Improved Methods for Extracting Frequent Itemsets from Interim-Support Trees

Frans Coenen
Department of Computer Science, University of Liverpool
Chadwick Building, Peach Street, Liverpool L69 7ZF, UK
frans@csc.liv.ac.uk

Paul Leng
Department of Computer Science, University of Liverpool
Chadwick Building, Peach Street, Liverpool L69 7ZF, UK
phl@csc.liv.ac.uk

Aris Pagourtzis*
Department of Computer Science, National Technical University of Athens
15780 Zografou, Athens, Greece
pagour@cs.ntua.gr

Wojciech Rytter†
Institute of Informatics, Warsaw University, Poland and
Department of Computer Science, New Jersey Institute of Technology, US
rytter@oak.njit.edu

Dora Souliou*
Department of Computer Science, National Technical University of Athens
15780 Zografou, Athens, Greece
dsouliou@cslab.ece.ntua.gr

Abstract

Mining association rules in relational databases is a significant computational task with lots of applications. A fundamental ingredient of this task is the discovery of sets of attributes (*itemsets*) whose frequency in the data exceeds some threshold value. In previous work [9] we have introduced an approach to this problem which begins by carrying out an efficient *partial* computation of the necessary totals, storing these interim results in a set-enumeration tree. This work demonstrated that making

*Aris Pagourtzis and Dora Souliou were partially supported for this research by "Pythagoras" grant of the Hellenic Ministry of Education, co-funded by the European Social Fund (75%) and National Resources (25%) under Operational Programme "Education and Initial Vocational Training" (EPEAEK II).

†Wojciech Rytter was supported for this research by grants 4T11C04425 and CCR-0313219.

use of this structure can significantly reduce the cost of determining the frequent sets.

In this paper we describe two algorithms for completing the calculation of frequent sets using an interim-support tree. These algorithms are improved versions of earlier algorithms described in the above mentioned work and in a consequent paper [7]. The first of our new algorithms (TTF) differs from its ancestor in that it uses a novel tree pruning technique, based on the notion of *(fixed-prefix) potential inclusion*, which is specially designed for trees that are implemented using only two pointers per node. This allows to implement the interim-support tree in a space efficient manner. The second algorithm (PTF) explores the idea of storing the frequent itemsets in a second tree structure, called the *total support tree* (T-tree); the improvement lies in the use of multiple pointers per node which provides rapid access to the nodes of the T-tree and makes it possible to design a new, usually faster, method for updating them.

Experimental comparison shows that these improvements result in considerable speedup for both algorithms. Further comparison between the two improved algorithms, shows that PTF is generally faster on instances with a large number of frequent itemsets, while TTF is more appropriate whenever this number is small; in addition, TTF behaves quite well on instances in which the densities of the items of the database have a high variance.

1 Introduction

An important data mining task initiated in [2] is the discovery of association rules over huge listings of sales data, also known as *basket data*. This task initially involves the extraction of *frequent* sets of items from a database of transactions, i.e. from a collection of sets of such items. The number of times that an itemset appears in transactions of the database is called its *support*. The minimum support an itemset must have in order to be considered as frequent is called the *support threshold*, a nonnegative integer denoted by t. The support of an association rule $A \implies B$, where A and B are sets of items, is the support of the set $A \cup B$. The *confidence* of rule $A \implies B$ is equal to $support(A \cup B)/support(A)$ and represents the fraction of transactions that contain B among transactions that contain A.

Association Rule Mining, in general, involves the extraction from a database of all rules that reach some required thresholds of support and confidence. The major part of this task is the discovery of the frequent itemsets; once the support of all these sets has been counted, determining the confidence of possible rules is trivial. Of course, there is no polynomial-time algorithm for this problem since the number of possible itemsets is exponential in the number of items. This problem has motivated a continuing search for effective heuristics for finding frequent itemsets and their support.

The best-known algorithm, from which most others are derived, is *Apriori* [4]. Apriori performs repeated passes of the database, successively counting the support for single items, pairs, triples, etc.. At the end of each pass, itemsets that fail to reach the support threshold are eliminated, and *candidate* itemsets

for the next pass are constructed as supersets of the remaining frequent sets. As no frequent set can have an infrequent subset, this heuristic ensures that all sets that may be frequent are considered. The algorithm terminates when no further candidates can be constructed.

Apriori remains potentially very costly because of its multiple database passes and, especially, the possible large number of candidates in some passes. Attempts to reduce the scale of the problem include methods that begin by partitioning [11] or sampling [12] the data, and those that attempt to identify *maximal* frequent sets [5] or *closed* frequent sets [13] from which all others can be derived. A number of researchers have made use of *set-enumeration tree* structures to organise candidates for more efficient counting. The *FP-growth* algorithm of Han et al. [10] counts frequent sets using a structure, the *FP-tree*, in which tree nodes represent individual items and branches represent itemsets. FP-growth reduces the cost of support-counting because branches of the tree that are subsets of more than one itemset need only be counted once. In contemporaneous work, we have also employed set-enumeration tree structures to exploit this property. Our approach begins by constructing a tree, the P-tree, [9, 8], which contains an incomplete summation of the support of sets found in the data. The P-tree, described in more detail below, shares the same performance advantage of the FP-tree but is a more compact structure. Results presented in [7] demonstrate that algorithms employing the P-tree can achieve comparable or superior speed to FP-growth, with lower memory requirements.

Unlike the FP-tree, which was developed specifically to facilitate the FP-growth algorithm, the P-tree is a generic structure which can be the basis of many possible algorithms for completing the summation of frequent sets. In this paper we describe and compare two algorithms for this purpose, namely:

1. The *T-Tree-First (TTF)* algorithm.
2. The *P-Tree-First (PTF)* algorithm.

Both algorithms make use of the incomplete summation contained in the P-tree to construct a second set-enumeration tree, the T-tree, which finally contains frequent itemsets together with their total support. The algorithms differ in the way they compute the total support: algorithm T-Tree-First iterates over the nodes of T-tree, and for each of them it traverses the P-tree; algorithm P-Tree-First starts by traversing the P-tree and for each node that it visits, it updates all relevant nodes at the current level of the T-tree.

TTF and PTF are improved versions of known algorithms, described in [9] and [7] respectively. Here we will refer to them as TTF-old and PTF-old respectively (note however that PTF-old was called Apriori-TFP in [7]).

The contribution of this work lies in the introduction of techniques that can considerably accelerate the process of computing frequent itemsets. In particular, the main improvement in the first of our algorithms (TTF) is a novel tree pruning technique, based on the notion of *fixed-prefix potential inclusion*, which is specially designed for trees that are implemented using only two pointers per node. This allows to implement the interim-support tree in a space efficient

manner. The second algorithm (PTF) differs from its predecessor in that it uses multiple pointers per node in the T-tree; this accelerates the access of the nodes of the T-tree and makes it possible to find and update appropriate T-tree nodes following a new, usually faster, strategy.

We perform experimental comparison of the two algorithms against their predecessors and show that in most cases the speedup is considerable. We also compare the two new algorithms to each other and discuss the merits of each. Our results show that PTF is faster than TTF if there are a lot of frequent itemsets in the database (small support threshold). On the other hand TTF gains ground as the support threshold increases and behaves even better for instances of variable item density which have been pre-sorted according to these densities.

2 Notation and Preliminaries

A database $\mathcal{D}$ is represented by an $m \times n$ binary matrix. The columns of D correspond to items (attributes), and the rows correspond to the transactions (records). The columns are indexed by consecutive letters a, b, ... of the alphabet. The set of columns (items) is denoted by $\mathcal{C}$. An *itemset* I is a set of items $I \subseteq \mathcal{C}$. For an itemset I we define:

- $E(I)$ (E-value of I) is the number of transactions that are exactly equal to I. This value is also called *exact support of* I.
- $P(I)$ (P-value of I) is the number of transactions that have I as a prefix. Also called *interim support of* I.
- $T(I)$ (T-value of I) is the number of transactions that contain I. Also called *total support* or, simply, *support of* I.

In this paper we consider the problem of finding all itemsets I with $T(I) \geq t$, for a given database $\mathcal{D}$ and threshold t.

For an item x we define the *density of* x *in* $\mathcal{D}$ to be the fraction of transactions of $\mathcal{D}$ that contain x, that is $T(\{x\})/m$. We also define the *density of a database* $\mathcal{D}$ to be the average density of the items of $\mathcal{D}$; note that the density of $\mathcal{D}$ is equal to the fraction of the total number of items appearing in the transactions of $\mathcal{D}$ over the size of $\mathcal{D}$ ($= nm$).

We will make use of the following order relations:

- *Inclusion order:* $I \subseteq J$, the usual set inclusion relation,
- *Lexicographic order:* $I \leq J$, I is lexicographically smaller or equal to J if seen as strings,
- *Prefix order:* $I \sqsubseteq J$, I is a prefix of J if seen as strings. Note that $I \sqsubseteq J \Leftrightarrow I \subseteq J \ \& \ I \leq J$.

We will also use the corresponding operators without equality: $I \subset J$, $I \sqsubset J$ and $I < J$.

Notice that for any itemset I:

$$T(I) = \sum_{I \subseteq J} E(J)$$

and therefore:

$$T(I) = \sum_{I \subseteq J \ \&\ I \leq J} E(J) + \sum_{I \subseteq J \ \&\ J < I} E(J) = P(I) + \sum_{I \subseteq J \ \&\ J < I} E(J) \qquad (1)$$

This property will play an important role in our algorithms.

3 The Interim-Support Tree

Both new algorithms TTF and PTF (as well as their predecessors TTF-old and PTF-old) have a common first part which is a pre-processing of the database that results in the storage of the whole information into a structure called the P-tree or *interim-support tree*. The P-tree is a set-enumeration tree the nodes of which are distinct itemsets of the database as well as some common prefixes of these itemsets. For each node, the interim support (P-value) of the corresponding itemset is also stored.

The notion of interim-support trees was introduced in [9], where details of the construction of the P-tree were given, and more fully in [8]. The tree is constructed in a single pass of $\mathcal{D}$. As each transaction is examined, the tree is traversed in a top-down (preorder) manner until either a node with identical itemset is found or a new node is created to represent the new itemset at an appropriate place in the tree. During this traversal, the support of all ancestors (preceding subsets) of the itemset is incremented. When a new itemset is inserted which shares a common prefix with an existing itemset, this prefix is created, if not already present, as a parent node, and inherits the support of its children.

The significance of the P-tree is that it performs a large part of the counting of support totals very efficiently in a single database pass. The size of the P-tree is linearly related to the original database, from which all relevant information is preserved, and will be smaller in cases where the data includes many duplicated itemsets. In results presented in [7], both the memory requirements and construction time for the P-tree were less than for a corresponding FP-tree [10].

For the sake of memory efficiency the P-tree is implemented using two pointers per node: *down* and *right*. For a node v, its down pointer links v to one of its children — the lexicographically smaller. This child's right pointer points to another child of v, and so on. For example, in the implementation of a P-tree containing itemsets 'a', 'ab', 'ac', and 'abc' node 'a' points down to 'ab' which in turn points down to 'abc' and right to 'ac'.

Note that, for simplicity, we often identify nodes of the P-tree with the itemsets they contain. This should cause no confusion since itemsets in the P-tree are unique.

4 The T-Tree-First (TTF) algorithm

TTF is an improved version of the algorithm in [9] (which we call TTF-old here). In this section we give a detailed description of the new algorithm.

The algorithm first scans the database and creates the P-tree, as explained in the previous section.

It then starts building the T-tree (recall that the T-tree will finally contain all frequent itemsets together with their total supports). Each level of the T-tree is implemented as a linear list, where itemsets appear in lexicographic order; nodes of such a list neither point to nor are pointed from nodes that are in the list of another level. In the beginning, the algorithm builds level 1 of the T-tree, which contains all frequent singletons; to this end it counts their support traversing the P-tree. It then builds the remaining T-tree level by level using procedure **Iteration(k)**.

The algorithm is presented below. A fundamental ingredient of TTF is function **CountSupport** which is described separately.

Algorithm T-Tree-First (TTF)
Input: Database $\mathcal{D}$, threshold t.
Output: The family $\mathcal{F}$ of frequent itemsets.

Build P-tree from database $\mathcal{D}$;
(Build the 1-st level of T-tree *)*
for $i = 1$ **to** n **do**
    if CountSupport(P-tree, $\{i\}$) $\geq t$ **then** add $\{i\}$ to $\mathcal{F}_1$;
(Build the remaining levels of T-tree *)*
for $k = 2$ **to** n **do**
    Iteration(k);
    if $\mathcal{F}_k = \emptyset$ **then exit**
    else $\mathcal{F} = \mathcal{F} \cup \mathcal{F}_k$;
return $\mathcal{F}$;

Some details of procedure **Iteration(k)** need to be clarified. Its goal is to build $\mathcal{F}_k$, that is, the k-th level of the T-tree. Itemsets in $\mathcal{F}_k$ must have all their $(k-1)$-size subsets in $\mathcal{F}_{k-1}$. Therefore, one can start from existing itemsets in $\mathcal{F}_{k-1}$ and try to augment them with one more item in order to create all potentially frequent itemsets. To avoid duplications the algorithm may proceed by considering for each frequent itemset X_{k-1} in $\mathcal{F}_{k-1}$ all X_{k-1}'s supersets $X_k = \{x\} \cup X_{k-1}$ for items x that are greater than any item of X_{k-1}.

Careful observation reveals that only if X_{k-1} and the node following it, denoted X'_{k-1}, differ at the last item it makes sense to consider such supersets.

The candidate superset X_k is then the union of X_{k-1} and X'_{k-1}. Then it is checked whether all the ($k-2$ many) remaining ($k-1$)-subsets of X_k are frequent; this is carried out by a special function called **ExistSubsets**, which we will not describe in detail here. If some of the examined subsets of X_k is not present in $\mathcal{F}_{k-1}$, X_k is not added to $\mathcal{F}_k$.

```
Procedure Iteration(k) (* Building the k-th level of T-tree *)
for each itemset X_{k-1} ∈ F_{k-1} do
    X'_{k-1} := next(X_{k-1});
    while X'_{k-1} ≠ NULL do
        if X_{k-1} and X'_{k-1} differ only at the last item then
            X_k := X_{k-1} ∪ X'_{k-1};
            if ExistSubsets(X_k, F_{k-1}) then
                T(X_k) := CountSupport(P-tree, X_k);
                if T(X_k) ≥ t then add X_k to F_k;
            X'_{k-1} := next(X'_{k-1});
        else exit while;
```

In order to complete the description of TTF it remains to describe its most critical part, that is, function **CountSupport**, which counts the total support of an itemset X in the P-tree in a recursive manner. An essential ingredient of **CountSupport** is the notion of *fixed-prefix potential inclusion*:

Fixed-Prefix Potential Inclusion. $I \overset{\text{pot}}{\subseteq}_K J$: $\exists J',\ commonprefix(J, J') = K \ \&\ I \subseteq J'$.

Examples: 'bdf' $\overset{\text{pot}}{\subseteq}_{\text{'ab'}}$ 'abc', 'bdf' $\overset{\text{pot}}{\not\subseteq}_{\text{'ab'}}$ 'abd'.

In words, $I \overset{\text{pot}}{\subseteq}_K J$ means that there is an itemset greater than J, sharing with J a common prefix K, that contains I.

A second interesting inclusion relation can be defined in terms of $\overset{\text{pot}}{\subseteq}_K$:

Potential Inclusion. $I \overset{\text{pot}}{\subseteq} J \overset{\text{def}}{=} I \overset{\text{pot}}{\subseteq}_J J$, i.e. $\exists J',\ J \sqsubseteq J' \ \&\ I \subseteq J'$.

Examples: 'bdf' $\overset{\text{pot}}{\subseteq}$ 'abde', 'bdf' $\overset{\text{pot}}{\not\subseteq}$ 'abdg'.

In words, $I \overset{\text{pot}}{\subseteq} J$ means that there is an extension of J that contains I.

The use of the above inclusion relations can significantly reduce the number of moves needed to count the support of an itemset in trees with two pointers per node. Suppose that we are looking for appearances (i.e. supersets) of an itemset I in the P-tree and we are currently visiting a node that contains itemset J:

- Nodes that are below the current node contain itemsets J' which have J as prefix. Therefore, if $I \overset{\text{pot}}{\not\subseteq} J$ there is no point visiting the subtree rooted at the current node.
- Nodes that are to the right of the current node (siblings) contain itemsets that have $par(J)$ (parent of J) as prefix — and so does J — and are

greater than J. If $I \overset{\text{pot}}{\not\subseteq}_{par(J)} J$ there is no point visiting the subtrees rooted at these nodes.

These two tests result in much better tree pruning comparing to the one applied by the TTF-old algorithm. As an example, suppose that we are trying to find the support of itemset X='bd' in a P-tree in which there is a node 'ab' with children 'abde' and 'abefg'. Then, once the tree traversal reaches node 'abde' it adds its support to $T(X)$ and does not move to the right, that is, it avoids visiting 'abefg'. On the other hand, TTF-old [9] would also examine 'abefg' (and other siblings if such existed) because it only terminates its search whenever it finds itemsets lexicographically equal or greater than X.

```
Function CountSupport(pnode, X): integer
(* Counts the total support of itemset X
in the subtree of P-tree rooted at pnode*)
T := 0;
if pnode ≠ NULL then
    J := pnode → itemset;
    if X ⊆pot J then (* makes sense to search children *)
        if X ⊆ J then T := T + P(J)
            (* inclusion is a special case of potential inclusion *)
        else T := T+ CountSupport(pnode → down, X);
    if X ⊆pot_par(J) J then (* makes sense to search right siblings *)
        T := T+ CountSupport(pnode → right, X);
return T;
```

Finally, let us explain how to check potential inclusion and fixed prefix potential inclusion. It can be shown that the following tests suffice. The proof is omitted.

- $X \overset{\text{pot}}{\subseteq} J$: if $X \subseteq J$ then $X \overset{\text{pot}}{\subseteq} J$ is true. Otherwise let x be the lexicographically smaller item of X that is not item of J (such x exists). If for all items j of J are lexicographically smaller than x then $X \overset{\text{pot}}{\subseteq} J$ is true otherwise it is false.
- $X \overset{\text{pot}}{\subseteq}_K J$: assume $K \sqsubseteq J$ (otherwise the inclusion $X \overset{\text{pot}}{\subseteq}_K J$ is obviously false). Let x be the first item of $X \setminus K$ and J be the first item of $J \setminus K$. If $x > j$ the inclusion $X \overset{\text{pot}}{\subseteq}_K J$ holds otherwise it is false.

5 The P-Tree-First (PTF) Algorithm

PTF is an improved version of the algorithm Apriori-TFP [7] (which we call PTF-old here). PTF also begins by constructing the P-tree exactly as TTF, but

then it follows an inverse approach in order to update the T-tree. In particular, during the processing of level-k of the T-tree, each node of the P-tree is visited once. Let I be the itemset of a visited node; the algorithm updates all nodes of level-k that are subsets of I, except for those that are also subsets of $par(I)$ (parent of I) — the latter have already been updated while visiting $par(I)$.

Following again the a-priori strategy [4], level-k itemsets of the T-tree are constructed from the itemsets of level-$(k-1)$, by adding single items to each of them. Then, the P-tree is traversed as described above in order to compute support for all nodes of level-k. Nodes with support smaller than the threshold are removed before the generation of level-$(k+1)$.

Algorithm P-Tree-First (PTF)
Input: Database $\mathcal{D}$, threshold t.
Output: The family $\mathcal{F}$ of frequent itemsets.

Build P-tree from database $\mathcal{D}$;
add $\emptyset$ to $\mathcal{F}_0$; *(* create a dummy level with one empty itemset *)*
(Build level-k of the T-tree *)*
for $k = 1$ **to** n **do**
    Iteration(k**)**;
    if $\mathcal{F}_k = \emptyset$ **then exit for**
    else $\mathcal{F} = \mathcal{F} \cup \mathcal{F}_k$;
return $\mathcal{F}$;

Our innovation here is the use of multiple pointers at each node of the T-tree in contrast to PTF-old where two pointers per node are used. This provides rapid access to the nodes of the T-tree which allows for a new strategy for T-tree update. In particular, while building level k, once a node I of the P-tree is visited, all its k-subsets (subsets of size k) are generated; once such a k-subset is generated, it is sought in the T-tree and, if present, its support is updated accordingly. Whenever such an itemset J has a subset J' which is not frequent (hence neither J can be frequent) the algorithm discovers this quite early and the update process terminates. For example, if the algorithm visits a node of the P-tree with itemset 'acdfghk' and the current level of the T-tree is level-6 the algorithm should update all size-6 subsets of 'acdfghk'. Consider 'acdfgh'; the algorithm will try to find this node starting from 'a' in level-1, continuing to 'ac' in level-2, and then to 'acd', 'acdf' and 'acdfg'. If 'acd' is non-frequent, i.e. does not exist in level-3, the algorithm stops and considers the lexicographically next size-6 subset of 'acdfghk'. In fact, it saves even more comparisons by considering 'acfghk' as next subset because there is no need to check any subset that contains 'acd'. On the other hand, PTF-old traverses a potentially large list of candidate itemsets in order to check whether any of them is a k-subset of I. This could be much slower than the above described procedure, especially if I has few k-subsets in that list. A detailed description of the update of level-k of the T-tree is given below.

```
Procedure Iteration(k) (* Building k-th level of T-tree *)
for each itemset X_{k-1} ∈ F_{k-1} do
    for each item x greater than all items of X_{k-1} do
        add X_k := X_{k-1} ∪ {x} to F_k;
        let the x-th down pointer of X_{k-1} point to X_k;
(* Update total supports of nodes in F_k *)
for each node I of the P-tree do
    non-frequent := {dummy};
    for each itemset J ⊆ I with |J| = k do
        if J ⊆ par(I) or non-frequent ⊆ J then
            proceed to the lex. next J ⊆ I such that
                J is not subset of par(I) and does not contain non-
frequent
        else
            descend the T-tree following prefixes of J
                until J is found or some J'⊑J is missing;
            if J is found then T(J) := T(J) + P(I)
            else non-frequent := J'; (* J' is not in the T-tree *)
remove from F_k all nodes with support < t (threshold);
```

6 Experimental Comparison

We implemented four algorithms in ANSI-C: TTF, TTF-old, PTF and PTF-old. We run several experiments using a Pentium 1.6 GHz PC. We first experimented with datasets created by using the IBM Quest Market-Basket Synthetic Data Generator (described in [4]). We follow a standard notation according to which a dataset is described by four parameters: T represents the average transaction length (roughly equal to the database density times the number of items), I represents the average length of maximal frequent itemsets, N represents the number of items, and D represents the number of transactions in the database. We generated datasets T10.I4.N50.D10K and T10.I4.N20.D100K and run experiments with all four algorithms. The execution time of each algorithm for these two datasets and threshold varying from 5% to 1% is shown in Figure 1.

These results show that both algorithms TTF and PTF are faster than their predecessors, except for rather large thresholds. As regards TTF and TTF-old, the reason for this behaviour is that TTF-old performs fewer tests at each P-tree node that it visits; thus, whenever a contiguous part of the tree is traversed by both TTF and TTF-old, it is TTF-old the one which does it faster. Now, whenever the frequent itemsets are few, they are also (most probably) of small size; a small itemset has higher chances to appear in a contiguous part of the P-tree which therefore cannot be pruned by TTF. PTF-old can also be faster than PTF if there are only few frequent itemsets because in such a case it can be faster to traverse the list of candidate itemsets than generating all subsets of a node.

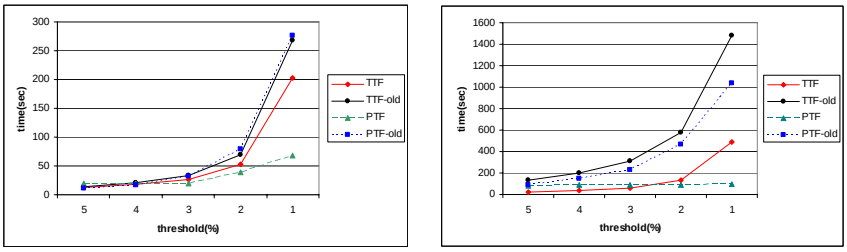

Figure 1: Results for datasets T10.I4.N50.D10K (left) and T10.I4.N20.D100K (right).

Comparing now the two new algorithms, we observe that PTF is faster than TTF for small thresholds ($\leq 2\%$). This is due to the fact that whenever the number of frequent itemsets is large, TTF performs a lot of P-tree traversals, while PTF performs only one full P-tree traversal per T-tree level. Since the size of the P-tree can be rather large (even comparable to the size of the database) its traversal is quite slow; hence, whenever TTF performs many traversals, even partial, the overall slowdown is considerable. On the other hand, PTF performs several T-tree traversals at each level but these are fast thanks to the use of multiple pointers. The two algorithms have comparable running time for thresholds above 2%. This is because for relatively sparse T-tree the P-tree traversals performed by TTF are few; in this case the economizing techniques of TTF balance, or even beat the advantages of PTF.

To further compare TTF and PTF we implemented a probabilistic generator in order to create datasets of *variable item density* (each item has a different expected density). This generator fills the i-th item of a row with probability $p_f - (i-1)p_s$, i.e., the probability decreases linearly as we move from the first to the last item of a row; p_f represents the probability of appearance of the first item and p_s is the decrement step. The expected density of the database is equal to $p_f - \frac{(n-1)}{2}p_s = \frac{p_f - p_l}{2}$, where p_l is the probability of appearance of the last item and n is the number of items in each row.

We have generated four variable-density datasets, one for each of the following four types (where letter 'V' stand for 'variable-density'): V.T4.N20.D10K, V.T6.N20.D10K, V.T4.N20.D100K, and V.T6.N20.D100K; the corresponding first item selection probabilities and decrement steps (in parentheses) are 0.4 (0.02), 0.6 (0.03), 0.4 (0.02), and 0.6 (0.03) respectively.

We run experiments with support thresholds ranging from 5% to 0.5%. For each dataset type / threshold combination we have measured the execution time of PTF and TTF, averaging over ten experiments, one for each dataset of the type.

Results for the datasets with 10K transactions appear in Figure 2. Figure 3 shows results for the datasets with 100K transactions.

Comparison of the algorithms for Variable-Density Datasets. The compar-

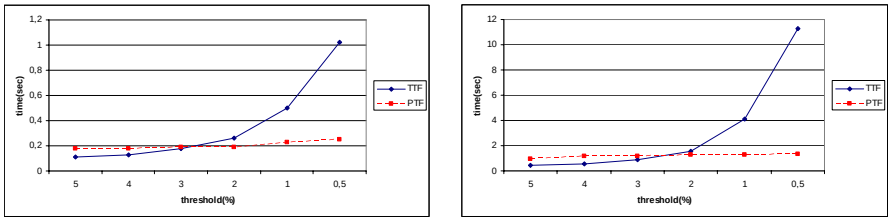

Figure 2: Results for datasets V.T4.N20.D10K (left) and V.T6.N20.D10K (right).

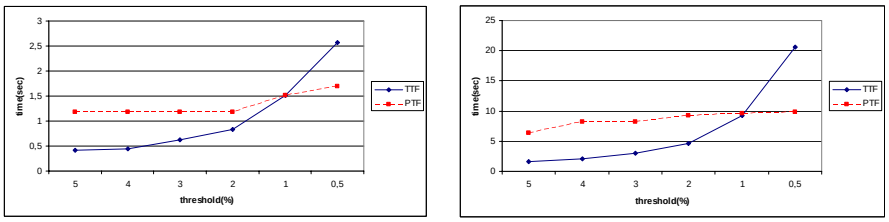

Figure 3: Results for datasets V.T4.N20.D100K (left) and V.T6.N20.D100K (right).

ison of the two algorithms is much more interesting when it comes to variable-density data sets. As before, PTF behaves better for small thresholds (roughly smaller than 2%) but TTF is faster for larger thresholds. Besides, PTF exhibits almost constant running time in most experiments. Now, whenever the T-tree is small and sparse, it happens that the few full P-tree traversals performed by PTF can take longer than the (more but not too many) partial P-tree traversals of TTF. The main reason is that potentially frequent itemsets consist mainly of lexicographically smaller items, hence the partial P-tree traversals of TTF are limited to a small part of the P-tree and are therefore much faster. On the other hand, TTF performs a full P-tree traversal at each level of the T-tree that contains potentially frequent itemsets, regardless of the number of these itemsets, hence it needs almost the same time as before, since it considers a similar number of levels.

Comparing the performance of the two algorithms with respect to uniformity of item densities one observes that while PTF exhibits roughly the same performance for both uniform and variable item densities, TTF is considerably faster on instances of variable item density; indeed, our results show that for variable-density datasets, TTF outperforms PTF for support thresholds above 3%, even above 2% or 1% in some cases. This is due to the fact that the performance of PTF is mainly determined by the rank of the higher level of frequent

itemsets, while the performance of TTF depends heavily on the part of the P-tree that must be visited each time — which is much smaller for variable density instances, because frequent itemsets consist mainly of lexicographically smaller items.

Let us note here that for our experiments we built the variable-density datasets in such a way that the lexicographically greater items are of smaller density. This property is essential for the performance of TTF, since it guarantees that most frequent itemsets consist mainly of lexicographically small items which appear in a small part of the P-tree. Therefore, to make TTF work well for real datasets, a sorting of the items in order of decreasing density should be performed in a preprocessing step.

7 Conclusions

In this work we have developed and implemented two *Apriori*-style algorithms for the problem of frequent itemsets generation, called T-Tree-First (TTF) and P-Tree-First (PTF), that are based on the interim-support tree approach [9]. The two algorithms follow inverse approaches: TTF iterates over the itemsets of T-tree, and for each of them traverses the relevant part of the P-tree in order to count its total support; PTF starts by traversing the P-tree and for each visited node it updates all relevant nodes at the current level of the T-tree.

Our algorithms are improved versions of known algorithms, described in [9] and [7]. We have introduced several new techniques that result in faster algorithms comparing to these earlier attempts. The most important of them are the *fixed-prefix potential inclusion* technique, which is used in algorithm TTF, and the use of *multiple pointers* in the T-tree, employed by PTF. The former allows faster support counting for P-trees that are built using only two pointers per node, thus being particularly memory-efficient. The latter provides fast access to the T-tree and makes PTF a generally efficient algorithm. We show experimentally that our new algorithms achieve considerable speedup comparing to their predecessors.

The main difference between the two algorithms is that TTF performs a partial P-tree traversal for each potentially frequent itemset, while PTF performs only one, but full, P-tree traversal for each level of potentially frequent itemsets. As a result, PTF is considerably faster than TTF in instances where there are a lot of frequent itemsets, while TTF gains ground in instances where there are fewer potentially frequent itemsets, especially if for each of them it suffices to check only a small part of the P-tree. For example, the latter case may occur whenever item densities have a high variance.

In conclusion, each of the two heuristics has its own merits and deserves further exploration. As a suggestion for further research, it would be interesting to investigate possible combinations of the two inverse approaches of TTF and PTF. For example, it seems reasonable to use PTF as long as the current level of the T-tree contains a lot of frequent itemsets, while it may be wise to turn to TTF once the current level becomes sparse, especially if the majority of the

potentially frequent itemsets can be only found in a small part of the P-tree.

References

[1] F. Angiulli, G. Ianni, L. Palopoli. On the complexity of inducing categorical and quantitative association rules, arXiv:cs.CC/0111009 vol. 1, Nov. 2001

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of ACM SIGMOD Conference on Management of Data*, Washington DC, May 1993.

[3] R. Agrawal, T. Imielinski, and A. Swami. Database mining: a performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, Dec 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.

[4] R. Agrawal and R. Srikant. Fast Algorithms for mining association rules. In *VLDB'94*, pp. 487–499.

[5] R. Agrawal, C. Aggarwal and V. Prasad. Depth First Generation of Long Patterns. In *KDD 2000*, ACM, pp. 108-118

[6] E. Boros, V. Gurvich, L. Khachiyan, K. Makino. On the complexity of generating maximal frequent and minimal infrequent sets, in *STACS* 2002.

[7] F. Coenen, G. Goulbourne, and P. Leng. Computing Association Rules using Partial Totals. In L. De Raedt and A. Siebes eds, *Principles of Data Mining and Knowledge Discovery* (Proc 5th European Conference, PKDD 2001, Freiburg, Sept 2001), Lecture Notes in AI 2168, Springer-Verlag, Berlin, Heidelberg: pp. 54–66.

[8] F. Coenen, G. Goulbourne and P. Leng. Tree Structures for Mining Association Rules. *Data Mining and Knowledge Discovery*, 8 (2004), pp. 25-51

[9] G. Goulbourne, F. Coenen and P. Leng. Algorithms for Computing Association Rules using a Partial-Support Tree. *Journal of Knowledge-Based Systems* 13 (2000), pp. 141–149.

[10] J. Han, J. Pei, Y.Yin and R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Mining and Knowledge Discovery*, 8 (2004), pp. 53-87

[11] A. Savasere, E. Omiecinski and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *VLDB 1995*, pp. 432-444

[12] H. Toivonen. Sampling Large Databases for Association Rules. In *VLDB 1996*, pp. 1-12.

[13] M. J. Zaki. Generating Non-Redundant Association Rules. In *Proc. SIGKDD-2000*, pp. 34-43, 2000.