Strategies for Partitioning Data in Association Rule Mining

Shakil Ahmed, Frans Coenen, and Paul Leng

Department of Computer Science, University of Liverpool
Liverpool L69 7ZF, United Kingdom
{shakil, frans, phl}@csc.liv.ac.uk

Abstract. The problem of extracting association rules from databases is well known. The most demanding part of the problem is the determination of the *support* for all those sets of attributes which occur often enough to be of possible interest. We have previously described methods we have developed that approach the problem by first constructing a tree (the P-tree) that contains a record of all the relevant information in the database and a partial computation of the support totals. This approach offers significant performance advantages over comparable alternative methods, which we have demonstrated experimentally with store-resident datasets. In practice, however, the real focus of interest is on much larger databases. In this paper we discuss strategies for partitioning the data in these cases, and present results of the performance analysis.

Keywords: Association Rules, Partial Support, Data Structures, Partitioning.

1 Introduction

A well-known method for knowledge discovery in databases (KDD) involves the extraction of *association rules* [**?**] from a binary database. In a database of this kind, each attribute simply records the presence or absence of some property, or *item*, in the record. An association rule is a probabilistic relationship, of the form $A \Rightarrow B$, between disjoint sets of these items, obtained from examination of the data. The task is to discover, in applications such as supermarket shopping-basket analysis, important associations among items such that the presence of some items in a transaction will imply the likely presence of other items in the same transaction; that is, rules that associate one set of attributes of a relation to another.

A rule is usually only of potential interest if the set of attributes it associates occurs together relatively frequently in the data being examined. We define these *frequent* sets to be those which exceed some defined level of *support*. The support for the rule $A \Rightarrow B$ is the number (or proportion) of database records which contain $A \cup B$. The principal problem in association rule mining is the search for sets which exceed the support threshold. Once these frequent sets have been found, it is relatively straightforward to identify the rules that are likely to be of interest.

Finding all the frequent sets is a computationally demanding problem because of the exponential scale of the search space. The well-known *Apriori* algorithm [?] was one of the first to address this problem, and many subsequent algorithms have been described, many based on Apriori. We have previously introduced methods that begin by preprocessing the data in a single pass of the database which performs a partial summation of the support totals. These partial counts are stored in a set-enumeration tree structure which we call the *P*-tree. We then apply to this structure an algorithm, Apriori-TFP, which completes the summation of the final support counts, storing these in a second, differently-ordered, set-enumeration tree (the *T*-tree).

In experiments comparing the performance of this approach with other published methods, Apriori-TFP has compared favourably. To provide a fair basis for comparison, it is usual to examine data which can be wholly retained in main memory during all the computation. In reality, however, most of the most demanding applications of association-rule mining will involve data on a much larger scale. Not all methods translate easily to an implementation involving non-store-resident data, and performance in these cases may not scale linearly with the database size.

In this paper we examine possible methods for partitioning data in order to apply the Apriori-TFP algorithm in implementations which assume the data cannot be store-resident. We consider both "horizontal" partitioning, which divides the original data into smaller sets of records, and "vertical" partitioning, which divides the data by attribute. We present an experimental evaluation of the approaches.

2 Background

The biggest problem in deriving association rules is the exponential complexity of the task of computing support counts for all possible combinations of items. All practical algorithms, therefore, try to identify those sets of items which may possibly exceed the required threshold of support. Most methods are based to a greater or lesser extent on the "Apriori" algorithm [?]. Apriori performs repeated passes of the database, successively computing support-counts for sets of single attributes, pairs, triplets, and so on. At the end of each pass, sets which 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. The method may still be computationally expensive, however, especially for low support thresholds and when the database is very densely-populated, as in these cases the number of candidates to be considered may become very large, and many passes of the data may also be required.

Strategies to improve on the basic Apriori algorithm include algorithms which begin by partitioning [?] or sampling [?] the database to estimate the likely candidate set, or by checking subsets dynamically [?]. These methods reduce the number of database passes involved, but tend to generate more candidates to be counted. Other methods [?] [?] [?] reduce the search space by aiming to identify *maximal* frequent sets without first examining all their subsets. These algorithms may cope better with densely-populated databases than the other algorithms described, but at the cost of multiple database passes.

We have previously described [?] a class of methods we have introduced which begin by performing a single database pass to perform a restructuring of the relevant data and a partial computation of the support totals. We construct a tree, which we call the *P-tree*, which contains all the sets of items present as distinct records in the database (plus a few additional sets required to maintain the tree structure efficiently). These sets are stored in lexicographic order, in a form such that each subtree contains only lexicographically following supersets of its parent node. This order makes it easy and efficient, as the tree is constructed, to count for each node the number of occurrences of lexicographically-following supersets in the data. Figure 1 illustrates a P-tree for the attribute-set $\{A, B, C, D\}$. The illustration is of a complete tree, that would arise if every combination of attributes were present in the database; in practice, this will not occur in real large-scale data.

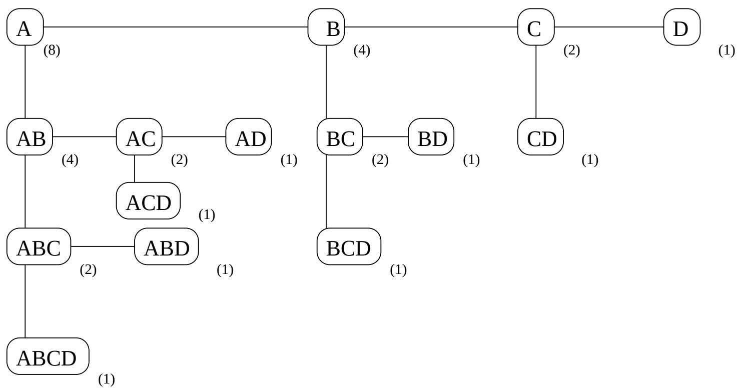

Fig. 1. Example of a P-tree

To complete the counting of support for a set S, then, we need only find S and its lexicographically-preceding immediate supersets in the P-tree. The algorithm we have used for this purpose, which we call Apriori-TFP, is described in [?]. Apriori-TFP (Total-from-Partial) uses the basic Apriori methodology, applied to the P-tree rather than to the original database. The candidate sets to be considered are stored in a second set-enumeration tree, the *T-tree*, ordered in the opposite way to the P-tree. On completion of the algorithm, the T-tree contains just the frequent sets, with their final support totals.

The performance of Apriori-TFP improves on the original Apriori for two reasons: firstly, the initial very efficient partial computation carried out when constructing the *P*-tree avoids some less efficient computation that Apriori must carry out repeatedly, on each pass; and secondly, the structure of the *T*-tree provides a very efficient way of locating the candidates for which support is to be counted in the passes of Apriori-TFP. Experimental results reported in [**?**] confirm this gain, and also demonstrate improved performance over the *FP-growth* [**?**] algorithm, which has some similar properties.

Like most experimental work on association-rule algorithms, our performance evaluation has been based on measurements using main-store-resident data, because this provides the firmest basis for performance measurement. It is clear, however, that for most applications only those methods that will cope effectively with much larger datasets will be suitable. Not all methods can be readily adapted to deal with non-store-resident data. For example, the otherwise very efficient *DepthProject* [**?**] algorithm is explicitly targeted at memory-resident data, while the multiply-linked structure of the *FP*-tree used in [**?**] also causes problems for non-store-resident implementations.

In this paper we examine strategies for implementing Apriori-TFP in cases when it will be impossible to contain all the data required in main memory. The memory requirement for Apriori-TFP, as for most other methods, is twofold. Firstly, the original data (in our case, transformed into a *P*-tree) must be contained in primary memory or else read repeatedly for each pass of the data. Secondly, we require to contain in primary memory the *candidate* sets of attributes whose support is being counted. The number of such candidates can become very large, especially if a low threshold of support is required. The overall memory requirement, therefore, includes both these factors. Two broad approaches for reducing this suggest themselves. First, "horizontal" partitioning: we can divide the database into a number of non-overlapping segments each of which is small enough to be handled in main memory. Alternatively or additionally, we can partition "vertically", dividing by attribute, and hence defining subsets of the candidates that can be counted separately. From these approaches, we have defined four methods for carrying out the counting of support totals needed, described in the following sections.

3 Method I: TFP with Horizontal Partitioning

Method 1 may be regarded as the 'natural' implementation of the Total- from Partial- support (TFP) method in this context. We begin by dividing the data into equal-sized segments, and create a separate *P*-tree for each segment. In this and the other methods described, the *P*-trees thus created enumerate all the sets present as distinct itemsets in the segment under consideration, recording for each an incomplete summation of their support-count. 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 *T*-tree. Detailed descriptions of the algorithms for building a P-tree in a single database pass and for counting support totals using a T-tree are given in [?] [?].

The first stage is to build a P-tree for each segment, storing this on backing store. For this purpose, the P-tree is converted to a table (array of arrays) form. The rows of the P-tree table consist of corresponding levels of the P-tree; so that all 1-itemsets of the P-tree are included in level 1 of the table, 2-itemsets in level 2, and so on. This is possible because the Apriori-TFP algorithm does not require the linkage between nodes in the tree to be maintained, and can traverse the P-tree in any order. Thus, it is possible to treat the stored P-trees as a single structure, divided into segments of manageable size.

To build a single T-tree from the P-trees stored on backing store, we use a form of the Apriori-TFP algorithm described in [?]. The algorithm is (for $K =$ 1, 2,. . .):

1. Build level K in the T-tree.
2. Read all the stored P-trees in turn, adding interim supports associated with individual P-tree nodes to the level K nodes established in (1).
3. Remove any level K T-tree nodes that do not have an adequate level of support.
4. Repeat steps (1), (2), and (3); until a level K is reached where no nodes are adequately supported.

The method creates a final single T-tree in main memory, that contains all the frequent sets and their support-counts. Note that the main memory requirement is for storage to contain the P-tree for a single segment of the data, plus the whole of the T-tree.

4 Method II: TFP using Negative Border

The method described above is the most obvious and simplest way of transforming our Apriori-TFP algorithm into a form applicable when data is not resident in main memory. The drawback, of course, is that the basic Apriori methodology requires repeated passes to be made of the database to compute the support for single items, pairs etc., in turn. For Apriori-TFP, the P-tree is used instead of the database, but we will still require to read each P-tree segment repeatedly from backing store.

The concept of the *negative border* [?] tries to avoid this by obtaining an early estimate of the possible frequent sets, which can then be counted in one pass. To do this, we take a single segment of data, find all the sets that are frequent in this segment, then verify the results with the rest of the database. Because there may be sets that are frequent in the entire database but not in our chosen segment, however, we need to enlarge the set of candidates to be considered. First, the support threshold is lowered so much that it is very unlikely that any frequent sets are missed. Then, after finding all the *locally frequent* sets in the segment being examined, this collection is augmented by adding its 'negative border'. The *negative border* is the collection of all candidates that are not frequent in the segment but all its subsets are. The significance of the negative border is that it defines a boundary between the frequent and non-frequent sets. If no set on the negative border is finally found to be frequent, then no sets 'outside' the border can be frequent either.

The application of this method in our case can be divided into three stages:

1. First, build and store P-trees for each segment of data, as for Method 1.
2. Build a T-tree with its negative border for the first segment of data (for simplicity, we will assume that the data is randomly distributed, so that the first segment can be taken as an accurate sample of the whole).
3. Update the T-tree for the other segments of data.

An initial T-tree with negative border is built, using a lowered support threshold, from the first stored P-tree segment, using a slightly modified Apriori-TFP algorithm. Although this will again require repeated passes of the P-tree segment, if this can be contained in main memory then it will be read once only. The negative border is easily obtained. In Apriori-TFP, a candidate set is added to the tree when its subsets are all frequent. If after counting its support, the set is found to be infrequent, it is deleted from the tree. In this variant, we retain these sets on the tree, but do not use them to construct further (superset) candidates. Thus, these sets are not themselves frequent, but have only frequent subsets, and have no supersets in the final tree.

For each of the other segments of data, we read the P-tree into main memory, then perform a single traversal of this to count the support of all the sets in the T-tree, adding these to the totals already stored. The final T-tree contains the final support-counts for all the candidate sets identified in the first section, including those on their negative border. Sometimes, unfortunately, it may happen that we find out that all necessary sets have not been evaluated. There has been a *failure* in the building if all frequent sets are not found in one pass. If there are no misses, then the method is guaranteed to have found all frequent sets. Misses indicate a potential failure; if there is a miss of any set, then some superset of the missed set might be frequent but not evaluated. A miss is identified by checking the negative border. If any superset can be constructed, all of whose subsets are frequent, then it will be necessary to perform another pass to count its support.

5 Methods III and IV: TFP with Vertical Partitioning

Both methods described above require main memory sufficient to contain the P-tree for a single segment of data together with the whole of the T-tree. Because there is no partitioning of the T-tree, these may face problems in dealing with very large candidate sets. A possible alternative way of reducing the requirement to manageable proportions is to divide the set of attributes under consideration into subsets, corresponding to which there is a partitioning of both the P-tree and the T-tree into subtrees that can be processed separately. The problem with this "vertical" partitioning strategy is that, although subtrees of the T-tree properly represent subsets of the candidate set, the subtrees of the P-tree do not map directly on to this. This is because although *succeeding* supersets of a set S are located in the subtree rooted at S, *predecessor* supersets are scattered throughout the preceding part of the P-tree, which must therefore be traversed in order to compute the support for S.

The problem can be overcome by increasing the amount of counting done during the construction of the P-tree. We begin by dividing the ordered attribute-set into (say) 4 subsequences, labeled 1, 2, 3, 4. We then proceed to construct separate *Partition-P*-trees, which we will label PP1, PP2, PP3, PP4 respectively, for these attribute-sets. PP1 is a normal P-tree for the attributes in subset 1. PP2, however, counts all those sets which include at least one item from subset 2, in a tree which includes both set 2 and its predecessors. PP4, finally, will include all attributes, but will count only those sets which include at least one item from subset 4. Suppose, for example, the attribute-set of figure 1 was used to define 4 PP-trees for items A, B C and D respectively. Then, a record comprising the itemset ACD would be counted in PP1 (as A), PP3 (AC), and PP4 (ACD).

The effect of this is that the total support for any set S can now be obtained from the PP-tree within which S is located. The apparent drawback is that we now need to construct not one but several P-trees, of increasing size. In our example, the "tree" PP1 will comprise only the single node A, whereas at first sight it appears that PP4 will be as large as the original P-tree. In practice, however, this is not the case as only those records which contain a D will be represented in PP4. The size of the largest PP-tree can be further reduced if we order the attributes by descending expected frequency (a heuristic that also, as we have found in previous work [?], improves the performance of Apriori-TFP).

Method 3, therefore, may be summarized thus:

1. Choose an appropriate partitioning of the attribute-set into sequences 1, 2,...k.
2. Perform k passes of the database to build *Partition-P*-trees PP1, PP2...PPk.
3. Read PP1 into memory, and build a T-tree to count the total support for all frequent sets formed from members of set 1 only.
4. Read PP2 into memory, and build a T-tree counting the support for frequent sets formed from members of set 2 with its predecessors.
5. Repeat step 4 for PP3, PP4, etc.

The method reduces the maximum primary memory requirement, as we now need to contain in memory only a *PP*-tree together with the subtree of the T-tree corresponding to that partition. A further advantage of the method is that more of the counting is done by the relatively efficient procedures used when constructing the P-tree. The chief drawback, however, is that repeated passes of the database are now required to construct the *PP*-trees. To overcome this, we can combine the "vertical" partitioning of Method 3 with a "horizontal" segmentation of the database as in Methods 1 and 2. In this case, (Method 4) we will first build *PP*-trees, as described for Method 3, for each segment of the database (choosing a segment size which allows this to be done entirely in primary memory). Then, the stored *PP*-trees are processed in the order of the vertical partitioning, to construct the complete T-trees for partition 1, 2 ...in turn. In this case the maximum memory requirement is for the *PP*-trees for one partition, plus the corresponding *T*-tree partition, and the *PP*-trees stored for each segment need be read once only.

6 Results

In this section we describe our experiments and the performance results for the methods described above. All the programs were written in standard C++ and run under the Linux operating system. We performed our 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 used synthetic data sets constructed using the QUEST generator described in [?]. This uses parameters T, which defines the average number of items found in a set; and I, the average size of the maximal supported set. Higher values of T and I in relation to the number of items (attributes) N correspond to a more densely-populated database. For all the experiments described here, we have used the parameters T =10, I =5, N = 500.

We first examined the two "horizontal" partitioning methods, 1 and 2. For these experiments, we generated a database of 500,000 records, and investigated partitions of this into 5, 10 and 20 equal segments, for each of which a separate *P*-tree was constructed and stored as described above. The time to create the *P*-trees was 30.25, 25.78 and 22.82 seconds for 5, 10 and 20 segments respectively. Interestingly, because the size of the trees reduces as the number of segments increases, the time to build the trees reduces for more segments. Table 1 then compares the performance of the methods (times in seconds) in using the stored *P*-trees to compute final support totals for all frequent sets, for varying support thresholds. For method 2, the actual support thresholds were reduced in each case to 2/3 of the chosen figure when constructing the initial *T*-tree, with the aim of avoiding failure to find all frequent sets in one pass.

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*
Method 1 (5 segs)	18.54	19.69	89.10	210.01
Method 1 (10 segs)	18.67	19.99	89.59	213.46
Method 1 (20 segs)	19.77	20.96	93.91	221.78
Method 2 (5 segs)	6.45	7.62	80.47	199.51(Fail)
Method 2 (10 segs)	6.30	7.22	75.76(Fail)	216.59(Fail)
Method 2 (20 segs)	6.59	7.45	89.01(Fail)	241.16(Fail)

Table 1: Times to find frequent sets for T10I5N500D500000

As can be seen, the performance of Method 1 varies only slightly for the three different segmentations, especially if the time to build the *P*-tree is taken into account. This suggests that the method will scale well for larger datasets, requiring more partitions. Method 2 outperforms Method 1 for relatively high support thresholds, gaining because of the reduced number of passes of the stored *P*-trees that are required. At lower support thresholds, however, the advantage begins to be offset by the increased cost of managing a larger T-tree, and, worse, the heuristic used to estimate the candidate set proves to be inadequate in some cases, so that the algorithm fails to find all the frequent sets in one pass. In this event, another full pass of the disk-resident data would be required to complete the counting of supports (the time for this is not included here).

Table 2 shows the storage requirements for these experiments. The figures tabulated, in megabytes, are the sum of the store required to contain the largest P-tree segment, plus the whole of the T-tree. Thus, in each case, the table records the maximum main memory requirement for the Method. These results illustrate the problem with these methods; at low support thresholds (or with data of high density), the number of candidates to be included in the T-tree increases sharply. This is the cause of the sharp increase in execution times observed in Figure 1 as the support threshold is reduced. This adverse scaling is, of course, a characteristic of all methods of finding frequent sets. The further effect, however, is that the reduction in memory requirement brought about by partitioning the data is ineffective as the store required for the candidate set becomes dominant, and in the most extreme cases, this may make the methods infeasible. Method 2 scales less well than Method 1 in this respect. because of the larger candidate sets imposed by the reduced support threshold and negative border.

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*
Method 1 (5 segs)	3.6	3.8	7.6	50.4
Method 1 (10 segs)	1.9	2.0	5.9	48.5
Method 1 (20 segs)	0.1	0.3	4.1	46.8
Method 2 (5 segs)	3.7	4.1	37.1	141.6
Method 2 (10 segs)	2.0	2.3	32.7	145.4
Method 2 (20 segs)	0.2	0.6	37.6	141.7

Table 2: Memory requirements for T10I5N500D500000

In examining Method 3, we first examined the effect of different degrees of partitioning. We investigated partitions comprising 50, 25, 10 and 1 items: that is, for N =500 attributes, dividing the set into 10, 20, 50 and 500 partitions. In each case the partitioning was naive; the attribute set was placed in descending order of frequency, and divided into equal-sized ranges. This is, of course, unlikely to be optimal because, as we have explained, the size of trees produced tends to increase towards the higher end of the attribute-sequence, although this effect is reduced by the ordering of attributes. In some preliminary experiments, we found the scaling of the methods was close to linear as the database size increased, so for experimental convenience the remaining experiments were carried out with a database size D = 50,000, while continuing to require that all trees are stored on disk and re-read whenever required.

Table 3 shows the times required to compute the frequent sets (from the stored PP-trees), and Table 4 the maximum memory requirements (largest PP-tree + largest T-tree partition) for these experiments. For lower support thresholds, a higher degree of partitioning appears to offer better results, both in time and memory requirements. This is offset, however, by the greater cost of constructing the PP-trees; because each tree requires a pass of the database, the time for this scales linearly with the number of partitions. Although in some applications it might be reasonable to write off the PP-tree construction as an initial preprocessing cost, in general this cost will be a significant weakness of Method 3.

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*	*0.01*	*0.005*
50 items/partition	2.26	2.32	3.44	8.05	31.09	61.80
25 items/partition	2.80	2.97	3.79	7.56	28.73	53.36
10 items/partition	3.43	3.50	4.23	6.72	25.97	46.89
1 item/partition	4.41	4.53	4.66	5.71	15.56	33.58

Table 3: Method 3: Times to find Frequent sets for T10I5N500D50000

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*	*0.01*	*0.005*
50 items/partition	1.2	1.2	1.5	3.9	24.7	48.2
25 items/partition	0.9	0.9	1.0	2.5	12.9	24.6
10 items/partition	0.5	0.5	0.6	1.4	6.5	10.9
1 item/partition	0.1	0.1	0.1	0.7	1.4	2.1

Table 4: Method 3: Memory requirements for T10I5N500D50000

This drawback is overcome in Method 4. Table 5 shows the results obtained for this, using "vertical" partitioning as for Method 3, together with a horizontal segmentation of the data into 5 segments. Thus, for each segment of 10,000 records, PP-trees were constructed in primary memory in a single pass.

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*	*0.01*	*0.005*
50 items/partition	2.44	2.50	3.67	8.91	33.71	63.97
25 items/partition	3.13	3.19	4.10	8.26	30.64	55.96
10 items/partition	3.56	3.79	4.59	7.16	27.77	48.94
1 item/partition	4.85	5.20	5.55	6.48	16.11	32.96

Table 5: Method 4: Times to find Frequent sets for T10I5N500D50000

The times for Method 4 were very close to those for Method 3, and the memory requirements for the method, not tabulated here, were also similar. In these experiments we are dealing with a relatively small database, but for larger databases the segmentation used will ensure that the memory required while constructing the PP-trees will continue to be manageable.

The major advantage of Method 4 is the reduced time taken to build the PP-trees. Table 6 shows this; Method 4 is significantly less, especially when the partitioning is increased, requiring more data passes in Method 3. By comparison, the overall P-tree construction time for Methods 1 and 2, for this data, was 1.92 seconds.

Items/partition	*50*	*25*	*10*	*1*
Method 3	10.58	15.08	24.63	153.90
Method 4 (5 segs)	7.01	8.71	10.46	15.67

Table 6: Time to construct PP-trees for T10I5N500D50000

The final tabulation compares the results for all four methods on this dataset. Here we have shown the results for a vertical partitioning of 1 item/partition (Methods 3 and 4), and a horizontal partitioning into 5 segments (Methods 1, 2 and 4). As we can see, at the lower support thresholds Methods 3 and 4 strongly outperform Methods 1 and 2 in both execution time and primary memory requirement. Even when the time taken to construct the *P*-trees is taken into account, Method 4 is clearly the best for this data.

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*	*0.01*	*0.005*
Method 1	1.95	2.02	8.74	18.27	61.45	185.80
Method 2	0.70	0.78(Fail)	10.66(Fail)	25.04(Fail)	-	-
Method 3	4.41	4.53	4.66	5.71	15.56	33.58
Method 4	4.85	5.20	5.55	6.48	16.11	32.96

Table 7: Times to find Frequent sets for T10I5N500D50000

Support threshold	*1.0*	*0.5*	*0.1*	*0.05*	*0.01*	*0.005*
Method 1	0.4	0.4	1.8	16.9	127.4	274.9
Method 2	0.4	0.6	19.5	62.5	overflow	-
Method 3	0.1	0.1	0.1	0.7	1.4	2.1
Method 4	0.1	0.1	0.2	0.7	1.4	2.1

Table 8: Maximal memory requirements for T10I5N500D50000

7 Conclusions

We have in this paper described the results of a number of experiments carried out to investigate the performance of methods for extracting association rules from databases in cases where the data is too large to be contained in main memory, demanding some strategy for partitioning the data. In all cases, we have been examining methods that use an initial preprocessing of the database into a reordered form that includes a partial computation of support totals: the *P*-tree structure. In previous work we have shown this to offer significant performance advantages to Apriori-based algorithms, and our aim here was to examine how best to apply the approach to non-store-resident data. Two of the methods examined are established methods, modified slightly to make use of our data structures, and two others are new methods, using the tree structures to produce a "vertical" partitioning of the data.

The first method examined is, essentially, a straightforward adaptation of Apriori in this context, involving a simple partitioning of the data into segments. We have shown that the method scales well for increasing number of partitions, but, like the original Apriori, its performance drawback is the repeated passes of disk-resident data that it involves, especially when low support thresholds and/or high-density data is involved. The sampling method of [?] was developed specifically to avoid the cost of multiple database passes when data is non-store-resident. Our results confirm its effectiveness for relatively high support thresholds, but, at very low support thresholds, as candidate sets become very large, the additional memory requirement of the method becomes an increasing overhead, and in the extreme the method requires an additional database pass to find all the frequent sets.

The new methods we have introduced aim to reduce the memory requirement by a partitioning of the attribute set, and a corresponding construction of trees that each contain some subset of the candidates to be counted. Our results show these methods to be extremely effective in limiting the maximum primary memory requirement, even at very low support thresholds, because they enable both the original data (as represented by PP-trees) and the candidate set to be partitioned for memory management. For the most computationally demanding cases, at low support thresholds, a high degree of partitioning appears to work best, and in these cases these methods are significantly faster than the others we have considered. With a very high degree of partitioning, the increased cost of preprocessing the data to produce PP-trees may become a problem. We have shown, however, that this problem can be overcome by applying a horizontal segmentation of the data together with the vertical partitioning. In our experiments, this substantially reduced the preprocessing time with little effect on performance in generating the frequent sets. At low support thresholds, this method significantly outperforms all others in both time and space requirements. Work remains to be done to establish the optimal degree of partitioning and segmentation in different cases.

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., Imielinski, T. and Swami, A. Mining Association Rules between Sets of Items in Large Databases. In Proc. of the ACM SIGMOD Conference on Management of Data, Washington, D.C., pages 207-216, May 1993.
3. Agrawal, R. and Srikant, R. Fast Algorithm for Mining Association Rules. In Proc. of the 20th VLDB Conference, Santiago, Santiago, Chile, pages 487-499, September 1994.
4. Bayardo, R.J. Efficiently Mining Long Pattern from Databases. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 85-93, 1998.
5. 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.

6. 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.
7. Coenen, F., Goulbourne, G., and Leng, P. Computing Association Rules using Partial Totals. PKDD 2001, pages 54-66, 2001.
8. Coenen, F. and Leng, P. Optimising Association Rule Algorithms Using Itemset Ordering. Research and Development in Intelligent Systems XVIII: Proc ES2001 Conference, eds M Bramer, F Coenen and A Preece, Springer, pp53-66.
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. Toivonen, H. Sampling Large Databases for Association Rules. In Proc. of the 22th VLDB Conference, Mumbai, India, pages 1-12, 1996.
11. Brin, S., Motwani, R., Ullman, J. D. and Tsur, S. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In Proc. of the ACM SIGMOD Conference on Management of Data, USA, pages 255-264, 1997.
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. 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.