A Tree Partitioning Method for Memory Management in Association Rule Mining

Shakil Ahmed, Frans Coenen, and Paul Leng

Department of Computer Science, The University of Liverpool
Liverpool L69 3BX, UK
{shakil,frans,phl}@csc.liv.ac.uk

Abstract

All methods of association rule mining require the *frequent sets* of items, that occur together sufficiently often to be the basis of potentially interesting rules, to be first computed. The cost of this increases in proportion to the database size, and also with its density. Densely-populated databases can give rise to very large numbers of candidates that must be counted. Both these factors cause performance problems, especially when the data structures involved become too large for primary memory. In this paper we describe a method of partitioning that 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.

1 Introduction

The most computationally demanding aspect of Association Rule Mining is identifying the *frequent sets* of attribute-values, or *items*, whose *support* (occurrence in the data) exceeds some threshold. The problem arises because the number of possible sets is exponential in the number of items. For this reason, almost all methods attempt to count the support only of c*andidate* itemsets that are identified as possible frequent sets. It is, of course, not possible to completely determine the candidate itemsets in advance, so it will be necessary to consider many itemsets that are not in fact frequent. Most algorithms involve several passes of the source data, in each of which the support for some set of candidate itemsets is counted. The performance of these methods, clearly, depends both on the size of the original database and on the number of candidates 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, where we may be searching for rules that associate rather rare items within quite densely-populated data, 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. In this paper we examine methods of partitioning to limit the total primary memory requirement, including that required both for the source data and for the candidate sets. We describe a new method of partitioning that exploits tree structures we have previously developed for Association Rule Mining. Experimental results are presented that show this method offers significant performance advantages.

2 Background

Most methods for finding frequent sets are based to a greater or lesser extent on the "Apriori" algorithm [2]. 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.

A problem of Apriori is that it requires the source data to be scanned repeatedly, which is especially costly if this cannot be contained in primary memory. Attempts to reduce this cost include the "Partition" algorithm [12], which partitions the data into a number of equal-sized segments of manageable size, and the strategy introduced by Toivonen [13], which first processes a small random sample of the data to identify candidates for counting across the full database. The drawback of both these approaches highlights the second weakness of Apriori: that the number of candidates whose support must be counted may become very large. Both methods increase the size of the candidate set, and also require all candidates to be retained in primary memory (for efficient processing) during the final database pass. Other methods [3] [4] [1] [14] 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. [14] partitions candidate sets 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 pre-processing 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 [9], which enumerates itemsets counted in lexicographic order. 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, consider a database with items {a,b,c,d,e}, and a set of 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. For example, suppose the record cde is the last to be added. The tree-construction

algorithm will traverse the nodes c and cd that are subsets of cde, incrementing the count for each, before appending cde as a child of cd.

We apply to this structure an algorithm, Apriori-TFP, which completes the summation of support counts, storing the results in a second set-enumeration tree (the *T*-tree), which 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 [6], where we explain the rationale of the approach, and present experimental results that demonstrate performance gains in comparison both with Apriori, and also the *FP-growth* [10] algorithm, which has some similar properties. The *FP*-tree used in [10] 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 proposed, which will be discussed further below. The CATS tree, an extension of the FP-tree proposed in [5], also assumes no limitation on main memory capacity. In this paper we consider implementations of Apriori-TFP when we cannot contain all the data required in main memory, requiring some strategy for partitioning this.

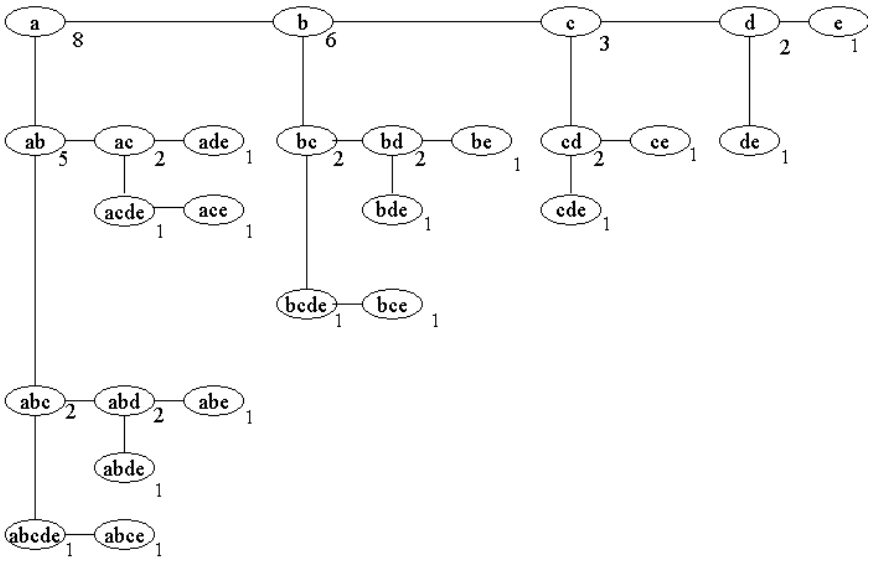

Figure 1: Example of a P-tree

3 Strategies for 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. Each pass of Apriori-TFP then requires each of the *P*-trees to be read in turn from secondary memory. The method creates a single final *T*-tree in primary memory, which contains all the frequent sets and their support-counts.

The drawback of this simple approach is that it replicates the two weaknesses of the Apriori methodology: all the data (now in the form of *P*-trees) must be re-read from secondary memory in each pass, and the entire *T*-tree, which contains candidates and, finally, all the frequent sets, must be kept in primary memory while counting proceeds. As we have noted, this tree may be very large. Even when it can be held in primary memory, a large set of candidates leads to slower counting, in Apriori-TFP just as for Apriori. The *P*-tree structure offers another possible form of partitioning, into subtrees that represent equivalence classes of the items represented. However, it is 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 Figure 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 Figure 1, we might define 3 sequences of items, {a,b}, {c,d} and {e}, labelled 1,2,3 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, 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 Figure 2. The *PP*-trees are, in effect, overlapping partitions of the *P*-tree of Figure 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 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,a. Using the same data as for Figures 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 Figure 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 the 23 of PP3 in Figure 2. Our previous work has shown [7] 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 2 counts (db+edb).

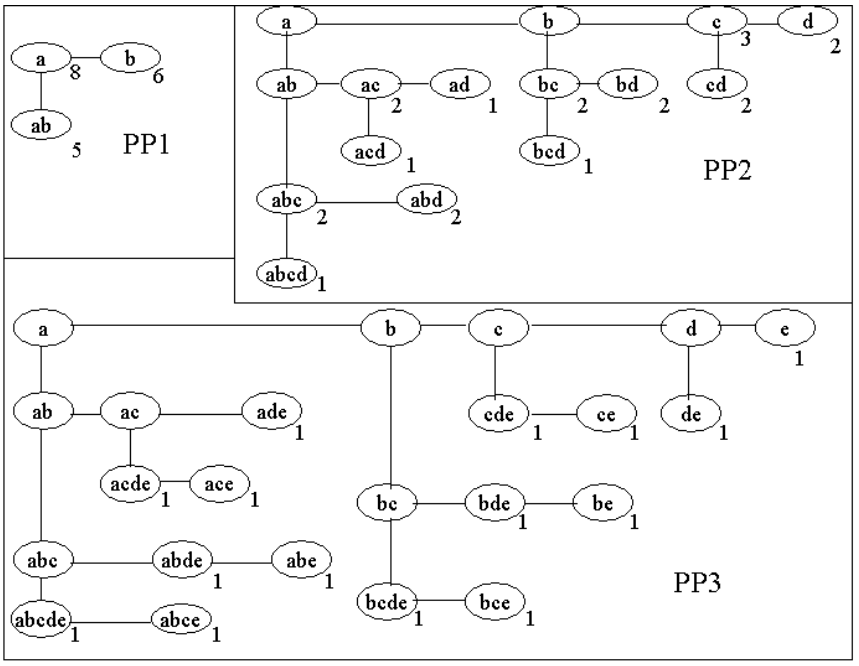

Figure 2: Partition-P-trees from figure 1

This form of partitioning offers us a way of dividing the source data into a number of *PP*-trees each of which may then be processed independently. The partitioning, as illustrated in Figure 3, is essentially similar to that obtained by the construction of *conditional* databases described in [10] and [11], and the COFI-trees proposed in [8]. The latter method also creates subtrees that can be processed independently, but requires an initial construction of an *FP*-tree that must be retained in primary memory for efficient processing. The partitioning strategy proposed in [11] for dealing with an *FP*-tree too large for primary storage, would first construct the *a-conditional* database corresponding to PP3, and after building the *FP*-tree for this, would copy relevant transactions (e.g. edbca), into the next (*c-conditional*) database, as edbc. The method we describe avoids this multiple copying by constructing all the trees in memory in a single pass, during which we also partially count support. 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. This approach is possible in our case because the *P*-tree structure is simpler and less pointer-rich than the corresponding *FP*-tree. Our method also allows us to define *PP*-trees which are, in effect, the union of a sequence of conditional databases, allowing more flexibility of partitioning.

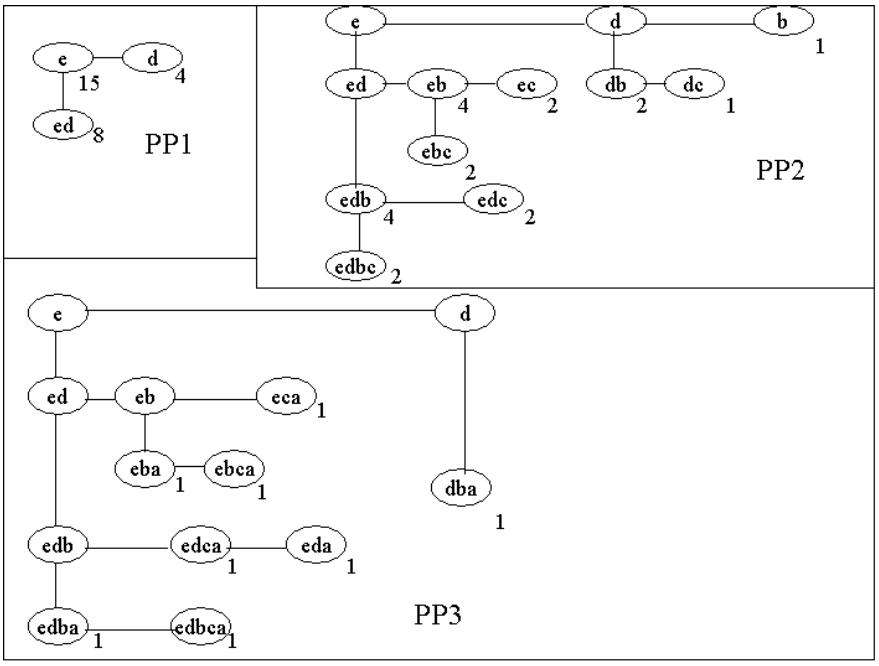

Figure 3: PP-trees after reordering of items

The overall method is as follows:

1. Obtain an (at least approximate) ordering of the frequency of items, and using this ordering, choose an appropriate partitioning of the items into *n* sequences 1, 2, 3,..etc.
2. Divide the source data into *m* segments.
3. For each segment of data, construct *n PP*-trees in primary memory, storing finally to disk. This involves just one pass of the source data.
4. For partition 1, read the PP1 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 PP1 trees for each segment of data to be read once only. The *T*-tree remains in memory throughout, finally being stored to disk.
5. Repeat step 4 for partitions 2, 3, ..*n*.

The method offers two advantages. First, we have now effectively reduced the number of disk passes to 2: one (step 3) to construct the *PP*-trees, and a second pass (of the stored trees) to complete the counting (steps 4 and 5). The second advantage is that we are now, at each stage, dealing with smaller tree structures, leading to faster traversal and counting.

4 Results

To investigate the performance of the method, we used synthetic data sets constructed using the QUEST generator described in [2]. The programs were

written in standard C++ and run on a 1.3 GHz machine with 256 Kb of cache, and 512 Mb of RAM. The data was stored on an NFS server (1Gb filestore).

We first establish that the method scales acceptably; i.e. 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 an average record size of 10 items. 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 all our experiments, the tree partitioning is naïve: 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 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 3 above) is usually more than compensated by the faster processing of smaller *T*-trees (steps 4 and 5).

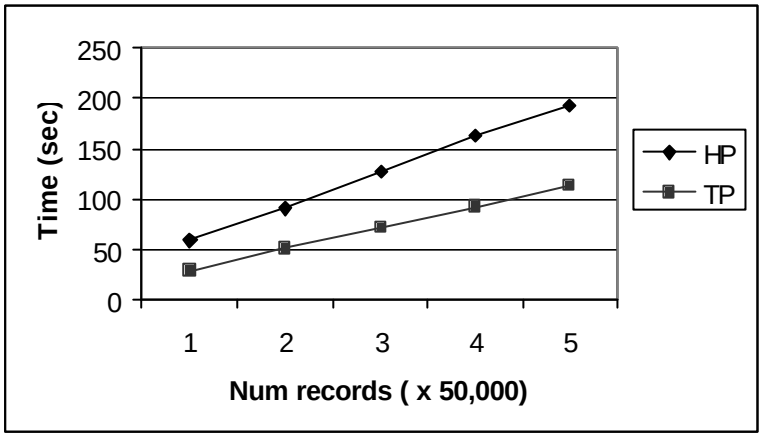

Figure 4: Execution times for T10.I5.N500 (0.01% support)

Figure 4 shows the overall time to generate frequent sets, with a support threshold of 0.01%, for datasets of increasing size, i.e. 1,2,3,4 and 5 segments. The figure compares the performance of the Tree-partitioning method (TP) with the simple method involving horizontal partitioning (segmentation) only (HP). The times illustrated include both the time to construct the trees and to execute the Apriori-TFP algorithm. As can be seen, TP offers substantially better performance than HP, and its performance scales linearly with the size of the dataset.

Importantly, this performance is achieved within conservative memory requirements. In steps 4 and 5 of the TP method it is necessary to contain in memory all the *PP*-trees for one partition, and the corresponding *T*-tree for that partition. In the experiment of Figure 4, this led to a maximum memory requirement for the TP method that varied from 1.38Mb (1 segment) to 1.6Mb (5 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, which could be a constraint during the construction of the *PP*-trees. If so, the problem can be overcome by imposing a greater degree of horizontal segmentation. We found that the total memory requirement to contain all the *PP*-trees for any one segment decreases from 7.8 Mb, when no segmentation is applied, to a maximum of 0.22 Mb when there are 50 segments, with almost no overall increase in execution time. The method scales well with increasing number of items, also. We found execution time increased slowly and linearly as the number of items increased from 500 to 2500, with no general upward trend in memory requirement. This is because the *P*-tree is a conservative structure, which stores (in general) only those itemsets that are actually needed, so increasing the number of items with no increase in density has little effect. 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 datasets is shown in figures 5 and 6. In these experiments, we generated databases with D=50000, N=500, varying the T and I parameters, and divided into 5 segments, with 10 items/partition. The small database size was chosen for convenience, but we imposed a requirement that only one partition can be retained in memory. We compared the TP method both with HP and with a method based on the "Negative Border" approach of [13] (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 2/3 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.

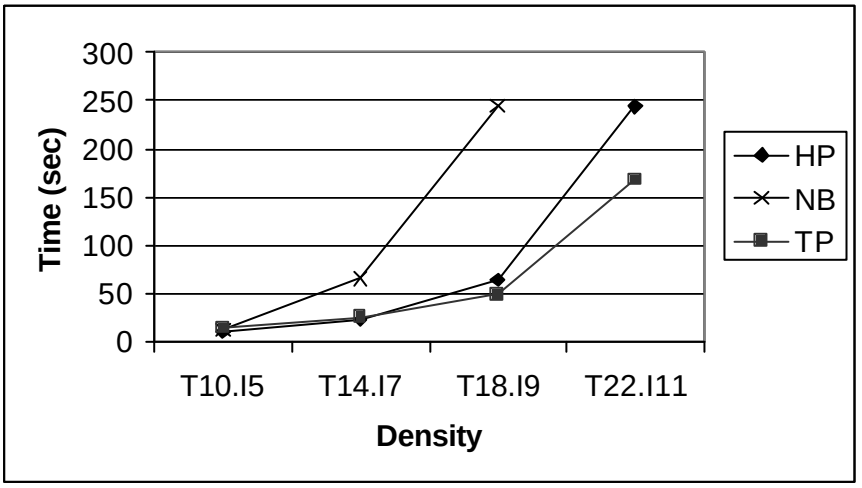

Figure 5: Execution times for various database characteristics

The figures show results for a support threshold of 0.1%, at which level (higher than in the previous experiments) there is little difference in the total execution times for the T10.I5 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 is apparent from the comparison of maximal memory requirements, shown in Figure 6. 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. A similar pattern was observed in experiments varying the support threshold (for the T10.I5 set). At a threshold of 1.0, the overhead of constructing the multiple *PP*-trees for the TP method leads to relatively poor execution times, and in this case, the NB method is 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, becoming faster than either at a threshold of 0.05, and twice as fast at threshold 0.01.

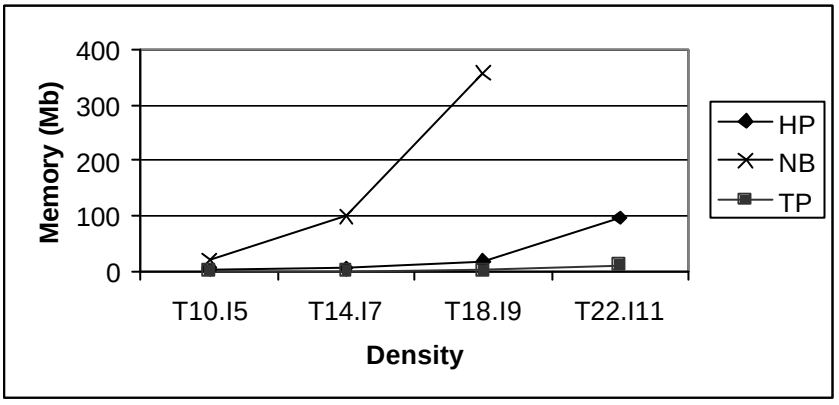

Figure 6: Memory requirements for various data

5 Conclusions

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 Tree Partitioning 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 a method that aims to complete counting in only two passes of the data. 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.

References

1. Agarwal, R., Aggarwal, C. and Prasad, V. Depth First Generation of Long Patterns. In Proc. of the ACM KDD Conference on Management of Data, Boston, pages 108-118, 2000.
2. Agrawal, R. and Srikant, R. Fast Algorithms for Mining Association Rules. In Proc. of the 20th VLDB Conference, Santiago, Santiago, Chile, pages 487-499, September 1994.
3. Bayardo, R.J. Efficiently Mining Long Pattern from Databases. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 85-93, 1998.
4. Bayardo, R.J., Agrawal, R. and Gunopulos, D. Constraint-Based Rule Mining in Large, Dense Databases. In Proc. of the 15th Int'l Conference on Data Engineering, 1999.
5. Cheung, W. and Zaiane, O.R. Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint. Proc. IDEAS 2003 Symposium 111-116
6. Coenen, F., Goulbourne, G., and Leng, P. Computing Association Rules using Partial Totals. PKDD 2001, pages 54-66, 2001.
7. Coenen, F. and Leng, P. Optimising Association rule Algorithms using Itemset Ordering. In 'Research and Development in Intelligent Systems XVIII (Proc ES2001 Conference, Cambridge), eds M Bramer, F Coenen and A Preece, Springer-Verlag, London, 2002, 53-66
8. El-Hajj, M. and Zaiane, O.R. Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations. Proc DAWAK 2003, pp 371-380
9. Goulbourne, G., Coenen, F. and Leng, P. Algorithms for Computing Association Rules Using a Partial-Support Tree. J. Knowledge-Based System 13 (2000), pages 141-149. (also Proc ES'99.)
10. Han, J., Pei, J. and Yin, Y. Mining Frequent Patterns without Candidate Generation. In Proc. of the ACM SIGMOD Conference on Management of Data, Dallas, pages 1-12, 2000.
11. Pei, J., Han, J. and Mao, R. CLOSET: an efficient algorithm for mining frequent closed itemsets. Proc ACM SIGMOD Workshop on Data Mining and Knowledge Discovery, 2000, 11-20
12. Savasere, A., Omiecinski, E. and Navathe, S. An Efficient Algorithm for Mining Association Rules in Large Databases. In Proc. of the 21th VLDB Conference, Zurich, Swizerland, pages 432-444, 1995.
13. Toivonen, H. Sampling Large Databases for Association Rules. In Proc. of the 22th VLDB Conference, Mumbai, India, pages 1-12, 1996.
14. Zaki, M.J. Parthasarathy, S. Ogihara, M. and Li, W. New Algorithms for fast discovery of association rules. Technical report 651, University of Rochester, Computer Science Department, New York. July 1997.