kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets

Claudio Lucchese[1], Salvatore Orlando[1], Paolo Palmerini[1,2],
Raffaele Perego[2], Fabrizio Silvestri[2,3]

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

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

[3]Dipartimento di Informatica
Università di Pisa
Pisa, Italy
silvestri@di.unipi.it

Abstract

This paper presents the implementation of kDCI*, an enhancement of* DCI *[10], a scalable algorithm for discovering frequent sets in large databases.*

The main contribution of kDCI *resides on a novel counting inference strategy, inspired by previously known results by Basted* et al. *[3]. Moreover, multiple heuristics and efficient data structures are used in order to adapt the algorithm behavior to the features of the specific dataset mined and of the computing platform used.*

kDCI *turns out to be effective in mining both short and long patterns from a variety of datasets. We conducted a wide range of experiments on synthetic and real-world datasets, both in-core and out-of-core. The results obtained allow us to state that* kDCI *performances are not over-fitted to a special case, and its high performance is maintained on datasets with different characteristics.*

1 Introduction

Despite the considerable amount of algorithms proposed in the last decade for solving the problem of finding frequent patterns in transactional databases (among the many we mention [1] [11] [6] [13] [14] [4] [3] [7]), a single *best* approach still has to be found.

The Frequent Set Counting (*FSC*) problem consists in finding all the set of items (itemsets) which occur in at least $s\%$ (s is called support) of the transactions of a database D, where each transaction is a variable length collection of items from a set I. Itemsets which verify the minimum support threshold are said to be *frequent*.

The complexity of the FSC problem relies mainly in the potentially explosive growth of its full search space, whose dimension d is, in the worst case, $d = \sum_{k=1}^{|t_{max}|} \binom{|I|}{k}$, where t_{max} is the maximum transaction length. Taking into account the minimum support threshold, it is possible to reduce the search space, using the well known *downward closure relation*, which states that an itemset can only be frequent if all its subsets are frequent as well. The exploitation of this property, originally introduced in the *Apriori* algorithm [1], has transformed a potentially exponentially complex problem, into a more tractable one.

Nevertheless, the *Apriori* property alone is not sufficient to permit to solve the FSC problem in a reasonable time, in *all* cases, i.e. on all possible datasets and for all possible interesting values of s. Indeed, another source of complexity in the FSC problem resides in the dataset internal correlation and statistical properties, which remain unknown until the mining is completed. Such diversity in the dataset properties is reflected in measurable quantities, like the total number of transactions, or the total number of distinct items $|I|$ appearing in the database, but also in some other more fuzzy properties which, although commonly recognized as important, still lack a formal and univocal definition. It is the case, for example, of the notion of how *dense* a dataset is, i.e. how much its transactions tend to resemble among one another.

Several important results have been achieved for specific cases. Dense datasets are effectively mined with compressed data structure [14], explosion in the candidates can be avoided using effective projections of the dataset [7], the support of itemsets in compact datasets

can be inferred, without counting, using an equivalence class based partition of the dataset [3].

In order to take advantage of all these, and more specific results, hybrid approaches have been proposed [5]. Critical to this point is *when* and *how* to adopt a given solution instead of another. In lack of a complete theoretical understanding of the FSC problem, the only solution is to adopt a heuristic approach, where theoretical reasoning is supported by direct experience leading to a strategy that tries to cover a variety of cases as wide as possible.

Starting from the previous DCI (Direct Count & Intersect) algorithm [10] we propose here kDCI, an enhanced version of DCI that extends its adaptability to the dataset specific features and the hardware characteristics of the computing platform used for running the FSC algorithm. Moreover, in kDCI we introduce a *novel counting inference* strategy, based on a new result inspired by the work of Bastide *et al.* in [3].

kDCI is a multiple heuristics hybrid algorithm, able to adapt its behavior during the execution. Since it origins from the already published DCI algorithm, we only outline in this paper how kDCI differs from DCI. A detailed description of the DCI algorithm can be found in [10].

2 The kDCI algorithm

Several considerations concerning the features of real datasets, the characteristics of modern hw/sw system, as well as scalability issues of FSC algorithms have motivated the design of kDCI. As already pointed out, transactional databases may have different characteristics in terms of correlations among the items inside transactions and of transactions among themselves [9]. A desirable feature of an FSC algorithm should be the ability to adapt its behavior to these characteristics.

Modern hw/sw systems need high locality for exploiting memory hierarchies effectively and achieving high performance. Algorithms have to favor the exploitation of spatial and temporal locality in accessing in-core and out-core data.

Scalability is the main concern in designing algorithms that aim to mine large databases efficiently. Therefore, it is important to be able to handle datasets bigger than the available memory.

We designed and implemented our algorithm kDCI keeping in mind such performance issues. The pseudo code of kDCI is given in Algorithm 1.

Algorithm 1 kDCI

```
Require: D, min_supp
  // During first scan get optimization figures
  F_1 = first_scan(D, min_supp)
  // second and following scans on a temporary db D'
  F_2 = second_scan(D', min_supp)
  k = 2
  while (D'.vertical_size() > memory_available()) do
    k++
    // count-based iteration
    F_k = DCP(D', min_supp, k)
  end while
  k++
  // count-based iteration + create vertical database VD
  F_k = DCP(D', VD, min_supp, k)
  dense = VD.is_dense()
  while (F_k ≠ ∅) do
    k++
    if (use_key_patterns()) then
      if (dense) then
        F_k = DCI_dense_keyp(VD, min_supp, k)
      else
        F_k = DCI_sparse_keyp(VD, min_supp, k)
      end if
    else
      if (dense) then
        F_k = DCI_dense(VD, min_supp, k)
      else
        F_k = DCI_sparse(VD, min_supp, k)
      end if
    end if
  end while
```

kDCI inherits from DCI the level-wise behavior and the hybrid horizontal-vertical dataset representation. As computation is started, kDCI maintains the database in horizontal format and applies an effective pruning technique to remove infrequent items and short transactions. A temporary dataset is therefore written to disk at every iteration. The first steps of the algorithm are described in [8] and [10] and remain unchanged in kDCI. In kDCI we only improved memory management by exploiting compressed and optimized data structures (see Section 2.1 and 2.2).

The effectiveness of pruning is related to the possibility of storing the dataset in main memory in vertical format, due to the dataset size reduction. This normally occurs at the first iterations, depending on the dataset, the support threshold and the memory available on the machine, which is determined at run time.

Once the dataset can be stored in main memory, kDCI switches to the vertical representation, and applies several heuristics in order to determine the most effective strategy for frequent itemset counting.

The most important innovation introduced in kDCI regards a novel technique to determine the itemset supports, inspired by the work of Bastide *et al.* [3]. As we will discuss in Section 2.4, in some cases the support of candidate itemsets can be determined without actually

counting transactions, but by a faster inference reasoning.

Moreover, kDCI maintains the different strategies implemented in DCI for sparse and dense datasets. The result is a multiple strategy approach: during the execution kDCI collects statistical information on the dataset that allows to determine which is the best approach for the particular case.

In the following we detail such optimizations and improvements and the heuristics used to decide which optimization to use.

2.1 Dynamic data type selection

The first optimization is concerned with the amount of memory used to represent itemsets and their counters. Since such structures are extensively accessed during the execution of the algorithm, is it profitable to have such data occupying as little memory as possible. This not only allows to reduce the spatial complexity of the algorithm, but also permits low level processor optimizations to be effective at run time.

During the first scan of the dataset, global properties are collected like the total number of distinct frequent items (m_1), the maximum transaction size, and the support of the most frequent item.

Once this information is available, we remap the survived (frequent) items to contiguous integer identifiers. This allows us to decide the best data type to represent such identifiers and their counters. For example if the maximum support of any item is less than 65536, we can use an `unsigned short int` to represent the itemset counters. The same holds for the remapped identifiers of the items. The decision of which is the most appropriate type to use for items and counters is taken at run time, by means of a C++ template-based implementation of all the kDCI code.

Before remapping item identifiers, we also reorder them in increasingly support ordering: more frequent items are thus assigned larger identifiers. This also simplifies the intersection-based technique used for dense datasets (see Section 2.3).

2.2 Compressed data structures

Itemsets are often organized in *collections* in many FSC algorithms. Efficient representation of such collections can lead to important performance improvements. In [8] we pointed out the advantages of storing candidates in directly accessible data structures for the first passes of our algorithm. In kDCI we introduce a compressed representation of an itemset collection, used to store in the main memory collections of candidate and frequent itemsets. This representation take advantage of prefix sharing among the lexicographically ordered itemsets of the collection.

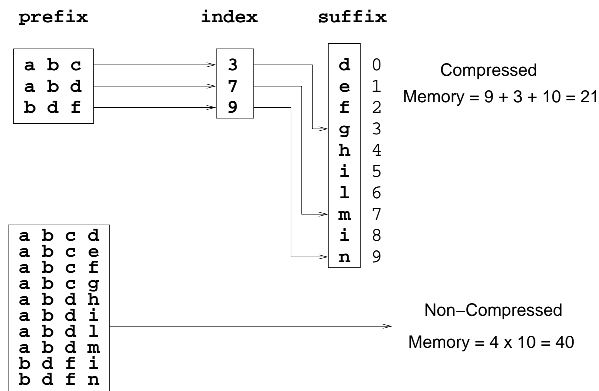


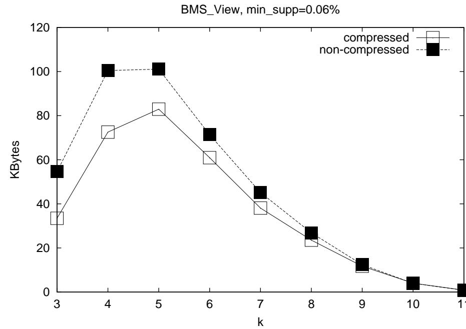
Figure 1. Compressed data structure used for itemset collection can reduce the amount of memory needed to store the itemsets.

The compressed data structure is based on three arrays (Figure 1). At each iteration k, the first array (`prefix`) stores the different prefixes of length $k-1$. In the third array (`suffix`) all the length-1 suffixes are stored. Finally, in the element i of the second array (`index`), we store the position in the `suffix` array of the section of suffixes that share the same prefix. Therefore, when the itemsets in the collection have to be enumerated, we first access the `prefix` array. Then, from the corresponding entry in the `index` array we get the section of suffixes stored in `suffix`, needed to complete the itemsets.

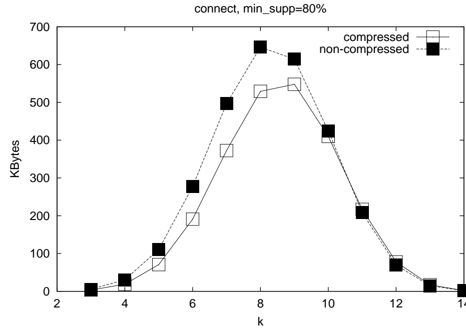
From our tests we can say that, in all the interesting cases – i.e., when the number of candidate (or frequent) iemsets explodes – this data structure works well and achieves up to 30% as compression ratio. For example, see the results reported in Figure 2.

2.3 Heuristics

One of the most important features of kDCI is its ability to adapt its behavior to the dataset specific characteristics. It is well known that being able to distinguish between sparse and dense datasets, for example, allows to adopt specific and effective optimizations. Moreover, as we will explain the Section 2.4, if the number of frequent itemsets is much greater than the number of closed itemsets, it is possible to apply a counting inference procedure that allows to dramatically reduce the time needed to determine itemset supports.



(a)



(b)

Figure 2. Memory usage with compressed itemsets collection representation for BMS with min_sup=0.06% (a) and connect with min_sup=80% (b)

In kDCI we devised two main heuristics that allow to distinguish between dense and sparse datasets and to decide whether to apply the counting inference procedure or not.

The first heuristic is simply based on the measure of the dataset density. Namely, we measure the correlation among the tidlists corresponding to the most frequent items. We require that the maximum number of frequent items for which such correlation is significant, weighted by the correlation degree itself, is above a given threshold.

As an example, consider the two dataset in Figure 3, where tidlists are placed horizontally, i.e. rows correspond to items and columns to transactions. Suppose to choose a density threshold $\delta = 0.2$. If we order the items according to their support, we have the most dense region of the dataset at the bottom of each figure. Starting from the bottom, we find the maximum number of items whose tidlists have a significant intersection. In the case of dataset (a), for example, a fraction $f = 1/4$ of the items share $p = 90\%$ of the transactions, leading

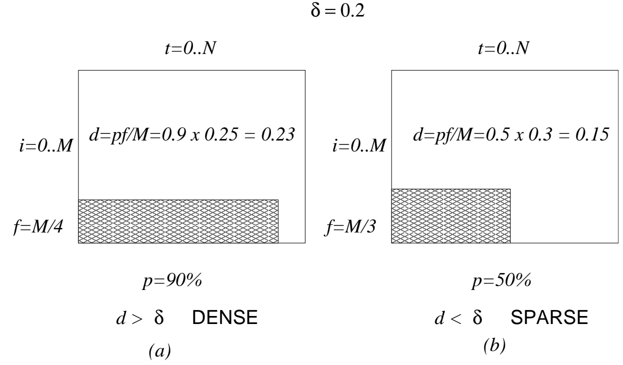


Figure 3. Heuristic to establish a dataset density or sparsity

to a density of $d = fp = 0.25 \times 0.9 = 0.23$ which is above the density threshold. For dataset (b) on the other hand, to a smaller intersection of $p = 50\%$ is common to $f = 1/3$ of the items. In this last case the density $d = fp = 0.3 \times 0.5 = 0.15$ is lower than the threshold and the dataset is considered as sparse. It is worth to notice that since this notion of density depends on the minimum support threshold, the same dataset can exhibits different behaviors when mined with different support thresholds.

Once the dataset density is determined, we adopted the same optimizations described in [10] for sparse and dense datasets. We review them briefly for completeness.

Sparse datasets. The main techniques used for sparse datasets can be summarized as follows:

– **projection**. Tidlists in sparse datasets are characterized by long runs of 0's. When intersecting the tidlists associated with the 2-prefix items belonging to a given candidate itemset, we keep track of such empty elements (words), in order to perform the following intersections faster. This can be considered as *a sort of raw projection* of the vertical dataset, since some transactions, i.e. those corresponding to zero words, are not considered at all during the following tidlist intersections.

– **pruning**. We remove infrequent items from the dataset. This can result in some transaction remaining empty or with too few items. We therefore remove such transactions (i.e. columns in the our bitvector vertical representation) from the dataset. Since this bitwise

pruning may be expensive, we only perform it when the benefits introduced are expected to balance its cost.

Dense datasets.

If the dataset is dense, we expect to deal with strong correlations among the most frequent items. This not only means that the tidlists associated with these *most frequent items* contain long runs of 1's, but also that they turn out to be very similar. The heuristic technique adopted by DCI and consequently by kDCI for dense dataset thus works as follows:

- we reorder the columns of the vertical dataset by moving identical segments of the tidlists associated with the most frequent items to the first consecutive positions;
- since each candidate is likely to include several of these most frequent items, we avoid repeated intersections of identical segments.

The heuristic for density evaluation is applied only once, as soon as the vertical dataset is built. After this decision is taken, we further check if the counting inference strategy (see Section 2.4) can be profitable or not. The effectiveness of the inference strategy depends on the ratio between the total number of frequent itemsets and how many of them are *key-patterns*. The closer to 1 this ratio is, the less advantage is introduced by the inference strategy. Since this ratio is not known until the computation is finished, we found that the same information can be derived from the average support of the frequent singletons (items), after the first scan. The idea behind this is that if the average support of the single items that survived the first scan is high enough, then longer patterns can be expected to be frequent and more likely the number of *key-patterns* itemsets will be lower than that of frequent itemsets. We experimentally verified that this simple heuristic gives the correct output for all datasets - both real and synthetic.

To resume the rationale behind kDCI multiple strategy approach, if the *key-patterns* optimization can be adopted, we use the counting inference method that allows to avoid many intersections. For the intersections that cannot be avoided and in the cases where the *key-patterns* inference method cannot be applied, we further distinguish between sparse and dense datasets, and apply the two strategies explained above.

2.4 Pattern Counting Inference

In this section we describe the count inference method, which constitute the most important innovation introduced in kDCI. We exploit a technique inspired by the theoretical results presented in [3], where the PASCAL algorithm was introduced. PASCAL is able to infers the support of an itemset without actually count its occurrences in the database. In this seminal work, the authors introduced the concept of *key pattern* (or *key itemset*). Given a generic pattern Q, it is possible to determine an equivalence class $[Q]$, which contains the set of patterns that have the same support and are included in the same set of database transactions. Moreover, if we define $min[P]$ as the set of the smallest itemsets in $[P]$, a pattern P is a *key pattern* if $P \in \min[P]$, i.e. no proper subset of P is in the same equivalence class. Note that we can have several key patterns for each equivalence class. Figure 4 shows an example of a lattice of frequent itemsets, taken from [3], where equivalence classes and key patterns are highlighted.

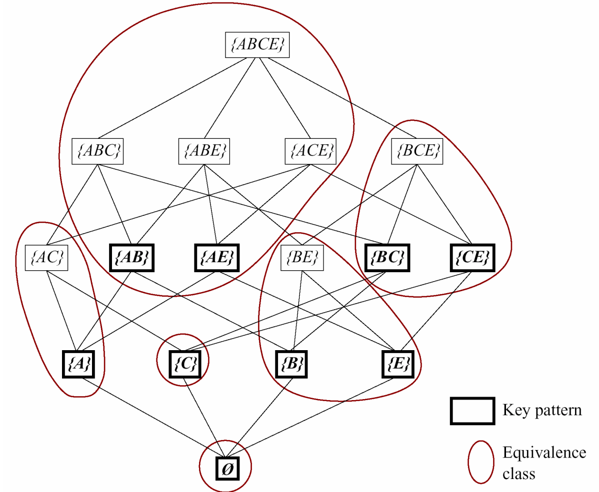


Figure 4. Example lattice of frequent items.

Given an equivalence class $[P]$, we can also define a corresponding *closed set* [12]: the closed set c of $[P]$ is equal to $\max[P]$, so that no proper supersets of c can belong to the same equivalence class $[P]$.

Among the results illustrated in [3] we have the following important theorems:

Theorem 1 *Q is a key pattern iff $supp(Q) \neq \min_{p \in Q}(supp(Q \setminus \{p\}))$.*

Theorem 2 *If P is not a key pattern, and $P \subseteq Q$, then Q is a non-key pattern as well.*

From Theorem 1 it is straightforward to observe that if Q is a non-key pattern, then:

$$supp(Q) = \min_{p \in Q}(supp(Q \setminus \{p\})). \quad (1)$$

Moreover, Theorem 1 says that we can check whether Q is a *key pattern* by comparing its support with the minimum support of its proper subsets, i.e. $\min_{p \in Q}(supp(Q \setminus \{p\}))$. We will show in the following how to use this property to make faster candidate support counting.

Theorems 1 and 2 give the theoretical foundations for the PASCAL algorithm, which finds the support of a *non-key* k-candidate Q by simply searching the minimum supports of all its $k-1$ subsets. Note that such search can be performed during the pruning phase of the Apriori candidate generation. DCI does not perform candidate pruning because its intersection technique is comparably faster. For this reason we will not adopt the PASCAL counting inference in kDCI.

The following theorem, partially inspired by the proof of Theorem 2, suggests a faster way to compute the support of a *non-key* k-candidate Q.

Before introducing the theorem, we need to define the function f, which assigns to each pattern P the set of all the transactions that include this pattern. We can define the support of a pattern in terms of f: $supp(P) = |f(P)|$. Note that f is a monotonically decreasing function, i.e. if $P_1 \subseteq P_2 \Rightarrow f(P_2) \subseteq f(P_1)$. This is obvious, because every transaction containing P_2 surely contains all the subsets of P_2.

Theorem 3 *If P is a non-key pattern and $P \subseteq Q$, the following holds:*

$$f(Q) = f(Q \setminus \ (P \setminus P')).$$

where $P' \subset P$, and P and P' belong to the same equivalence class, i.e. $P, P' \in [P]$.

PROOF. Note that, since P is a *non-key pattern*, it is surely possible to find a pattern P', $P' \subset P$, belonging to the same equivalence class $[P]$.

In order to demonstrate the Theorem we first show that $f(Q) \subseteq f(Q \setminus \ (P \setminus P'))$ and then that also $f(Q) \supseteq f(Q \setminus \ (P \setminus P'))$ holds, thus proving the Theorem hypotheses.

The first assertion $f(Q) \subseteq f(Q \setminus \ (P \setminus P'))$ holds because $(Q \setminus (P \setminus P')) \subseteq Q$, and f is a monotonically decreasing function.

To prove the second assertion, $f(Q) \supseteq f(Q \setminus \ (P \setminus P'))$, we can rewrite $f(Q)$ as $f(Q \setminus (P \setminus P') \cup (P \setminus P'))$, which is equivalent to $f(Q \setminus (P \setminus P')) \cap f(P \setminus P')$.

Since f is decreasing, $f(P) \subseteq f(P \setminus P')$. But, since $P, P' \in [P]$, then we can write $f(P) = f(P') \subseteq f(P \setminus P')$. Therefore $f(Q) = f(Q \setminus (P \setminus P')) \cap f(P \setminus P') \supseteq f(Q \setminus (P \setminus P')) \cap f(P')$. The last inequality is equivalent to $f(Q) \supseteq f(Q \setminus (P \setminus P') \cup P')$. Since $P' \subseteq (Q \setminus (P \setminus P'))$ clearly holds, it follows that $f(Q \setminus (P \setminus P') \cup P') = f(Q \setminus (P \setminus P'))$. So we can conclude that $f(Q) \supseteq f(Q \setminus (P \setminus P'))$, which completes the proof.
□

The following corollary is trivial, since we defined $supp(Q) = |f(Q)|$.

Corollary 1 *If P is a non-key pattern, and $P \subseteq Q$, the support of Q can be computed as follows:*

$$supp(Q) = supp(Q \setminus \ (P \setminus P'))$$

where P' and P, $P' \subset P$, belong to the same equivalence class, i.e. $P, P' \in [P]$.

Finally, we can introduce Corollary 2, which is a particular case of the previous one.

Corollary 2 *If Q is k-candidate (i.e. $Q \in C_k$) and P, $P \subset Q$, is a frequent non-key* (k-1)*-pattern (i.e. $P \in F_{k-1}$), there must exist $P' \in F_{k-2}$, $P' \subset P$, such that P and P' belong to the same equivalence class, i.e. $P, P' \in [P]$ and P and P' differ for a single item: $\{p_{diff}\} = P \setminus P'$. The support of Q can thus be computed as:*

$$supp(Q) = supp(Q \setminus \ (P \setminus P')) = supp(Q \setminus \{p_{diff}\})$$

Corollary 2 says that to find the support of a *non-key candidate pattern* Q, we can simply check whether $Q \setminus \{p_{diff}\}$ belongs to F_{k-1}, or not. If $Q \setminus \{p_{diff}\} \in F_{k-1}$, then Q inherits the same support as $Q \setminus \{p_{diff}\}$ and is therefore frequent. Otherwise we can conclude that $Q \setminus \{p_{diff}\}$ is not frequent.

Using the theoretical result of Corollary 2, we adopted the following strategy in order to determine the support of a candidate Q at step k.

In kDCI, we store with each itemset $P \in F_{k-1}$ the following information:

- $supp(P)$;
- a flag indicating if P is a *key pattern* or not;
- if P is *non-key pattern*, also the item p_{diff} such that $P \setminus \{p_{diff}\} = P' \in [P]$.

Note that p_{diff} must be one of the items that we can remove from P to obtain a proper subset P' of P, belonging to the same equivalence class.

During the generation of a generic candidate $Q \in C_k$, as soon as kDCI discovers that one of the subsets of Q,

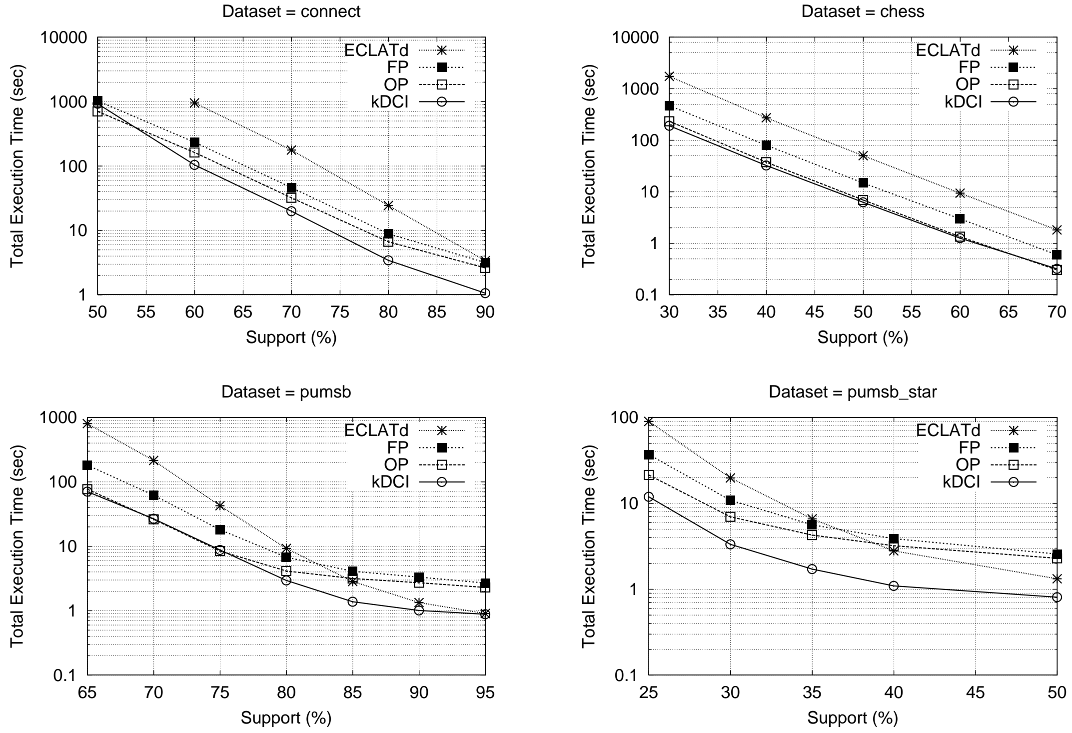


Figure 5. Total execution time of OP, FP, Eclatd, and kDCI on various datasets as a function of the support threshold.

say P, is a *non-key pattern*, kDCI searches in F_{k-1} the pattern $Q \setminus \{p_{diff}\}$, where p_{diff} is stored with P.

If $Q \setminus \{p_{diff}\}$ is found, then Q is a frequent *non-key pattern* (see Theorem 2), its support is $supp(Q \setminus \{p_{diff}\})$, and the item to store with Q is exactly p_{diff}. In fact, $Q' = Q \setminus \{p_{diff}\} \in [Q]$, i.e. p_{diff} is one of the items that we can remove from Q to obtain a subset Q' belonging to the same equivalence class.

The worst case is when all the subsets of Q in F_{k-1} are *key patterns* and the support of Q cannot be inferred from its subsets. In this case kDCI counts the support of Q as usual, and applies Theorem 1 to determine if Q is a *non-key-pattern*. If Q is a *non-key-pattern*, its support becomes $supp(Q) = \min_{p \in Q}(supp(Q \setminus \{p\}))$ (see Theorem 1), while the item to be stored with Q is p_{diff}, i.e. the item to be subtracted from Q to obtain the pattern with the minimum support.

The impact of this counting inference technique on the performance of an FSC algorithm becomes evident if you consider the Apriori-like candidate generation strategy adopted by kDCI. From the combination of every pair of itemsets P_a and $P_b \in F_{k-1}$, that share the same *(k-2)*-prefix (we called them *generators*), kDCI generates a candidate k-itemset Q. For very dense datasets, most of the frequent patterns belonging to F_{k-1} are *non-key patterns*. Therefore one or both patterns P_a and P_b used to generate $Q \in C_k$ are likely to be *non-key patterns*. In such cases, in order to find a *non-key pattern* and then apply Corollary 2, it is not necessary to check the existence of further subsets of Q. For most of the candidates, a single binary search in F_{k-1}, to look for the pattern $Q \setminus \{p_{diff}\}$, is thus sufficient to compute $supp(Q)$. Moreover, often $Q \setminus \{p_{diff}\}$ is exactly equal to one of the two *k-1*-itemsets belonging to the generating pair (P_a, P_b): in this case kDCI does not need to perform any search at all to compute $supp(Q)$.

We conclude this section with some examples of how the counting inference technique works. Let us consider Figure 4. Itemset $Q = \{A, B, E\}$ is a *non-key pattern* because $P = \{B, E\}$ is a *non-key pattern* as well. So, if $P' = \{B\}$, kDCI will store $p_{diff} = E$ with P. We have that the $supp(\{A, B, E\}) = supp(\{A, B, E\} \setminus (\{B, E\} \setminus \{B\})) = supp(\{A, B, E\} \setminus \{p_{diff}\} = supp(\{A, B\})$. From the Figure you can see that $\{A, B, E\}$ and $\{A, B\}$ both belong to the same

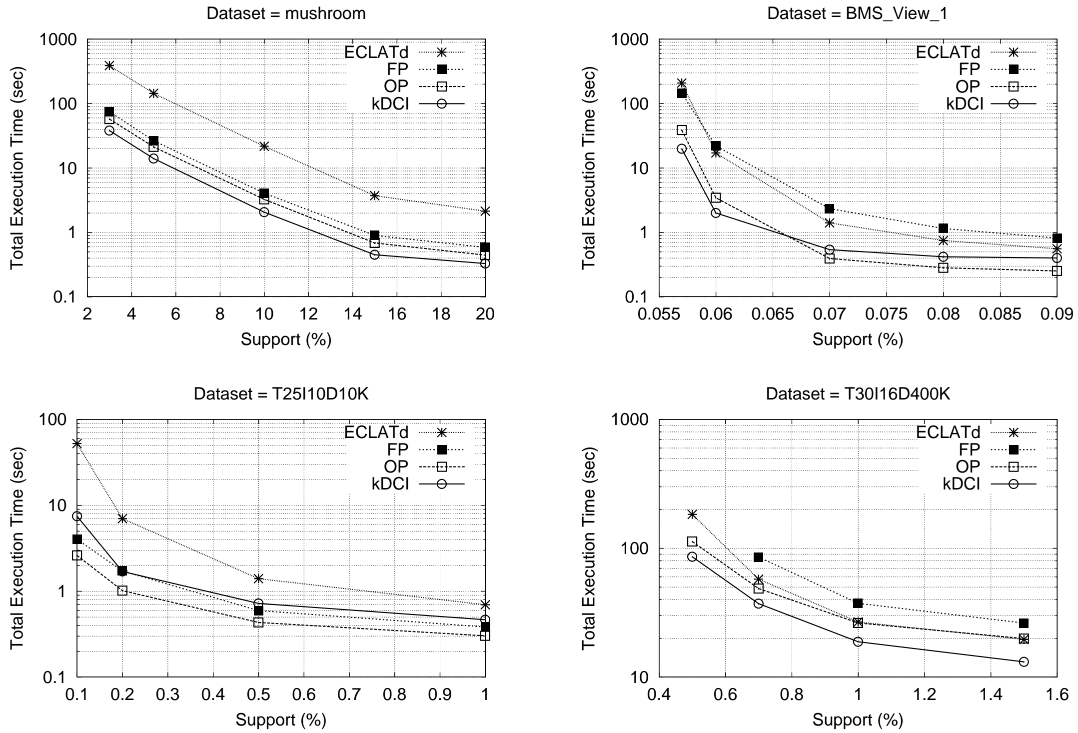


Figure 6. Total execution time of OP, FP, Eclatd, and kDCI on various datasets as a function of the support threshold.

equivalence class.
Another example is itemset $Q = \{A, B, C, E\}$, that is generated by the two *non-key patterns* $\{A, B, C\}$ and $\{A, B, E\}$. Suppose that $P = \{A, B, C\}$, i.e. the first generator, while $P' = \{A, B\}$. In this case kDCI will store $p_{diff} = C$ with P. We have that the $supp(\{A, B, C, E\}) = supp(\{A, B, C, E\} \setminus (\{A, B, C\} \setminus \{A, B\})) = supp(\{A, B, C, E\} \setminus \{p_{diff}\} = supp(\{A, B, E\}$, where $\{A, B, E\}$ is exactly the second generator. In this case, no search is necessary to find $\{A, B, E\}$. Looking at the Figure, it is possible to verify that $\{A, B, C, E\}$ and $\{A, B, E\}$ both belong to the same equivalence class.

3 Experimental Results

We experimentally evaluated kDCI performances by comparing its execution time with respect to the original implementations of state of the art FSC algorithms, namely FP-growth (FP) [6], Opportunistic Projection (OP) [7] and Eclat with diffsets (Eclatd) [14], provided by their respective authors.

We used a MS-WindowsXP workstation equipped with a Pentium IV 1800 MHz processor, 368MB of RAM memory and an eide hard disk. For the tests, we used both synthetic and real-world datasets. All the synthetic datasets used were created with the IBM dataset generator [1], while all the real-world datasets but one were downloaded from the UCI KDD Archive (`http://kdd.ics.uci.edu/`). We also extracted a real-world dataset from the TREC WT10g corpus [2]. The original corpus contained about 1.69 millions of WEB documents. The dataset for our tests was built by considering the set of all the terms contained in each document as a transaction. Before generating the transactional dataset, the collection of documents was filtered by removing HTML tags and the most common words (*stopwords*), and by applying a *stemming* algorithm. The resulting `trec` dataset is huge. It is about 1.3GB, and contains 1.69 millions of short and long transactions, where the maximum length of a transaction is $71,473$ items.

kDCI performance and comparisons. Figure 5 and 6 report the total execution time obtained running FP, Eclatd, OP, and kDCI on various datasets as a func-

tion of the support threshold s. On all datasets in Figure 5, `connect`, `chess` `pumsb` and `pumsb_star`, kDCI runs faster than the others algorithms. On `pumsb` its execution time is very similar to the one of OP. For high support thresholds kDCI can drastically prune the dataset, and build a compact vertical dataset, whose tidlists presents large similarities. Such similarity of tidlists is effectively exploited by our strategy for compact datasets. For smaller supports, the benefits introduced by the counting inference strategy become more evident, particularly for the `pumsb_star` and `connect` datasets. In these cases the number of frequent itemsets is much higher than the number of *key-patterns*, thus allowing kDCI to drastically reduce the number of intersections needed to determine candidate supports.

On the datasets `mushroom` and `T30I16D400K` (see Figure 6), kDCI outperforms the other competitors, and this also holds on the real-world dataset `BMS_View_1` when mined with very small support thresholds (see Figure 6). On only a dataset, namely `T25I10D10K`, FP and OP are faster than kDCI for all the supports. The reason of this behavior is the size of the candidate set C_3, which for this dataset is much larger than F_3. While kDCI has to carry out a lot of useless work to determine the support of many candidate itemsets which are not frequent, FP-growth and OP take advantage of the fact that they do not require candidate generation.

Furthermore, differently from FP, Eclatd, and OP, kDCI can efficiently mine huge datasets such as `trec` and `USCensus1990`. Figure 7 reports the total execution time required by kDCI to mine these datasets with different support thresholds. The other algorithms failed in mining these datasets due to memory shortage, also when very large support thresholds were used. On the other hand, kDCI was able to mine such huge datasets since it adapts its behavior to both the size of the dataset and the main memory available.

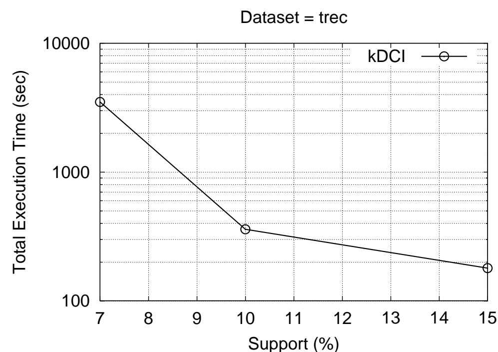
4 Conclusions and Future Work

Due to the complexity of the problem, a good algorithm for FSC has to implement multiple strategies and some level of adaptiveness in order to be able to successfully manage diverse and differently featured inputs.

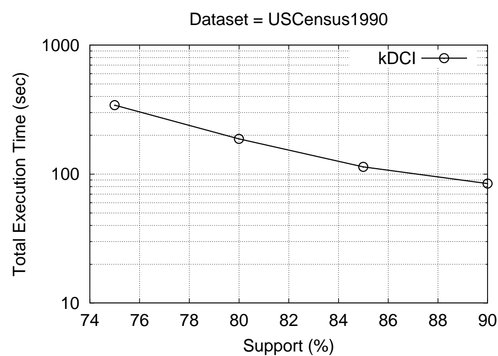
kDCI uses different approaches for extracting frequent patterns: count-based during the first iterations and intersection-based for the following ones.

Moreover, a new counting inference strategy, together with, adaptiveness and resource awareness are the main innovative features of the algorithm.

On the basis of the characteristics of the mined dataset, kDCI chooses which optimization to adopt for



(b)



(a)

Figure 7. Total execution time of kDCI: on datasets `trec` (a) and `USCensus1990` (b) as a function of the support.

reducing the cost of mining at run–time. Dataset pruning and effective out-of-core techniques are exploited during the count-based phase, while the intersection-based phase, which starts only when the pruned dataset can fit into the main memory, exploits a novel technique based on the notion of *key-pattern* that in many cases allows to infer the support of an itemset without any counting.

kDCI also adopts compressed data structures and dynamic type selection to adapt itself to the characteristics of the dataset being mined.

The experimental evaluation demonstrated that kDCI outperforms FP, OP, and Eclatd in most cases. Moreover, differently from the other FSC algorithms tested, kDCI can efficiently manage very large datasets, also on machines with limited physical memory.

Although the variety of datasets used and the large amount of tests conducted permit us to state that the performance of kDCI is not highly influenced by dataset

characteristics, and that our optimizations are very effective and general, some further optimizations and future work will reasonably improve kDCI performance. More optimized data structures could be used to store itemset collections in order to make faster searches in such collections. Note that such fast searches are very important in kDCI, which bases its count inference technique at level k on searching for frequent itemset in F_{k-1}. Finally, we can benefit from a higher level of adaptiveness to the available memory on the machine, either with fully memory mapped data structures or with out-of-core ones, depending on the data size. This should allow a better scalability and a wider applicability of the algorithm.

5 Acknowledgments

We acknowledge J. Han, Y. Pan, M.J. Zaki and J. Liu for kindly providing us the latest versions of their FSC software.

References

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

[2] P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for Web retrieval experiments. *Information Processing and Management*. to appear.

[3] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *ACM SIGKDD Explorations Newsletter*, 2(2):66–75, December 2000.

[4] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 05.

[5] B. Goethals. *Efficient Frequent Itemset Mining*. PhD thesis, Limburg University, Belgium, 2003.

[6] 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.

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

[8] S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of 3rd Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 01) - Munich, Germany*, volume 2114 of *LNCS*, pages 71–82. Springer, 2001.

[9] S. Orlando, P. Palmerini, and R. Perego. On Statistical Properties of Transactional Datasets. In *2004 ACM Symposium on Applied Computing (SAC 2004)*, 2004. To appear.

[10] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and Resource-Aware Mining of Frequent Sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 338–345, 2002.

[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 Int. Conf. on Management of Data*, pages 175–186, 1995.

[12] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *Lecture Notes in Computer Science*, 1540:398–416, 1999.

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

[14] M. J. Zaki and K. Gouda. Fast Vertical Mining Using Diffsets. In *9th Int. Conf. on Knowledge Discovery and Data Mining, Washington, DC*, 2003.