Data Structure for Association Rule Mining: T-Trees and P-Trees

Frans Coenen, Paul Leng, and Shakil Ahmed

Abstract—Two new structures for Association Rule Mining (ARM), the T-tree, and the P-tree, together with associated algorithms, are described. The authors demonstrate that the structures and algorithms offer significant advantages in terms of storage and execution time.

Index Terms—Association Rule Mining, T-tree, P-tree.

1 Introduction

Association Rule Mining (ARM) obtains, from a binary valued data set, a set of rules which indicate that the *consequent* of a rule is likely to apply if the *antecedent* applies [1]. To generate such rules, the first step is to determine the *support* for sets of items (I) that may be present in the data set, i.e., the frequency with which each combination of items occurs. After eliminating those I for which the support fails to meet a given *minimum support threshold*, the remaining *large* I can be used to produce ARs of the form $A \Rightarrow B$, where A and B are disjoint subsets of a large I. The ARs generated are usually pruned according to some notion of confidence in each AR. However this pruning is achieved, it is always necessary to first identify the "large" I contained in the input data. This in turn requires an effective storage structure.

In this paper, an efficient data storage mechanism for itemset storage, the *T-tree*, is described. The paper also considers data preprocessing and describes the *P-tree*, which is used to perform a partial computation of support totals. The paper then goes on to show that use of these structures offers significant advantages with respect to existing ARM techniques.

2 The Total Support Tree (T-Tree)

The most significant overhead when considering ARM data structures is that the number of possible combinations represented by the items (columns) in the input data scales exponentially with the size of the record. A partial solution is to store only those combinations that actually appear in the data set. A further mechanism is to make use of the *downward closure* property of itemsets—"if any given itemset I is not large, any superset of I will also not be large." This can be used effectively to avoid the need to generate and compute support for all combinations in the input data. However, the approach requires: 1) a number of passes of the data set and 2) the construction of candidate sets to be counted in the next pass.

The most well-known ARM algorithm that makes use of the downward closure property is Agrawal and Srikant's Apriori algorithm [1]. Agrawal and Srikant used a *hash tree* data structure, however, Apriori can equally well be implemented using alternative structures such as set enumeration trees [6]. Set enumeration trees impose an ordering on items and then enumerate the itemsets according to this ordering. If we consider a data set comprised of just three records with combinations of six items: $\{1,3,4\}$, $\{2,4,5\}$, and $\{2,4,6\}$ (and a very low support threshold), then the tree would include one node for each large I (with its support count). The top level of the tree records the support for 1-itemsets, the second level for 2-itemsets, and so on.

The implementation of this structure can be optimized by storing levels in the tree in the form of arrays, thus reducing the number of links needed and providing direct indexing. For the latter purpose, it is more convenient to build a "reverse" version of the tree, as shown in Fig. 1a. The authors refer to this form of *compressed set enumeration tree* as a T-tree (Total support tree). The implementation of this structure is illustrated in Fig. 1b, where each node in the T-tree is an object ($TtreeNode$) comprised of a support value (sup) and a reference ($chldRef$) to an array of child T-tree nodes. The Apriori T-tree generation algorithm is presented in Fig. 2, where $start$ is a reference to the start of the top-level array, $\Re$ is the input data set, N the number of attributes (columns), D the number of records and K a level in the T-tree (the Boolean variable $isNewLevel$ is a field in the class initialized to the value $false$). The method $TtreeNode()$ is a constructor to build a new $TtreeNode$ object.

3 The Partial Support Tree (P-Tree)

A disadvantage of Apriori is that the same records are repeatedly reexamined. In this section, we introduce the concept of *partial support* counting using the "P-tree" (Partial support tree). The idea is to copy the input data (in one pass) into a data structure, which maintains all the relevant aspects of the input, and then mine this structure. In this respect, the P-tree offers two advantages: 1) It merges duplicated records and records with common leading substrings, thus reducing the storage and processing requirements for these and 2) it allows *partial counts* of the support for individual nodes within the tree to be accumulated effectively as the tree is constructed.

The overall structure of the P-tree is that of a *compressed* set-enumeration tree. The top level is comprised of an array of nodes (instances of the class $PtNodeTop$), each index describing a 1-itemset, with child references to body P-tree nodes (instances of the class $PtNode$). $PtNodeTop$ instances are comprised of: 1) a field (sup) for the support value and 2) a link ($chdRef$) to a $PtNode$ object. Instances of the $PtNode$ class have: 1) a support field (sup), 2) an array of short integers (I) for the itemset that the node represents and 3) child and sibling links ($chdRef$ and $sibRef$) to further P-tree nodes.

To construct a P-tree, we pass through the input data record by record. When complete, the P-tree will contain all the itemsets present as distinct records in the input data. The sup stored at each node is an incomplete support total, comprised of the sum of the supports stored in the subtree of the node. Because of the way the tree is ordered, for each node in the tree, the contribution to the support count for that set which derives from all its lexicographically succeeding supersets has been included.

The complete algorithm is given in Fig. 3, where $\Re$ is the input data set, N the number columns/attributes, D the number of

• *The authors are with the Department of Computer Science, University of Liverpool, Liverpool, L69 3BX. E-mail: {frans, phl, shakil}@csc.liv.ac.uk.*

Manuscript received 10 June 2003; revised 21 Oct. 2003; accepted 28 Jan. 2004.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0091-0603.

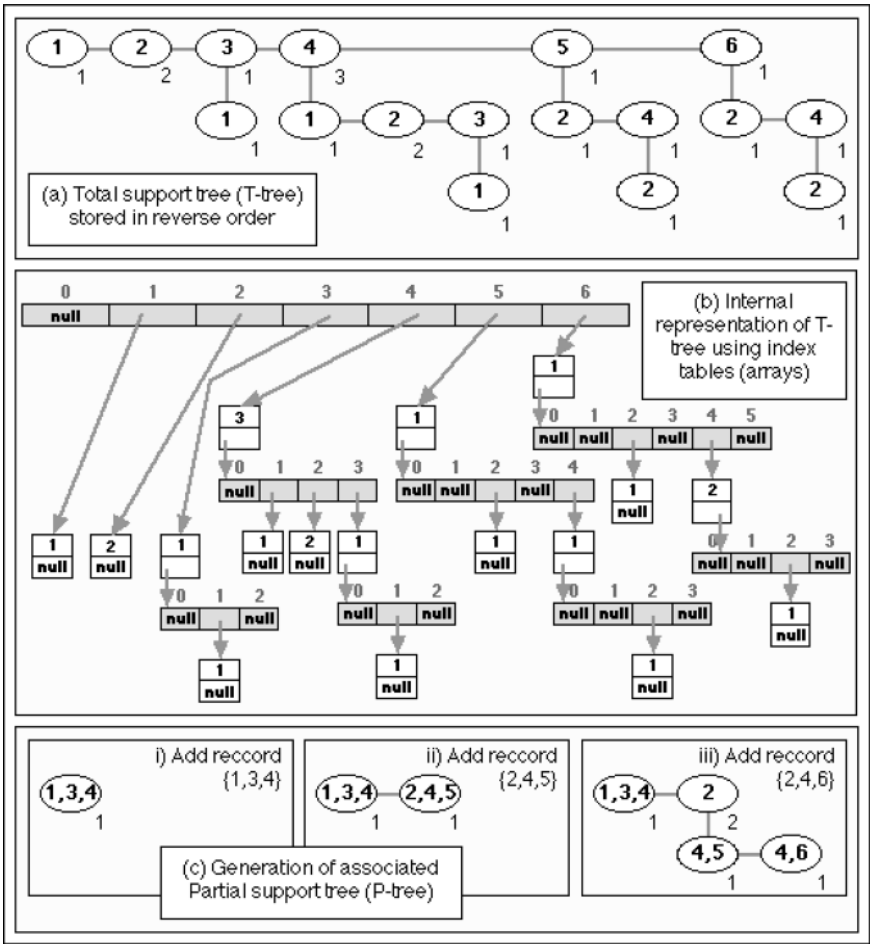

Fig. 1. The T-tree (Total support tree). Note that, for both clarity and ease of processing, items/attributes are enumerated commencing with 1.

records, ref a reference to the current node in the P-tree, $start$ a reference to the P-tree top-level array, $ref.I$ a reference to an itemset represented by a Ptree node, I_{lss} a leading substring of some item set I, and f a flag set according to whether a new node should be inserted at the top level ($f = 0$), as a child ($f = 1$), or sibling ($f = 2$). The $<$ and $>$ operators should be interpreted as *lexicographically* before and after. The method $del1(I)$ returns I with its first element removed. The method $delN(I_1, I_2)$ returns I_1 with the leading substring I_2 removed. The methods $PtNodeTop$ and $PtNode$ are constructors, the latter with two arguments—the node label and the support. As nodes are inserted into the P-tree to maintain the overall organisation of the tree, it may be necessary to: 1) create a "dummy" node representing a common leading substring and/or 2) "move up" siblings from the current node to become siblings of a new node.

An example of the construction of the P-tree, using the same data presented in Section 2, is given in Fig. 1c. Note that, on completion, the tree includes the full count for itemset $\{2\}$ and partial counts for the itemsets $\{1,3,4\}$, $\{4,5\}$, and $\{4,6\}$. Note also that, for reasons of computational effectiveness, the P-tree is in fact initialized with the complete set of one item sets expressed as an array (see above and Fig. 3).

4 Apriori-TFP

We can generate a T-tree from a P-tree in a similar Apriori manner to that described in Section 2. The algorithm for this (almost identical to that given in Fig. 2) is referred to as the Apriori-TFP (Total-from-Partial) algorithm. Note that the structure of the P-tree is such that, to obtain the complete support for any I, we need only add to the P-tree partial support for I the partial supports for those supersets of I that are lexicographically before it. Thus, for each pass of the T-tree, for each P-tree node P, we update only those level k T-tree nodes that are in P but not in the parent node of P.

An alternative "preprocessing" compressed set enumeration tree structure to the P-tree described here is the FP-tree proposed by Han et al. [4]. The FP-tree has a similar organization to the T-tree/P-tree, but stores only a single item at each node, and includes additional links to facilitate processing. These links start from a *header table* and link together all nodes in the FP-tree which store the same "label", i.e., item identifier.

5 Experimental Results

In this section, some of the experimental results obtained using the QUEST generator [1] are presented. Note that all ARM algorithms considered have been implemented, in Java `j2sdk 1.4.0`, to the

```
createTtree() {
   createTtreeTopLevel();
   prune(start, 1);
   genLevelN(start, 1, 1, null);
   K = 2;
   while (isNewLevel) {
      addSupport(K);
      prune(start, K);
      isNewLevel = false;
      genLevelN(start, 1, K, {});
      K++;
      }
   }
createTtreeTopLevel() {
   ∀s_i ∈ I • 0 < i ≤ N
      start[i] = new TtreeNode();
   ∀r_i ∈ ℜ • 0 ≤ i ≤ D
      ∀s_j ∈ r_i • 0 ≤ j ≤ |r_i|
         start[s_j].sup++;
   }
prune(ref, K) {
   if (K ≡ 1) ∀t ∈ T • T = {t|0 < t ≤ |ref|}
      if (ref[t] ≠ null &
            ref[t].sup < minSup)
         ref[t] = null;
   else ∀t ∈ T • T = {t|0 < t ≤ |ref|}
      if (ref[t] ≠ null &
            ref[t].chdRef ≠ null)
         prune(ref[t].chdRef, K − 1);
   }
addSupport(K) {
   ∀r_i ∈ ℜ • 0 ≤ i ≤ M
      addSup(start, K, |r_i|, r_i);
   }
addSup(ref, K, end, r) {
   if (K == 1)
      ∀s_i ∈ r • 0 ≤ i < end
         if (ref[s_i] ≠ null) ref[s_i].sup++;
   else
      ∀s_i ∈ r • 0 ≤ i < end
         if (ref[s_i] ≠ null)
            addSup(ref[s_i].chdRef, K − 1,
                  i, r);
   }
genLevelN(ref, K, newK, I) {
   if (K ≡ newK)
      ∀i ∈ M • M = {m|2 ≤ m < |ref|}
         if (ref[i] ≠ null) genLevel(ref,
                  i, append({i}, I));
   else
      ∀i ∈ M • M = {m|2 ≤ m < |ref|}
         if (ref[i] ≠ null)
            genLevelN(ref[i].chdRef,
                  K + 1, newK, append({i}, I));
   }

genLevel(ref, end, I) {
   // Create new array of T-tree nodes
   ref[end].chdRef = new TtreeNode[end];
   // Initialise elements where appropriate
   ∀i ∈ M • M = {m|1 ≤ m < end}
      if (ref[i] ≠ null)
         newI = append({i}, I);
         if (testCombinations(newI))
            ref[end].chdRef[i] =
                  new TtreeNode();
            isNewLevel = true;
         else ref[end].chdRef[i] = null;
   }

testCombinations(I) {
   if (|I| < 3) return(true);
   I_1 = {I[1], I[0]}
   I_2 = delN(I, 2);
   return(combinations(null, 0, 2, I_1, I_2));
   }

combinations(I, start, end, I_1, I_2) {
   if (end > |I_2|)
      testSet = append(I, I_1);
      return(findInTtree(testSet));
   else
      ∀i ∈ M • M = {m|start ≤ m < end}
         tempSet = append(I_2[i], I);
         if (¬combinations(tempSet,
                  i + 1, end + 1, I_1, I_2))
            return(false);
   return(true);
   }
```

Fig. 2. Basic Apriori T-tree algorithm.

best ability of the authors according to published information. The evaluation has been carried out on machines using RedHat Linux 7.1 OS; and fitted with AMD K6-2 CPUs running at 300MHz, with 64Kb of cache and 512 Mb of RAM. A sequence of graphs describing the key points of this evaluation are presented in Fig. 4.

Figs. 4a and 4b show a comparison of Apriori using Hash trees and T-trees with $T20I10D250kN500$ (chosen because it is representative of the data sets used by other researchers using the QUEST generator) and a range of support thresholds. The plots demonstrate that the T-tree significantly outperforms the hash-tree approach in terms of both storage and generation time. This is due to the indexing mechanism used by the T-tree and its reduced "housekeeping" overheads.

Figs. 4c and 4d compare the generation of P-trees with FP-trees using the data set used in the plots from Figs. 4a and 4b but varying the range of values for D. The plots show that the P-tree generation time and storage requirements are significantly less than for the FP-tree, largely because of the extra links included during FP-tree generation.

Fig. 4e shows a comparison between Apriori using a Hash tree (A-HT), Apriori using a T-tree (A-T), Apriori-TFP (A-TFP), Eclat (E) and Clique (C) [7], DIC [3], and FP-growth (FP-tree), with respect to execution time using the input set $T10I5D250kN500$. This data set was chosen partly because it is representative of the data sets used by other researchers, but also because it is a relatively sparse data set (density 2 percent). From the plots, it can be seen that Clique and Eclat perform badly because of the size of the vertical data arrays that must be processed. FP-growth also does not perform particularly well, while Apriori-TFP, Apriori-T, and DIC do better.

```
createPtree() {
   ∀r_i ∈ ℜ • 0 ≤ i ≤ M
      addToPtreeTopLevel(r_i);
   }

addToPtreeTopLevel(r) {
   if (start[r_0] ≡ null)
      start[r_0] = new PtNodeTop();
   else start[r_0].sup++;
   if (|r_i| > 1) addToPtree(0,
         start[r_0].chdRef, del1(r_i), null);
   }

addToPtree(f, ref, r, oldRef) {
   if (ref ≡ null)
      newRef = new PtNode(r, 1);
      Link newRef to oldRef as per f
   else
      if (r ≡ ref.I) ref.sup++;
      elseif (r < & ⊂ ref.I) (parent)
         parent(f, ref, r, oldRef);
      elseif (r < & ⊄ ref.I) (eld. sib.)
         eldSib(f, ref, r, oldRef);
      elseif (r > & ⊃ ref.I) (child)
         child(ref, r);
      else (r > & ⊅ ref.I) (young. sib.)
         yngSib(f, ref, r, oldRef);
   }

parent(f, ref, r, oldRef) {
   newRef = new PtNode(r, ref.sup + 1);
   newRef.chdRef = ref;
   Link newRef to oldRef as per f
   ref.I = delN(ref.I, r);
   Move up any siblings of ref to be
   sibings of newRef if necessary
   }

child(ref, r) {
   ref.sup = ref.sup++;
   if (ref.chdRef ≡ null)
      newRef = new PtNode(delN(r, ref.I), 1);
      ref.chdRef = newRef;
   else addToPtree(1, ref.chdRef,
         delN(r, ref.I), ref);
   }

eldSib(f, ref, r, oldRef) {
   Generate I_lss
   if(I_lss ≠ ∅ & ≠ oldRef.I)
      newPref = new PtNode(I_lss, ref.sup + 1)
      Link newPref to oldRef as per f
      r = delN(r, I_lss);
      newPref.chdRef = new PtNode(r, 1);
      newPref.chdRef.sibRef = ref;
      Move up any sibings of ref to be
      sibings of newPref if necessary
   else
      newSref = new PtNode(r);
      newSref.sibRef = ref;
      Add newSref to oldRef as per f
   }

yngSib(f, ref, r, oldRef) {
   if (ref.sibRef ≡ null)
      yngSib1(f, ref, r, oldRef);
   else yngSib2(f, ref, r, oldRef);
   }

yngSib1(f, ref, r, oldRef) {
   Generate I_lss
   if(I_lss ≠ ∅ & I_lss ≠ oldRef.I)
      newPref = new PtNode(I_lss, ref.sup + 1)
      Link newPref to oldRef as per f
      ref.I = delN(ref.I, I_lss);
      newPref.chdRef = ref;
      ref.sibRef = new PtNode(delN(r,
            I_lss), 1);
   else ref.sibRef = new PtNode(r, 1);
   }

yngSib2(f, ref, r, oldRef) {
   Generate I_lss
   if(I_lss ≠ ∅ & I_lss ≠ oldRef.I)
      newPref = new PtNode(I_lss, ref.sup + 1)
      Link newPref to oldRef as per f
      ref.I = delN(ref.I, I_lss);
      newPref.chdRef = ref;
      tempRef = ref.sibRef;
      ref.sibRef = new PtNode(r, 1);
      ref.sibRef.sibRef = tempRef;
      Move up any sibings of ref to be
      siblings of newPref if necessary
   else addToPtree(2, ref.sibRef, r, ref);
   }
```

Fig. 3. P-Tree Generation Algorithm.

Fig. 4f shows the results of experiments using a much denser data set than that used above (density 50 percent). Dense data favors both the P-tree and FP-tree approaches, which outperform the other algorithms, with FP Growth giving the best result. However, FP Growth, which recursively produces many FP-trees, requires significantly more storage than Apriori-TFP.

6 Conclusions

In this paper, we have described the T-tree and P-tree ARM data structures (and associated algorithms). Experiments show that:

1. The T-tree offers significant advantages in terms of generation time and storage requirements compared to hash tree structures,
2. The P-tree offers significant preprocessing advantages in terms of generation time and storage requirements compared to the FP-tree,
3. The T-tree is a very versatile structure that can be used in conjunction with many established ARM methods, and
4. The Apriori-TFP algorithm proposed by the authors performs consistently well regardless of the density of the input data set. For sparse data, Apriori-T (also developed by the authors) and DIC perform better than Apriori-TFP, while FP-growth performs less well. For dense data, FP-growth performs significantly better than Apriori-TFP, while Apriori-T and DIC perform less well.

A further advantage offered by the P-tree and T-tree structures is that branches can be considered independently and therefore the structures can be readily adapted for use in parallel/distributed ARM.

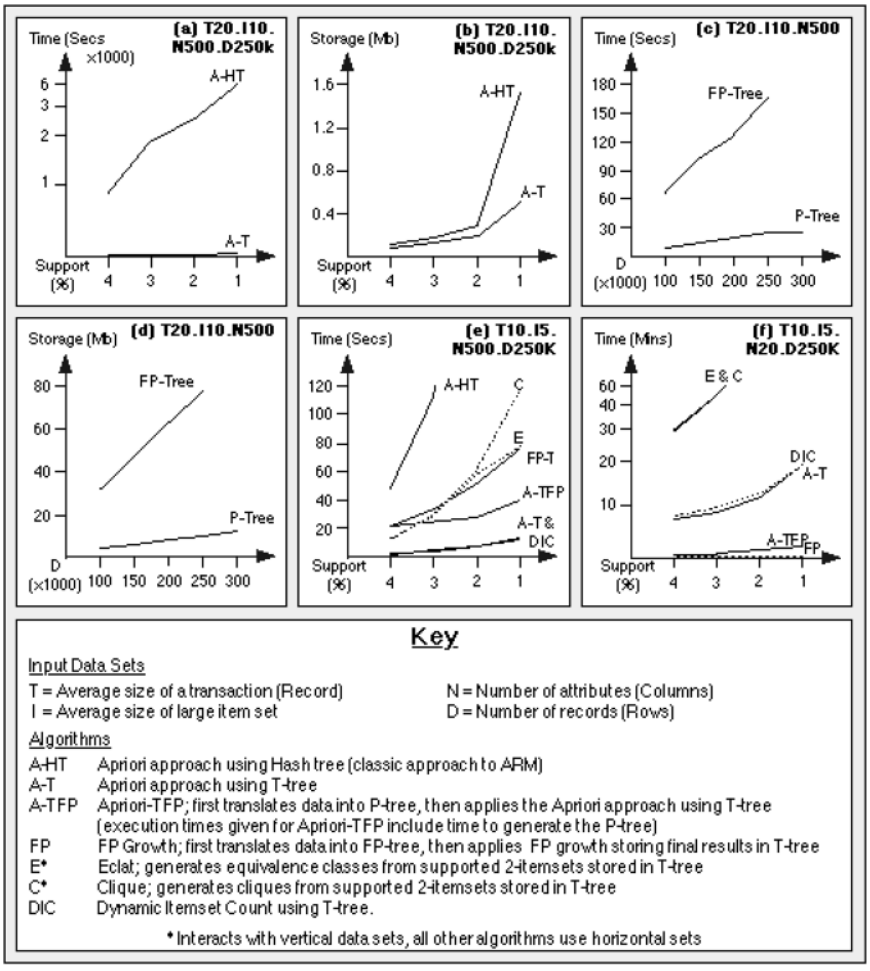

Fig. 4. Evaluation of ARM data structures and algorithms.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Very Large Databses (VLDB) Conf.*, pp. 487-499, 1994.

[2] R.J. Bayardo, "Efficiently Mining Long Patterns from Datasets," *Proc. ACM SIGMOD, Int'l Conf. Management of Data*, pp. 85-93, 1998.

[3] S. Brin, R. Motwani, J. Ullman, and S. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data," *Proc. ACM SIGMOD, Int'l Conf. Management of Data*, pp. 255-264, 1997.

[4] J. Han, J. Pei, and Y. Yiwen, "Mining Frequent Patterns Without Candidate Generation," *Proc. ACM-SIGMOD Int'l Conf. Management of Data*, pp. 1-12, 2000.

[5] Quest project, http://www.almaden.ibm.com/cs/quest/, IBM Almaden Research Center.

[6] R. Rymon, "Search Through Systematic Set Enumeration," *Proc. Third Int'l Conf. Principles of Knowledge and Reasoning*, pp. 539-550, 1992.

[7] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for Fast Discovery of Association Rules," *Proc. Third Int'l Conf. Knowledge Discovery and Data Mining*, 1997.