**DCI: a Hybrid 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 propose **DCI**, a new algorithm for solving the Frequent Set Counting problem. Similarly to **Apriori**, **DCI** adopts a level-wise approach, according to which at each iteration $k$ all the frequent $k$-itemsets and the associated supports are determined. Itemset supports are computed by using a *hybrid* technique, which exploits a very effective *counting*-based method during the first iterations, and a very fast *intersection*-based method during the last ones. During its *counting*-based phase, **DCI** uses an innovative method for storing candidate itemsets and accessing them to count their support. It also exploits effective pruning techniques to reduce the size of the dataset as execution progresses. As soon as the pruned dataset becomes small enough to fit into the main memory, **DCI** builds on the fly a vertical transaction database, and starts using a very efficient *intersection*-based technique to determine the support of larger itemsets. We implemented and engineered **DCI** and other algorithms belonging to the **Apriori** class. When possible, locality of data and pointer dereferencing were optimized due to their importance with respect to developments in computer architectures. In-depth experimental evaluations were conducted by taking into account not only execution time, but also virtual memory usage and I/O activity. The experimental results show that **DCI** remarkably outperforms **Apriori** and others previously proposed algorithms. We obtained encouraging results for synthetically generated datasets characterized by both short and long length frequent patterns.

1 **Introduction**

Association Rule Mining (ARM) is one of the most popular topic in the KDD field. The process of generating association rules has historically been adopted for *market-basket analysis*, where transactions are records representing point-of-sale data, while items represent products on sale. The importance for marketing decisions of association rules like “the 80% of customers which buy products $x_1$ and $x_2$ also buy $y$” is intuitive, and explains the strong interest for ARM [10, 11, 20].

Given a database of transactions $D$, an association rule has the form $X \Rightarrow Y$, where $X$ and $Y$ are sets of items (itemsets), and $X \cap Y = \emptyset$. A rule $X \Rightarrow Y$ holds in $D$ with a minimum confidence $c$ and a minimum support $s$, if at least $c\%$ of all the transactions containing $X$ also contain $Y$, and $X \cup Y$ is present in at least $s\%$ of all the transactions of the database. The ARM process can thus be subdivided into two main steps. The former is concerned with the Frequent Set Counting (FSC) problem. During this step, the set $F$, including all the *frequent* itemsets, is built, where an itemset is frequent if its support is greater than a fixed threshold $s$, i.e. the itemset occurs in at least $min\_sup$ transactions ($min\_sup = s/100 \cdot n$, where $n$ is the number of transactions in $D$). In the latter step the association rules satisfying both the minimum support and the minimum confidence are identified on the basis of the knowledge of both the frequent itemsets and their supports. This step is less expensive than FSC, since it only requires to check, for every $X \in F$, the confidence of all the rules $X \setminus Y \Rightarrow Y$, where $Y \subseteq X$, and $Y \neq \emptyset$. Since all
the subsets of a frequent itemset are frequent as well, and the support values of all frequent itemsets are known from the previous step, computing the confidence of the generated rules is straightforward.

In this paper we concentrate our attention on the FSC problem [3]. Its search space is exactly $\mathcal{P}(M)$, the power set of $M$, where $M$ is the set of items contained in the transactions of $D$. Although $\mathcal{P}(M)$ is exponential in $m = |M|$, effective pruning techniques exist for reducing it. Unfortunately, for small support thresholds, pruning becomes less effective, thus making the FSC problem very expensive to solve both in time and space. A lot of proposals regard the efficient solution of the FSC problem [3, 4, 5, 12, 14, 15, 16, 19, 21, 22, 23]. The main goals of these algorithms are to efficiently prune or partition $\mathcal{P}(M)$, and to provide effective strategies for traversing it.

The capability of effectively pruning the search space derives from the intuitive observation that none of the superset of an infrequent itemset can be frequent. The search for frequent itemsets can be thus restricted to itemsets whose subsets are all frequent. This observation suggests a level-wise, or breadth-first (BF), visit of the lattice corresponding to $\mathcal{P}(M)$, whose partial order is specified by the subset relation (⊆) [23]. The Apriori algorithm [5], one of the most important FSC algorithms, exactly adopts a BF visit of $\mathcal{P}(M)$ for counting the supports of the various itemsets. Other algorithms [7, 2] adopt instead a depth-first (DF) visit of $\mathcal{P}(M)$. The goal in this case is to discover long frequent itemsets first, thus saving the work needed for counting the support of itemsets included in long ones. This approach is very effective, but it does not allow the supports of all frequent itemsets to be exactly determined. Unfortunately, the knowledge of the exact supports is needed to correctly compute association rule confidences.

Although a number of other solutions have been proposed, the Apriori algorithm [5] is still the most commonly recognized reference to evaluate FSC algorithm performances. Apriori iteratively searches frequent itemsets: at each iteration $k$, the set $F_k$ of frequent itemsets of $k$ items ($k$-itemsets) is identified, where $\mathcal{F} = \bigcup F_k$. In other words, at iteration $k$, Apriori only visits the level $k$ of the lattice corresponding to $\mathcal{P}(M)$. In order to identify $F_k$, however, Apriori does not consider all the possible $k$-itemsets of $\mathcal{P}(M)$, but a pruned subset. In particular, the $k$-itemsets considered are only those included in a candidate set $C_k$, constructed as the set of all the $k$-itemsets whose subsets of $k-1$ items belong to $F_{k-1}$. In order to determine the support of the candidate itemsets, Apriori directly counts their occurrences in all the transactions of $D$. At the end of a complete scan of the database, the candidate $k$-itemsets with minimum support are included in $F_k$, and the next iteration is started. The algorithm terminates when either $F_k$ or $C_k$ results to be empty, i.e. when no frequent set of $k$ items (or larger) is present in the database.

Several variations to the original Apriori algorithm, as well as many parallel implementations, have been proposed in the last years. We can recognize two main methods for determining the supports of the various itemsets present in $\mathcal{P}(M)$: a counting-based [4, 5, 12, 19, 7, 1] and an intersection-based [21, 9, 23] one. The former one, also adopted by Apriori, exploits a horizontal dataset, where the various transactions, containing information about the items included, are stored sequentially. The method is based on counting how many times each candidate $k$-itemset occurs in every transaction. The intersection-based method, on the other hand, exploits a vertical dataset, where a tidlist, i.e. a list of the identifiers of all the transactions which contains a given item, is associated with the identifier of the item itself. To determine the support of any $k$-itemset we must in this case compute the cardinality of the tidlist resulting from the $k$-way intersection of the $k$ tidlists associated with the corresponding items. If we are able to buffer the tidlists of previously computed frequent $(k-1)$-itemsets, we can further speed up the computation since the support of a generic candidate $k$-itemset $X$ can be simply computed by intersecting the tidlists of two $(k-1)$-itemsets whose union is $X$.

The counting-based approach is quite efficient from the point of view of memory occupation, since only requires enough main memory to store $C_k$ along with the data structures exploited to make the access to candidate itemsets faster (e.g. hash-trees or prefix-trees). On the other hand, the intersection-based method is much more computational effective than its counting-based counterpart [21]. Unfortunately, it
may pay the reduced computational complexity with an increase in memory requirements, in particular to buffer the tidlists of previously computed frequent \((k - 1)\)-itemsets.

The algorithms of the \textit{Apriori} class are often criticized because they require a number of database scans equal to the cardinality of the longest frequent itemset. However, for some datasets and small support thresholds, \textit{Apriori} algorithms become \textit{compute-bound}, so that techniques as those illustrated in [6] can be effectively exploited to overlap I/O time with useful computation. In these cases the performance penalty of \textit{Apriori} with respect to other approaches is only partially due to the multiple dataset scans involved. A crucial and not fully investigated aspect of \textit{Apriori}-like algorithms is instead the method exploited for 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, \textit{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 holds for datasets where the maximal frequent itemsets are not very long.

In this paper we propose \textbf{DCI} (Direct Count \& Intersect), a new algorithm that, like \textit{Apriori}, exploits a level-wise visit of the search space of the FSC problem. It adopts a \textit{hybrid} approach to determine the support of frequent itemsets, by exploiting a very effective \textit{counting}-based method during the first iterations, and a very fast \textit{intersection}-based method during the last ones. When the counting method is employed, \textbf{DCI} relies on an innovative technique for storing and accessing candidate itemsets. The method is a generalization of the \textit{Direct Count} technique adopted to determine the support of singleton itemsets, and allows to strongly reduce both in time and space the cost of the initial iterations of the algorithm. Moreover, \textbf{DCI} adopts a simple and effective pruning of \(\mathcal{D}\). Even if the benefits of dataset pruning were already recognized by DHP [19], \textbf{DCI} does not exploit the complex and expensive DHP hash filter technique, but uses a simpler and efficient one. During the counting-based phase, the candidates are maintained in-core, while the out-of-core dataset is accessed in blocks. Only when the pruned dataset is small enough to fit into the main memory, \textbf{DCI} changes its behavior, and adopts an intersection-based approach to determine frequent sets. The representation of the dataset is thus transformed from horizontal to vertical, and the new dataset is stored in-core. Using this approach, the support of each candidate itemset generated can be determined on-the-fly by intersecting the corresponding tidlists. Our intersection approach is very efficient, and only requires a limited (and configurable) amount of memory. Tidlists are actually represented as vectors of \textit{bits} accessed with high locality, and can efficiently be intersected without using expensive comparison and conditional branch instructions. To reduce the complexity of intersection, \textbf{DCI} reuses most of the intersections previously done, by caching them in a fixed-size buffer for future use. Even if, in the worst case, the lists corresponding to all the \(k\) items included in a candidate \(k\)-itemset have to be intersected \((k\)-way intersection\), our caching method is able to strongly reduce the number of intersections actually performed. In Section 4 we show that the number of intersections actually carried out is very close to the minimum obtained when a 2-way intersection approach is adopted. The 2-way intersection approach, however, requires much more memory to store, at each iteration \(k\), the tidlists associated with all the frequent \((k - 1)\)-itemsets.

To validate our proposal, we conducted several experiments by taking into account not only execution times, but also virtual memory usage, 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. Our test bed was a Pentium-based Linux workstation, while the datasets used for tests were generated synthetically.
The paper is organized as follows. In Section 2 we report general assumptions and notations used throughout the paper. Section 3 introduces DCI, and discusses its features in depth. In Section 4 we detail the method used to generate the synthetic datasets used in the tests, we analyze how modern OSs and computer architectures optimize the data access pattern of FSC level-wise algorithms, and discuss the encouraging results obtained with DCI in comparison with other algorithms. In Section 5 we review some of the most recent results in the FSC field, and compare the DCI approach with others. Section 6 draws some conclusions and outlines future work. Finally, Appendix A reports and discusses the pseudo code of DCI.

<table>
<thead>
<tr>
<th>( \mathcal{D} )</th>
<th>transaction database</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>number of transactions in ( \mathcal{D} )</td>
</tr>
<tr>
<td>( M )</td>
<td>set of items in ( \mathcal{D} )</td>
</tr>
<tr>
<td>( m )</td>
<td>number of items ( (m =</td>
</tr>
<tr>
<td>( t )</td>
<td>a generic transaction of ( \mathcal{D} )</td>
</tr>
<tr>
<td>( t_i )</td>
<td>an item identifier appearing at position ( i ) in transaction ( t )</td>
</tr>
<tr>
<td>( F_k )</td>
<td>set of frequent ( k )-itemsets</td>
</tr>
<tr>
<td>( C_k )</td>
<td>set of candidate ( k )-itemsets</td>
</tr>
<tr>
<td>( c )</td>
<td>a generic candidate itemset belonging to ( C_k )</td>
</tr>
<tr>
<td>( c_i )</td>
<td>an item appearing at position ( i ) in candidate itemset ( c )</td>
</tr>
<tr>
<td>( \mathcal{D}_k )</td>
<td>pruned transaction database read at iteration ( k ) ( (\mathcal{D}_1 = \mathcal{D}) )</td>
</tr>
<tr>
<td>( \mathfrak{n}_k )</td>
<td>number of transactions in ( \mathcal{D}_k ) ( (\mathfrak{n}_1 = n) )</td>
</tr>
<tr>
<td>( \mathfrak{M}_k )</td>
<td>set of significant items in ( \mathcal{D}_k ) ( (\mathfrak{M}_1 = M) )</td>
</tr>
<tr>
<td>( \mathfrak{m}_k )</td>
<td>cardinality of ( \mathfrak{M}_k ) ( (\mathfrak{m}_1 = m) )</td>
</tr>
</tbody>
</table>

### 2 Notation and general assumptions

To make easier the readability of the paper, Table I reports the most important symbols used throughout the paper. Moreover, in this section we discuss our assumptions about database representation and layout.

Without lacking of generality, we can associate integer identifiers with database transactions and items appearing in transactions. In particular, each of the \( n \) transactions in \( \mathcal{D} \) will be identified by a distinct TID \( \in \{1, \ldots, n\} \), while each of the \( m \) items will be identified by a distinct IID \( \in \{1, \ldots, m\} \). In the following, we will refer to an item \( i \) to mean the item associated with the identifier IID = \( i \). Databases can be stored in either a horizontal or vertical format:

**Horizontal.** Each one of the \( n \) records of \( \mathcal{D} \) corresponds to a different transaction TID, and stores boolean information about the presence or the absence in the transaction of the various items. If we refer to market-basket data, each record represents an individual customer purchase transaction, while the items present in the transaction are products on sale bought by the customer.

**Vertical.** A different record is associated with each item IID (e.g., with a product on sale), and stores boolean information about the presence (or the absence) of that item within the various transactions. The total number of records is thus \( m \).

In both the horizontal and vertical layouts, variable or fixed length records can be adopted:

**Variable length.** Only the presence of items (or transactions) is explicitly coded. A list of identifiers, in particular a list of IID's in the horizontal case, and a list of TID's in the vertical one, is stored in each record.
**Fixed length.** Each record occupies a fixed number of bits. In the horizontal case, the record stores a distinct TID and a vector of \( m \) bits, where the \( i \)-th bit is 1 or 0 to respectively indicate the presence or the absence of item \( i \) within transaction TID. In the vertical case, on the other hand, each record stores a different IID and a vector of \( n \) bits. The \( i \)-th bit is set only if item IID is present in the transaction with \( TID = i \).

In the remainder of the paper we will use the terms *transaction* and *tidlist* to refer to a generic record of a horizontal or vertical database, respectively.

Moreover, we assume that each transaction of a horizontal database with variable length records is stored as a vector of item identifiers sorted according to an increasing numerical ordering. Similarly, itemsets, either candidate or frequent ones, are ordered vectors of item identifiers. Finally, we assume that \( k \)-itemsets stored within sets \( C_k \) and \( F_k \) are lexicographically ordered.

### 3 The DCI algorithm

DCI adopts a counting-based approach during the first iterations, and uses an efficient intersection-based technique to determine larger itemsets. As other algorithms of the Apriori class, DCI uses a level-wise approach, according to which at each iteration \( k \) all the frequent \( k \)-itemsets and the associated supports are determined. During its counting-based phase, DCI exploits a horizontal layout database with variable length records, while a vertical layout database with fixed length records is constructed on the fly when DCI switches to its intersection-based phase. During the former counting-based phase, DCI uses a technique, similar to the one adopted by Apriori, to generate \( C_k \) from \( F_{k-1} \). In this construction DCI exploits the lexicographic order of \( F_{k-1} \) to find pairs of \((k-1)\)-itemsets sharing a common \((k-2)\)-prefix. Due to this order, in fact, the various pairs occur in close positions within \( F_{k-1} \). The union of each pair becomes a candidate \( c \in C_k \) only if all its subsets turn out to be included in \( F_{k-1} \). Also in this case we exploit the lexicographic order of \( F_{k-1} \), since we check whether all the subsets of \( c \) are included in \( F_{k-1} \) in logarithmic time. The main innovations introduced by DCI are thus summarized below:

**Pruning.** During its counting-based phase, DCI trims the transaction database as execution progresses. In particular, a pruned dataset \( D_{k+1} \) 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 \( D_k \) is always smaller than the size of \( D_{k-1} \). 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.** DCI does not use a hash tree data structure for counting frequent sets. Instead it exploits directly accessible data structures, thus avoiding complex and expensive pointer dereferencing. Finally, DCI exploits high spatial locality in accessing its counting data structures.

**Intersecting.** The counting of itemset occurrences is limited at early iterations. As soon as the pruned dataset becomes small enough to fit into the main memory, DCI adaptively changes its behavior, and adopts an intersection-based approach to determine frequent sets. Note, however, that DCI continues to have a level-wise behavior, so that the search space is still traversed breadth-first [14]. Our intersection-based approach is very efficient, and requires only a limited (and configurable) amount of memory. Tidlists are actually represented as vectors of bits accessed with high locality, and without using expensive comparison and conditional branch instructions. Finally, DCI reuses most of the intersections previously done by caching them in a fixed-size buffer for future use. Even if, in the worst case, the lists corresponding to all the \( k \) items included in a candidate \( k \)-itemset have to be intersected \((k\text{-way intersection})\), our caching technique is able to highly reduce
the number of intersections. In Section 4 we show that, due to our simple caching technique, the number of intersections actually carried out is very close to the minimum obtained when a 2-way intersection approach is adopted. It is worth noting, however, that the 2-way intersection approach requires much more memory to store, at each iteration $k$, the tidlists associated with all the frequent $(k - 1)$-itemsets.

In the following we discuss in depth all these innovative features of DCI, while the pseudo code of the algorithm is reported and discussed in Appendix A.

### 3.1 Pruning the dataset

Two different pruning techniques are exploited. Dataset global pruning which transforms a generic transaction $t$, read from $D_k$ into a pruned transaction $\tilde{t}$, and Dataset local pruning which further prunes the transaction, and transforms $\tilde{t}$ into $\tilde{t}$ before writing it to $D_{k+1}$. 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 $t \in D_k$ to generate $\tilde{t}$. 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 $I$ 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 $\tilde{t}$ 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 $G_{k-1}[\cdot]$ that is updated on the basis of $F_{k-1}$. Each counter of $G_{k-1}[\cdot]$ is associated with one of the $m$ items of $D_k$. 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 the frequent $(k - 1)$-itemsets have been scanned, $G_{k-1}[j] = x$ means that item $j$ appears in $x$ frequent itemsets of $F_{k-1}$.

Counters $G_{k-1}[\cdot]$ are thus used at iteration $k$ as follows. An item $t_i \in t$ is copied to the pruned transaction $\tilde{t}$ only if $G_{k-1}[t_i] \geq k - 1$. Then, if $|\tilde{t}| < k$, 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 $\tilde{t}$ during subset counting. The arguments this pruning technique is based on, are similar to those of its global counterpart. Transaction $\tilde{t}$ 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 $I \subseteq \tilde{t}$ belong to $C_k$. This check could be made locally during subset counting of transaction $\tilde{t}$.

Note that to implement the check above we should have to maintain, for each transaction $\tilde{t}$, information about the inclusion of all the $k$-subsets of $\tilde{t}$ in $C_k$. Since storing this information may be expensive, we adopted the simpler technique already proposed in [19], whose pruning effect is however smaller:

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

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

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 \( I \subseteq t \) 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 \( k \).

### 3.2 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.

**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 [23]. Unfortunately, for large values of \( m \), this simple technique may waste a lot of memory. In fact, since \( |F_2| \) is usually less than \( m \) and \( C_2 = F_1 \times F_1 \), we have that \( |C_2| = \binom{|F_1|}{2} \ll m^2 \).

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 \( M_k = |M_k| \), where \( m_k < m \). In particular, for \( k = 2 \) we have that \( M_2 = F_1 \), so that \( m_2 = |F_1| \).

Our technique for counting frequent 2-itemsets is thus based upon the adoption of vector COUNTS[ ], which contains \( |C_2| = \binom{|M_2|}{2} = \frac{|M_2|(|M_2|-1)}{2} \) 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 \( \mathcal{T}_2 : M_2 \rightarrow \{1, \ldots, m_2\} \). 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 \( 1 \leq x_1 < x_2 \leq m_2 \). 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:

\[
\Delta_2(c_1, c_2) = F_2^{m_2}(x_1, x_2) = \sum_{i=1}^{x_1-1} (m_2 - i) + (x_2 - x_1) = m_2(x_1 - 1) - \frac{x_1(x_1 - 1)}{2} + x_2 - x_1, \quad (1)
\]

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 \( \{1, x_2\}, 2 \leq x_2 \leq m_2 \) are stored in the first \( m_2 - 1 \) positions of vector COUNTS, while the counters corresponding to the various pairs \( \{2, x_2\}, 3 \leq x_2 \leq m_2 \), are stored in the next \( m_2 - 2 \) positions, and so on. Moreover, the pair of counters relative to \( \{x_1, x_2\} \) and \( \{x_1, x_2 + 1\} \), where \( 1 \leq x_1 < x_2 \leq m_2 - 1 \), 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 \( m_k \) decreases with \( k \), the amount of memory needed to store
($m_k$) counters might be huge, and in general ($m_k$) >> $|C_k|$. Before detailing the technique exploited by DCI for $k > 2$, remember that, at step $k$, for every transaction $t$, we have to check whether any of its $|t|^k$ $k$-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 [17]. In DCI 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. Since $C_k$ is lexicographically ordered, all the candidates sharing a common 2-item prefix are stored in a contiguous section of $C_k$. To efficiently implement our prefix tree, a vector PREFIX$_k[ ]$ of size ($m_k$/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 $T_k : M_k \rightarrow \{1,\ldots,m_k\}$. $\Delta_k(c_1,c_2)$ can be thus defined as follows:

$$
\Delta_k(c_1,c_2) = \mathcal{F}_2^{m_k}(x_1,x_2)
$$

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

DCI 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 $\{t_i,t_j\}, t_i < t_j$, is selected, the possible completions of this prefix needed to build a $k$-subsets of $t$ can only be found in $\{t_{i+1},t_{i+2},\ldots,t_{|t|}\}$. The contiguous section of $C_k$ which must be visited is thus delimited by PREFIX$_k[\Delta_k(t_i,t_j)]$ and PREFIX$_k[\Delta_k(t_i,t_j) + 1]$. 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 within $C_k$.

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 \( c = \{c_1, \ldots, c_r\} \) to be checked against \( t \), we can derive that \( c \not\subseteq t \), 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 \( c \not\subseteq t \) because \( c \) contains other \((k - i)\) items greater than \( c_i \), but such items in \( t \) are only \((|t| - POS[c_i]), (|t| - POS[c_i]) < (k - i)\).

### 3.3 Tidlist intersections

**DCI** starts using its intersection-based method when the vertical representation of the pruned dataset \( D_k \) may entirely fit into the main memory. Since \( D_k \) is stored in horizontal form, it has to be transformed first into its vertical counterpart. **DCI** starts checking whether the vertical dataset may fit into main memory at the third iteration. When the pruned dataset is small enough, its vertical representation is built on the fly, while the transactions are read to count the support of candidate itemsets. More specifically, **DCI** builds and uses an in-core vertical layout dataset, with fixed length records stored as *bit vectors*. The whole dataset is thus stored as a bidimensional bit-array \( VD[|] \), whose rows correspond to records (tidlists), each associated with a *not pruned item*. Given the set of not pruned items \( M_k \), and the set of not pruned transactions \( T_k \), let be \( \overline{m}_k = |M_k| \) and \( \overline{n}_k = |T_k| \). The amount of memory required to store \( VD[|] \) is thus \( \overline{m}_k \cdot \overline{n}_k \) bits.

At each iteration \( k \geq 3 \), **DCI** computes \( \overline{m}_k \cdot \overline{n}_k \) in order to decide whether the vertical dataset \( VD[\overline{m}_k][\overline{n}_k] \) can fit into the main memory. If this holds, **DCI** allocates and assigns \( VD[\overline{m}_k][\overline{n}_k] \). This construction occurs while the various transactions belonging to \( D_k \) are processed for subset counting. More specifically, for every item \( i \) included in each transaction TID, the bit \( VD[T_k(i)][H_k(TID)] \) is set, where \( T_k() : M_k \rightarrow \{1, \ldots, \overline{m}_k\} \) and \( H_k() : T_k \rightarrow \{1, \ldots, \overline{n}_k\} \) are two strictly monotonous increasing functions. Starting from the following iteration, **DCI** starts using the vertical database \( VD \) to determine the support of each candidate itemset \( c \). First the candidates are generated and stored in \( C_k \). Then, for each candidate \( k \)-itemset \( c \in C_k \), **DCI** and-intersects the \( k \) associated bit vectors and computes the support of \( c \) by counting the number of bits set in the resulting tidlist.

At first glance, the strategy above might seem inefficient. If we had enough memory to maintain the tidlists associated with all the frequent \((k - 1)\)-itemsets, we could build the tidlist of any \( k \)-itemset \( c \) by intersecting only two tidlists, i.e. those associated with a pair of frequent \((k - 1)\)-itemsets whose set union is exactly \( c \). In other words, we might exploit 2-way intersections instead of more expensive \( k \)-way ones. Unfortunately maintaining the tidlists of all frequent \((k - 1)\)-itemsets poses strong constraints on the applicability of the 2-way approach, due to the huge amount of memory required. Nevertheless, a number of effective optimizations can still be applied to our \( k \)-way intersection method in order to save work, and speed up the overall computation:

1. all the intermediate intersections that have been computed to determine the support of each candidate itemset \( c \in C_k \) are "cached", by storing them in a bidimensional bit-array \( V[j][|] \) of size \((k - 2) \cdot \overline{n}_k \). More specifically, the bit vector \( V[j][1 : \overline{m}_k] \), \( 2 \leq j \leq (k - 1) \), is used to store the results of the intersections relative to the first \( j \) items of \( c \). Since itemsets in \( C_k \) are lexicographically ordered, with high probability two consecutive candidates, e.g. \( c \) and \( c' \), share a common prefix. If we have to determine the support of \( c' \), and \( c \) and \( c' \) share a common prefix of length \( h \geq 2 \), we save work by reusing the intermediate result stored in \( V[h][1 : \overline{n}_k] \). This simple optimization only requires additional \((k - 2) \cdot \overline{n}_k \) bits, and results in a consistent saving in the number of intersections actually executed.
2. bit vectors are generally sparse and characterized by long runs of 0’s intermixed by few 1’s. Moreover, any and-intersection operation produces a vector with fewer 1’s. This property can be exploited to speedup intersections as follows. While we compute the intersection of the bit vectors relative to the first two items $c_1$ and $c_2$ of a generic candidate itemset $c = \{c_1, c_2, \ldots, c_k\}$, we maintain information about the positions of the 1’s in the resulting bit vector $V[2][1 : \bar{m}_k]$. Further intersections performed for processing itemset $c$ (as well as other itemsets sharing the same 2-item prefix) skip the runs of 0’s, so that only vector segments which may contain 1’s are actually intersected. Note that this optimization results to be very effective, since the positions of $V[2][1 : \bar{m}_k]$ containing runs of 0’s are computed only once, and the same information reused many times.

3. dataset transaction pruning is important also for vertical databases due to the consequent reduction in the length of the bit vectors, and thus in the intersection costs. A transaction, i.e. a column of $VD$, can be removed from the vertical database when no one of its items is included in $F_k$. In this case the column of $VD$ to be removed contain only 0’s in correspondence of all the items included in $F_k$. To determine prunable columns (transactions), we initialize a bit vector $ToPrune[\cdot]$ to zero at each iteration $k$. Then, each time we find that a given candidate itemset is frequent, its resulting bit vector is or-ed with $ToPrune$. To save work, also in this case the or operation only regards the non zero segments of vector $V[2][1 : \bar{m}_k]$. At the end of iteration $k$, when the construction of $F_k$ has been completed, $ToPrune$ is used to remove from $VD$ all the columns $i$ for which $ToPrune[i] = 0$ holds.

**Further optimizations.** Further effective optimizations have been introduced in the intersection-based part of DCI. The first one consists in reassigning item identifiers when the vertical dataset is built in memory. At this purpose, before building $VD$, $M_k$ is sorted in increasing order of item support, and new consecutive integer identifiers $\{1, \ldots, \bar{m}_k\}$ are assigned to the ordered items. The effect of this reordering is twofold. From one hand, since only the items with the highest supports can presumably belong to largest frequent itemsets, we can expect that as the iteration index $k$ increases, most DCI references concentrate on tidlists associated with items with high supports (i.e., high integer identifiers). The section of $VD$ actually accessed thus shrinks progressively, and spatial locality is consequently enhanced. On the other hand, since candidate itemsets are lexicographically ordered, and intersections are carried out respecting this order, tidlists with less 1’s are intersected first, with obvious repercussions on the second optimization discussed above.

An ordering step is also performed during pruning. When $VD$ is pruned, its columns are reordered with the purpose of increasing the length of the runs of 0’s and 1’s present in the various rows. The goal is once more making the second optimization discussed above even more effective. To reorder $VD$ columns, the first item $\bar{1}_1$ belonging to $M_k$ (i.e. the least frequent one) is considered first, and the columns (transactions) that contain it are moved in front of $\bar{1}_1$’s tidlist, thus obtaining a single initial run of 1’s followed by all 0’s. The second less frequent item $\bar{1}_2$ is then considered, and the columns (transactions) containing $\bar{1}_2$ but not $\bar{1}_1$ are grouped in the same way. This process proceeds until the last column of $VD$ is reached and no other column can be moved in front.

However, we verified that transaction reordering and the whole pruning process are effective only when many (and long) frequent patterns are discovered in the database. Pruning and reordering overheads otherwise overwhelm the advantages of dealing with shorter and ordered tidlists. A simple and effective heuristic was thus introduced in DCI that starts pruning and transaction reordering at the end of each intersection-based iteration $k$ only if $|C_k| >> \bar{m}_k$. 

10
4 Performance evaluation

To evaluate DCI performances we conducted several experiments with different FSC level-wise algorithms. Table II reports the main characteristics of the algorithms tested. Apriori$_{DP}$ refers to a version of the classic Apriori algorithm enhanced with the dataset pruning technique introduced in DCI. DCP (Direct Count of candidates & Pruning transactions) [18] is an implementation of DCI that continues to adopt a counting-based approach for all the algorithm iterations, and thus does not switch to an intersection-based approach for the last iterations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>FSC approach</th>
<th>FSC data structure</th>
<th>Dataset pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apriori</td>
<td>counting itemset occurrences</td>
<td>hash tree</td>
<td>no</td>
</tr>
<tr>
<td>Apriori$_{DP}$</td>
<td>counting itemset occurrences</td>
<td>hash tree</td>
<td>yes</td>
</tr>
<tr>
<td>DHP</td>
<td>counting itemset occurrences</td>
<td>hash tree</td>
<td>yes</td>
</tr>
<tr>
<td>DCP</td>
<td>counting itemset occurrences</td>
<td>prefix tree</td>
<td>yes</td>
</tr>
<tr>
<td>DCI</td>
<td>hybrid: counting &amp; intersect</td>
<td>prefix tree &amp; tidvectors</td>
<td>yes</td>
</tr>
</tbody>
</table>

The test bed architecture was a Linux-based workstation, equipped with a Pentium III processor running at 450MHz, 512MB RAM, and an Ultra2 SCSI disk. The experiments were conducted by varying the minimum support threshold $s$ and the characteristics of the transaction databases. The datasets we used in our experiments were created with one of the most commonly adopted synthetic dataset generator [5], and are characterized by the parameters reported in Table III, 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 potentially frequent itemsets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$T$</th>
<th>$P$</th>
<th>$n$</th>
<th>$m$</th>
<th>$L$</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200k±20p+4n1k</td>
<td>20</td>
<td>4</td>
<td>200k</td>
<td>1k</td>
<td>2000</td>
<td>18</td>
</tr>
<tr>
<td>400k±10p+8n1k</td>
<td>10</td>
<td>8</td>
<td>400k</td>
<td>1k</td>
<td>2000</td>
<td>18</td>
</tr>
<tr>
<td>400k±10p+8n10k</td>
<td>10</td>
<td>8</td>
<td>400k</td>
<td>10k</td>
<td>2000</td>
<td>18</td>
</tr>
<tr>
<td>400k±20p+8n1k</td>
<td>30</td>
<td>8</td>
<td>400k</td>
<td>1k</td>
<td>2000</td>
<td>50</td>
</tr>
<tr>
<td>400k±30p+16n1k</td>
<td>30</td>
<td>16</td>
<td>400k</td>
<td>1k</td>
<td>2000</td>
<td>50</td>
</tr>
<tr>
<td>800k±30p+8n1k</td>
<td>30</td>
<td>8</td>
<td>800k</td>
<td>1k</td>
<td>2000</td>
<td>100</td>
</tr>
<tr>
<td>5000k±20p+8n1k</td>
<td>20</td>
<td>8</td>
<td>5000k</td>
<td>1k</td>
<td>2000</td>
<td>438</td>
</tr>
<tr>
<td>10k±25p+10n1k</td>
<td>25</td>
<td>10</td>
<td>10k</td>
<td>1k</td>
<td>2000</td>
<td>1</td>
</tr>
</tbody>
</table>

**DCI computational costs** We analytically analyzed the advantages of adopting an intersection-based approach, which is used by DCI after a few iterations, over the exploitation of the same counting-based approach for all the iterations of the algorithm. To this end, we compared the costs of the last iterations of DCI and DCP. The computational costs of each DCP iteration is dominated by subset counting. At most $k – 2$ comparisons are necessary in order to check whether a given candidate $k$-itemset is included into a transaction $t$ or not. Hence, the number of operations actually performed by DCP at iteration $k$ is approximately:

$$N_{CS} \cdot k$$

where $N_{CS}$ is the total number of candidates actually visited for counting the supports of all the transactions in $D_k$. On the other hand, the computational cost of each DCI iteration is proportional to the number of and operations needed to determine the supports of all candidate itemsets. The number of and depends on both the average length of tidlists and the number of candidates itemsets. In addition,
DCI performs or operations to prepare the subsequent pruning of the vertical dataset. Therefore, the number of operations actually performed by DCI at iteration $k$ is approximately:

$$ (N_{AND} + N_{OR}) \cdot N_{VD} $$

(3)

where $N_{AND}$ and $N_{OR}$ are, respectively, the total numbers of tidlist pairs which are actually and-ed or or-ed, while $N_{VD}$ is the average number of operations needed for and-ing or or-ing a pair of tidlists. In principle we can say that $N_{VD}$ depends on the average length of tidlists, but we have to consider that DCI exploits several optimizations aimed at reducing the number of operations actually performed (see Section 3.3).

This simple analysis is confirmed by our experimental evaluation. In Figure 2 the measured per-iteration execution times, i.e. $T_{DCP}$ and $T_{DCI}$, are plotted against their analytic estimates above, i.e. Equations (2) and (3), as a function of the iteration index $k$. The dataset considered was 400k, with minimum support $s = 0.25\%$. The actual values of $N_{CS}$, $N_{AND}$, $N_{OR}$ and $N_{VD}$ were determined by profiling the executions of the two programs. These experimental results confirm that the intersection-based method performs better than its counting-based counterpart. This is only partially due to its in-core behavior, since the vertical-layout dataset is entirely stored in the main memory. More importantly, we have proved that the intersection-based approach is computationally more efficient than the counting-based one.

![Operations and Execution Time: 400k, s = 0.25](image)

Figure 2: DCI and DCP theoretical and measured computational costs on dataset 400k, with minimum support $s = 0.25\%$. Times $T_{DCI}$ and $T_{DCP}$ are measured in seconds according to the scale on the right hand $y$ axis. Millions of operations are instead reported on the left hand $y$ axis.

Caching of intersection results In Section 3.3 we discussed DCI intersection method, and described the technique used to cache the results of previous and intersections, thus reducing the cost for determining the candidate support. To evaluate the effectiveness of our caching policy, we counted the actual number of intersections carried out by DCI on the dataset 400k, with $s = 0.25\%$. We compared this number with the number of intersections performed by two alternative techniques. The former one consists in adopting a 2-way intersection approach, which is only possible if we can fully cache the tidlists associated with all the frequent $(k - 1)$-itemsets. The latter technique regards the adoption of a pure $k$-way intersection method, i.e. a method that does not exploit caching at all, and determines the support of a candidate $k$-itemset by intersecting the tidlists associated with all the $k$ items.

Figure 3.(a) plots the results of this analysis. The caching policy of DCI turns out to be very effective, since the actual number of intersections performed results to be very close to the minimum obtained when a 2-way intersection approach is adopted. Moreover, memory requirements for the three approaches are
plotted in Figure 3.(b). As expected, DCI requires orders of magnitude less memory than a pure 2-way intersection approach, thus better exploiting memory hierarchies.

**Pruning effectiveness.** An important characteristic of FSC level-wise algorithms is their ability of trimming dataset size as execution progresses. DHP is the most effective algorithm in this regard. It prunes both the candidates and the dataset using an expensive hash filtering technique [19]. The plot reported in Figure 4.(a) quantifies the effectiveness of DHP dataset pruning. The Figure plots, for $s = 0.25\%$ and various datasets, the size of $\mathcal{D}_{k+1}$ as a function of the iteration index $k$. As it can be seen, the reduction in the size of the processed dataset is notable even with this low support threshold. For the same datasets and support, Figure 4.(b) shows the effectiveness of our simpler pruning technique. We can see that DHP prunes slightly more the dataset in early iterations, but the differences becomes rapidly neglectable.

![Graphs](image.png)

**Figure 4:** DHP (a) and DCI (b) dataset pruning efficacy for $s = 0.25\%$ and various datasets.

Dataset pruning has important implications on performances. Even if the size of the transaction database is initially huge, we can forecast that after few iterations the pruned dataset may entirely fit into the main memory. If this is the case, the original out-of-core algorithm could become in-core. Note that this transformation is automatically achieved by modern OSs that exploit the main memory left unused by the kernel and the processes as a buffer cache for virtualizing accesses to block devices such as...
disks [6, 8]. In order to evaluate the impact of the OS caching policy on FSC level-wise algorithms which exploits dataset pruning, we conducted synthetic tests on our test bed architecture. The benchmark used for these tests simulates the data access pattern of FSC level-wise algorithms. Specifically, at each iteration $k = \{1, \ldots, 17\}$, it reads the dataset in blocks of 4KB, performs some computation on each block read, and write back to the disk only part of these blocks. The size of the original dataset was 505 MB. For this dataset and for a given $s$, we first measured the effect of the DHP pruning technique, i.e. the actual reduction in the dataset size at each iteration. Our benchmark reduces the dataset size using the same rate measured for DHP. In the benchmark, however, the computational granularity - i.e. the per-block computation time - can artificially be varied in order to study its effect on the total elapsed time. The difference between the total elapsed time and the process CPU time can be considered as an approximated measure of the I/O cost. Figure 5 shows the result of this test, where the per-block computation time was varied from 0 to 0.9 ms. The $x$ and $y$ axes report the total CPU and elapsed time, respectively. Note that an approximated measure of the actual I/O cost paid for repeatedly reading and writing the dataset can be deduced from the value plotted for $x = 0$, i.e. null per-block CPU time. From the curves plotted in Figure 5, we can observe that the total elapsed time increases slowly when small computational grains are considered, since the elapsed time is dominated by I/O time, and this time cannot be overlapped with useful computation. For larger computational grains, corresponding to real FSC problem instances with low supports, the actual I/O cost becomes a constant, which turns out to only be a few percent of the total elapsed time.

![Figure 5: Elapsed time vs. computational granularity for a 505MB dataset iteratively elaborated and pruned.](image)

**Per-iteration Execution Times.** The plots reported in Figure 6 show the per-iteration execution times of DCI, DCP, Apriori, Apriori$_DP$, and DHP. The plots refer to tests conducted on different datasets, for various values of $s$. The most evident result is that, for $k = 2$ and $k = 3$, DCI and DCP always outperform Apriori, Apriori$_DP$, and DHP of at least one order of magnitude. This is due to the different counting technique exploited, which results to be much more efficient than the one based on a hash tree. By looking at DCP curves, slighter improvements due to our counting method can be noted also for larger values of $k$.

The other interesting result is that DCI remarkably outperforms the other implementations. On datasets characterized by medium/long patterns, and for several iterations, DCI achieves a performance gain over the counting-based algorithms of even two order of magnitude, thus demonstrating the lower complexity of the intersection-based approach.
Figure 6: Per-iteration execution times of DHP, DCP, Apriori, Apriori$_{DP}$, and DCI for different datasets and supports.
Figure 7: Total execution times for \textit{Apriori}, \textit{DHP}, \textit{Apriori}_{DP}, and \textit{DCI} on various datasets as a function of the support value.
Total Execution Times. Figure 7 reports the total execution times obtained running Apriori, DHP, AprioriDP, DCP, and DCI on various datasets as a function of \( s \). In all the tests conducted DCI remarkably outperforms the other implementations. Improvements of more than one order of magnitude were achieved for small values of support. Moreover, DCP performance resulted always better than DHP and AprioriDP ones, thus demonstrating the efficiency of our counting approach. The low execution times obtained with DCI are therefore due to both the efficiency of the counting method exploited during early iterations, and the effectiveness of the intersection-based approach used when the pruned vertical dataset fits into the main memory. Finally, it is worth noting that for very small supports \( (s = 0.25\%) \) some tests with DHP and AprioriDP were not able to allocate all the memory needed. DCI and DCP on the other hand, require less memory than their competitors, which exploit a hash tree for counting candidates. In this regard, Figure 8 plots the maximum amount of memory allocated by the various algorithms during the tests on two different datasets.

![Figure 8: Maximal sizes of physical memory allocated during the execution for Apriori, DHP, AprioriDP, and DCI on dataset 400k_t30_p8_m1k (a), 400k_t10_p100k (b) for different supports.](image)

Figure 9: Relative execution times on datasets t20_p8_m1k \( (s = .25\%) \) with varying number of transactions.

Scale up. Finally, we tested the behavior of the algorithms when the amount of data processed is increased, keeping everything else constant. In Figure 9 we plotted the execution times of the various implementations obtained by varying the number of transactions, and keeping their lengths fixed, for \( s = .25\% \). The times reported are normalized with respect to the execution time of DCI on dataset...
All the algorithms show the same qualitative behavior: we can see that there is a
gradual increase in the execution time as the dataset size is increased, and DCI always outperforms the
other algorithms.

5 Related work

The algorithms of the Apriori class [4, 5, 12, 19] adopt a level-wise, counting-based approach, according
to which, at each iteration $k$, the occurrences of candidate $k$-itemsets set-included in each transaction are
counted. The approach involves multiple scans of a horizontal database, and a hash tree is used to speedup
the counting phase. An alternative approach, used by several other algorithms, is the intersection-based
one [21, 9, 23]. In this case the database is stored in a vertical layout, where each record is associated with
a distinct item and stores information about the transactions containing the item itself. Partition, an
intersection-based algorithm that solves several FSC local problems on distinct partitions of the dataset
is discussed in [21]. Dataset partitions are chosen which are small enough to fit in the main memory, so
that all these local FSC problem can be solved with a single dataset scan, by hopefully using an efficient
2-way intersection method. 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 [16] some methods to reduce the effects
of this problem are discussed.

Dataset sampling [24, 22] 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.

Since algorithms that reduce dataset scans [21, 16, 22] increase the amount of work carried out at
each iteration, we argue that further work has to be done to quantitatively analyze advantages and
disadvantages of adopting these algorithms rather than level-wise ones. This is particularly true if we
consider that techniques for database pruning like ours are able to highly reduce the dataset size and the
related I/O costs.

Recently, some new algorithms for solving the FSC problem have also been proposed [23]. 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 that
only mine 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
easily 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 [13], a very interesting algorithm that is 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 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], a counting-based algorithm which, like DCP (and DCI during the its first iterations), prunes the transactions before counting the support of candidate itemsets. 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 for each specific group of candidates. To this end, candidates are subdivided into groups, where each group shares a common prefix. To determine the various projections of a transaction at level \( k \), the transaction has to be compared with each frequent itemset at level \( k - 2 \). These itemsets exactly determine the common prefix of a given group of candidates of length \( k \). This projection phase is thus the most expensive part of the algorithm. Once each projection has been determined, counting is very fast, since it only requires to access a small matrix of counters. The counting method used by DCP (and by DCI when the counting-based approach is adopted) is different from Tree Projection). For each transaction 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 pruned transaction \( t \), we look up a directly accessible prefix table of \( C_k \), and then sequentially 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 low. We verified experimentally that a large part of the candidates checked by DCP against \( t \) turn out to be actually included in \( t \). In other words, DCP is able to optimally select the candidates to be checked against \( t \).

6 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 algorithms are not I/O-bound. When this occurs, the computational granularity becomes 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 inefficient.

Unfortunately, for problem datasets from which long patterns can be mined, we have an explosion of the number of candidate and frequent itemsets, since all the $2^l$ subsets of each long maximal frequent itemset of length $l$ have to be produced. In this case, the counting-based approach becomes very expensive, since the supports of frequent itemsets become very large, and the various candidates turn out to be set-included in a lot of transactions. Possible solutions to reduce the burden of the counting-based approach are to use lower bound techniques to determine whether an itemset is frequent without actually counting its exact support, or to find maximal frequent sets by looking ahead throughout the search space [7, 2]. 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. An alternative solution to speed up the counting of the exact support for long frequent patterns regards instead the adoption of a intersection-based approach.

Our ideas to design DCI, a new hybrid algorithm for solving the FSC problem, originate from all the above considerations. While a counting-based approach is used during the first iterations, DCI uses an efficient intersection-based technique to determine the support of longer patterns.

During its counting-based phase, DCI 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 DCI is to adopt a blocking technique [1] to improve temporal locality as well. In fact, for each transaction $t$, DCI explores specific sections of a vector storing $C_k$ in order to count the support of candidates. Instead of 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.

DCI starts using its intersection-based method when the vertical representation of the pruned dataset may entirely fit into the main memory. Tidlists are actually represented as vectors of bits accessed with high locality, and can efficiently be and-intersected without using expensive comparison and conditional branch instructions. To reduce the complexity of intersection, DCI reuses most of the intersections previously done by caching them in a fixed-size buffer for future use.

As a result of its optimized design, DCI significantly outperforms other level-wise algorithms. For many datasets the performance improvement is even more than one order of magnitude. More importantly, due to its well-balanced use of system resources, DCI can be used to efficiently find frequent itemsets with very low supports in databases characterized by the presence of both short and long patterns.

References


20


[13] 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.


A Pseudo code of DCI

The first iteration of DCI, 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 $t \in D$.

Figure 10 shows the pseudo code of the second iteration of DCI, which exploits the direct count technique discussed in Section 3.2. 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 $G_k[i]$, 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 $k - 1$ (line 9). Note that, since during the first iteration of the algorithm the dataset cannot be pruned, we still read transactions from $D$. 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 to the pruned dataset $D_3$ (line 17).

1: set global_counter($G_1, F_1$)  
2: $k \leftarrow 2$  
3: $m_2 \leftarrow |F_1|$  
4: for all $i \in \{1, \ldots, m\}$ do  
5: COUNTS[$i$] $\leftarrow 0$  
6: end for  
7: $D_3 \leftarrow \emptyset$  
8: for all $t \in D$ do  
9: $\hat{t} = \text{global pruning}(t, G_1, k)$  
10: if $|\hat{t}| \geq 2$ then  
11: for all $\{t_{i_1}, t_{i_2}\} \subseteq \hat{t} | 1 \leq i_1 < i_2 \leq |\hat{t}|$ do  
12: $\Delta = \Delta_2(t_{i_1}, t_{i_2})$  
13: COUNTS[$\Delta$] $\leftarrow$ COUNTS[$\Delta$] + 1  
14: end for  
15: end if  
16: if $|\hat{t}| \geq 3$ then  
17: $D_3 \leftarrow D_3 \cup \hat{t}$  
18: end if  
19: end for  
20: $F_2 = \{i_1, i_2\} \in C_2 | $ COUNTS[$\Delta_2(i_1, i_2)$] $\geq \min_{sup}$  
21: $k \leftarrow 3$

Figure 10: Pseudo code of the second iteration of DCI.

Figure 11 shows the pseudo-code of the following iterations $k, k \geq 3$, which still work on the horizontal database. This loop terminates when either its is impossible to find larger frequent itemsets, or the vertical dataset has been built in memory on the fly. The flag HorizontalLayout is used to control when to switch to the intersection-based approach whose pseudo-code is shown in Figure 13. First we set the global counters on the basis of $F_{k-1}$ (line 3), and we build $C_k$ by adopting the same procedure as in Apriori (line 4). Once the candidates are generated, we decide, on the basis of the values of $m_k$ and $m_2$, whether to allocate the vertical dataset $VD[|\hat{t}|][\hat{t}]$ or not (lines 8-11). After building the prefix tree discussed in Section 3.2 (line 12), we start processing each transaction. The global pruning technique is first applied (line 15). When required, the transaction $\hat{t}$ is inserted into $VD[|\hat{t}|]$ by subroutine Set_bit_vector() (line 16-18). Then we count the occurrences of candidates in $\hat{t}$ (lines 19-27). 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 22-25). Then the subroutine count_candidates() is called (line 26 and Figure 12). It scans the contiguous section of $C_k$ identified by start and end. The scanning of the various candidates against $\hat{t}$ employs the vector POS[], which has been initialized with the relative positions of the items included in $\hat{t}$ (line 21). Note that subroutine count_candidates() also updates $L_k[\hat{t}]$, the per-transaction vector of counters exploited by the
1: HorizontalLayout ← True
2: while (F_{k-1} ≠ ∅) and HorizontalLayout do
3: set global counters (G_{k-1}, F_{k-1})
4: C_k = apriori gen(F_{k-1})
5: if C_k = ∅ then
6: return
7: end if
8: if (∏_{k} - ∏_{k-1}) ≤ MAX_SZ then
9: HorizontalLayout ← False
10: Allocate a zeroed Vertical Dataset of ∏_{k} - ∏_{k-1} bits: \( VD|\∏_{k}||\∏_{k-1} |\)
11: end if
12: PREFIX_{k}[ ] = init candidates(k, C_k)
13: D_{k+1} ← ∅
14: for all \( t ∈ D_{k} \) do
15: \( \tilde{l} = global\_pruning(t, G_{k-1}, k) \)
16: if not HorizontalLayout then
17: set bit vector \( t, VD[|]|) \)
18: end if
19: if |\( \tilde{l} \)| ≥ k then
20: Initialize local counters \( L_{k}[ ] \)
21: POS[ ] = init positions(\( \tilde{l} \))
22: for all \( \{t_1, t_2\} ⊆ \tilde{l} | 1 ≤ t_1 < t_2 ≤ |\( \tilde{l} \)| - k + 2 \) do
23: \( Δ = Δ_k(t_1, t_2) \)
24: start = PREFIX_k[Δ]
25: end = PREFIX_k[Δ + 1] - 1
26: count candidates(\( \tilde{l} \), k, C_k, POS, start, end, L_k)
27: end for
28: if HorizontalLayout then
29: \( \tilde{l} = local\_pruning(\( \tilde{l}, L_k \)) \)
30: if |\( \tilde{l} \)| ≥ (k + 1) then
31: \( D_{k+1} ← D_{k+1} ∪ \tilde{l} \)
32: end if
33: end if
34: end if
35: end for
36: \( F_k = \{c ∈ C_k | c.COUNTS ≥ min\_supp\} \)
37: \( k ← k + 1 \)
38: end while

Figure 11: Pseudo code of the counting-based iterations of DCI (k ≥ 3).

Subroutine count candidates(\( \tilde{l} \), k, C_k, POS, start, end, L_k)

1: for all \( c = \{i_1, \ldots, i_k\} | C_k[\text{start}] ≤ c ≤ C_k[\text{end}] \) do
2: /* \( c \) is included in the ordered segment of candidates comprised between \( C_k[\text{start}] \) and \( C_k[\text{end}] \) */
3: found ← True
4: \( j ← 3 \)
5: while ( \( j ≤ k \) ) \& \( \& \) \( \text{found} \) ) do
6: if ( \( (POS[i_j] = 0) \) \& \( \& \) (|\( \tilde{l} \)| - \( POS[i_j] \) < \( k - j \)) then
7: found ← False
8: else
9: \( j ← j + 1 \)
10: end if
11: end while
12: if found then
13: \( c.COUNTS ← c.COUNTS + 1 \)
14: for all \( i_j ∈ c \) do
15: \( L_k[i_j] ← L_k[i_j] + 1 \)
16: end for
17: end if
18: end for
end Subroutine

Figure 12: Pseudo code of the subroutine count candidates().
local pruning technique.

Figure 13 shows the pseudo-code of DCI final loop, which adopts the intersection-based approach discussed above. After generating $C_k$, we allocate $V[k][\overline{\pi_k}]$, to store partial intersection results, and $ToPrune[\overline{\pi_k}]$, used to prune the columns of the vertical dataset $VD[\overline{\pi_k}]$ (lines 7-8). The various candidate $k$-itemsets $c_i \in C_k$ are scanned, and their supports are computed by subroutine $compute\_support\_intersect()$. The last two parameters of the subroutine are the candidate itemset to be considered, and the vertical layout dataset $VD[\overline{\pi_k}]$, respectively. The first parameter $h$ of the subroutine determines whether the partial intersection results stored in $V[\overline{\pi_k}]$ can be used to speed up the intersection job. When $h$ is lower than 2, as when the first candidate $c_1 \in C_k$ is considered (line 12), $V[\overline{\pi_k}]$ is only used to store intersection intermediate results. On the other hand, when $h$ is greater than 1 (i.e. $c_i$ and $c_{i-1}$ share a common prefix of length greater or equal to 2), the partial results previously stored in $V[h][1: \overline{\pi_k}]$ are used to speed-up the intersection job relative to $c_i$. The subroutine $compute\_support\_intersect()$ returns the support of candidate $c_i$. If this number is equal or greater than $\min\_sup$, the candidate is added to $F_k$, and the resulting vector is or-ed with $ToPrune[\overline{\pi_k}]$ (lines 13-16 and 20-23). Finally, if $|C_k| >> \overline{\pi_k}$, the dataset is pruned and reordered as discussed in Section 3.3, and the new value of $\pi_k$ (i.e. the number of non zero columns of $VD[\overline{\pi_k}]$ ) is computed (line 26).