

Shakil Ahmed · Frans Coenen · Paul Leng

# Tree-based partitioning of data for association rule mining

Received: 28 January 2004 / Revised: 12 May 2005 / Accepted: 12 June 2005 /  
Published online: 24 March 2006  
© Springer-Verlag London Limited 2006

**Abstract** The most computationally demanding aspect of Association Rule Mining is the identification and counting of support of the frequent sets of items that occur together sufficiently often to be the basis of potentially interesting rules. The task increases in difficulty with the scale of the data and also with its density. The greatest challenge is posed by data that is too large to be contained in primary memory, especially when high data density and/or low support thresholds give rise to very large numbers of candidates that must be counted. In this paper, we consider strategies for partitioning the data to deal effectively with such cases. We describe a partitioning approach which organises the data into tree structures that can be processed independently. We present experimental results that show the method scales well for increasing dimensions of data and performs significantly better than alternatives, especially when dealing with dense data and low support thresholds.

**Keywords** Association rules · Partial support · Data structures · Set-enumeration tree

## 1 Introduction

An *association rule* [2] is an implication of the form  $A \rightarrow B$ , relating disjoint sets of database attributes, which is interpreted to mean “if the set of attribute-values  $A$  is found together in a database record, then it is likely that the set  $B$  will be present also”. Association Rule Mining involves the discovery, in a tabular database, of all such rules that satisfy defined threshold requirements. Of these requirements, the most fundamental concerns *frequency*: a rule is likely to be applicable only if the relationship it describes occurs sufficiently often in the data.

The *support* for the rule  $A \rightarrow B$  is the number (or proportion) of database records within which the set of attribute-values  $A \cup B$  is found. The *frequent sets* are those sets for which the support exceeds some threshold value. Association Rule Mining requires that all frequent sets are identified and their support determined, so that other properties of rules such as *confidence* and *lift* [6] can be calculated.

It is recognised that identifying the frequent sets is the most computationally demanding aspect of Association Rule Mining. The problem arises because the number of possible sets is exponential in the number of possible attribute-values. Continuously valued attributes can be dealt with by discretization, and, for convenience of processing, most methods convert multiple-valued attributes into a number of binary attributes, or *items*, each of which can be said to be present or absent in each record. For most real data, the number  $n$  of such items is likely to be such that counting the support of all  $2^n$  sets of items (*item sets*) is infeasible. For this reason, almost all methods attempt to count the support only of candidate item sets that are identified as possible frequent sets. It is, of course, not possible to completely determine the candidate item sets in advance, and it will therefore be necessary to consider many item sets that are not in fact frequent.

In general, algorithms for finding frequent sets involve one or (usually) several passes of the source data, in each of which the support for some set of candidate item sets is counted. The performance of these methods, clearly, depends both on the size of the original database, typically millions or billions of records, and on the number of candidate item sets being considered. The number of possible candidates increases with increasing density of data (greater number of items present in a record) and with decreasing support thresholds. In applications such as medical epidemiology, we may be searching for rules that associate rather rare items within quite densely-populated data, and in these cases the low support thresholds required may lead to very large candidate sets. These factors motivate a continuing search for efficient algorithms.

Performance will be affected, especially, if the magnitudes involved make it impossible for the algorithm to proceed entirely within primary memory. In these cases, some strategy for *partitioning* the data may be required to enable algorithmic stages to be carried out on primary-memory-resident data. Effective partitioning will reduce the number of accesses to secondary memory. In this paper, we examine methods of partitioning to limit the total primary memory requirement, including those required both for the source data and for the candidate sets. We consider both ‘horizontal’ partitioning, which divides the source data into sets of records and ‘vertical’ partitioning, which partitions records into sets of items. We describe a new method of vertical partitioning that exploits tree structures we have previously developed for Association Rule Mining. Experimental results are presented that show this method offers significantly better performance than horizontal partitioning.

## 2 Background

Most methods for finding frequent sets are to a greater or lesser extent based on the “Apriori” algorithm [3]. Apriori performs repeated passes of the database, successively computing support counts for sets of single items, pairs, triplets and so on. At the end of each pass, sets that fail to reach the required support threshold

are eliminated, and candidates for the next pass are constructed as supersets of the remaining (frequent) sets. Since no set can be frequent which has an infrequent subset, this procedure guarantees that all frequent sets will be found.

One of the inherent performance weaknesses of Apriori is that it requires the source data to be scanned repeatedly; in principle, the number of passes required is one greater than the size of the largest frequent set. This is especially a problem if, as is likely to be the case in many applications, the source data cannot be contained in primary memory. An early refinement of the method attempted to reduce this cost by partitioning the data into a number of equal-sized segments that can be so contained. The “Partition” algorithm [18] applies the Apriori procedure to each data segment in turn, retaining the segment in primary storage throughout its repeated passes. For each segment, thus, a set of *locally* frequent item sets is determined, each of which reaches the proportionate threshold of support in that segment. A second pass of the complete database is required to establish which of the locally frequent sets are (globally) frequent. Similar thinking motivated the strategy introduced by Toivonen [19]. Here, a random sample of the source data, small enough to contain in primary memory, is first processed using the Apriori procedure, with a modified support threshold and other modifications designed to make it likely that all the globally frequent sets will be identified in the sample. The sets thus found become candidates for a single full pass of the source data to verify this.

The drawback of both these approaches highlights the second weakness of Apriori that the number of candidates whose support is to be counted may become very large, especially when the data is such that the frequent sets may contain many items (the “long pattern” problem [1]). If, for example, there is just one set of 20 items that reaches the threshold of support, then the method inescapably requires the support for all the  $2^{20}$  subsets of this set to be counted. In the Partition algorithm, this is exacerbated because there may be many more sets that are locally frequent in some partition, even though they are not globally frequent. In Toivonen’s method also, it is necessary for the initial processing of the sample to identify an enlarged candidate set, to give a reasonable probability that all the actual frequent sets will be included. Both these methods also require all candidates to be retained in primary memory (for efficient processing) during the final database pass.

Other methods [1, 4, 5, 20] aim to identify *maximal* frequent sets without first examining all their subsets. These algorithms may cope better with densely-populated databases and long patterns than the others described, but again usually involve multiple database passes. The DepthProject [1] algorithm bypasses the problem by explicitly targeting memory-resident data. The method of Zaki et al. [20] is of interest, as it introduces a different kind of partitioning. In this, candidate sets are partitioned into clusters which can be processed independently. The problem with the method is that, especially when dealing with dense data and low support thresholds, expensive preprocessing is required before effective clustering can be identified. The partitioning by *equivalence class*, however, is relevant to the methods we will describe.

Our methods begin by performing a single pass of the database to perform a partial summation of the support totals. These partial counts are stored in a tree structure that we call the *P*-tree, which enumerates item sets counted in

lexicographic order. The term *P*-tree has been used elsewhere (e.g. [15]) to describe other structures: here we use it to denote the structure we first introduced in [12], representing a tree of *Partial* support counts. The *P*-tree contains all the sets of items present as distinct records in the database, plus some additional sets that are leading subsets of these.

To illustrate this, consider a database with items  $\{a, b, c, d, e\}$  and 20 records:

$$\{abcde, abce, abd, abde, abe, acde, ace, ade, b, bcde, bce, bd, bde, be, cd, cde, ce, d, de, e\}$$

(Not necessarily in this order.) For convenience, we will use the notation *abd*, for example, to denote the set of items  $\{a, b, d\}$ . Figure 1 shows the *P*-tree that would be constructed. The counts stored at each node are *incomplete* support totals, representing support derived from the set and its succeeding supersets in the tree.

We then apply to this structure an algorithm, Apriori-TFP, which completes the summation of the final support counts, storing the results in a second set-enumeration tree (the *T*-tree, of *Total* support counts), ordered in the opposite way to the *P*-tree. The *T*-tree finally contains all frequent sets with their complete support counts. The algorithm used, essentially a form of Apriori that makes use of the partial counting that has already been done, is described in [8], where we also explain the rationale of the approach and its advantages. Experimental results reported in [8] demonstrate significant performance gains in comparison with Apriori and also some improvements over the *FP-growth* [13] algorithm, which uses somewhat similar structures and has some similar properties. The *FP*-tree used in [13] is a more pointer-rich structure than the *P*-tree, leading to greater difficulties in dealing with non-memory-resident data, although strategies for this have been

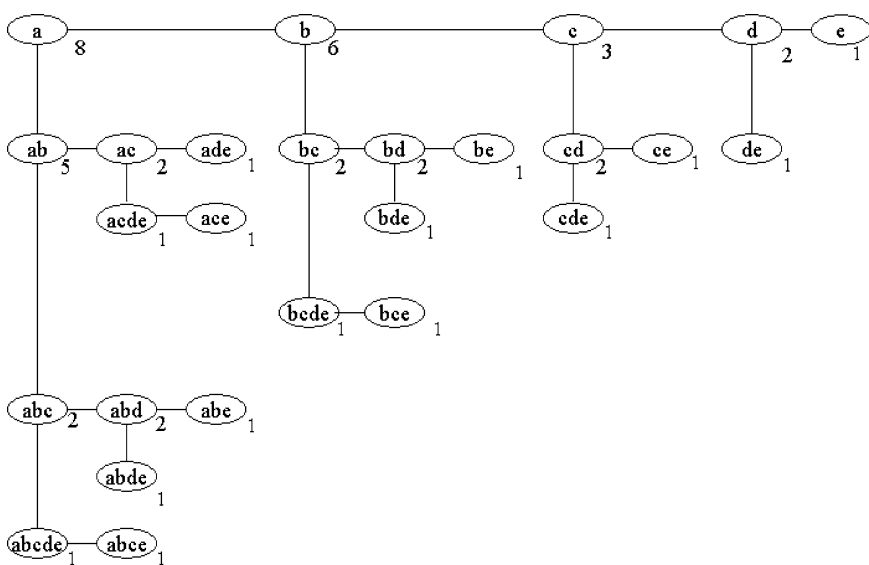


Fig. 1 Example of a *P*-tree

proposed, which will be discussed further later. The CATS tree, an extension of the *FP*-tree proposed in [7], also assumes no limitation on main memory capacity. In this paper, we consider implementations of Apriori-TFP in cases when it will be impossible to contain all the data required in main memory, requiring some strategy for partitioning this.

### 3 Strategies for partitioning

#### 3.1 Horizontal partitioning

The natural implementation of Apriori, when source data cannot be contained in primary memory, requires all the data to be read from secondary memory in each pass. The equivalent for Apriori-TFP, because the first stage of the method involves the construction of a *P*-tree, requires a partitioning of the data into segments of manageable size. We will refer to this form of partitioning, in which each segment contains a number of complete database records, as ‘horizontal’ partitioning (HP), or *segmentation*. We first take each segment of data separately and create for it a *P*-tree that is then stored in secondary memory. The stored *P*-trees are then treated as a composite structure from which we compute the final support totals for all the frequent sets, storing these in a single *T*-tree. Each pass of Apriori-TFP requires each of the *P*-trees to be read in turn from secondary memory. The method creates a final *T*-tree in primary memory, which contains all the frequent sets and their support counts.

#### 3.2 Vertical partitioning

The drawback of the simple approach outlined earlier in the paper is that it replicates the two weaknesses of the Apriori methodology. As with Apriori, all the source data (now in the form of *P*-trees) must be re-read from secondary memory in each pass. The second problem is that the entire *T*-tree, which finally contains all the frequent sets, must be contained in primary memory while counting proceeds. As we have noted, this tree may itself become very large, especially when long frequent patterns are encountered. Even if the tree is not too large to be contained in primary memory, a large set of candidates leads to slower counting, in Apriori-TFP just as for Apriori.

A possible alternative way of partitioning the data is to divide the set of items under consideration into subsets, each of which defines a *vertical partition* of the data set. The problem with this is that, in general, the sets for which support is to be counted contain items from several partitions. The *P*-tree structure offers another form of vertical partitioning into subtrees that represent equivalence classes of the items represented. In this case, again, it is still not possible to compute the support for a set by considering only the subtree in which it is located. Although succeeding supersets of a set *S* in the *P*-tree are located in the subtree rooted at *S*, predecessor supersets are scattered throughout the preceding part of the *P*-tree. For example, consider the support for the set *bd* in the data used for Fig. 1. In the subtree rooted at *b*, we find a partial support total for *bd*, which includes the total for its superset *bde*. To complete the support count for *bd*, however, we must add

in the counts recorded for its preceding supersets  $bcde$ ,  $abd$  (incorporating  $abde$ ) and  $abcde$ , the latter two of which are in the subtree headed by  $a$ .

The problem can be overcome by a different partitioning of the  $P$ -tree structure. Our Tree Partitioning (TP) method begins by dividing the ordered set of items into subsequences. For example, for the data used in Fig. 1, we might define three sequences of items  $\{a, b\}$ ,  $\{c, d\}$  and  $\{e\}$  labelled as 1, 2 and 3, respectively. For each sequence we define a *Partition-P-tree* ( $PP$ -tree), labelled PP1, PP2 and PP3. The construction of these is a slight modification of the original method. The first partition tree, PP1, is a proper  $P$ -tree that counts the partial support for the power set of  $\{a, b\}$ . PP2, however, counts all those sets that include a member of  $\{c, d\}$  in a tree that includes just these items and their predecessors. The third tree, PP3, will count all sets that include any member of  $\{e\}$ . The three trees obtained, from our example, are illustrated in Fig. 2. The  $PP$ -trees are, in effect, overlapping partitions of the  $P$ -tree of Fig. 1, with some restructuring resulting from the omission of nodes when they are not needed.

The effect of this is that the total support for any set  $S$  can now be obtained from the  $PP$ -tree corresponding to the last item within  $S$ ; for example, we now find all the counts contributing to the support of  $bd$  are included in PP2. The drawback is that the later trees in the sequence are of increasing size; in particular, PP3 in our example is almost as large as the original  $P$ -tree. We can overcome this, however, by a suitable reordering of the items. In descending order of their frequency in the data, the items of our example are  $e, d, b, c$  and  $a$ . Using the same data as for

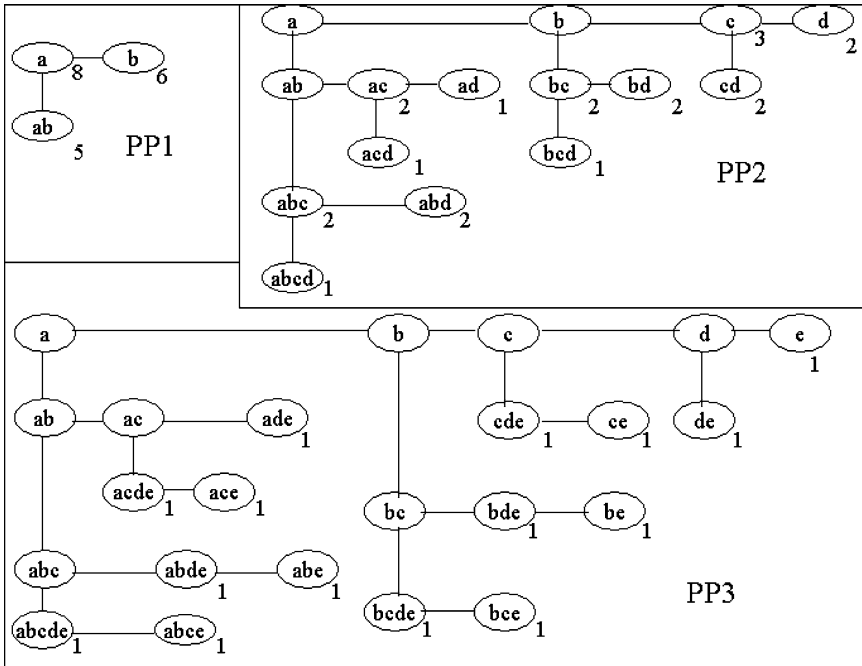


Fig. 2 Partition- $P$ -trees from Fig. 1

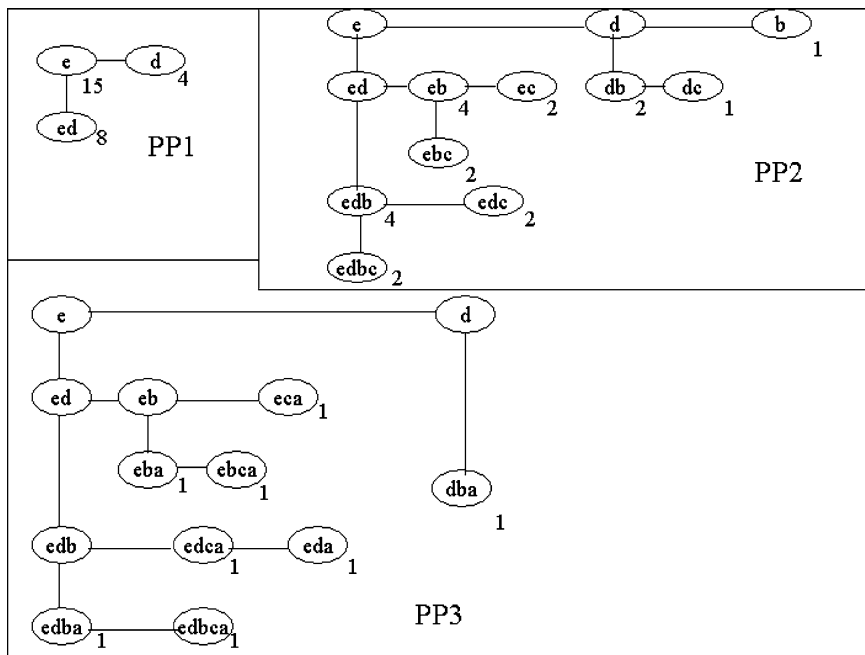


Fig. 3 PP-trees after reordering of items

Figs. 1 and 2, we will construct PP-trees using this ordering, for the sets of items  $\{e, d\}$ ,  $\{b, c\}$  and  $\{a\}$  respectively.

The results are shown in Fig. 3. Now, because the less frequent items appear later in the sequence, the trees become successively more sparse, so that PP3 now has only 13 nodes, compared with 23 of PP3 in Fig. 2. In fact, our previous work has shown [9] that ordering items in this way leads to a smaller P-tree and faster operation of Apriori-TFP. The additional advantage for partitioning is that the PP-trees become more compact and more equal in size. The total support count for  $bd$  (now ordered as  $db$ ) is again to be found within PP2, but now requires the addition of only two counts ( $db + edb$ ).

### 3.3 Counting total support using PP-trees

The form of partitioning we have described offers us a way of dividing the source data into a number of PP-trees each of which may then be processed independently. With a sufficiently large data set, it will of course still not be possible to construct the PP-trees within primary memory. We can, however, combine this approach with a (horizontal) segmentation of the original data into segments small enough to allow the corresponding PP-trees to be contained in primary store.

The overall method is as follows. For clarity, we will use the term *segment* when we refer to the horizontal division of the data into sets of records and *partition* when we refer to the vertical division into sets of items and the corresponding tree structures:

1. Obtain an (at least approximate) ordering of the frequency of items.
2. Using this ordering, choose an appropriate partitioning of the items into  $n$  sequences 1, 2, 3,  $\dots$ , etc.
3. Divide the source data into  $m$  segments.
4. For each segment of data, construct  $n$   $PP$ -trees in primary memory, storing finally to disk. This construction phase involves just one pass of the source data.
5. For partition 1, read the  $PP$  trees for all segments into memory and apply the Apriori-TFP algorithm to build a  $T$ -tree that finds the final frequent sets in the partition. This stage requires the  $PP$  trees for each segment of data to be read once only. The  $T$ -tree remains in memory throughout, finally being stored to disk.
6. Repeat step 5 for partitions 2, 3,  $\dots$ ,  $n$ .

The method offers two speed advantages over simple horizontal segmentation. First, we have now effectively reduced the number of disk passes to two: one (step 4) to construct the  $PP$ -trees, and a second pass (of the stored trees) to complete the counting (steps 5 and 6). The second advantage is that we are now, at each stage, dealing with smaller tree structures, leading to faster traversal and counting.

### 3.4 Comparison with other methods

As we have noted earlier, our  $P$ -tree structure, first presented in [12], has many properties in common with the  $FP$ -tree structure developed independently and contemporaneously by Han et al. [13]. The principal differences are two. First, the nodes of the  $FP$ -tree correspond to individual items, whereas in the  $P$ -tree a sequence of items which is partially closed (i.e. which has no leading subsequence with greater support in the tree) will be stored as a single tree node. Thus, for example, two transactions  $\{a, b, c, d, e\}$  and  $\{a, b, c, x, y\}$ , which share a common prefix  $\{a, b, c\}$ , would require in all seven nodes in the  $FP$ -tree. In the  $P$ -tree, conversely, only three nodes would necessarily be created: a parent for  $\{a, b, c\}$  and child nodes for  $\{d, e\}$  and  $\{x, y\}$ . The second difference is that, in order to implement the  $FP$ -growth algorithm, the  $FP$ -tree must store pointers at each node to link all nodes representing the same item and also to link a node to its parent and child nodes. The Apriori-TFP algorithm, however, treats the  $P$ -tree essentially as a set of nodes which can be processed in any order. This makes it possible, once the tree has been constructed, to store it in a tabular form in which no pointers are required.

Both these differences lead to a more compact tree structure. Furthermore, the absence of pointers allows us easily to use horizontal partitioning in order to build a succession of  $P$ -trees, each of which is vertically partitioned into  $PP$ -trees, as described earlier. It is then straightforward to collect all the  $PP$ -trees representing a single partition from their separate segments and use these trees in any order to construct the final  $T$ -tree for that partition.

Partitioning the  $FP$ -tree is necessarily more complex. The partitioning we describe, as illustrated in Fig. 3, is essentially similar to that obtained by the construction of *conditional* databases described in [13, 17]. In [14] two strategies are proposed for dealing with an  $FP$ -tree too large for primary storage. In the first of



these, *parallel projection*, the original database is partitioned into a set of projected databases, one for each item. Each projected database contains only transactions in which the item is present with some predecessors in the item ordering. Thus, each projected database essentially represents the same information as would be contained in a corresponding *PP*-tree in our method. The second method described, *partition projection*, would (using our example to illustrate) first construct the *a*-conditional database corresponding to *PP*3, and after building the *FP*-tree for this, would copy relevant transactions (e.g.  $\{e, d, b, c.a\}$ ), into the next (*c*-conditional) database, as  $\{e, d, b, c\}$ . This reduces the total size of the projected databases at a cost of some additional processing.

A number of other researchers have made use of the *FP*-tree or variants thereof. The *CFP*-tree described in [16] stores frequent closed item sets in a form that facilitates subsequent query processing. The construction algorithm is similar to that used for *FP*-tree construction, using conditional databases as described in [13] to partition the data and produce separate trees for query processing. The focus of this work is on the form of a structure that can be re-used efficiently, rather than on the efficiency of the construction algorithm. Reusability is also a feature of our *P*-tree structures, which retain all relevant information from the original data as well as performing part of the support counting. In [15] a structure is described, also (coincidentally) called a *P*-tree, which is quite similar to our *P*-tree but (like the *FP*-tree) stores only one item at each node. The *P*-tree of [15] shares with ours the property of retaining all the information from the database needed for counting of support at any threshold rather than just counts of frequent sets. The approach described in [15] constructs *FP*-trees from the *P*-tree rather than from the original data; results presented show that this offers significant performance gains when multiple *FP*-trees are required. The partitioning strategy described for dealing with large databases is essentially horizontal, dividing the data into segments for each of which a *P*-tree is constructed. The method produces a single overall *FP*-tree, however, for which further partitioning might become necessary.

The COFI-trees proposed in [10] also create subtrees that can be processed independently but require an initial construction of an *FP*-tree that must be retained in primary memory for efficient processing. In [11], a method is described for building COFI-trees from an inverted database structure called “Inverted Matrix”. In this structure, each item is represented by a row of the matrix which lists all transactions in which the item occurs. Each element of the list contains a pointer to the next item in the transaction. Constructing the Inverted Matrix requires two passes of the original database and will, in general, lead to an expansion in size because of the need to store a pointer with every item occurrence. The COFI-trees constructed from this have similar properties to the conditional *FP*-trees produced by database projection, as in [13]. However, to produce the COFI-trees it is necessary to mine the large inverted matrix, and the pointers between rows of this imply that no simple partitioning of this is possible. The approach seems to work well with very sparse databases, but for dense data it seems likely that following the links through a disk-resident matrix will be a costly overhead.

In general, all methods that use *FP*-tree-like structures require to employ projection of conditional databases in some manner in order to avoid the problem of dealing with a single large *FP*-tree or its equivalent in primary memory. In our

experiments, described later, we compare the performance of Apriori-TFP using our TP method with that of *FP-growth* using database projection.

## 4 Results

We first consider some general performance properties of our method. To investigate performance, we have used synthetic data sets constructed using the QUEST generator described in [3]. The programs were written in standard C++ and run under the Linux operating system. We performed these experiments on an AMD Athlon workstation with a clock rate of 1.3 GHz, 256 KB of cache and 512 MB of RAM. The data was stored on an NFS server (1 GB filestore).

We first need to establish that the method scales acceptably; that is, that the partitioning strategy successfully constrains the maximum requirement for primary memory, without leading to unacceptable execution times. For this purpose, we generated data sets with parameters  $T10.I5.N500$ : i.e 500 items, with an average record length of 10 items and an expected maximal frequent pattern size of 5. We divided the data into segments of 50,000 records, and within each segment generated 500 partitions, i.e. a *PP*-tree for each item. In this and all other experiments, the TP is naive: after ordering the items, we partition into sequences of equal length (in this case, 1). In fact, our experiments seem to show that increasing the degree of (vertical) partitioning always reduces the primary memory requirement (as would be expected) and also almost always reduces execution time. The latter, less obvious result arises because the increased time taken to construct a greater number of *PP*-trees (step 4 of the algorithm outlined earlier) is usually more than compensated by the faster processing of smaller *T*-trees (steps 5 and 6).

Figure 4 shows the overall time to generate frequent sets, with a support threshold of 0.01%, for data sets of increasing size, i.e. one, two, three, four and five segments. The figure shows the performance of the TP method in comparison with the simple method involving horizontal partitioning (segmentation) only (HP). The times illustrated include both the time to construct the *P*-trees (HP method) or *PP*-trees (TP method) and to execute the Apriori-TFP algorithm. As can be seen, TP offers substantially better performance than horizontal partitioning and its performance scales linearly with the size of the data set.

Importantly, this performance is achieved within conservative requirements for primary storage. In the second phase of the TP method (steps 5 and 6 of the

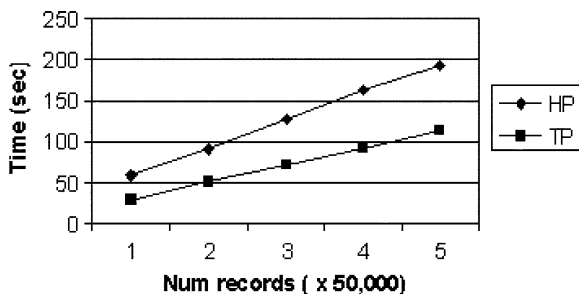


Fig. 4 Execution times for  $T10.I5.N500$  (0.01% support)

algorithm outlined earlier), it is necessary to contain in memory all the *PP*-trees for one partition and the corresponding *T*-tree containing the frequent sets in that partition. In the experiment of Fig. 4, this led to a maximum memory requirement for the TP method that varied from 1.38 MB (one segment) to 1.6 MB (five segments). In general, larger data sets, requiring greater horizontal segmentation, lead to some increase in the combined size of the *PP*-trees, but this is relatively slight. By contrast, the HP method requires the *P*-tree for one data segment and the whole of the *T*-tree to be contained in primary store, leading to a maximum memory requirement of between 116 and 128 MB in the case illustrated.

The combined sizes of the *PP*-trees for any one segment are, of course, greater than the size of a corresponding *P*-tree. In the experiment of Fig. 4, the sum of the sizes of the *PP*-trees for any one segment was about 15.56 MB (varying little between segments), compared to a *P*-tree size of about 1.85 MB. This was not the dominant store requirement in the case we have illustrated, but in other cases, could be a constraint during the construction of the *PP*-trees (step 4 of the method described earlier). If this is so, the problem can easily be overcome by imposing a greater degree of horizontal segmentation. Our experiments show that increasing the number of segments has little effect on execution times, while reducing memory requirements during the *PP*-tree construction. Figure 5 shows the results of experiments with increasing segmentation of *T* 10.15.N500.D50000, again with a support threshold of 0.01%. Having first established the linear scaling of the method, in the results presented earlier, this experiment for convenience used this relatively small database, but in order to replicate the problems of dealing with large databases, we imposed a requirement that only one segment can be retained in primary store at one time.

In this case, we imposed a vertical partitioning of 50 items per partition (10 partitions in all), while varying the number of segments. Figure 5 shows that the overall execution time (for both methods) increases only slightly with increasing segmentation. The total memory requirement to contain all the *PP*-trees for any one segment decreases, as one would expect, from 7.8 MB (1 segment) to a maximum of 0.22 MB (50 segments).

The method scales well with increasing number of items also. Figure 6 shows the execution times for the TP method for *T* 10.15.D50000, with a support threshold of 0.01, a vertical partitioning of 10 items per partition, and horizontal partitioning of 10,000 records per segment (i.e. five segments). The maximum memory requirement in this case also remained small (between 6 and 15 MB, with no

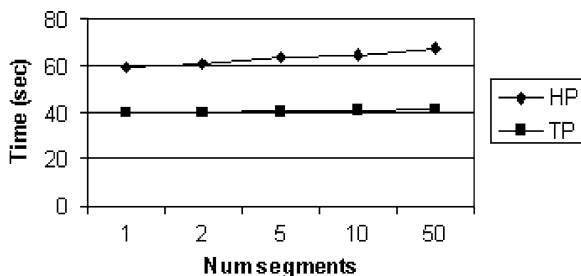


Fig. 5 Effect of increasing segmentation

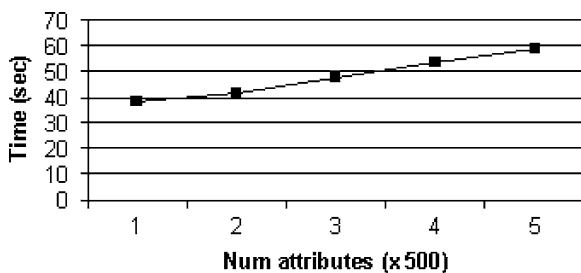


Fig. 6 Performance of TP with increasing number of items

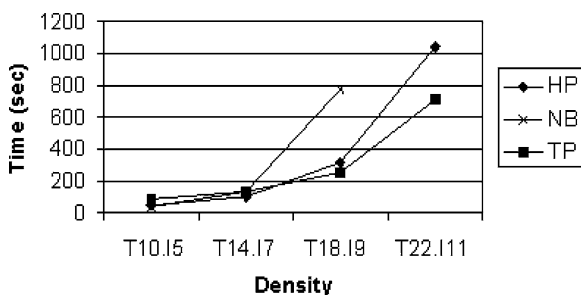


Fig. 7 Execution times for various database characteristics

general upward trend). Conversely, the simple HP method has a rapidly increasing memory requirement in this case because of the greater size of the unpartitioned  $T$ -tree.

The performance of the method when dealing with more dense data sets is shown in Figs. 7 and 8. In these experiments, we generated databases with  $D = 250,000$ ,  $N = 500$ , varying the  $T$  and  $I$  parameters. Here we compared the TP method both with HP and with a method based on the “Negative Border” approach of [19] (labelled NB in the figures). In the latter,  $P$ -trees are first constructed for all segments, as for the HP method. Then, all the proportionately frequent sets in the first segment of data are found using a support threshold reduced to two-third of the original and also retaining the negative border of the sets found, i.e. those sets which, although not themselves frequent, have no infrequent subsets. The frequent sets, with their negative border, are stored in a  $T$ -tree which is kept in store to complete the counts for these sets for the remaining segments. The reduced support threshold and the inclusion of the negative border make it very likely that all the finally frequent sets will be included in this tree, in which case the method requires the disk-stored  $P$ -trees to be read once only.

The figures show the results with a support threshold of 0.1% using 5 segments and 10 items per partition. With this support threshold (higher than in the previous experiments), there is little difference in the total execution times for the  $T10.15$  data: in this case, the faster computation of the frequent sets by the TP method is offset by the longer time taken to construct the  $PP$ -trees. With more dense data, however, the latter factor becomes decreasingly significant and the advantage of the TP method becomes increasingly apparent. This is principally because of the much smaller candidate sets that are involved. This becomes apparent from the

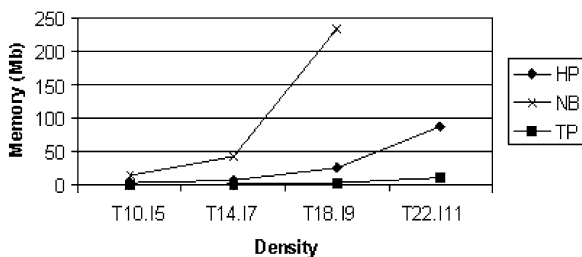


Fig. 8 Memory requirements for various data

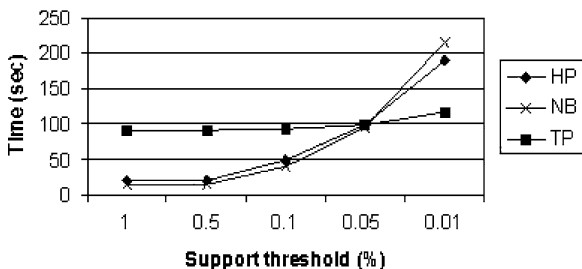


Fig. 9 Execution times for T 10.15.N500.D250000

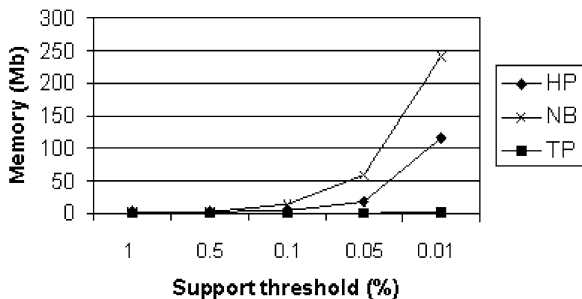


Fig. 10 Memory requirements for T 10.15.N500.D250000

comparison of maximal memory requirements shown in Fig. 8. This reflects the growing size of the candidate sets (and hence the *T*-tree) as the data density increases, leading both to larger memory requirements and to longer times to find candidates. The problem is particularly acute with the ‘Negative Border’ method. This works well with relatively sparse data, but at high density, the reduced support threshold and the inclusion of the negative border lead to very large candidate sets.

In Figs. 9 and 10, we compare the performance of the three methods for different support thresholds. These results again relate to the *T 10.15.N500.D250000* data, for support thresholds decreasing from 1.0 through to 0.01. In these experiments, we divided the data into five segments, and for the TP method used 500 partitions (1 item per partition). Here again, as can be seen from Fig. 9, the overhead of constructing the multiple *PP*-trees for the TP method leads to relatively poor execution times when the support threshold is high; in this case, the NB method is

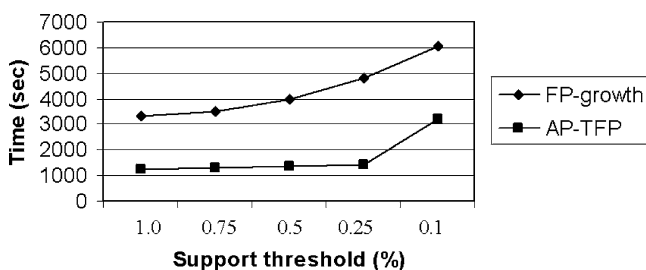


Fig. 11 Comparison of Apriori-TFP with *FP-growth*

fastest. As the support threshold is reduced, however, the increasing cost of servicing a growing candidate set leads to rapidly increasing memory requirements and execution times for the alternative methods, whereas the TP method scales much better.

Finally, we compare our method with an implementation of *FP-growth*, using the *FP-tree* structure of [13]. For this purpose, we implemented both Apriori-TFP, using our TP method, and an *FP-growth* algorithm in Java with the aim of obtaining as fair a comparison as possible. In this experiment, we used the data set *T20.I10.N500.D500K*, which was divided into five equal segments. For *FP-growth*, we processed each segment in turn to generate conditional databases for each of the 500 items, using the parallel projection method described in [14]. The conditional database segments were then combined in order to build an overall conditional *FP-tree* for each item in turn, to which the *FP-growth* algorithm was applied to produce frequent sets. For Apriori-TFP, each segment is used to construct 500 *PP-trees*, as described earlier. The experiments were in this case run on a 1.2 GHz Intel Celeron CPU with 512 MB RAM.

The overall execution times for the two methods are shown in Fig. 11, for support thresholds from 1.0 down to 0.1. As can be seen, on this data Apriori-TFP with TP outperforms a comparable implementation of *FP-growth* partitioned by parallel projection. We believe the advantage of Apriori-TFP in this case arises from the relatively efficient iterative processing of the simple *P-tree* structures, in comparison with the recursive generation of multiple *FP-trees* required by *FP-growth*. This is especially expensive with relatively dense data because of the greater depth of recursion required.

## 5 Conclusions

Because Data Mining is principally concerned with obtaining information from data of very large dimensions, it is important that methods used should scale effectively to deal with the most extreme cases. In this paper, we have examined ways of partitioning data for Association Rule Mining. Our aim has been to identify methods that will enable efficient counting of frequent sets in cases where the data is much too large to be contained in primary memory and also where the density of the data means that the number of candidates to be considered becomes very large. Our starting point was a method which makes use of an initial preprocessing of the data into a tree structure (the *P-tree*) which incorporates a partial

counting of support totals. In previous work, we have shown this method to offer significant performance advantages. Here, we have investigated ways of applying the approach in cases that require the data to be partitioned for primary memory use. We have, in particular, described a method that involves a partitioning of the tree structures involved to enable separate subtrees to be processed independently. The advantage of this approach is that it allows both the original data to be partitioned into more manageable subsets and also partitions the candidate sets to be counted. The latter results in both lower memory requirements and also faster counting.

The experimental results we have reported here show that the TP method described is extremely effective in limiting the maximal memory requirements of the algorithm, while its execution time scales only slowly and linearly with increasing data dimensions. Its overall performance, both in execution time and especially in memory requirements, is significantly better than that obtained from either simple data segmentation or from other methods considered. The advantage increases with increasing density of data and with reduced thresholds of support – i.e. for the cases that are in general most challenging for Association Rule Mining. Furthermore, a relatively high proportion of the time required by the method is taken up in the preprocessing stage during which the *PP*-trees are constructed. Because this stage is independent of the later stages, in many applications it could be accepted as a one-off data preparation cost. In this case, the gain over other methods becomes even more marked. Note also that the *P*-tree construction, and the partitioning thereof, is essentially generic: it leads to no loss of relevant information and so could be used as the first stage of other quite different algorithms for completing the support counts. A further advantage, not examined here, is that the independent processing of subtrees can be carried out in parallel.

## References

1. Agarwal R, Aggarwal C, Prasad V (2000) Depth first generation of long patterns. In: Proceedings of the ACM KDD conference on management of data, Boston, pp 108–118
2. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the ACM SIGMOD conference on management of data, Washington, DC, pp 207–216
3. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB conference, Santiago, Santiago, Chile, pp 487–499
4. Bayardo RJ (1998) Efficiently mining long pattern from databases. In: Proceedings of the ACM SIGMOD conference on management of data, pp 85–93
5. Bayardo RJ, Agrawal R, Gunopulos D (1999) Constraint-based rule mining in large, dense databases. In: Proceedings of the 15th international conference on data engineering, pp 188–197
6. Berry MJ, Linoff GS (1997) Data mining techniques for marketing, sales and customer support. Wiley, New York
7. Cheung W, Zaiiane OR (2003) Incremental mining of frequent patterns without candidate generation or support constraint. In: Proceedings of seventh international database engineering and applications symposium (IDEAS'03), pp 111–116
8. Coenen F, Goulbourne G, Leng P (2001) Computing association rules using partial totals. In: Proceedings of the 5th European conference on principles of data mining and knowledge discovery (PKDD'01). LNCS, vol 2168. Springer, Berlin Heidelberg New York, pp 54–66
9. Coenen F, Leng P (2001) Optimising association rule algorithms using itemset ordering. In: Bramer M, Coenen F, Preece A (eds) Research and development in intelligent systems XVIII (Proceedings of ES2001). Springer-Verlag, London, pp 53–66

10. El-Hajj M, Zaiane OR (2003) Non recursive generation of frequent K-itemsets from frequent pattern tree representations. In: Proceedings of 5th international conference on data warehousing and knowledge discovery (DaWaK 2003). LNCS, vol 2737. Springer-Verlag, Berlin Heidelberg New York, pp 371–380
11. El-Hajj M, Zaiane OR (2003) Inverted matrix: efficient discovery of frequent items in large datasets in the context of interactive mining. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining (KDD 2003), ACM, pp 109–118
12. Goulbourne G, Coenen F, Leng P (2000) Algorithms for computing association rules using a partial-support tree. *J Knowl Based Syst* 13:141–149 (also in Proceedings ES'99, December 1999. Springer, London, pp 132–147)
13. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the ACM SIGMOD conference on management of data, Dallas, pp 1–12
14. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Min Knowl Discov* 8(1):53–87
15. Huang H, Wu X, Relue R (2002) Association analysis with one scan of databases. In: Proceedings of the 2002 IEEE international conference on data mining (ICDM'02), pp 629–632
16. Liu G, Lu H, Lou W, Yu J (2003) On computing, storing and querying frequent patterns. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining (KDD 2003), ACM, pp 607–612
17. Pei J, Han J, Mao R (2000) CLOSET: an efficient algorithm for mining frequent closed itemsets. In: Proceedings of ACM SIGMOD workshop on data mining and knowledge discovery, pp 11–20
18. Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21th VLDB conference, pp 432–444
19. Toivonen H (1996) Sampling large databases for association rules. In: Proceedings of the 22th VLDB conference, pp 1–12
20. Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) New algorithms for fast discovery of association rules. Technical report 651, University of Rochester, Computer Science Department, New York



**Shakil Ahmed** received a first class BSc (Hons) degree from Dhaka University, Bangladesh, in 1990; and an MSc (first class), also Dhaka University, in 1992. He received his PhD from The University of Liverpool, UK, in 2005. From 2000 onwards he is a member of the Data Mining Group at the Department of Computer Science of the University of Liverpool, UK. His research interests include data mining, Association Rule Mining and pattern recognition.





**Frans Coenen** has been working in the field of Data Mining for many years and has written widely on the subject. He received his PhD from Liverpool Polytechnic in 1989, after which he took up a post as a RA within the Department of Computer Science at the University of Liverpool. In 1997, he took up a lecturing post within the same department. His current Data Mining research interests include Association rule Mining, Classification algorithms and text mining. He is on the programme committee for ICDM'05 and was the chair for the UK KDD symposium (UKKDD'05).



**Paul Leng** is professor of e-Learning at the University of Liverpool and director of the e-Learning Unit, which is responsible for overseeing the University's online degree programmes, leading to degrees of MSc in IT and MBA. Along with e-Learning, his main research interests are in Data Mining, especially in methods of discovering Association Rules. In collaboration with Frans Coenen, he has developed efficient new algorithms for finding frequent sets and is exploring applications in text mining and classification.