and the size of $\mathcal{D}$.

The DHP algorithm is advantageous over the classic *Apriori* only if the construction of the hash table at each iteration entails a large pruning of either C_k or $\mathcal{D}_k$. If this pruning does not occur, the construction of the hash table only introduces overheads. For this reason, DHP uses the hash filtering only for the first iterations of the algorithm, when C_k or $\mathcal{D}_k$ are very large. The condition to switch to the classic *Apriori* is finding a small number of buckets with minimum support in the hash table. This

[†]Note that F_k is still under construction, so it can not be used at this purpose.

means that for the following iterations the number of frequent itemsets is becoming small, and thus also the technique used by *Apriori* is able to generate small candidate sets without incurring in the hash filtering overheads.

4 The DCP algorithm

In this section we will discuss our new algorithm, **DCP** (candidate Direct Count & transaction Pruning), for solving the FSC problem. The main enhancements regard the exploitation of DHP-*like* pruning techniques, and the use of an innovative method for storing candidate itemsets and counting their support:

Pruning. Similarly to DHP, we introduced dataset pruning into *Apriori*. The level of pruning is not the same as in DHP, but in our approach we do not pay the cost of constructing the hash table;

Counting. We did not use a hash tree data structure for counting frequent sets. Instead we based our algorithm on directly accessible data structures. Note that *Apriori* already adopts a "direct count" technique for the first iteration of the algorithm, when frequent singleton itemsets are discovered. Finally, it is worth remarking that **DCP** exploits both spatial and temporal locality in accessing its data structures, also avoiding complex and expensive pointer dereferencing.

4.1 Pruning the dataset

As DHP, **DCP** generates at each iteration k a pruned dataset $\mathcal{D}_{k+1}$ which will be used at the next iteration. In general, $\mathcal{D}_{k+1}$ will contain fewer transactions than $\mathcal{D}_k$, and the average length of transactions will be smaller as well. 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 been already 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 arguments: t may contain a frequent k-itemset I only if all its $(k-1)$-subsets belong to F_{k-1}.

Since searching F_{k-1} for all the $(k-1)$-subsets of any $I \in t$ 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 $I \in t$ are exactly k, but each item belonging to t only appears in $k-1$ of these k itemsets. Therefore, we can derive that a necessary (but weaker) condition to keep a given item in t is that it appears in at least $k-1$ frequent itemsets belonging to F_{k-1}. To check this condition, we build a *global* vector $G_{k-1}[\,]$ of counters on the basis of F_{k-1}. Each counter 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. At the end we have that if the value stored in the counter $G_{k-1}[t_i]$ associated with a given item t_i, $1 \le i \le m$, is x, then t_i 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 belonging to t is kept in $\hat{t}$ only if $G_{k-1}[t_i]$ is greater than or equal to $k-1$. At the end of pruning, if $|\hat{t}| < k$, the transaction is skipped, because it can not surely contain any frequent k-itemset.

Dataset local pruning. The *Dataset local pruning* technique is applied during subset counting to each transaction $\hat{t}$. The idea on which this pruning technique is based has arguments similar to its global counterpart. Transaction $\hat{t}$ may contain a frequent $(k+1)$-itemset I only if all its k-subsets belong to F_k. All the items of $\hat{t}$ which do not appear in frequent k-subsets can be thus pruned. Unfortunately, F_k is not known when *Dataset local pruning* is applied. However, since C_k is a superset of F_k, a *weaker* necessary condition for pruning $\hat{t}$ is to check whether all the k-subsets of any $(k+1)$-itemset $I \in \hat{t}$ belong to C_k. Only the items of $\hat{t}$ at least included in a $(k+1)$-itemset I whose all k-subsets belong to C_k are kept in $\ddot{t}$. This pruning condition could *locally* be checked 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 [11], whose pruning effect is however smaller.

The technique is simply based on an array of $|t|$ *local* counters $L_k[\,]$. In particular, for each transaction $t = \{\hat{t}_1, \ldots, \hat{t}_{|t|}\}$ to be counted against C_k, we build a distinct array of counter $L_k[\,]$, 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-itemsets 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 it at least contains $k+1$ items.

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

4.2 Direct count of frequent $k-$itemsets

As discussed above, most part of the execution time of *Apriori* is spent on the first iterations, when the smallest frequent itemsets are searched for. Experimentally it can be seen that in most cases itemsets of cardinality lower than 4 contribute for more than 90% of the total execution time. While for $k = 1$ the direct count technique exploited within *Apriori* is very efficient, for $k = 2$ and 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. Moreover, the pruning technique adopted by *Apriori* during candidate generation is not effective for $k = 2$, and C_2 is exactly equal to $F_1 \times F_1$. The DHP hash filtering technique is able to reduce the cardinality of C_2. Nevertheless, in some cases this reduction is not enough and the hash tree used for counting can not be exploited efficiently. Moreover, the hash filtering technique is expensive, both in time and space.

Starting from 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.

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 be actually incremented [15]. Unfortunately, for large values of m, this simple technique may waste a lot of memory. In fact, we can note that $C_2 = F_1 \times F_1$ and thus $|C_2| = \binom{|F_1|}{2}$ which is lower than m^2.

Moreover, the items actually present in F_1 are less than m. In general, at each iteration k, we can identify the set M_k, which only contains the significative items that are not pruned by the *Dataset global pruning* technique at iteration k. Let $\overline{m}_k$ be equal to $|M_k|$, where $\overline{m}_k < m$. For $k = 2$ we have that $M_2 = F_1$, so that $\overline{m}_2 = |F_1|$.

Our technique for counting frequent 2-itemset is thus based upon the adoption of a vector COUNTS[ ] of $|C_2| = \binom{\overline{m}_2}{2} = \binom{|F_1|}{2}$ counters, which 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 from M_2 to $\{1, \ldots, \overline{m}_2\}$. A generic itemset $c \in C_2$, $c = \{c_1, c_2\}$, where $1 \leq c_1 < c_2 \leq m$, 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 $1 \leq x_1 < x_2 \leq \overline{m}_2$.

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

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

where $x_1 = \mathcal{T}_2(c_1)$ and $x_2 = \mathcal{T}_2(c_2)$. Equation 2 can easily be derived by considering how the counters associated with the various 2-itemsets are stored in vector COUNTS[ ] (see Figure 4.(a)). We assume, in fact, that the counters relative to the various pairs $\{1, x_2\}$, $2 \leq x_2 \leq \overline{m}_2$ are stored in the first $(\overline{m}_2 - 1)$ positions of vector COUNTS, while the counters corresponding to the various pairs $\{2, x_2\}$, $3 \leq x_2 \leq \overline{m}_2$, 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 $1 \leq x_1 < x_2 \leq \overline{m}_2 - 1$, are stored in contiguous positions of COUNTS[ ].

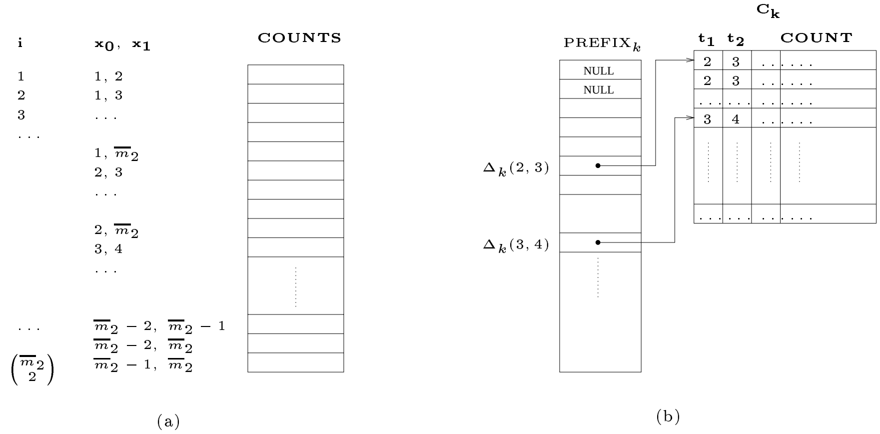


Figure 4: Data structures used to count (a) 2-itemsets and (b) $k > 2$-itemsets.

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 to store $\binom{\overline{m}_k}{k}$ counters might become huge, since $\binom{\overline{m}_k}{k}$ can become much larger than $|C_k|$.

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 at the same purpose [10]. 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 C_k is ordered, each subset of candidates sharing a common 2-item prefix is stored in a *contiguous section* of C_k. To efficiently implement our *prefix tree*, a directly accessible vector PREFIX$_k$[ ] of size $\binom{\overline{m}_k}{2}$ is allocated (see Figure 4.(b)). Each location in PREFIX$_k$[ ] contains the pointer to the first candidate in C_k characterized by the associated 2-item 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 $\mathcal{T}_k$ from M_k to $\{1, \ldots, \overline{m}_k\}$. $\Delta_k(c_1, c_2)$ 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 (2).

DCP exploits $PREFIX_k$[ ] as follows. We select all the possible prefixes of length 2 of any k-subsets of 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 all the k-subsets of t sharing this common prefix are exactly $\{t_{i_2+1}, t_{i_2+2}, \ldots, t_{|t|}\}$. The contiguous section of C_k which must be visited to check these k-subsets is delimited by PREFIX$_k[\Delta_k(t_{i_1}, t_{i_2})]$ and the next entry PREFIX$_k[\Delta_k(t_{i_1}, t_{i_2}) + 1]$. Moreover, the check can be limited to the suffix of t starting from item t_{i_2+1}, since the 2-item prefix $\{t_{i_1}, t_{i_2}\}$ is surely contained in each candidate belonging to the segment just selected. Note that our technique exploit high spatial locality. Subsequent memory references are directed to contiguous addresses, thus resulting in an efficient use of the memory hierarchies.

In addition, we highly optimized the code which checks each candidate itemset against $\{t_1, \ldots, t_{|t|}\}$. This check is in fact performed with 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. This knowledge allows in fact 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 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 an item c_i such that $POS[c_i] = 0$.

Moreover, since since both C_k and t are ordered, we can deduce that a candidate itemset is not included in a transaction without checking all the items.

This happen when, given a candidate itemset $c = \{c_1, \ldots, c_k\}$ to be checked against a transaction $t = \{t_1, \ldots, t_{|t|}\}$, an item c_i of c appears in t (i.e. $POS[c_i] \neq 0$), but its position $POS[c_i]$ is such that

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

In this case c cannot in fact be included in t since in c there are other $(k - i)$ items greater than c_i, while in t such items are only $(|t| - POS[c_i]) \ < \ (k - i)$.

Remarks. Our technique based on a directly accessible, limited prefix tree is particularly efficient for small values of k, where it effectively reduces the search space within C_k. Moreover, the technique adopted enhances locality exploitation, and for large values of k the above discussed use of a vector storing the item positions within a transaction permits the number of item comparisons to be considerably reduced.

```
1:  global_counter(G_1, F_1)
2:  k ← 2
3:  m̄_2 ← |F_1|
4:  for all i ∈ [1, m̄_2] do
5:     COUNTS[i] ← 0
6:  end for
7:  D_3 ← ∅
8:  for all t ∈ D_2 do
9:     t̂ = global_pruning(t, G_1, 2)
10:    if |t̂| ≥ 2 then
11:       for all {t_{i1}, t_{i2}} ∈ t̂ | 1 ≤ i1 < i2 ≤ |t̂| do
12:          Δ = Δ_2(t_{i1}, t_{i2})
13:          COUNTS[Δ] ← COUNTS[Δ] + 1
14:       end for
15:    end if
16:    if |t̂| ≥ 3 then
17:       D_3 ← D_3 ∪ t̂
18:    end if
19: end for
20:
21: F_2 = {c_1, c_2 ∈ C_2 | COUNTS[Δ_2(c_1, c_2)] ≥ min_sup}
22: k ← 3
```

Figure 5: Pseudo code of the second iteration of **DCP**.

4.3 Pseudo code of DCP.

Figure 5 shows the pseudo code of the second iteration of **DCP**, which exploits the direct count technique discussed in Section 4.2. 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 counter are greater than 1 (line 9). 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 6. 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 are generated, the limited prefix tree described in the above section is built (line 7). We then process each transaction. 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). To 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 called. It scans the contiguous

```
1: while F_{k-1} ≠ ∅ do
2:   global_counter(G_{k-1}, F_{k-1})
3:   C_k = apriori_gen(F_{k-1})
4:   if C_k = ∅ then
5:     return
6:   end if
7:   PREFIX_k[ ] = init_candidates(k, C_k)
8:   D_{k+1} ← ∅
9:   for all t ∈ D_k do
10:    t̂ = global_pruning(t, G_{k-1}, k)
11:    if |t̂| ≥ k then
12:      Initialize local counters L_k[ ]
13:      POS[ ] = init_positions(t̂)
14:      for all {t_{i_1}, t_{i_2}} ∈ t̂ | 1 ≤ i_1 < i_2 ≤ |t̂| − k + 2 do
15:        Δ = Δ_k(t_{i_1}, t_{i_2})
16:        start = PREFIX_k[Δ]
17:        end = PREFIX_k[Δ + 1] - 1
18:        count_candidates(|t̂|, k, C_k, POS, start, end, L_k)
19:      end for
20:      ẗ = local_pruning(t̂, L_k)
21:      if |ẗ| ≥ (k + 1) then
22:        D_{k+1} ← D_{k+1} ∪ ẗ
23:      end if
24:    end if
25:  end for
26:  F_k = {c ∈ C_k | c.COUNTS ≥ min_supp}
27:  k ← k + 1
28: end while
```

Figure 6: Pseudo code of a generic iteration of **DCP** for $k \geq 3$.

section of C_k identified by *start* and *end*. The fast scanning 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 7 shows the pseudo-code for subroutine *count_candidates*() which exploits the technique previously discussed.

5 Experimental results

The results we present in this section were obtained running our implementations of *Apriori*, DHP, and **DCP**. In addition, we also 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 [3]. The datasets we used in our experiments are characterized by the parameters reported in Table II, where T indicates the average transaction size, I 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.

The test bed architecture used in our experiments was a Linux-based workstation, equipped with a Pentium III running at 450MHz, 512MB RAM, and a 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 is bigger in **DCP** than in DHP. However,

[‡]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*$(|\hat{t}|, k, C_k, POS, start, end, L_k)$

for all $c = \{c_1, \ldots, c_k\} \;\mid\; C_k[start] \leq c \leq C_k[end]$ **do**
  // c is included in the ordered segment of candidates
  // comprised between $C_k[start]$ and $C_k[end]$
  found $\leftarrow True$
  $i \leftarrow 3$
  while ($(i \leq k)$ $\;AND\;$ $found$) **do**
    if ($(POS[c_i] = 0)$ $\;OR\;$ $(|\hat{t}| - POS[c_i] \;<\; k - i)$) **then**
      found $\leftarrow False$
    else
      $i \leftarrow i + 1$
    end if
  end while
  if found **then**
    c.COUNTS $\leftarrow c$.COUNTS $+ 1$
    for all $c_i \in c$ **do**
      $L_k[c_i] \leftarrow L_k[c_i] + 1$
    end for
  end if
end for

end Subroutine

Figure 7: Pseudo code of the subroutine *count_candidates*().

Database	T	I	n	m	L	Size (MB)
200k_t10_m1k	20	4	200k	1k	2000	10
400k_t10_m1k	10	8	400k	1k	2000	18
400k_t10_m100k	10	8	400k	100k	2000	18
400k_t30_m1k	30	8	400k	1k	2000	50
400k_t30_m100k	30	8	400k	100k	2000	50
800k_t30_m1k	30	8	800k	1k	2000	100
2000k_t20_m1k	20	4	2000k	1k	2000	180
5000k_t20_m1k	20	8	5000k	1k	2000	438

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

due to the *global dataset pruning* technique, **DCP** further reduces the size of each transaction t as soon as it is read from $\mathcal{D}_k$. Finally, looking at the two tables we can see that, after a certain dimension of the candidate set ($k = 12$ in both cases), the effect of the two dataset pruning techniques in 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 take advantage of modern OS features such as caching and prefetching [4], thus limiting I/O overhead. 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 main memory, for possible future reuses.

To show the benefits of I/O prefetching, we conducted some synthetic tests whose results are plotted in Figure 8. In these tests, we read a file of 256 MB in blocks of 4KB, and used a 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 for I/O activity. The plots in Figure 8.(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

k	N. Trans DCP	N. Trans DHP	D. Size (bytes) DCP	D. Size (bytes) 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)

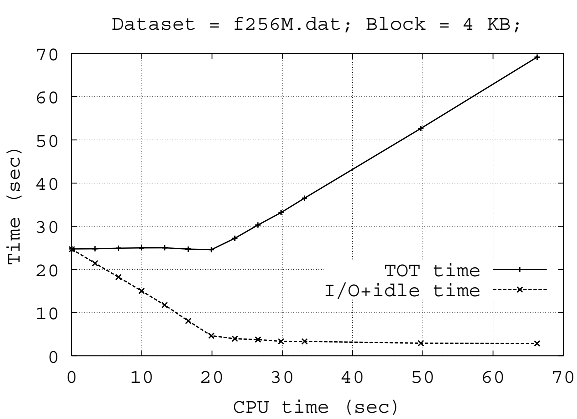
k	N. Trans DCP	N. Trans DHP	D. Size (bytes) DCP	D. Size (bytes) DHP
1	338603	338603	19695500	19695500
2	338388	338399	19432864	9484108
3	334452	229062	5792664	3198796
4	119390	76960	2748736	2464004
5	59367	57135	1943112	1943112
6	40790	43309	1341392	1341392
7	27037	27786	1049884	1049884
8	18651	20666	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)

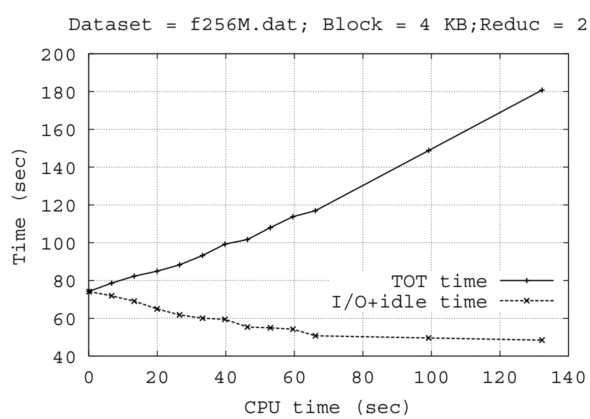
Table III: Pruning effect in **DCP** and DHP for (a) dataset 400k_t30_m1k and $min\_sup = 0.75\%$, and for (b) dataset 400k_t10_m1k and $min\_sup = 0.25\%$.

the program is large enough to allow the OS to completely overlap computation and I/O.

From the consideration above, we can deduce that, when an application is compute-bound, we can have a quasi complete overlapping between I/O activity and useful computation. In our case, an *Apriori* algorithm results to be compute-bound when the activity of counting candidate 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.



(a)



(b)

Figure 8: 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 a iterated elaboration of the dataset, which is re-written at each step. Only half of the read blocks are however written back on disk and used at the next iteration.

We repeated the experiment above by introducing disk writing, and by also iterating the elaboration performed on the dataset. Specifically, we only wrote half of the blocks read each time, reproducing in this way, the situation we have to face with when the transaction dataset is pruned, as in DHP and **DCP**. The dataset read at each iteration is thus the one written at the previous iteration. Our test is iterated till the dataset becomes empty. Figure 8.(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 main memory size, it can be completely contained in the buffer cache so that blocks can be read without actually accessing the disk. Finally, note that even if blocks are accessed in 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 determine sensibly different behaviors.

The plots reported in Figure 9 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 *min_sup*. First note that **DCP** always outperforms the other algorithms due to its more efficient counting technique. The performance improvements is impressive for small values of k. In particular from Figure 9.(a) we can see that the second iteration of **DCP** takes about 21 sec. with respect to the 853 and 1321 sec. of 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 the hash table introduces useless overhead. For a larger support ($min\_sup = 0.75\%$), in fact, DHP outperforms $Apriori_{DP}$ (see Figure 9.(a)), since it is able to prune more candidates than $Apriori_{DP}$. For a lower support ($min\_sup = 0.50\%$), since only few transactions and items can be pruned, DHP only pays the overhead of constructing the hash table (see Figure 9.(b)). In other words, for low supports and small values of k, we have that almost all the candidates selected by $Apriori_{DP}$ are found to be frequent.

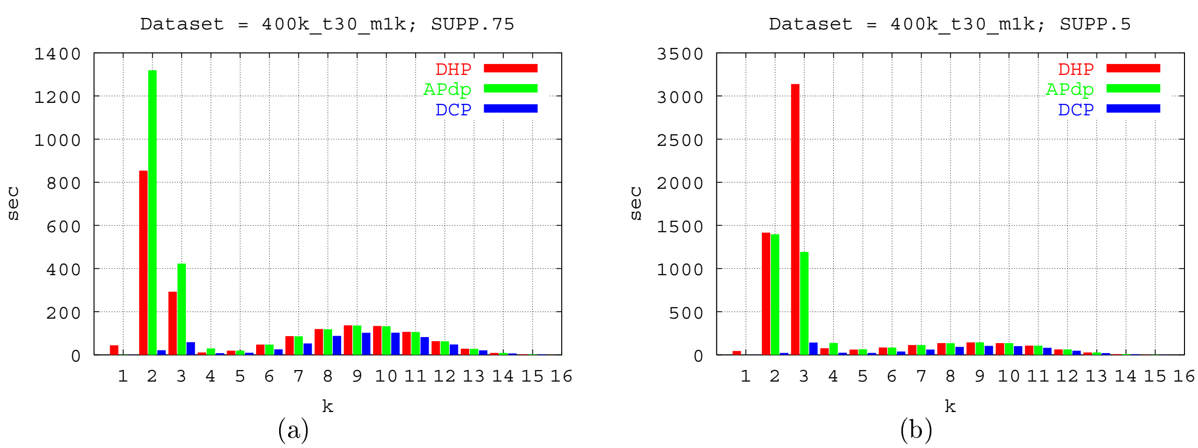
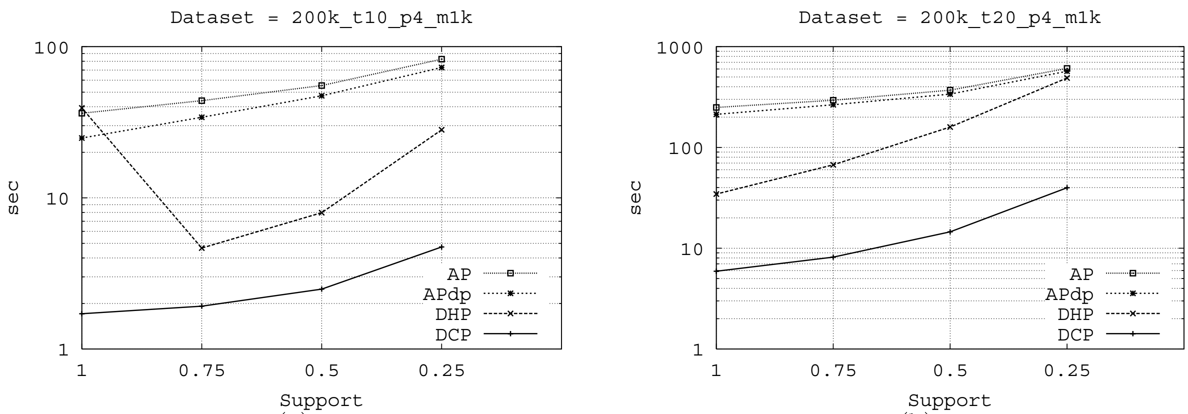


Figure 9: Per-iteration execution times of DHP, $Apriori_{DP}$, and **DCP** on dataset 400k_t30_n1k with $min\_sup = 0.75\%$ (a) and $min\_sup = 0.50\%$ (b).

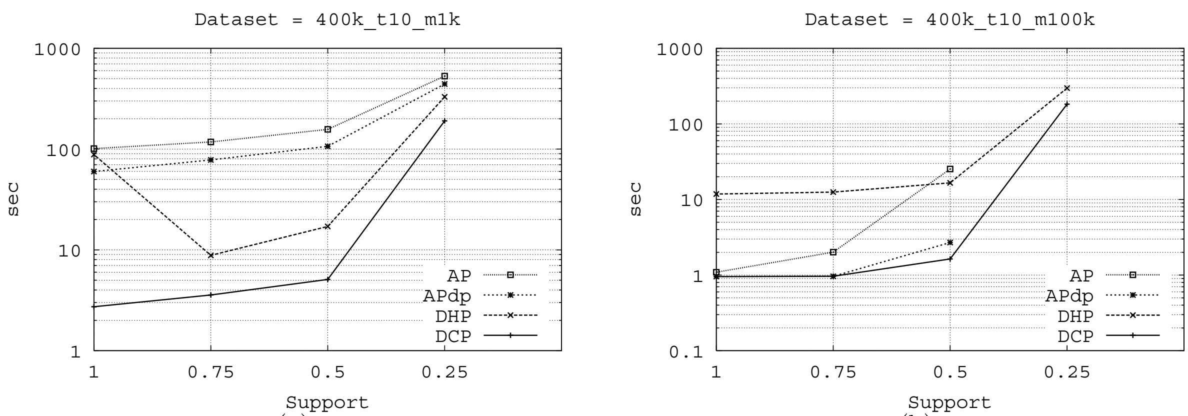
Total Execution Times. Figure 10 reports the total execution time obtained running *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on a dataset containing a small number n of transactions as a function of *min_sup*. Figure 10.(a) and (b) refer to datasets where the average transaction size is 10 and 20, with a fixed n. Changing the average transaction size has the twofold effect of increasing the dataset size, and, at the same time, increasing the average size of the maximal frequent itemset. In all the tests **DCP** showed better performances. It also reveals to be more stable to possible correlations in the dataset that can cause a heavier computational load. To this regard, note that DHP, whose pruning technique is more effective than ours, is not able to effectively handle dataset 200k_t10_n1k for $min\_sup = 1\%$. This is due to the high number of candidates of C_2 which DHP is not able to further reduce.

We also studied the influence of the total number m of items present in $\mathcal{D}$. As we increase m, we expect to find smaller maximal frequent itemsets. In other words, the effect of increasing m is the reduction of



(a) (b)

Figure 10: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on dataset 200k_t10_n1k (a), 200k_t20_n1k (b) for different supports.



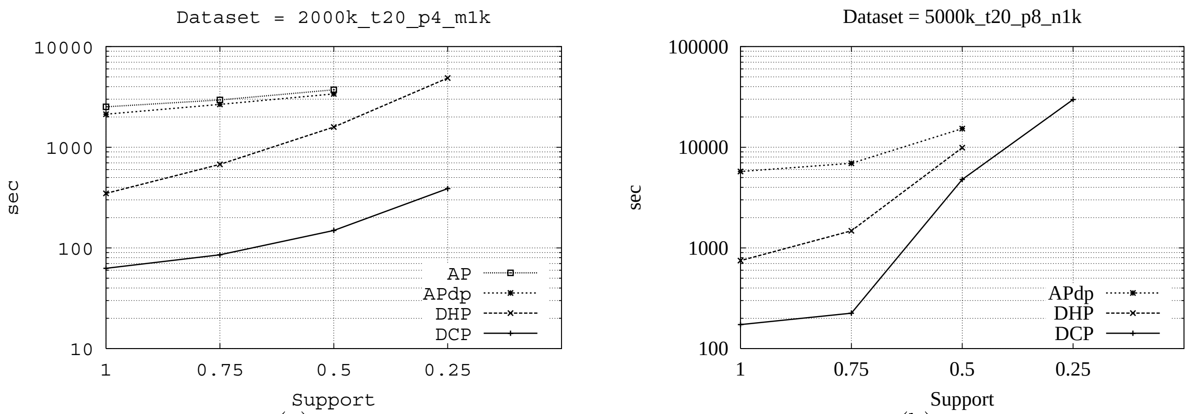
(a) (b)

Figure 11: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on dataset 400k_t10_n1k (a), 400k_t10_n100k (b) for different supports.

the number of algorithm iterations. This is confirmed by our experiments, whose results are reported in Figure 11. In particular, for $m = 100k$ and $min\_sup \in \{1, 0.75\}$, we have observed that $F_2 = \emptyset$. This is the reason why DHPis particularly penalized in this case, since the additional cost of the hash table construction at the second iteration is surely useless.

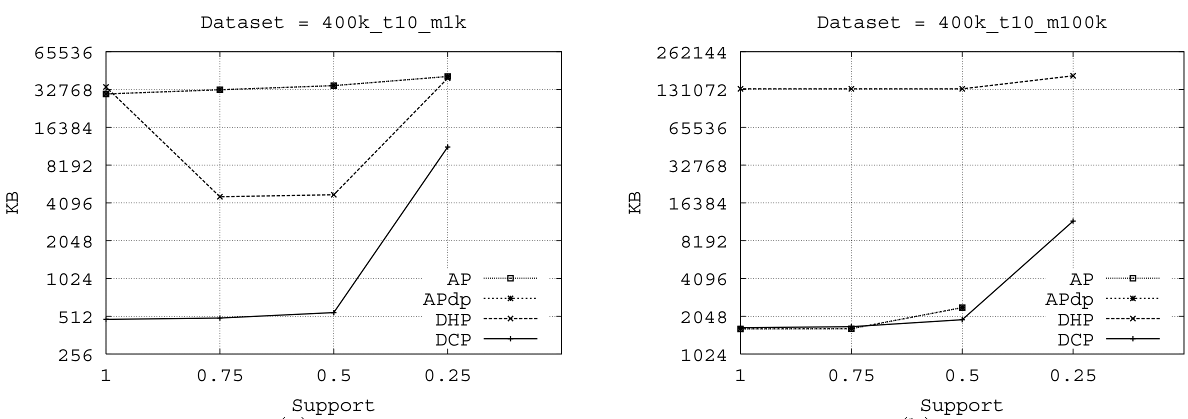
Finally, we tested the algorithm behaviors on large datasets (see Figure 12). Specifically, dataset 5000k_t20_n1k is about as large as the total physical memory available on the workstation used. This test is important, since the disk buffer cache is not surely able to contain the whole dataset. Thus we cannot take advantage from 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 of 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.

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 13 plots the maximum amount of memory allocated by the various algorithms during the tests on two different datasets.



(a) (b)

Figure 12: Total execution times for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on dataset 2000k_t20_p4_n1k (a), 5000k_t20_p8_n1k (b) for different supports.



(a) (b)

Figure 13: Maximal sizes of physical memory allocated during the execution for *Apriori*, DHP, $Apriori_{DP}$, and **DCP** on dataset 400k_t10_n1k (a), 400k_t10_n100k (b) for different supports.

6 Conclusions

In this paper we reviewed the *Apriori* class of algorithms proposed for solving the FSC problem. These algorithms have been often criticized because of their level-wise behavior which requires a number of scans of the dataset equal to the cardinality of the largest frequent itemset discovered. We demonstrated instead that the FSC is not an I/O-bound problem. In many cases, in fact, its computational granularity is large enough to take advantage of the features of modern OSs which allow computation and I/O to be effectively overlapped. Moreover, as the DHP algorithm demonstrates, counting the number of dataset scans as a measure of *Apriori* algorithms complexity does not consider that very effective dataset pruning techniques can be devised. These pruning techniques can rapidly reduce the size of the dataset until it fits in main memory. Nevertheless, our experimental results showed that the efforts to reduce the size of the dataset and the number of candidates are partially worthless if the counting procedure is not efficient. Our proposal of a new algorithm for solving the FSC problem goes in this direction.

DCP uses effective 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 technique enhances spatial and temporal locality in accessing data structures, also avoiding complex and expensive pointer dereferencing. As a result of its accurate design, **DCP** sensibly outperforms both DHP and *Apriori*: on many problem instances the performance improvement is even more than one order of magnitude. More importantly, **DCP** exhibits better scalability. Due to its counting efficiency and low memory requirements, it can efficiently manage large datasets to find frequent sets with very low support.

Future work has to be done to compare the **DCP** algorithm with FSC algorithms which exploit a *tid-list* organization of the dataset [15]. Such algorithms seem very interesting and efficient in discovering frequent itemsets with very low support, but a deeper experimental evaluation is required to analyze advantages and disadvantages of adopting these algorithms rather than level-wise ones.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] M. Beck et al. *Linux Kernel Internals, 2nd ed.* Addison-Wesley, 1998.

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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

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