Computing Association Rules Using Partial Totals

Frans Coenen, Graham Goulbourne and Paul Leng

Department of Computer Science, The University of Liverpool,
Chadwick Building, P.O. Box 147, Liverpool L69 3BX, England.
{frans,graham_g,phl}@csc.liv.ac.uk

Abstract. The problem of extracting all association rules from within a binary database is well-known. Existing methods may involve multiple passes of the database, and cope badly with densely- packed database records because of the combinatorial explosion in the number of sets of attributes for which incidence-counts must be computed. We describe here a class of methods we have introduced that begin by using a single database pass to perform a *partial* computation of the totals required, storing these in the form of a set enumeration tree, which is created in time linear to the size of the database. Algorithms for using this structure to complete the count summations are discussed, and a method is described, derived from the well-known *Apriori* algorithm. Results are presented demonstrating the performance advantage to be gained from the use of this approach.
Keywords: Association Rules, Set Enumeration Tree, Data Structures.

1 Introduction

A well-established approach to Knowledge Discovery in Databases (KDD) involves the identification of *association rules* [**?**] within a database. An association rule is a probabilistic relationship, of the form $A \rightarrow B$, between sets of database attributes, which is inferred empirically from examination of records in the database. In the simplest case, the attributes are boolean, and the database takes the form of a set of records each of which reports the presence or absence of each of the attributes in that record. The paradigmatic example is in supermarket shopping-basket analysis. In this case, each record in the database is a representation of a single shopping transaction, recording the set of all items purchased in that transaction. The discovery of an association rule, $PQR \rightarrow XY$, for example, is equivalent to an assertion that "shoppers who purchase items P, Q and R are also likely to purchase items X and Y at the same time". This kind of relationship is potentially of considerable interest for marketing and planning purposes.

More generally, assume a set I of n boolean attributes, $\{a_1, \cdots, a_n\}$. and a database table each record of which contains some subset of these attributes, which may equivalently be recorded as a n-bit vector reporting the presence or absence of each attribute. An association rule R is of the form $A \rightarrow B$, where A, B are disjoint subsets of the attribute set I. The *support* for the rule R is the number of database records which contain $A \cup B$ (often expressed as a proportion of the total number of records). The *confidence* in the rule R is the ratio of the support for R to the support for its antecedent, A. A rule is described as "frequent" or "interesting", if it exceeds some defined levels of support and confidence. The fundamental problem in association rule mining is the search for sets which exceed the support threshold: once these frequent sets have been identified the confidence can be immediately computed.

In this paper we describe a class of methods for identifying frequent sets of attributes within a database. For the databases in which we are interested, the number of attributes is likely to be 500 or more, making examination of all subsets computationally infeasible. Our methods use a single pass of the database to perform a partial summation of support totals, with time and space requirements that are linear to the number of database records. The partial counts are stored in a set-enumeration tree structure (the *'P-tree'*) which facilitates efficient completion of the final totals required. We describe an algorithm for performing this computation, using a second tree structure (the *'T-tree'*) to store the support-counts. Results are presented which illustrate the performance gain achieved by this approach.

2 Background

The central problem in deriving association rules is the exponential time- and space-complexity of the task of computing support counts for all 2^n subsets of the attribute set I. Hence, practicable algorithms in general attempt to reduce the search space by computing support-counts only for those subsets which are identified as potentially interesting. The best-known algorithm, "Apriori" [**?**], does this by repeated passes of the database, successively computing support-counts for single attributes, pairs, triples, and so on. Since any set of attributes can be "interesting" only if all its subsets also reach the required support threshold, the *candidate set* of sets of attributes is pruned on each pass to eliminate those that do not satisfy this requirement. Other algorithms, AIS [**?**] and SETM [**?**], have the same general form but differ in the way the candidate sets are derived.

Two aspects of the performance of these algorithms are of concern: the number of passes of the database that are required, which will in general be one greater than the number of attributes in the largest interesting set, and the size of the candidate sets which may be generated, especially in the early cycles of the algorithm. The number of passes may be reduced to 2 by strategies which begin by examining subsets of the database [**?**], or by sampling the database to estimate the likely candidate set [**?**]. The drawback of these methods is that the candidate set derived is necessarily a superset of the actual set of interesting sets, so again the search space may become very large, especially with densely packed database records. Large candidate-set sizes create a problem both in their storage requirement and in the computation required as each database record is examined. The implementation described for the Apriori algorithm stores the candidate set in a hash-tree, which is searched for each database record in turn to identify candidates that are subsets of the set of attributes included in the record being considered.

The computation involved in dealing with large candidate sets has led researchers to look for methods which seek to identify *maximal* interesting sets without first examining all their smaller subsets. Zaki et al [**?**] do this by partitioning the search space into *clusters* of associated attributes; however, this approach breaks down if the database is too densely-populated for such clusters to be apparent. Bayardo's [**?**] Max-Miner algorithm also searches for maximal sets, using Rymon's set enumeration framework [**?**] to order the search space as a tree. Max-Miner reduces the search space by pruning the tree to eliminate both supersets of infrequent sets and subsets of frequent sets. In a development from Max-Miner, the Dense-Miner algorithm [**?**] imposes additional constraints on the rules being sought to reduce further the search space in these cases. These algorithms cope better with dense datasets than the other algorithms described, but again require multiple database passes. For databases which can be completely contained in main memory, the DepthProject algorithm of [**?**] also makes use of a set-enumeration structure. In this case the tree is used to store frequent sets that are generated in depth-first order via recursive projections of the database. However, because of the combinatorial explosion in the number of candidates which must be considered, and/or the cost of repeated access to the database, no existing algorithm copes fully with large databases of densely-packed records.

In the method we describe here, we also make use of Rymon's set enumeration tree, to store *interim* support- counts in a form that facilitates completion of the computation required. The approach is novel but generic in that it can be used as a basis for implementing improved variants of many existing algorithms.

3 Partial Support and The P-Tree

The most computationally expensive part of Apriori and related algorithms is the identification of subsets of a database record that are members of the candidate set being considered; this is especially so for records that include a large number of attributes. We avoid this, at least initially, by at first counting only sets occurring in the database, without considering subsets.

Let i be a subset of the set I (where I is the set of n attributes represented by the database). We define P_i, the *partial support* for the set i, to be the number of records whose contents are identical with the set i. Then T_i, the *total support* for the set i, can be determined as:

$$T_i = \sum P_j \qquad (\forall j, j \supseteq i)$$

For a database of m records, the partial supports can, of course, be counted simply in a single database pass, to produce m' partial totals, for some $m' \leq m$. We use Rymon's set enumeration framework [**?**] to store these counts in a tree; Figure 1 illustrates this for $I = \{A, B, C, D\}$. To avoid the potential exponential scale of this, the tree is built dynamically as the database is scanned so as to include only those nodes that represent sets actually present as records in the database, plus some additional nodes created to maintain tree structure when necessary. The size of this tree, and the cost of its construction are linearly related to m rather than 2^n.

Taking advantage of the structural relationships between sets of attributes apparent from the tree, we also use the construction phase to begin the computation of total supports. As each set is located within the tree during the course of the database pass, it is computationally inexpensive to augment *interim* support-counts, Q_i stored for subsets which precede it in the tree ordering; thus:

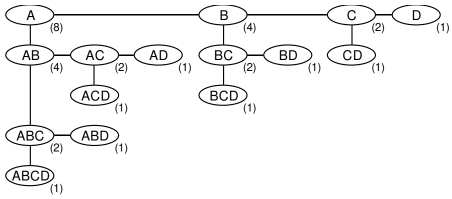

Fig 1: Tree storage of subsets of $\{A, B, C, D\}$

$$Q_i = \sum P_j \quad (\forall j, j \supseteq i,\ j\ follows\ i\ in\ lexicographic\ order)$$

It then becomes possible to compute total support using the equation:

$$T_i = Q_i + \sum P_j \quad (\forall j, j \supset i,\ j\ precedes\ i\ in\ lexicographic\ order)$$

The numbers associated with the nodes of Figure 1 are the interim counts which would be stored in the tree arising from a database the records of which comprise exactly one instance of each of the 16 possible sets of attributes; thus, for example, $Q(BC) = 2$, derived from one instance of BC and one of BCD. Then:

$$T(BC) = Q(BC) + P(ABC) + P(ABCD) = Q(BC) + Q(ABC)$$

We use the term *P-tree* to refer to this incomplete set- enumeration tree of interim support-counts. An algorithm for building the P-tree, counting the interim totals, is described in detail in [**?**]. Because the P-tree contains all the relevant data stored in the original database, albeit in a different form, we can in principle apply versions of almost any existing algorithm to complete the summation of total supports. Use of the P-tree as a surrogate for the original database, however, offers three potential advantages. Firstly, when n is small ($2^n \ll m$), then traversing the tree to examine each node will be significantly faster than scanning the whole database. Secondly, even for large n, if the database contains a high degree of duplication ($m' \ll m$) then using the tree will again be significantly faster than a full database pass, especially if the duplicated records are densely-populated with attributes. Finally, and most generally, the computation required in each cycle of the algorithm is greatly reduced because of the partial summation already carried out in constructing the tree. For example, in the second pass of Apriori (considering pairs of attributes), a record containing r attributes may require the counts for each of its $r(r-1)/2$ subset-pairs to be incremented. When examining a node of the P-tree, conversely, it is necessary only to consider only those subsets not already covered by a parent node, which in the best case will be only $r-1$ subsets.

To illustrate this, consider the node $ABCD$ in the tree of Figure 1. The partial total for $ABCD$ has already been included in the interim total for ABC, and this will be added to the final totals for the subsets of ABC when the latter node is examined. Thus, when examining the node $ABCD$, we need only consider those subsets not covered by its parent, i.e. those including the attribute D. The advantage gained from this will be greater, of course, the greater the number of attributes in the set being considered.

A rather similar structure to our *P-tree* has been described independently by [**?**]. This structure, the *FP-tree*, has a different form but quite similar properties to the P-tree, but is built in two database passes, the first of which eliminates attributes that fail to reach the support threshold, and orders the others by frequency of occurrence. Each node in the FP-tree stores a single attribute, so that each path in the tree represents and counts one or more records in the database. The FP-tree also includes more structural information, including all the nodes representing any one attribute being linked into a list. This structure facilitates the implementation of an algorithm, "FP- growth", which successively generates subtrees from the FP-tree corresponding to each frequent attribute, to represent all sets in which the attribute is associated with its predecessors in the tree ordering. Recursive application of the algorithm generates all frequent sets. The two structures, the FP-tree and our P-tree, which have been developed independently and contemporaneously, are sufficiently similar to merit a detailed comparison, which we discuss in section 5.

4 Computing Total Supports

The construction of the P-tree has essentially performed, in a single pass, a reorganisation of the relevant data into a structured set of counts of sets of attributes which appear as distinct records in the database. For any candidate set T of subsets of I, the calculation of total supports can be completed by walking this tree, adding interim supports as required according to the formulae above.

We can also take advantage of the structure of the P-tree to organise the computation of total supports efficiently, taking advantage of the fact that the counts for each set in the P-tree already incorporate contributions from their successor-supersets. Figure 2 illustrates the dual of Figure 1, in which each subtree includes only supersets of its root node which contain an attribute that precedes all those of the root node. We will call this the T-tree, representing the target sets for which the total support is to be calculated, as opposed to the interim-support P-tree of Figure 1. Observe that for any node t in the T-tree, all the subsets of t which include an attribute i will be located in that segment of the tree found between node i and node t in the tree ordering. This allows us to use the T- tree as a structure to effect an implementation of an algorithm to sum total supports:

```
Algorithm TFP (Compute Total- from Partial- supports)
        for each node j in P-tree do
        begin k = j - parent (j);
            i = first attribute in k;
            starting at node i of T-tree do
            begin if i ⊆ j then add Q_j to T_i;
                        if i = j then exit
                        else recurse to child node;
                proceed to sibling node;
            end
        end
```

To illustrate the application of the algorithm, consider the node ACD in the tree of Figure 1. TFP first obtains the difference of this node from its parent, AC, i.e. D, and begins traversing the T-tree at node D. From this point the count associated with ACD will be added to all nodes encountered that are subsets of ACD, i.e. D, AD, CD and ACD, the traversal terminating when the node ACD is reached. Note that the count for the node BD which is not a subset of ACD will not be updated, nor will its subtree be traversed.

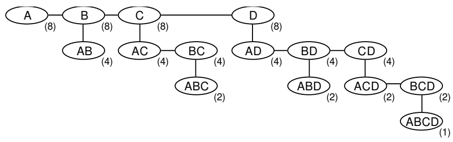

Fig 2. Tree with predecessor-subtrees

Of course, to construct the entire T-tree would imply an exponential storage requirement. In any practical method, however, it is only necessary to create that subset of the tree corresponding to the current candidate set being considered. Thus, for example, a version of the Apriori algorithm using these structures would consider candidates which are singletons, pairs of attributes, triples, etc., in successive passes. This algorithm, which we will call Apriori- TFP, has the following form:

1. Build level K in the T-tree.
2. "Walk" the P-tree, applying algorithm TFP to add 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 algorithm begins by constructing the top level of the T-tree, containing all the singleton subsets, i.e. the single attributes in I. A first pass of algorithm TFP then counts supports for each of these in a single traversal of the P-tree. Note again that identification of the relevant nodes in the T-tree is trivial and efficient, as these will be located in a (usually short) segment of the level-1 list. In practice, it is more efficient to implement level 1 of the T-tree as a simple array of attribute-counts, which can be processed more quickly than is the case for a list structure. A similar optimisation can be carried into level 2, replacing each branch of the tree by an array, and again this is likely to be more efficient when most of the level 2 nodes remain in the tree.

Following completion of the first pass, the level 1 T-tree is pruned to remove all nodes that fail to reach the required support threshold, and the second level is generated, adding new nodes only if their subsets are contained in the tree built so far, i.e. have been found to have the necessary threshold of support. The new level of the tree forms the candidate set for the next pass of the algorithm TFP. The complete algorithm is described formally in Table 1 (Part 1) and Table 2 (Part 2). This uses a function, $endDigits$, that takes two arguments P and N (where N is the current level) and returns a set comprising the last N attributes in the set P; thus $endDigits(ABC, 2) = BC$. The significance of this is that BC is the last subset of ABC at level 2 that need be considered.

$$
\begin{array}{l}
\forall P \in Ptree\ \ where\ \ (numAttributes(P) \geq requiredLevel) \\
\quad P' = P \setminus P_{parent} \\
\forall T_{1,j}\ \ (nodes\, at\, level\, 1)
\end{array}
$$

$loop\ while\ P' \neq null$
  $if\ T_{1,j} < P'\ \ j++$
  $if\ T_{1,j} \equiv P'$
    $if\ (requiredLevel \equiv 1)\ \ T_{sup} = T_{sup} + P_{sup}$
    $else$ **Part 2**
    $P' = null$
  $if\ \left(T_{1,j} \subset P'\right)$
    $if\ (requiredLevel \equiv 1)\ \ T_{sup} = T_{sup} + P_{sup}$
    $else$ **Part 2**
    $P' = P' \setminus firstAttribute(P') \cdot j++$

Table 1: Total Support Algorithm (Part 1)

$P'' = \ endDigits(P, currentLevel)$
$loop\ while\ T_{i,j} \neq null$
  $if\ T_{i,j} < P''$
    $if(T_{i,j} \subset P)$
      $if\ currentLevel \equiv requiredLevel\ \ T_{sup} = T_{sup} + P_{sup}$
      $else\ recursivelycall\ Part\ 2\ commencing\ with\ T_{i++,1}$
    $j++$
  $if\ T_{i,j} \equiv P''$
      $if\ currentLevel \equiv requiredLevel\ \ T_{sup} = T_{sup} + P_{sup}$
      $else\ recursivelycall\ Part\ 2\ commencing\ with\ T_{i++,1}$
    $stop$
  $if\ T_{i,j} > P''$
    $stop$

Table 2: Total Support Algorithm (Part 2)

5 Results

To evaluate the algorithms we have described, we have compared their performance with that for our implementations of two published methods: the original Apriori algorithm (founded on a hash tree data structure), and the FP- growth algorithm described in [**?**]. In both cases, the comparisons are based on our own implementations of the algorithms, which follow as closely as we can judge the published descriptions. All the implementations, including those for our own algorithms, are experimental prototypes, unoptimised low-performance Java programs.

The first set of experiments illustrate the performance characteristics involved in the creation of the P-tree. Figure 3 shows the time to build the P-tree, for databases of 200,000 records with varying characteristics. The graphs of storage requirements also have exactly the same pattern. The three cases illustrated represent synthetic databases constructed using the QUEST generator described in [**?**]. This uses parameters T, which defines the average number of attributes found in a record, and I, the average size of the maximal supported set. Higher values of T and I in relation to the number of attributes N correspond to a more densely-populated database. These results show that the cost of building the P-tree is almost independent of N, the number of attributes. As is the case for all association-rule algorithms, the cost of the P-tree is greater for more densely-populated data, but in this case the scaling appears to be linear.

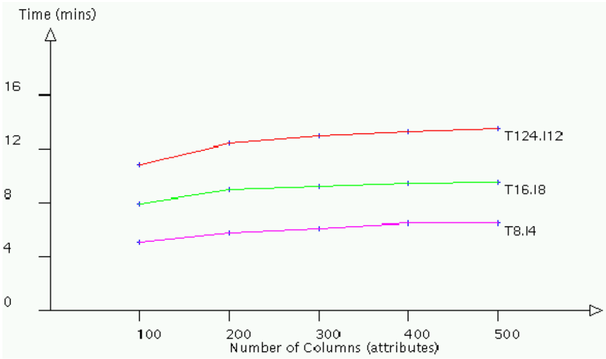

Fig 3: Graph showing effort (time) to generate P-tree for data sets with number of rows fixed at 200000

Figure 4 examines the P-tree storage requirement for databases of 500 attributes, with the same sets of parameters, as the number of database records is increased. This shows, as predicted, that the size of the tree is linearly related to the database size. Again, this is also the case for the construction time. The actual performance figures for the P- tree construction could easily be improved from a more efficient implementation, and it would also be possible and probably worthwhile to use this first pass to compute total support counts for the single attributes. However, the construction of the P-tree is essentially a restructuring of the database, the effect of which will be realised in all subsequent data mining experiments.

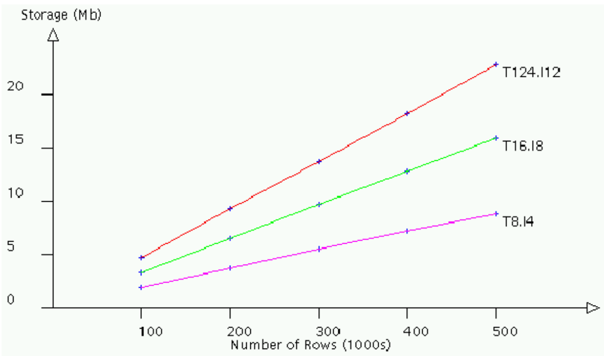

Fig 4: Graph showing P-tree storage requirements for data sets with number of columns fixed at 500

In Table 3 we examine the cost of building the P-tree in comparison with that for the FP-tree of [**?**]. The figures tabulated are for two different datasets:

1. **quest.T25.I10.N1K.D10K**: A synthetic data set, also used in [**?**], generated using the Quest generator (N=1000 attributes, D=10000 records).
2. **fleet.N194.D9000**: A genuine data set, not in the public domain, provided by a UK insurance company. Note that this set is much denser than quest.T25.I10.N1K.D10K

	quest.T25.I10.N1K.D10K		*fleet.N194.D9000*	
	Storage (Bytes)	*Time (Mins)*	*Storage (Bytes)*	*Time (Mins)*
P-tree	1,020,690	0.65	582,196	0.36
FP-tree (Sup 5%)	1,566,838	3.43	767,062	1.24
FP-tree (Sup 4%)	2,283,360	5.53	912,918	1.39
FP-tree (Sup 3%)	3,028,082	9.36	1,334,146	2.04
FP-tree (Sup 2%)	3,974,482	20.00	1,704,990	3.28
FP-tree (Sup 1%)	4,567,480	34.83	1,754,990	3.35

Table 3: *P-tree and FP-tree generation characteristics*

With respect to Table 3 it should be noted that the procedure for building the FP-tree eliminates all single attributes that fail to reach the support threshold, so figures for a range of support thresholds are tabulated against the (constant) characteristics of the P-tree. As can be seen, the P-tree is a significantly more compact structure, and its construction time lower than that of the FP-tree. The greater size of the FP-tree arises from the greater number of nodes it creates, and the additional links required by the FP-growth algorithm. The FP-tree stores each attribute of a record as a separate node, so that, for example, two records $ABCDE$ and $ABCXY$, with a common prefix ABC, would require in all 7 nodes. The P-tree, conversely, would create only 3 nodes: a parent ABC, and child nodes DE and XY. Each node in the FP-tree also requires two additional links not included in the P-tree. One, the “node-link”, connects all nodes representing the same attribute, and the other, which links a node to its parent, appears to be necessary to effect an implementation of FP-growth. The greater construction time for the FP-tree is unsurprising, given its more complex structure and that it requires two passes of the source data. In these trials, this data is main-memory resident: in the case of a dataset too large for this to be possible, the cost of the additional pass would of course be much greater.

Finally, to evaluate the performance of the method for computing final support-counts, we have compared the Apriori-TFP algorithm we have described with our implementations of the original Apriori (founded on a hash tree data structure) and of FP-growth. The results are presented in Figure 5, for quest.T25.I10.N1K.D10K and Figure 6, for fleet.N194.D9000, In all cases, to give the fairest basis for comparison, we have used data which is main-memory resident throughout. The performance time presented with respect to Apriori-TFP and FP-growth do not include the time to produce the P-tree or FP-tree respectively.

As we would expect, Apriori-TFP strongly outperforms our implementation of Apriori. This improvement arises from a combination of two factors. The first, as described above, is the lower number of support-count updates that will be required when examining a P-tree node, as opposed to the number required in Apriori from examination of the records from which the node is made up. This gain will be greatest when there are clusters of records including significant numbers of shared attributes (as we might hope to find when mining potentially interesting data), and, especially, if there are significant numbers of duplicated records. Secondly, the effect is compounded by the more efficient localisation of candidates obtained by using the T-tree for the TFP algorithm, as opposed to the hash-tree used by Apriori. The cost of accessing the hash-tree to locate candidates for updating increases rapidly as the candidate set increases in size, as is the case for lower support thresholds, and is greatest when examining a record which includes many attributes and hence many potential candidates for updating.

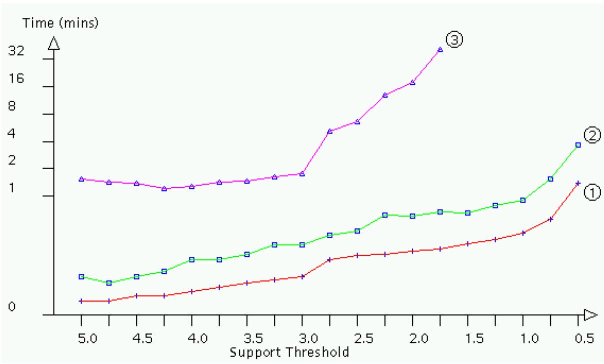

Fig 5: Graph showing processing time to mine (1) the P-tree, (2) the FP-tree and (3) to perform the same operation using a traditional Apriori algorithm using quest.T25.I10.N1K.D10K

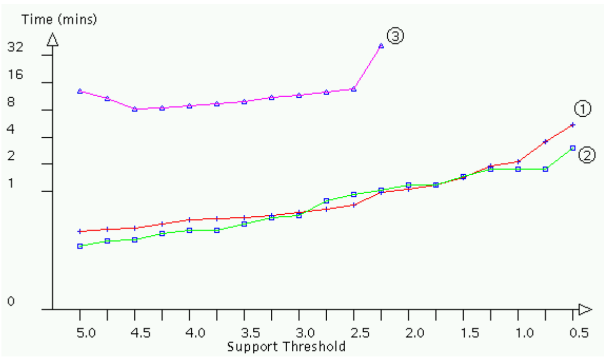

Fig 6: Graph showing processing time to mine (1) the P-tree, (2) the FP-tree and (3) to perform the same operation using a traditional Apriori algorithm using fleet.N194.D9000

Apriori-TFP also outperforms our implementation of FP- growth using quest.T25. I10.N1K.D10K although the difference here is much less. We believe that the performance gain here is a consequence of the cost of the recursive construction of successive conditional FP-trees, which, in our straightforward implemention, is much slower than the simple iterative building of the T-tree. In the case of the fleet.N194.D9000 data set similar performance times for both Apriori-TFP and FP growth are recorded with one outperforming the other on some occasions and vice versa. However, if the P-tree/FP-tree generation times are included Apriori- TFP clearly outperforms FP-growth.

Althoughit is possible that some of the advantage is an artefact of our implementations, the results appear to show that the simpler P-tree structure offers at least as good performance as the more complex FP-tree. Moreover, the above experiments use memory-resident data only; we believe that the additional structural links in the FP-tree, and the need for repeated access to generate subtrees, will create problems for efficient implementation in cases for which the tree is too large to hold in main memory. For the simpler P-tree structure, conversely, it is easy to describe an efficient construction process which will build separate trees for manageable segments of the database, prior to a final merging into a single tree. Nor is it necessary, in general, for the P-tree to be retained in main memory throughout the calculation of final support totals. The only structural information necessarily retained is the relationship of a node to its parent. For example, if a node representing the set $ABDFG$ is present in the tree as a child of the node ABD, all the relevant information can be recorded by a node representation of the form $ABD.FG$. In this form, the "tree" can in fact be stored finally as a simple array in any convenient order, depending on the needs of the algorithm to compute the final support totals. In the case of the Apriori-TFP algorithm, the tree/array is processed element-by-element in any order, causing no problems even when it is necessary to hold it in secondary memory.

6 Conclusions

We have presented here an algorithm for computing support counts using as a starting point an initial, incomplete computation stored as a set- enumeration tree. Although the actual algorithm described here to compute the final totals is based on the Apriori algorithm, the method itself is generic, in that, once the P-tree has been created, a variety of methods may be applied to complete the summation. Many of these methods, like the one we have illustrated, will be able to take advantage of the partial computation already carried out in the initial database pass to reduce the cost of further multiple passes.

Note, however, that the advantage gained from this partial computation is not equally distributed throughout the set of candidates. For candidates early in the lexicographic order, most of the support calculation is completed during the construction of the P-tree; for example, for the attributes of Figure 1, support for the sets A, AB, ABC and $ABCD$ will be counted totally in this first stage of the summation. This observation allows us to consider methods which maximise the benefit from this by a suitable ordering of the attribute set. This is, of course, the heuristic used by [**?**], and also, in various ways, by [**?**], [**?**] and [**?**].

We could also increase the proportion of the summation which is completed during the initial scan of the database by a partitioning of the P-tree. For example, it would be possible to separate the tree of Figure 1 into four subtrees, rooted at the nodes A, B, C and D, and for the first pass to accumulate interim supports within each of these subtrees independently. In this case, a record containing the set ABD, for example, would increment the support-counts for ABD within the A-tree, BD within the B-tree, and D within the (single-node) D-tree. Again, the effect of this is similar to that for a set of conditional FP-trees produced by FP-growth. The advantage offered is that is provides a means of reducing the size of trees required for processing. The size of the complete subtree corresponding to an attribute a_i that is in position i in the tree ordering is 2^{n-i}. However, the P-tree construction method we use will produce an incomplete subtree, the size of which will be of order m', where $m' \leq T_{a_i}$, the number of records in the database which contain a_i (again, reduced by the existence of duplicates). Thus, the storage requirement for each subtree is less than or equal to min $\{2^{n-i}, T_{a_i}\}$. The requirement for any single subtree can be minimised by ordering the attributes in reverse order of their frequency, so that the most common attributes are clustered at the high-order end of the tree structure.

Partitioning the tree in this way would allow us (in one pass) to organise the data into sets each of which can be processed independently and may be small enough to be retained in central memory. At the high-order end of the organisation, i.e. for values of i close to n, the 2^{n-i} limit becomes computable. Thus, for large i, it may be more efficient to store partial supports in a complete array of subset-counts, and to use an exhaustive algorithm to compute total supports efficiently. Conversely, for smaller i, the conservative P-tree storage method, and an algorithm such as Apriori-TFP can be applied. We are presently investigating this and other heuristics to produce effective hybrid algorithms of this kind.

References

1. Agarwal, R., Aggarwal, C. and Prasad, V. Depth First Generation of Long Patterns. Proc ACM KDD 2000 Conference, Boston, 108-118, 2000.
2. Agrawal, R. Imielinski, T. Swami, A. Mining Association Rules Between Sets of Items in Large Databases. SIGMOD-93, 207-216. May 1993.
3. Agrawal, R. and Srikant, R. Fast Algorithms for Mining Association Rules. Proc 20th VLDB Conference, Santiago, 487-499. 1994
4. Bayardo, R.J. Efficiently Mining Long Patterns from Databases. Proc ACM-SIGMOD Int Conf on Management of Data, 85-93, 1998
5. Bayardo, R.J., Agrawal, R. and Gunopolos, D. Constraint-based rule mining in large, dense databases. Proc 15th Int Conf on Data Engineering, 1999
6. Brin, S., Motwani. R., Ullman, J.D. and Tsur, S. Dynamic itemset counting and implication rules for market basket data. Proc ACM SIGMOD Conference, 255-256, 1997
7. Goulbourne, G., Coenen, F. and Leng, P. Algorithms for Computing Association Rules using a Partial-Support Tree. J. Knowledge-Based Systems 13 (2000), 141-149. (also Proc ES'99.)
8. Han, J., Pei, J. and Yin, Y. Mining Frequent Patterns without Candidate Generation. Proc ACM SIGMOD 2000 Conference, 1-12, 2000.
9. Houtsma, M. and Swami, A. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Centre, San Jose, October 1993.
10. Rymon, R. Search Through Systematic Set Enumeration. Proc. 3rd Int'l Conf. on Principles of Knowledge Representation and Reasoning, 1992, 539-550.
11. Savasere, A., Omiecinski, E. and Navathe, S. An efficient algorithm for mining association rules in large databases. Proc 21st VLDB Conference, Zurich, 432-444. 1995.
12. Toivonen, H. Sampling large databases for association rules. Proc 22nd VLDB Conference, 134-145. Bombay, 1996.
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.