How to find frequent patterns?

Wim Pijls[*] Walter A. Kosters[†]

Econometric Institute Report EI 2005-24

June 1, 2005[‡]

Abstract

An improved version of $\mathcal{DF}$, the depth first implementation of Apriori as devised in [7], is presented. Given a database of (e.g., supermarket) transactions, the $\mathcal{DF}$ algorithm builds a so-called trie that contains *all* frequent itemsets, i.e., all itemsets that are contained in at least *minsup* transactions with *minsup* a given threshold value. In the trie, there is a one-to-one correspondence between the paths and the frequent itemsets. The new version, called $\mathcal{DF}^+$, differs from $\mathcal{DF}$ in that its data structure representing the database is borrowed from the FP-growth algorithm. So it combines the compact FP-growth data structure with the efficient trie-building method in $\mathcal{DF}$.

1 Introduction

Finding frequent itemsets, also called patterns, in large databases has become a major research topic in the past decade. Initially, frequent itemsets were used for searching association rules in supermarket transaction data. An example of such an association rule is: 15% of the transactions that include item X and item Y, also include item Z. Besides, frequent itemsets are used in several other data mining issues such as classification, clustering, correlations, episodes and so on. Application domains nowadays are, among others, direct mailing, customer relationship management and fraud detection.

One of the oldest algorithms for finding frequent itemsets is Apriori [1, 2], which is based upon the *Apriori-property* that, for an itemset to be frequent, certainly any subset must be frequent. The Apriori algorithm finds the frequent patterns in a breadth-first way. In 1999 we proposed $\mathcal{DF}$[7], a depth first algorithm based upon this same property. A major step forward was the FP-growth algorithm, also denoted as $\mathcal{FP}$ [4]. In this algorithm the database is represented by a so-called Frequent Pattern tree (in short *FP-tree*), a compact data structure that contains all information needed to find the frequent itemsets. This paper presents a new variant of $\mathcal{DF}$, called $\mathcal{DF}^+$. In the original $\mathcal{DF}$ implementation the database was kept in memory as a two-dimensional array. In the new algorithm this array

[*]Econometric Institute, Erasmus University Rotterdam, P.O.Box 1738, 3000 DR Rotterdam, The Netherlands, e-mail: pijls@few.eur.nl

[†]Leiden Institute of Advanced Computer Science, Universiteit Leiden, P.O. Box 9512, 2300 RA Leiden, The Netherlands, e-mail kosters@liacs.nl

[‡]This paper was presented at: Operations Research 2004, Annual International Conference of the German Operations Research Society, Tilburg, The Netherlands, September 2004.

is replaced with an FP-tree. This requires less memory as well as less memory inspections. For sparse transaction databases, the combination of $\mathcal{DF}$ with an FP-tree turns out to be fast.

Preliminaries. A transaction database consists of rows that represent transactions or records, and columns with each column corresponding to an item. Each entry contains a boolean value, denoting whether an item is included or not in a transaction. Any database with only discrete values can be transformed into such a format. A transaction or record R is said to support an itemset I if R contains the full set I. The *support* of an itemset I is the number of transactions that support I. An itemset is *frequent* if its support exceeds the minimum support *minsup*, a threshold value given in advance.

Overview. This paper is organized as follows. In Section 2 we discuss $\mathcal{DF}$. Section 3 first introduces the FP-tree and the FP-growth algorithm. Next, we show how an FP-tree can be incorporated into the depth first algorithm, leading to the improved $\mathcal{DF}^+$ algorithm. Section 4 shows results of experiments; Section 5 concludes.

2 The Depth First Algorithm

Suppose that we are given the database on the left hand side of Figure 1, which serves as our running example. Each line represents a transaction. Let us take the support threshold equal to 3. The frequent itemsets, the aim of our algorithm, are presented in the right hand side of the figure. Abusing notation, we omit curly braces and commas from the common set notation.

Database

nr.	items
1	A D
2	A B C E
3	A B C E
4	A B C D
5	A B C D E
6	B D E

Frequent itemsets for $minsup = 3$

support	frequent itemsets
5	A, B
4	AB, C, AC, BC, ABC, D, E, BE
3	AD, BD, AE, CE, ABE,
	ACE, BCE ABCE

Figure 1: Example database with its frequent itemsets.

An appropriate data structure to store the frequent itemsets of a given database is the well-known *trie*. The trie of frequent patterns is shown in Figure 2. The entries (or cells) in a node of a trie are usually called *buckets*, as is also the case for a hash-tree. Each bucket can be identified with its path to the root and hence with a unique frequent itemset. The example final trie has 9 nodes and 18 buckets, representing the 18 frequent itemsets. As an example, the frequent itemset {A,B,E,F} can be seen as the leftmost path in the trie; an infrequent set as {A,B,C} is not present.

The $\mathcal{DF}$ algorithm starts by determining the support for all single items. For this purpose the database is examined once. The items are sorted in decreasing order. Let the n frequent items in decreasing order be denoted by $i_1, i_2, \ldots, i_n$. Next, the code from Figure 3 is executed. Figure 4 illustrates the consecutive steps of the algorithm applied to our example. The single items surpassing the minimum support threshold 3 are: $i_1 =$ A, $i_2 =$ B, $i_3 =$ C, $i_4 =$ D and $i_5 =$ E. In the figure, the shape of T after each iteration of the main loop is shown. Also the infrequent itemsets to be deleted at the end of an iteration

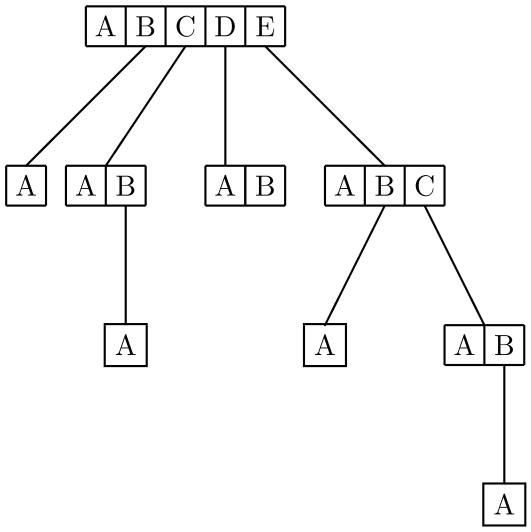

Figure 2: The trie of frequent item sets (without support counts).

```
(1)   $T :=$ the trie including only bucket $i_1$;
(2)   for $m := 2$ to $n$ do
(3)       $T' := T$;
(4)       $T := T'$ with $i_m$ added to the right and
                a copy of $T'$ appended to $i_m$;
(5)       $S := T \backslash T'$ (= the subtrie rooted in $i_m$);
(6)       count$(S, i_m)$;
(7)       delete the infrequent itemsets from $S$;

(9)   procedure count(subtrie $S$, item $i$):
(10)      for every transaction $R$ containing $i$ do
(11)          for every itemset (path) $I$ in $S$ do
(12)              if $R$ supports $I$ then increase $I$.support;
```

Figure 3: The $\mathcal{DF}$ algorithm.

are mentioned. At the start of the iteration with index m, the root of trie T consists of the 1-itemsets $i_1, \ldots, i_{m-1}$. (We denote a 1-itemset by the name of its only item.) A new trie T is composed by adding bucket i_m to the root and by appending a copy of T' (the former T) to i_m. The newly added buckets are the new candidates and they make up a subtrie S. In Figure 4, the candidate set S is in the right part of each trie and is drawn in bold. Notice that the final trie (after deleting infrequent itemsets) is identical to Figure 2.

The procedure *count*(S, i) determines the support of each itemset (i.e., bucket) in the subtrie S. In lines *(11)* and *(12)* of the code, each path (itemset) I of S is traversed and compared with transaction R, increasing the appropriate support counters by 1. The traversal of I is aborted as soon as an item outside R is found on I. All paths of subtrie S are traversed using backtracking.

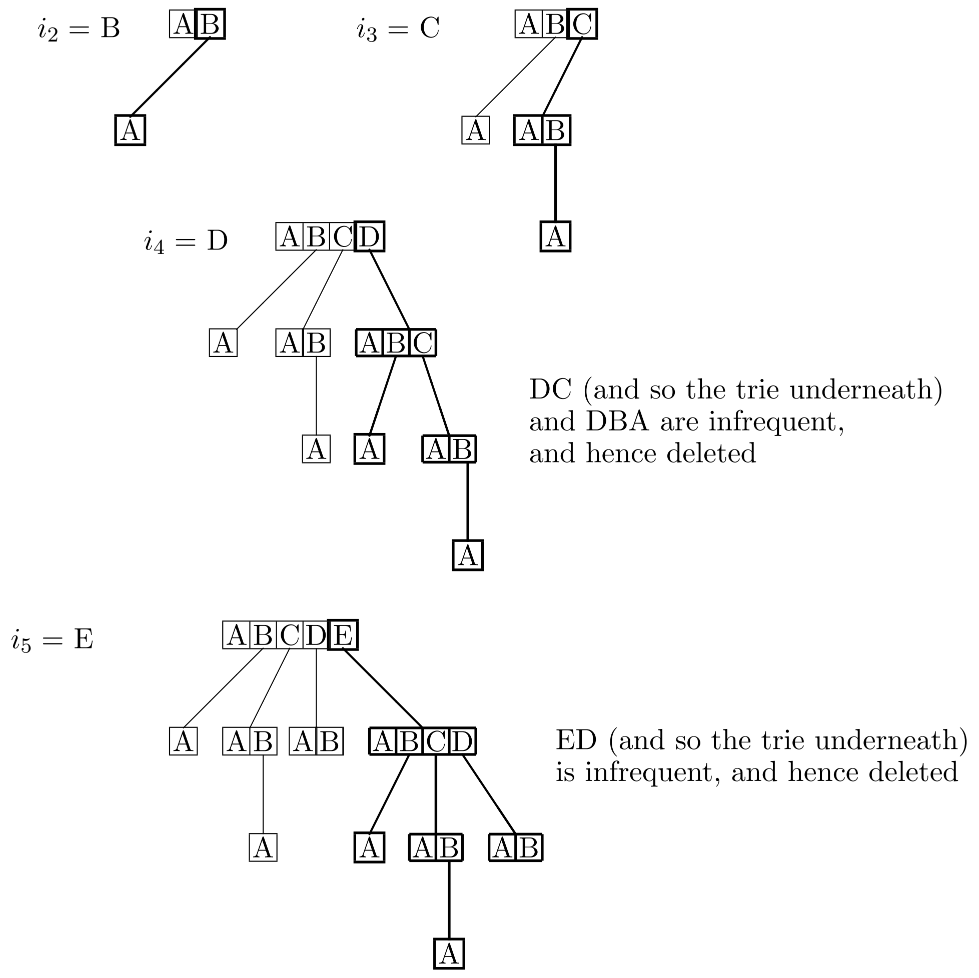
$i_2 = $ B
$i_3 = $ C
$i_4 = $ D
DC (and so the trie underneath)
and DBA are infrequent,
and hence deleted
$i_5 = $ E
ED (and so the trie underneath)
is infrequent, and hence deleted

Figure 4: Illustrating the $\mathcal{DF}$ algorithm.

3 The FP-Tree Improvement

An *FP-tree* is an efficient and compact data structure for a transaction database. In an FP-tree each node contains an item, a counter, pointers to its children (if any), a pointer to its parent and an "extra" pointer to be discussed later on. A transaction is inserted into the FP-tree constructed so far, by following a unique path, meanwhile increasing all counters on this path by 1. The path consists of the *frequent* items in the transaction in the prescribed order; if necessary, new nodes are created. To be precise, if the transaction includes item i, we look for child i of the current node in the FP-tree; if this child does not exist, it is created (with counter initialized to 1), otherwise its counter is incremented by 1; the current node is set to the (new) child, and the insertion process proceeds. In this way, transactions with the same prefix follow the same path as long as they coincide. The FP-tree for our example database is depicted in Figure 5. The dotted lines from left to right indicate the "extra" pointers that connect all occurrences of one item in the tree. Note that itemset {A,B,C} has support 4, while the support of an itemset like {B,D} is recorded as $2 + 1 = 3$.

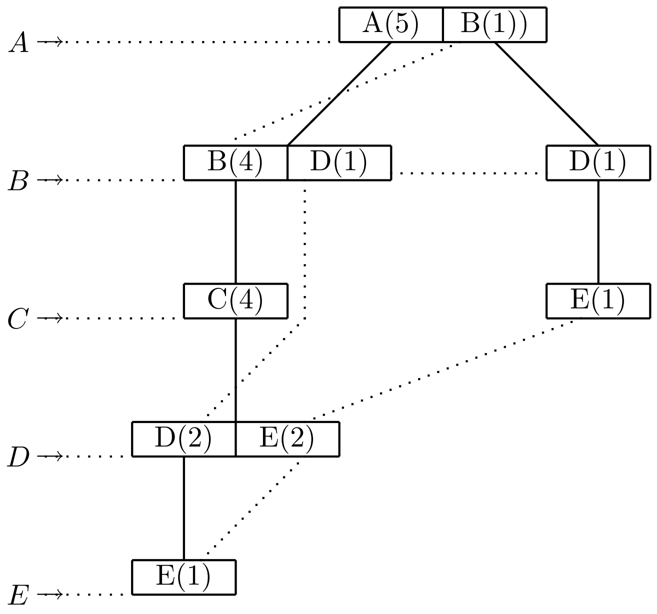

Figure 5: The FP-tree of the database.

FP-growth. As mentioned earlier, the FP-tree plays a crucial role in the FP-growth algorithm, see Figure 6. The main call of FP-growth is *FP-growth($\emptyset, D, I$)* with D the given database and I the original set of frequent items. A precondition of the procedure *FP-growth* is that I is the set of frequent items in D. An essential feature of FP-growth is that parameter D is given as an FP-tree in any (sub)call. An extra action (*clean*) is performed in line *(7)*: the items in the set $\{i_1, i_2, \ldots, i_{k-1}\}$ with a support lower than *minsup* in D' are not inserted into I' to make sure that the precondition of the subsequent subcall is fulfilled. Note that *FP-growth* is a typical instance of backtracking.

```
(1)  procedure FP-growth(pattern P, database D, itemset I)
(2)  if $D = \emptyset$ then exit;
(3)  let I be $\{i_1, i_2, \ldots, i_n\}$;
(4)  for $k := 1$ to $n$ do
(5)      "output" a new pattern $P' := P \cup \{i_k\}$;
(6)      $D'$ := the set of transactions in D that support $i_k$;
(7)      $I'$ := clean($\{i_1, i_2, \ldots, i_{k-1}\}$);
(8)      FP-growth($P', D', I'$);
```

Figure 6: The FP-growth algorithm.

A new algorithm. Although FP-growth is fast, a major drawback is that in each recursive call a new FP-tree D' has to be built. The depth first algorithm has another drawback. In each outer iteration, a call of the procedure *count* is performed. The execution of this procedure traverses the whole database. Now, we propose a new algorithm, called $\mathcal{DF}^+$, which solves both problems. The code of Figure 3 is performed, where the database is stored in memory not as a two-dimensional array, but as an FP-tree. So, before the code of Figure 3 starts, the whole database is put into memory as an FP-tree. The procedure *count* utilizes this FP-tree. The execution of the call *count(S,i)* in $\mathcal{DF}^+$ includes considerably less steps compared with $\mathcal{DF}$. The latter one must inspect all transactions of the database, whereas the former visits only the paths between the FP-root and the buckets containing item i, using the extra dashed pointers (so line *(10)* from Figure 3 uses paths instead of transactions). Note that the counter in the FP-tree node is used in line *(12)* from Figure 3. Consequently, using an FP-tree results in a significant speed up. An extra speedup is achieved by (in the main loop from Figure 3) first discarding the infrequent itemsets with 2 elements (one of them being i_m); in the example tree {D,C} and {E,D} are deleted in an early stage, thereby eliminating much counting in their respective subtries.

4 Experiments

The following databases, all available through [3], are used:

- `chess` (342 kB, with 3,196 transactions),
- `accidents` (25.5 MB, with 340,183 transactions),
- `T10I4D100K` (4.0 MB, with 100,000 transactions),
- `T40I10D100K` (15.5 MB, with 100,000 transactions).

These databases have either few, but coherent records (`chess`), or many records (`accidents` and the two synthetic `T`-databases).

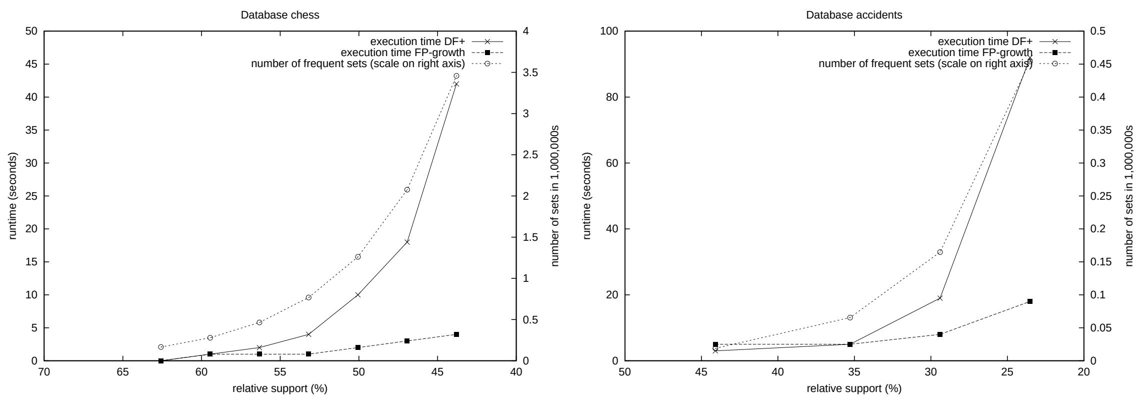

Figure 7: Experimental results for databases `chess` and `accidents`.

The experiments were conducted at a Pentium-IV machine with 512 MB memory at 2.8 GHz, running Red Hat Linux 7.3. The programs were developed under the GNU C++ compiler, version 2.96.

The following statistics are plotted in the graphs: the execution time in seconds of the $\mathcal{DF}^+$ and $\mathcal{FP}$ algorithms (scale on left axis), and the total number of frequent itemsets: in all figures the corresponding axis is on the right hand side. The execution time excludes preprocessing: in this phase the database is read in order to detect the frequent items (see before); also excluded is the time needed to print the resulting itemsets. These actions together usually only take a few seconds. The number of frequent 1-itemsets has range 31–38 for the experiments on the database `chess`, 28–40 for `accidents`, 844–869 for

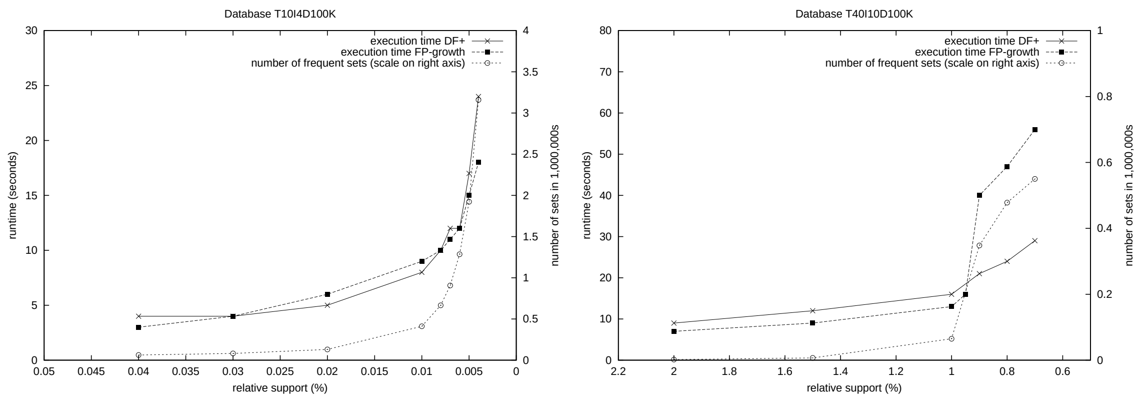

Figure 8: Experimental results for databases `T10I4D100K` and `T40I10D100K`.

`T10I4D100K` and 610–804 for `T40I10D100K`. Note the very high support thresholds for `chess` and `accidents`.

The experiments show that in the case of the very coherent databases, such as `chess` and, in case of a low threshold, `accidents`, $\mathcal{FP}$ performs better, but for `T40I10D100K` $\mathcal{DF}^+$ shows better results in case of low support thresholds; for `T10I4D100K` the behaviour of $\mathcal{DF}^+$ is slightly better. Another problem for $\mathcal{FP}$ is the required memory. If a large database has very long patterns, a machine with a large memory is needed, as we experienced while conducting our experiments.

5 Concluding remarks

We have addressed $\mathcal{DF}^+$, a depth first implementation of Apriori using the data structure of $\mathcal{FP}$. It turns out that $\mathcal{DF}^+$ competes with well-known algorithms, especially when applied to sparse databases. Real-world transaction databases of supermarkets mostly belong to this type of database. However, for dense databases, e.g., `chess`, $\mathcal{FP}$ is still preferable, provided that memory requirements are not a problem.

There is another major difference between $\mathcal{FP}$ and $\mathcal{DF}+$. The former generates a file of all frequent patterns, whereas the latter at first generates a trie, which is transformed into a file subsequently. If the collection of all frequent sets is processed further, a trie implementation is preferable to a file.

In the FIMI contest [3] faster results have been achieved. Most of the well-performing submissions were based upon $\mathcal{FP}$. As far as we understand it, this is however due to very sophisticated implementation details. In this paper we are rather comparing the underlying algorithms, using similar programming styles.

Complexity issues are left out of consideration in this paper. Some results on time complexity are mentioned in [6, 5]. The final aim would be to understand the behaviour expressed in terms of parameters of the databases, in view of recent results like [8]. In this paper an important complexity result was proved: counting the number of *maximal* frequent itemsets is #P-complete. A maximal frequent itemset has the property that every subset is frequent and no superset has this property. In a practical sense, pattern finding has been solved. In the theoretical field several research topics are left.

References

[1] Agrawal R., Mannila H., Srikant R., Toivonen H., Verkamo A.I. (1996) Fast Discovery of Association Rules. In: Fayyad U.M., Piatetsky-Shapiro G., Smyth P., Uthurusamy R. (Eds.) Advances in Knowledge Discovery and Data Mining, AAAI/MIT Press, pages 307–328

[2] Agrawal R., Srikant R. (1994) Fast Algorithms for Mining Association Rules. In: Bocca J.B., Jarke M., Zaniolo C. (Eds.) Proceedings of the 20th International Conference on Very Large Databases, pages 487–499

[3] Goethals B., Zaki M.J. (Eds.) (2003) Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations. CEUR Workshop Proceedings, `http://fimi.cs.helsinki.fi/fimi03/`.

[4] Han J., Pei J., Yin Y. (2000) Mining Frequent Patterns Without Candidate Generation. In: Chen W., Naughton J.F., Bernstein P.A. (Eds.) Proceedings 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00), ACM, pages 1–12

[5] Kosters W.A., Pijls W. (2003) Apriori: A Depth First Implementation. In: [3]

[6] Kosters W.A., Pijls W., Popova V. (2003) Complexity Analysis of Depth First and FP-Growth Implementations of Apriori. In: Perner P., Rosenfeld A. (Eds.) Machine Learning and Data Mining in Pattern Recognition, Proceedings MLDM 2003, Springer Lecture Notes in Artificial Intelligence 2734, Springer Verlag, pages 284–292

[7] Pijls W., Bioch J.C. (1999) Mining Frequent Itemsets in Memory-Resident Databases. In: Postma E., Gyssens M. (Eds.) Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC1999), pages 75–82

[8] Yang G. (2004) The Complexity of Mining Maximal Frequent Itemsets and Maximal Frequent Patterns. Accepted for the Tenth ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'04), August 2004, Seattle