A Tree Projection Algorithm For Generation of Frequent Itemsets

Ramesh C. Agarwal, Charu C. Aggarwal, V.V.V. Prasad
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

E-mail: { agarwal, charu, vvprasad }@watson.ibm.com

In this paper we propose algorithms for generation of frequent itemsets by successive construction of the nodes of a lexicographic tree of itemsets. We discuss different strategies in generation and traversal of the lexicographic tree such as breadth-first search, depth-first search or a combination of the two. These techniques provide different trade-offs in terms of the I/O, memory and computational time requirements. We use the hierarchical structure of the lexicographic tree to successively project transactions at each node of the lexicographic tree, and use matrix counting on this reduced set of transactions for finding frequent itemsets. We tested our algorithm on both real and synthetic data. We provide an implementation of the tree projection method which is up to one order of magnitude faster than other recent techniques in the literature. The algorithm has a well structured data access pattern which provides data locality and reuse of data for multiple levels of the cache. We also discuss methods for parallelization of the *TreeProjection* algorithm.

Key Words: association rules, data mining, caching, itemsets

CONTENTS

1. INTRODUCTION

The problem of finding association rules was first introduced by Agrawal, Imielinski, and Swami [3]. This problem is concerned with finding relationships between different items in a database containing customer transactions. Such information can be used for many sales purposes such as target marketing, because the buying patterns of consumers can be inferred from one another.

Let I be the set of all items in the database of transactions. A transaction $T \subseteq I$ is defined to be a set of items which are bought together in one operation. An association rule between a set of items $X \subseteq I$ and another set $Y \subseteq I$ is expressed as $X \Rightarrow Y$, and indicates that the presence of the items X in the transaction also indicates a strong possibility of the presence of the set of items Y. The measures used to indicate the strength of an association rule are *support* and *confidence*. The support of the rule $X \Rightarrow Y$ is the fraction of the transactions containing both X and Y. The confidence of the rule $X \Rightarrow Y$ is the fraction of the transactions containing X which also contain Y. A set of items is referred to as an *itemset*. We shall refer to an itemset with k items in it as a k-itemset.

In the association rule problem we wish to find all rules above a minimum level of support and confidence. The primary concept behind most association rule algorithms is a two phase procedure: In the first phase, all *frequent itemsets* are found. An itemset is said to be *frequent* if it satisfies a user-defined minimum support requirement. The second phase uses these frequent itemsets in order to generate all the rules which satisfy the user specified minimum confidence.

Since its initial formulation, considerable research effort has been devoted to the association rule problem. A number of algorithms for frequent itemset generation have been proposed [1, 2, 4, 6, 11, 7, 12, 13, 16, 17]. Variations of association rules such as generalized association rules, quantitative association rules and multilevel association rules have been studied in [9, 14, 15]. In this paper, we present a method which represents frequent itemsets as nodes of a lexicographic tree. We count the support of frequent itemsets by projecting the transactions onto the nodes of this tree. This significantly improves the performance of counting the number of transactions containing a frequent itemset. In a hierarchical manner, we look only at that subset of transactions which can possibly contain that itemset. This is done by traversing the lexicographic tree in a top down fashion. The technique of using carefully chosen lexicographic extensions in order to generate itemsets have been discussed in [3, 6]. Our aim is to use the lexicographic tree as a framework upon which different strategies for finding frequent itemsets can be based. Most other algorithms have utilized a hash tree to count frequent itemsets contained in a transaction. We believe our approach based on projecting transactions on nodes of a lexicographic tree and then eventually counting against a matrix structure provides the most efficient method of counting itemsets at very low levels of support. The advantage of using matrix counting is that efficient cache implementations can be provided which improve substantially over the current implementations. The experimental results support this conclusion.

This paper is organized as follows. In the next section we discuss the concept of the lexicographic tree and a few key ideas which lay the groundwork for the description of our algorithm. In section 3, we will discuss the primary strategies for the creation of the lexicographic tree, and the various trade-offs associated with each strategy. We propose strategies for parallelization of the *TreeProjection* algorithm in section 4. In section 5, we discuss the empirical results, while section 6 contains the conclusion and summary.

2. THE LEXICOGRAPHIC TREE OF ITEMSETS

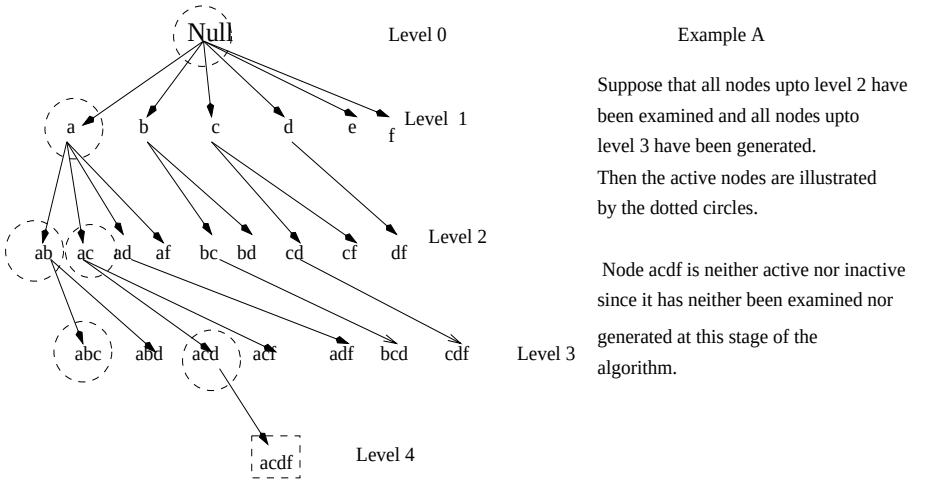


FIG. 1. The lexicographic tree

We assume that a lexicographic ordering exists among the items in the database. In order to indicate that an item i occurs lexicographically earlier than j, we will use the notation $i \leq_L j$. The lexicographic tree is a structural representation of the frequent itemsets with respect to this ordering. The lexicographic tree is defined in the following way:

(1) A vertex exists in the tree corresponding to each frequent itemset. The root of the tree corresponds to the null itemset.

(2) Let $I = \{i_1, \ldots i_k\}$ be a frequent itemset, where $i_1, i_2 \ldots i_k$ are listed in lexicographic order. The parent of the node I is the itemset $\{i_1, \ldots i_{k-1}\}$.

The goal in this paper is to use the structure of the lexicographic tree in order to substantially reduce the CPU time for counting frequent itemsets. An example of the lexicographic tree is illustrated in Figure 1. A frequent 1-extension of an itemset such that the last item is the contributor to the extension will be called a *frequent lexicographic tree extension*, or simply a tree extension. Thus, each edge in the lexicographic tree corresponds to an item corresponding to its frequent lexicographic tree extension. We will denote the set of frequent lexicographic tree extensions of a node P by $E(P)$. In the example illustrated in Figure 1, the frequent lexicographic extensions of node a are b, c, d, and f.

Let Q be the immediate ancestor of the itemset P in the lexicographic tree. The set of *candidate branches* of a node P is defined to be those items in $E(Q)$ which occur lexicographically after the node P. These are the *possible* frequent lexicographic extensions of P. We denote this set by $R(P)$. Thus, we have the following relationship: $E(P) \subseteq R(P) \subset E(Q)$. The value of $E(P)$ in Figure 1, when $P = ab$ is $\{c, d\}$. The value of $R(P)$ for $P = ab$ is $\{c, d, f\}$, and for $P = af$, $R(P)$ is empty.

The *levels* in a lexicographic tree correspond to the sizes of the different itemsets. The various levels for the example in Figure 1 are indicated. We shall denote the set of itemsets corresponding to the nodes at level k by $\mathcal{L}_k$.

A node is said to be *generated*, the first time its existence is discovered by virtue of the extension of its immediate parent. A node is said to have been *examined*, when its frequent lexicographic tree extensions have been determined. Thus, the process of examination of a node P results in generation of further nodes, unless the set $E(P)$ for that node is empty. Obviously a node can be examined only after it has been generated. This paper will discuss a set of algorithms which construct the lexicographic tree in a top-down fashion by starting at the node *null* and successively generating nodes until all nodes have been generated and subsequently examined. At any point in the algorithm, a node in the lexicographic tree is defined to be *inactive*, if it has been determined that the sub-tree rooted at that node can not be further extended. Otherwise, the node is said to be *active*. Thus, the event of a node being *active* or *inactive* is dependent on the current state of the algorithm which is generating the nodes. A node which has just been generated is usually born active, but it becomes inactive later when all its descendents have been determined. For example, in the case of the Figure 1, let us assume that all nodes up to and including level-2 have already been examined. (Consequently, all nodes up to and including level-3 have been generated.) In this case, the set of active nodes would be *abc*, *acd*, *ab*, *ac*, *a*, and *null*. Thus, even though there are 23 nodes corresponding to the top three levels which have been generated, only 6 of them are active. Note that we have not labeled the unexamined nodes *abd* and *acf* as active since even the set of candidate branches for these nodes is empty. We will be using this example repeatedly in our paper. For the sake of notational convenience, we shall label it A.

An active node is said to be a *boundary node* if it has been generated but not examined. In example A, the active boundary node set is $\{abc, acd\}$. As we can see from the complete tree in Figure 1, the subsequent examination of the node *abc* will not lead to any further extensions, while the examination of the node *acd* will indeed lead to the node *acdf*.

The extension set $E(P)$ was produced when P was first examined. As the algorithm progresses, some of these 1-extensions are no longer active. We introduce the term $AE(P)$ to denote the subset of $E(P)$ which is currently *active*. We call these *active extensions*. These represent the branches at node P which are *currently active*. Next, we introduce the concept of *active items*.

The set of *active items* $F(P)$ at a node P is recursively defined as follows:

(1) If the node P is a boundary node, then $F(P) = R(P)$.

(2) If the node P is not a boundary node, then $F(P)$ is the union of $AE(P)$ with *active items* of all nodes included in $AE(P)$.

Clearly, $AE(P) \subseteq F(P) \subseteq E(P)$. The first condition is true because of the nature of the relationship between $AE(P)$ and $F(P)$. $F(P)$ is a set which reduces in size when more itemsets are generated, since fewer number of items form active extensions. For example A, for the *null* node, the only active extension is a, and the set of active items is $\{a, b, c, d, f\}$. For node a, its active extensions are $\{b, c\}$, and the set of active items is $\{b, c, d, f\}$.

In the next section, we will discuss an algorithm which constructs the lexicographic tree. The following information is stored at each node during the process of this construction:

(1) The itemset P at that node.

(2) The set of lexicographic tree extensions at that node which are *currently active* - $AE(P)$.

(3) The set of active items $F(P)$ at that node. $F(P)$ and $AE(P)$ will be updated whenever the set of boundary nodes changes.

Let P be a node in the lexicographic tree at level-m, and let all levels of the lexicographic tree upto level $k > m$ have already been generated. Then, for a transaction T we define the *projected transaction* $T(P)$ to be equal to $T \cap F(P)$. However, if T does not contain the itemset corresponding to node P then $T(P)$ is null. If $T(P)$ has less than $(k - m + 1)$ items then also it is eliminated. This is because a transaction $T(P)$ at a level-m node with less than $(k-m+1)$ items does not have enough items to extend to a node at level-k. The rationale for this is that any successive projection to lower levels strictly reduces the number of items in the projected transaction, and hence, if the projected transaction at node P contains less than $(k - m + 1)$ items, then the same transaction will not contain any item, when projected to a node at level-k. For a set of transactions $\mathcal{T}$, we define the projected transaction set $\mathcal{T}(P)$ to be the set of projected transactions in $\mathcal{T}$ with respect to active items $F(P)$ at P.

Consider the transaction $abcdefghk$. Then, for the example A of Figure 1, the projected transaction at node *null* would be $\{a, b, c, d, e, f, g, h, k\} \cap \{a, b, c, d, f\} = abcdf$. The projected transaction at node a would be $bcdf$. For the transaction $abdefg$, its projection on node ac is null because it does not contain the required itemset ac.

Let P be a level-m node of the lexicographic tree, and let us assume that the top $k > m$ levels have already been generated. We emphasize the following points:

(1) An inactive node does not provide any extra information which is useful for further processing. Thus, it can be eliminated from the lexicographic tree.

(2) For a given transaction T, the information required to count the support of any itemset which is a descendant of a node P is completely contained in its projection $T(P)$.

(3) The number of items in a projected transaction $T(P)$ is typically much smaller than the original transaction. This number continues to reduce as the algorithm progresses, more frequent itemsets are generated, and $F(P)$ reduces. As $F(P)$ shrinks, and k increases, then at some point, $T(P)$ contains less than $(k - m + 1)$ items. At that time, $T(P)$ becomes null.

(4) For a given transaction set $\mathcal{T}$ and node P, the ratio of the number of transactions in $\mathcal{T}(P)$ and $\mathcal{T}$ is approximately determined by the support of the itemset corresponding to P. Since projected transactions with less than $(k - m + 1)$ items are not included, the ratio is actually much smaller.

2.1. Notational Reference Guide

We will briefly repeat the terminology introduced in the above section for ease in reader reference:

$E(P)$: Frequent lexicographic extensions of node P.
$R(P)$: Candidate branches of node P.
$\mathcal{L}_k$: Large itemsets (of length k) corresponding to level k.
$AE(P)$: Currently active extensions of node P.
$F(P)$: Currently active items at node P.
$T(P)$: Projection on node P of transaction T.
$\mathcal{T}(P)$: Projection on node P of transaction set $\mathcal{T}$.

3. ALGORITHMIC STRATEGIES FOR LEXICOGRAPHIC TREE CREATION

Various algorithmic strategies are feasible for lexicographic tree creation. Either all nodes at level-k may be created before nodes at level-$(k+1)$, or longer patterns may be discovered earlier in order to remove some of the other branches of the tree. For example, in breadth-first search all nodes at level-k will be created before nodes at level-$(k+1)$. On the other hand, in depth-first creation, all frequent descendents of a *given node* will be determined before any other node. The various strategies provide different trade-offs in I/O, memory, and CPU performance.

3.1. Breadth First Creation of the lexicographic tree

In breadth-first search, all nodes at level-k are created before nodes at level-$(k+1)$. At any given level-k, the information regarding the possible items which can form frequent lexicographic extensions of it can be obtained from its parent at level-$(k-1)$. A given item i can be a frequent lexicographic extension of a node only if it is also a frequent lexicographic extension of its immediate parent and occurs lexicographically after it. Thus, while finding $(k+1)$-itemsets, we look at all possible frequent lexicographic extensions of each $(k-1)$-itemset. For a given node at level-$(k-1)$, if there are m such extensions, then there are $\binom{m}{2}$ possible $(k+1)$-itemsets which are descendants of this $(k-1)$-itemset. In order to count these $\binom{m}{2}$ possible extensions, we will use projected transaction sets which are stored at that node. The use of projected transaction sets in counting supports is important in the reduction of the CPU time for counting frequent itemsets. A flowchart indicating the overall process is illustrated in Figure 2. The algorithmic description is illustrated in Figure 3. The process of creating the matrices at level-$(k-1)$ of the tree and subsequently executing the subroutine *AddCounts()* is performed in Step 3, the process of addition of new nodes to the lexicographic tree (*AddTree()*) is discussed in Step 4, whereas the process of pruning inactive nodes (*PruneTree()*) from the tree is performed in Step 5.

The process of counting the support of the $(k+1)$-itemsets is accomplished as follows: Let P be any $(k-1)$-itemset whose frequent extensions $E(P)$ (nodes at level-k) have already been determined. At each such node P, a matrix of size $|E(P)| * |E(P)|$ is maintained. A row and column exists in this matrix for each item $i \in E(P)$. An entry exists in this matrix which indicates the count of itemset $P \cup \{i, j\}$. (Since the matrix is symmetric, we maintain only the lower triangular part.) For each item pair $\{i, j\}$ in the projected transaction $T(P)$, we increment the corresponding entry of this matrix by one unit. Thus, the total time for counting at each node is equal to the sum of the squares of the projected transaction sizes at that node. Once the process of counting is complete, the frequent $(k+1)$-itemsets which

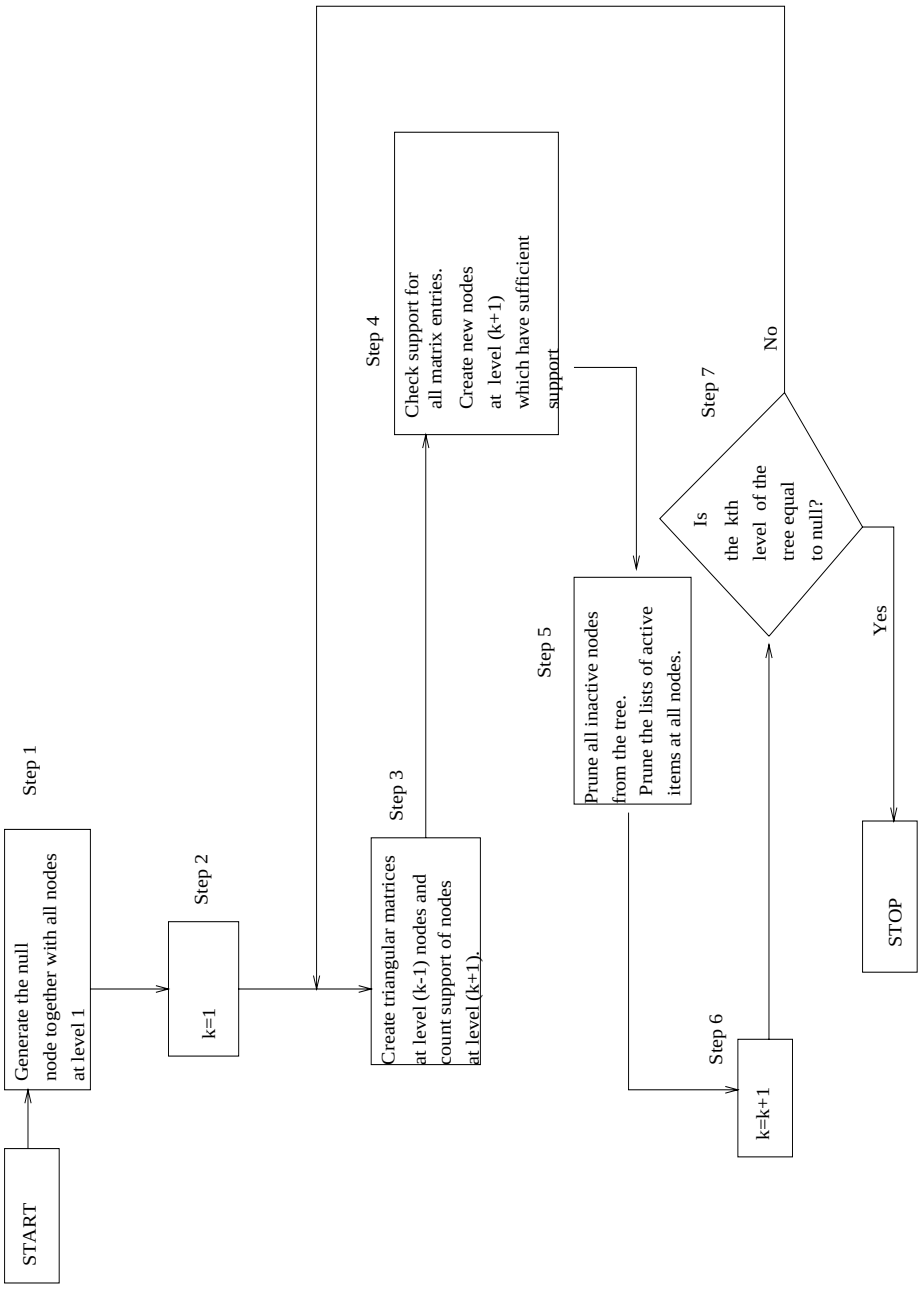


FIG. 2. Flowchart for breadth-first creation of the lexicographic tree

```
Algorithm BreadthFirst(minsupport : s, Database : T)
begin
  { L_k denotes the set of all frequent k-itemsets };
  L_1 = All frequent 1-itemsets;
  E(null) = set of items in L_1;
  Construct the top level of the lexicographic tree;
  k = 1;
  while level-k of the tree is not null do
    begin
    Create matrices at all level-(k − 1) nodes of the lexicographic tree;
    for each transaction T ∈ T do AddCounts(T);
    AddTree(k); { Add itemsets to the set of frequent
        (k + 1)-itemsets L_{k+1} and the lexicographic tree }
    PruneTree(k); { Delete all inactive nodes in the tree
              up to and including level-(k + 1) }
    k = k + 1;
    end
end
```

FIG. 3. Breadth First Creation of the Lexicographic Tree

are descendants of P may be determined by identifying those entries in the matrix which have support larger than the required minimum support s. The process of generating frequent $(k+1)$-itemsets from k-itemsets is repeated for increasing k until the level-k of the tree is null. Another way of rewording the termination criterion is that the active list is empty at the null node. The basic flowchart for performing the counting is illustrated in Figure 4.

The hierarchical structure of the lexicographic tree is useful in creating a set of projected transactions for the level-$(k-1)$ nodes. This is quite important in the reduction of CPU time for itemset counting. The transactions can be projected recursively down the tree in order to create all the projected sets up to the level-$(k-1)$. This projected set is a small subset of the original transaction set *for each node*. However, the total space occupied by the projected transactions over all nodes may be much larger than the original database size. Thus, the algorithm reads a transaction from the database into main memory, recursively projects the transactions down the lexicographic tree in depth first order, and updates the matrices maintained at level-$(k-1)$. (In actual implementation, the algorithm reads a *block* of transactions into a memory buffer at one time.) The process of performing the recursive transaction projection and counting for a single transaction is illustrated in Figure 5. This can also be considered an implementation of Step 3 of Figure 4. This flowchart illustrates how to perform the transaction projection and counting for all descendants of a given node P. The flowchart uses a recursive call to itself. The first call to the flowchart is from the *null* node. The algorithmic description is provided in Figure 6. An example of the matrix along with associated counts at the null node is illustrated in the Figure 9. A more detailed description of the process will be discussed in the section on transaction projection and counting. We have used the concept of a single transaction for ease in exposition. In reality, it is a block of transactions which is projected at one time. More details will be provided in the section on memory requirements.

Once all the counts of the matrices at level-$(k-1)$ have been determined, we compare these counts against the required minimum support, and add the new

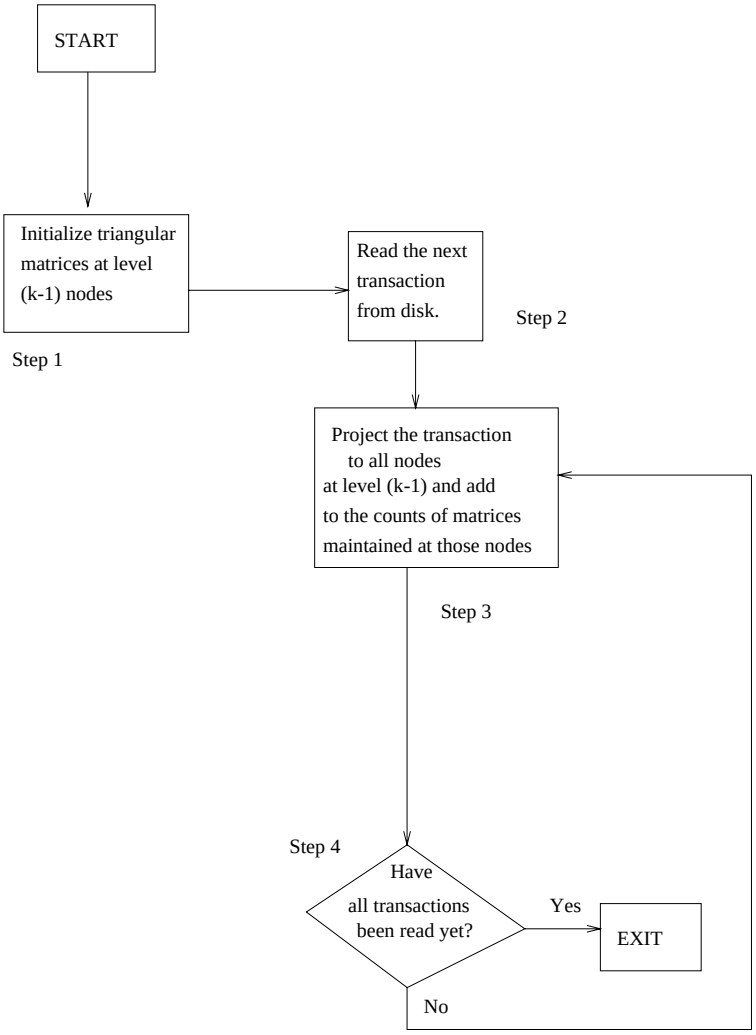


FIG. 4. Illustrating the process of performing counting in the lexicographic tree

Flowchart for recursive projection of transaction T to all descendants of a node P.

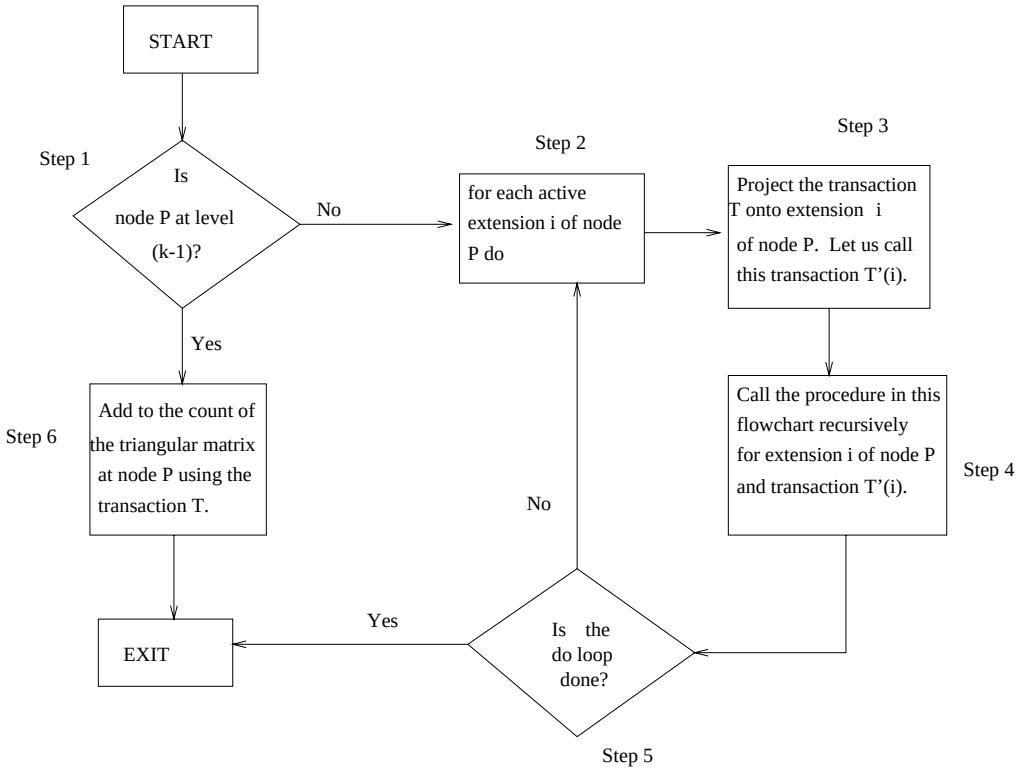


FIG. 5. An implementation of the *AddCounts()* procedure

Algorithm AddCounts(T)
begin
{ Add to the counts of the matrices maintained at
the nodes at level-$(k-1)$ by using a depth first
projection of the transaction T }
end

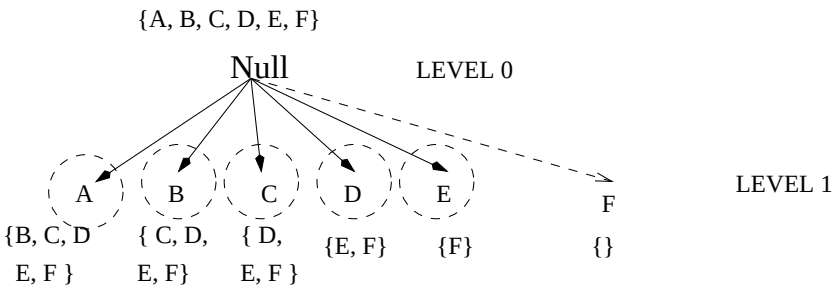
FIG. 6. Incrementing Itemset Counts

Algorithm AddTree(k)
begin
  $\mathcal{L}_{k+1}$ = All $(k+1)$-itemsets which satisfy the minimum support requirement;
  Add nodes to the tree at level-$(k+1)$;
end

FIG. 7. Adding New Itemsets to the Tree

Algorithm PruneTree(k)
begin
  Remove all inactive nodes at level-$(k+1)$;
  Set the active item list at each level-$(k+1)$ node to
    the set of candidate branches at that node;
  for $r = k$ **down to** 0 **do**
  begin
    Remove all inactive nodes at level-r;
    Update active item lists of nodes at level-r to the
      union of their active extensions along with
        active item lists of their children;
  end;
end

FIG. 8. Pruning Inactive Nodes



Transactions:

ACDF
ABCDEF
BDE
CDEF

AB(1)
AC(2) BC(1)
AD(2) BD(2) CD(3)
AE(1) BE(2) CE(2) DE(3)
AF(2) BF(1) CF(3) DF(3) EF(2)

Matrix for counting supports for the candidate level-2 nodes
This matrix is maintained at the null node
Matrix counts for the 4 transactions on the right are indicated

FIG. 9. The matrix structure at the *null* node

nodes at level-$(k+1)$. The new nodes at level-$(k+1)$ are added by the procedure *AddTree()* of Figure 7. The algorithm prunes all those nodes which have been determined to be unnecessary for further counting and tree generation (Figure 8). The process of pruning inactive nodes proceeds by first removing all inactive nodes at level-$(k+1)$, then all inactive nodes at level-k, and so on, until the *null* node. At the same time the active item lists for the nodes are updated. Thus, in the next iteration, when $(k+2)$ itemsets are being counted, the time for projecting transactions is greatly reduced. This is because only active items need to be used in the projection. Another way to look at it is that at any node P, as the algorithm progresses, the projected transaction set $\mathcal{T}(P)$ keeps shrinking both in terms of the number of transactions as well as the number of items in transactions. This is because the active item list $F(P)$ also shrinks as the algorithm progresses. If level-k of the tree has already been generated, then for a node P at level-m, the projection of a transaction $T(P)$ must have at least $(k-m+1)$ items for it to be useful to extend the tree to level-$(k+1)$. It it has fewer items, it is eliminated. For a given value of m, as k increases, fewer transactions satisfy the $(k-m+1)$-items criterion.

3.2. Depth First Creation of the lexicographic Tree

In depth-first search, we create the nodes of the lexicographic tree in depth-first order. At any point in the search, we maintain the projected transaction sets for all nodes (and their siblings) on the path from the root to the node which is currently being extended. The root of the tree contains the entire transaction database. The key point to understand is that once we have projected all the transactions at a given node, then finding the sub-tree rooted at that node is a completely independent itemset generation problem with a *substantially reduced* transaction set. Furthermore, it is possible to prune other branches of the tree quickly. For example, in the Figure 1, once the node *acdf* has been discovered by the depth-first search procedure, it is possible to prune off all other sub-trees hanging at c, d, e and f, since none of these generate any itemsets whose existence is not implied by *acdf*.

In breadth-first search, it is necessary to initiate the process of transaction projection in each iteration, starting from the *null* node. Depth-first search has the advantage that it is not necessary to re-create the projected transactions for each level-k. The problem with this technique is that since the entire transaction database needs to be carried down the tree at one time, disk I/O may be necessary in order to read and write the projected transactions. Thus, although the CPU times are reduced, the disk I/O times may increase so substantially, that the method may often be infeasible for large databases, except for lower levels of the tree where the projected transactions fit in memory. A very efficient version of the depth first algorithm has been proposed in [2] which is capable of finding very long patterns. This algorithm is more than one order of magnitude faster than the *MaxMiner* algorithm [6] for finding long patterns.

3.3. Combined Depth First and Breadth First Creation

In this case, we process the first few levels of the tree using breadth-first search. At some stage of the algorithm, the projected transaction set at individual boundary

level nodes is small enough to fit in memory. At that point, we write all the projected transactions to disk in separate files for each boundary node. These files are read one at a time and the sub-tree rooted at that node is created in the depth-first order. This is done entirely in memory and therefore, does not require any additional I/O. This process is repeated for all boundary nodes. Thus, the inefficiencies associated with each approach can be avoided by using the correct strategy for the case in question. Many other hybrid schemes are possible, each having different trade-offs in terms of I/O, memory, and CPU time.

The implementation discussed in this paper uses a pure breadth-first strategy since it is geared towards short itemsets in large databases.

3.4. Transaction Projection Strategy

As in tree creation, several strategies are possible in projecting transactions to boundary nodes where the eventual counting is done. The most important factors are available memory and cache size. For simplicity, let us assume that the tree has been created in breadth-first order and all nodes up to level-k have been created. To create nodes at level-$(k+1)$, for all nodes P at level-$(k-1)$ we allocate storage for triangular matrices of size $|E(P)*E(P)|$. The collective amount of memory required to hold all these matrices is likely to be far larger than cache size of the machine. However, it is expected that they will fit in the amount of available memory. Any remaining amount of memory is available to hold projected transactions. One possible strategy is to process one transaction at a time and project it to all nodes at level-$(k-1)$. In practice, it will get eliminated at several intermediate nodes of the tree, reaching only a small fraction of the nodes at level-$(k-1)$. If the projected transaction at a level-$(k-1)$ node has m items, then it will require $\binom{m}{2}$ operations to update the triangular matrix stored at that node. This is likely to result in a very poor cache performance. However, if a large fraction of the transaction database is projected to that node, then it is quite likely that many transactions will project to that node. Then, we can set up a loop to count contributions of all these transactions to the triangular matrix. This results in a much better cache performance for the triangular matrices. This approach requires a larger amount of memory. When a transaction is simultaneously projected to all nodes at level-$(k-1)$, the total amount of memory required for all the projections is far larger. However, this problem can be substantially avoided by following the depth-first strategy in projecting the transactions, even though the tree has been created in breadth first order. The discussion of this section can be considered an implementation of Figure 5, using a block of transactions instead of a single transaction T. The actual implementation of the algorithm, as discussed in the empirical section used a block of transactions at one time. The size of this block was determined by the memory availability.

3.5. Implementation Details

The performance of this algorithm is quite sensitive to the lexicographic ordering of the items in the tree. When the ordering of the items in the tree is such that the least frequent item occurs first, the running times are best. This has also been observed earlier by Bayardo [6]. The reason for this is that the average size of the set $E(P)$ is sensitive to the nature of the ordering. If least frequent items are picked

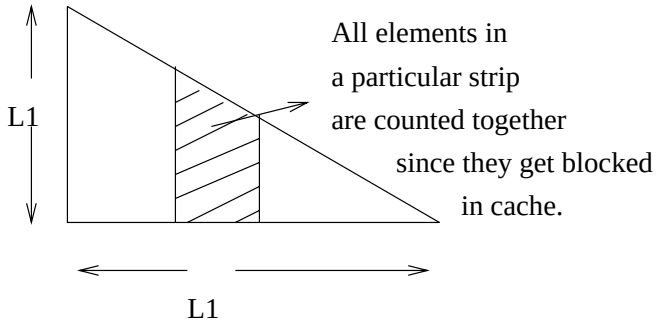


FIG. 10. Cache blocking for counting $\mathcal{L}_2$

first, then the size of $E(P)$ is small. This contributes to the reduced running times. In reality, it is possible to improve the performance even further by re-ordering the items at lower level nodes based on the support counts of the corresponding frequent extensions. In our paper, we choose to maintain a fixed ordering from least to most support after counting 1-itemsets.

Our current implementation is primarily based on pure breadth-first strategy for tree generation, combined with the depth-first strategy for transaction projection and counting. The counting of the support for frequent $(k+1)$-itemsets which are descendents of a $(k-1)$-itemset P was performed in a structured way so that the cache performance was optimized. We have discussed earlier that a block of transactions is used at one time in order to perform the counting. Here we will discuss the method for counting $\mathcal{L}_2$. The general technique of counting $\mathcal{L}_{k+1}$ is very similar. We assume that the database which is used is expressed in terms of $\mathcal{L}_1$ for counting frequent 2-itemsets. This corresponds to the projection at the *null* node. Consider the case when $|\mathcal{L}_1| = 10000$. In that case, the number of elements in the matrix of size $|\mathcal{L}_1| * |\mathcal{L}_1|$ is equal to 49995000. The memory required to hold this triangle is far larger than cache size of a typical machine. Therefore, *cache-blocking* was performed.

In *cache-blocking*, the current buffer of transactions is counted against one strip of elements in the matrix at a time (see Figure 10). Thus, it is more likely for this entire strip to remain in cache when the counting is performed. Each cache block represents a set of items. (An item is represented by a column in this trapezoidal cache block.) This strip of elements was chosen such that the total number of elements in it fits in the cache. The number of cache blocks NC is obtained by dividing the total amount of memory required by the cache size of the machine. We say that a transaction *touches* a block, if it has at least one item in the cache block. If a transaction touches a block, then we maintain pointers which carry the information for the first and the last item in the transaction from each block, as well as the last item in the entire transaction. Thus, three pointers are maintained for each transaction which touches a cache-block. Thus, the transactions are *implicitly* projected to the cache-blocks by storing this pointer information. For a given cache block, let there be M transactions which touch it. For this block, we maintain a list of $3 \cdot M$ pointers for these M transactions. Similar lists are maintained for each of the blocks. Memory buffers are allocated in order to hold the lists of pointers for

each cache block. These pointers may be used to perform the counting effectively. Let $p_1(C,T)$ and $p_2(C,T)$ be the pointers to be first and last item respectively at which a transaction T touches a cache block C, and $p_3(C,T)$ be the pointer to the end of the transaction. Note that the value of $p_3(C,T)$ is the same for each cache block C. We first determine the projections for each transaction in the buffer onto each cache block C:

```
for each transaction T in the buffer do
  for each cache block C touched by transaction T do
    determine and store p1(C,T), p2(C,T), and p3(C,T)
```

The loop for performing the counting using the pointers for the projection of a transaction on a trapezoidal cache block is as follows:

```
for each cache block C do
  for each transaction T in the buffer which touches cache block C do
    for OuterPointer=p1(C,T) to p2(C,T) do
      for InnerPointer=OuterPointer+1 to p3(C,T) do
        Add to the count of the matrix entry corresponding to the items which
          are pointed to by OuterPointer and InnerPointer
```

The code above ensures that the counts for each trapezoidal cache block are done at one time. Cache blocking is more critical for the higher levels of the tree since the matrix sizes are larger at those levels.

3.6. Performance effect of caching

One of the aspects of the *TreeProjection* algorithm is to effectively utilize the memory hierarchy in a way which improves the performance of the algorithm substantially. The speed of cache operations has continued to outpace the speed of main memory operations in the past few years. Therefore, it is valuable to make efficient use of caches both for present and future implementations.

The use of a lexicographic tree facilitates the use of matrix counting techniques for the *TreeProjection* algorithm. Matrix counting provides the short inner loop structure which creates the repeated cache accesses to the same memory addresses. The bottleneck operation of the *TreeProjection* algorithm is in counting. Since the caching aspect of the algorithm improves this bottleneck operation substantially, it follows that this can lead to an overall performance improvement by an order of magnitude.

Another caching advantage is obtained by the depth-first nature of the projection. This is because in the depth-first order, the support of a node and all its descendants is counted before counting the support of any other node. When the set of projected transactions at a node fit in the cache, then the depth-first order is likely to make repeated cache accesses to this small set of transactions while performing the counting for the entire sub-tree rooted at that node. Since most of the time spent by the algorithm is in counting the support of the candidate extensions of lower level nodes, it follows that the performance improvement will be reflected over a large portion of the counting process.

The depth-first projection algorithm also works well with multiple levels of cache. The algorithm automatically exploits multiple cache sizes. Typically, in one pass of the algorithm, we read a large block of data from disk. This block will not fit even in a very large cache. However, the block size is designed to fit in main memory.

It is projected to the null node using the active item list at the null node. This will result in a large reduction in the buffer size. In the next step, it is projected to all currently active level-1 nodes. Each of these projections is likely to be an order of magnitude smaller and may fit in the highest level of cache of the machine (say L3). Next, the block at the leftmost level-1 node is projected to all its active children. These projections are likely to be another order of magnitude smaller and may fit in L2. These blocks are again projected in depth first order to lower level nodes. At this point, these projections may fit in L1 and and all subsequent processing is done with data in L1. Note that the algorithm does not need to know sizes of various levels of cache. The depth-first projection automatically adjusts to various levels and sizes of cache. For the higher levels of tree where the transaction buffer does not fit in cache, cache performance is still good because transactions are accessed with stride one. To summarize, our tree projection counting algorithm provides good data locality and exploits multiple levels of cache.

3.7. Further Optimizations

A number of optimizations can be performed on this algorithm. For example, when the matrix $E(P) * E(P)$ is counted, it is also possible to count the support of the itemset $P \cup E(P)$. If this itemset is discovered to be frequent, then it is possible to stop exploring the sub-tree rooted at P. This implies that all subsets of $P \cup E(P)$ are frequent.

Another technique which may be used to speed up the performance is the method of counting the lower branches of the tree. In this case, we can use *bucket counting* instead of matrix counting. Whenever the size of the $E(P)$ is below a certain level, a set of $2^{|E(P)|} - 1$ buckets may be used in order to find the sub-tree rooted at P. Since there are only $2^{|E(P)|}-1$ possible *distinct* projected transactions at a node, we can find a one-to-one mapping between the buckets and the distinct transactions. The count of a bucket is equal to the number of projected transactions which map on to it. The bucket counting is a two-phase operation. In the first phase, the number of transactions corresponding to each bucket are counted. In the second phase, the counts from various buckets are aggregated together to form counts for all the nodes of the sub-tree rooted at P. In this sub-tree, only those nodes which meet the desired support condition are retained. This technique can result in a substantial reduction in the CPU time and I/O requirements when the number of projected transactions at a given node are large, but the size of $E(P)$ is relatively small.

3.8. A note on memory requirements

In this subsection, we will discuss the memory requirements of the *TreeProjection* algorithm. The memory requirement of the *TreeProjection* is equal to the sum of the memory requirements for triangular matrices at each level-$(k-1)$ node of the lexicographic tree. At first sight, this might seem like a rather large requirement. However, this is proportional to the *number* of *candidate (k+1)-itemsets* at that level. (Assuming that each entry of the matrix requires 4 bytes, the proportionality factor will be 4.) Most other itemset generation algorithms require the *explicit generation* of the candidate itemsets in some form or the other. Thus, the memory requirements for maintaining the matrices in the *TreeProjection* algorithm are quite

comparable to other schemes in the literature. In particular, the lexicographic tree requires much less memory than the hash tree implementation of the *Apriori* algorithm because of the following two reasons:

- Since each node of the hash tree has fewer number of children on the average, it has a larger number of nodes.
- Each node of the hash tree needs to contain multiple hash slots, some of which are usually empty. This is quite wasteful.

TABLE 1
Memory and CPU time requirements for the matrices at the kth level (Retail Data)

Level	Number of Matrices	Average Size	No of Matrix Entries	CPU Time (seconds)
0	1	5706	16276365	23.49
1	1668	30	3521972	25.44
2	4210	5	219269	9.76
3	305	3	8131	2.56
4	12	2	115	1.55

An example of the memory requirements are illustrated in Table 1. This example is for the retail data set (to be introduced in the next section) at a support level of 0.1 percent. As we see, the memory requirements of the algorithm rapidly diminish with increasing level in the tree. Thus, the maximum amount of memory is required for the matrix at level 0, which corresponds to the candidate 2-itemsets. This is generally true for data in which the patterns are relatively short.

It is true that the number of candidates of size k (for $k \geq 3$) scored by *Apriori* is less than the *TreeProjection* algorithm because of certain pruning methods [4] which require that all subsets of a candidate should be frequent. However, our experiments on retail data showed that our candidate sets were only slightly larger compared to fully pruned candidates of *Apriori*. Furthermore, in most of our runs, *Apriori* usually ran out of memory much earlier than the *TreeProjection* algorithm.

The only other memory requirement is to project a block of transactions to a set of nodes in depth-first order. This requirement is flexible, and the block size can be adjusted to fit the amount of available memory. In order to project a block of transactions down the lexicographic tree, we first decide the amount of memory which needs to be allocated to each level. The memory for a given level is distributed among the siblings at a given level based on the support counts. Note that in depth-first projection of a block, only one memory buffer is needed at each level, and typically these buffers become smaller as you go down the tree.

4. EXTENSIONS TO PARALLEL ASSOCIATION RULE MINING

In this section we provide a brief description of how the algorithm may be parallelized. We assume that there are m processors, labelled $1 \ldots m$, The *Apriori*

algorithm may be parallelized to two algorithms: namely, the *Count Distribution Algorithm* and the *Data Distribution Algorithm*. The *Count Distribution Algorithm* [5] scales linearly, and has excellent speedup behavior with respect to the number of transactions. In this algorithm, the database is equally divided among the P different processors. Each processor independantly runs the kth pass of the *Apriori* algorithm, and builds the hash tree in its own memory using its own small portion of the database. After each iteration, a global reduction operation is performed in which the global counts of candidates are computed by summing these individual counts across the different processors [10]. Although the algorithm has excellent scaleup properties with the number of processors, it has the disadvantage that each processor needs to hold the entire hash tree in memory. When the hash tree cannot fit in memory, the data distribution algorithm may be used in order to partition the hash tree among the different processors. In the *Data Distribution Algorithm*, [5] the candidate itemsets are partitioned among the different processors, and each processor is reponsible for computing the counts of its locally stored subset of the candidate itemsets. The performance of the algorithm is sensitive to the distribution of the candidate itemsets among the different processors. It is desirable to distributed the candidate itemsets among the different processors in such a way that the load is very well balanced. The *Intelligent Data Distribution* algorithm of Han et. al. [8] provides some interesting bin packing techniques for performing this load balancing operation effectively. The parallelization techniques for both the Count Distribution and Data distribution algorithms may be used for the *TreeProjection* algorithm.

(1) **Count Distributed TreeProjection Algorithm:** This algorithm is implemented in exactly the same way as the *TreeProjection* Algorithm. In this case, each processor builds the lexicographic tree in its memory instead of the hash tree. Again, the same global sum reduction operation is performed in order to add the support of the candidates across the different processors. The method is linearly scalable with the number of processors. The only disadvantage with the algorithm is that it is likely to work well when the lexicographic tree fits in memory of a single processor. When this is not the case, we need to use a method akin to the Intelligent Data Distributed Algorithm of Han et. al. [8] in order to perform the lexicographic tree generation and counting. This memory constraint is less severe for the parallelized version of the *TreeProjection* algorithm because of the fact that the lexicographic tree requires substantially less memory than the hash tree.

(2) **Intelligent Data Distributed TreeProjection Algorithm:** In this algorithm, we distribute the lexicographic tree among the different processors based on the first item in the tree. Thus, each processor has its own set of *first items* denoted by $S(i)$. The lexicographic tree is distributed among the processors in such a way so that the sum of the supports of the branches assigned to each processor is almost evenly balanced. The techniques for moving the data among the different processors for performing the counting are very similar to those discussed in [8]. We do not discuss these techniques for lack of space in our paper. A more detailed discussion and empirical testing of these parallelization techniques will be provided in future work.

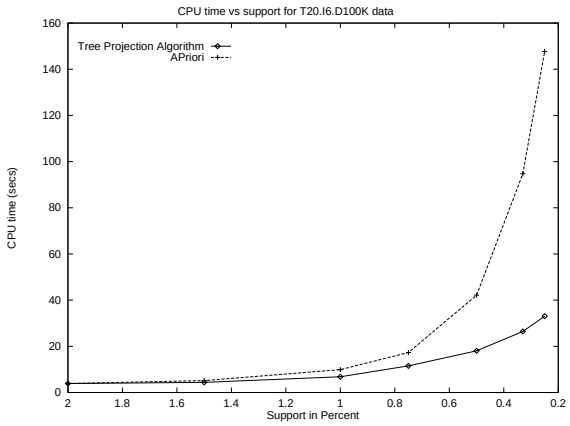


FIG. 11. CPU Time versus support for synthetic data

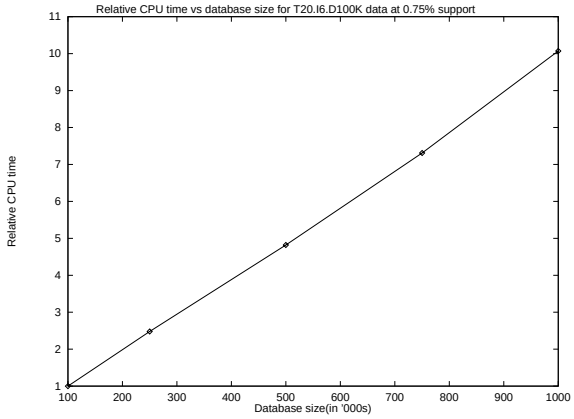


FIG. 12. CPU Time versus Database size for synthetic data

5. EMPIRICAL RESULTS

The experiments were performed on an IBM RS/6000 43P-140 workstation with a CPU clock rate of 200 MHz, 128 MB of main memory, 1MB of L2 Cache and running AIX 4.1. The data resided in the AIX file system and was stored on a 2GB SCSI drive. We tested the algorithm for both synthetic and real data.

The synthetic data set which we used for the purpose of our experiments was T20.I6.D100K [4]. The corresponding performance curves are illustrated in Figures 11 and 12. In Figure 11, we have illustrated the performance of the algorithm versus the support both for the *Apriori* algorithm [4] and our method (we will call it the *TreeProjection* algorithm) on the synthetic data set T20.I6.D100K. As we see, the *TreeProjection* algorithm quickly outstrips the *Apriori* method when support

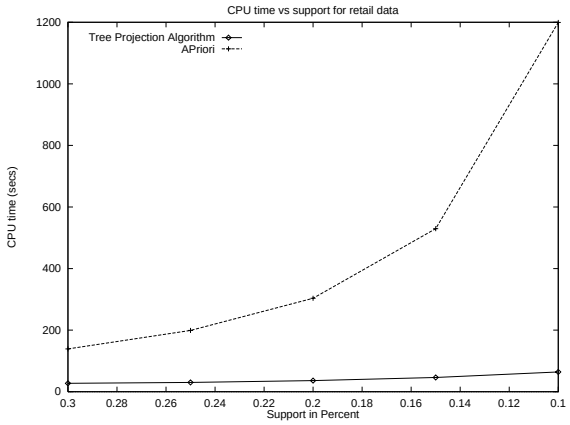


FIG. 13. CPU Time versus support for retail data

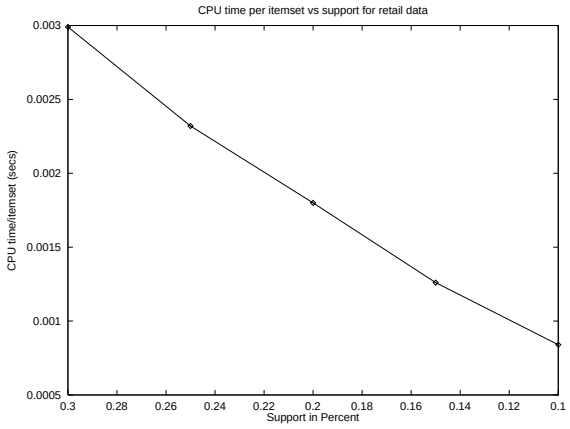


FIG. 14. CPU Time per itemset versus support for retail data

TABLE 2
Number of frequent itemsets found for the splice data set

Support in percent	No of sets	Longest Set
6	19112	7
5.5	26932	7
5	35047	7
4.5	45727	7
4	62258	7
3.5	93928	7
3	164584	8
2.5	309091	8
2	702065	8

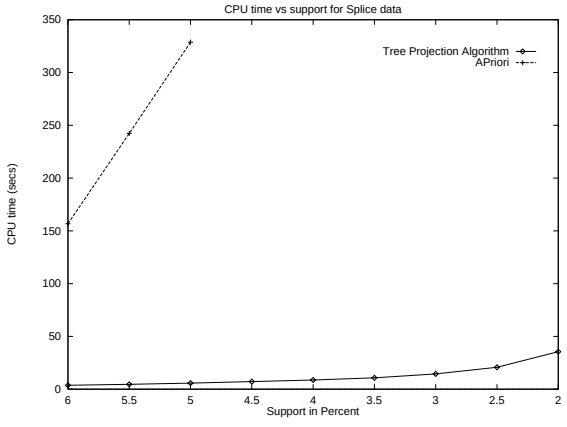


FIG. 15. CPU Time versus support for splice data

values are low. For the lowest support value of 0.25 percent, the *Apriori* algorithm required 147.87 seconds, while our technique required only 33.05 seconds. On the other hand, when the support values are high, there was not much of a difference between the *TreeProjection* and *Apriori* algorithms. This is because the times for performing I/O dominate at the higher levels of support. The scaling with database size was linear for both methods.

We tested the algorithm on two real data sets. One of them was a retail data set which was first used in [6] to test the efficiency of association algorithms. This data set is not publicly available for proprietary reasons. The retail data set contains 213,972 transactions with an average transaction length of 31. The performance gap between the two methods was even more substantial for the case of the retail

data set. The corresponding performance curves are illustrated in Figures 13 and 14. In Figure 13, we have illustrated the performance variation with support. At the lowest support value of 0.1 percent, the *TreeProjection* algorithm required only 64.28 seconds in comparison with the 1199.44 seconds required by the *Apriori* algorithm. This difference is more than an order of magnitude. The time required for counting at each level of the tree for a support of 0.1% is illustrated in Table 1. In addition, 1.48 seconds were required for $\mathcal{L}_1$ counting. Note that very little CPU time is spent in counting support at lower levels of the tree. The reason for this is that our *PruneTree(·)* method removes a very large fraction of the nodes. An even more interesting characteristic of the *TreeProjection* algorithm was the CPU time required per frequent itemset generated. The *TreeProjection* algorithm improved in terms of the time required in order to generate each frequent itemset, when the support was reduced. It is easy to see why this is the case. As the required support level drops, more nodes are generated at the lower levels of the tree. At these levels, the projected transaction database is very small and therefore much smaller effort is required to do the counting per frequent itemset.

We also tested the algorithm on the *splice* dataset. The splice data set was taken from the University of California at Irvine (UCI) machine learning repository(*http* : *//www.ics.uci.edu/~mlearn/MLRepository.html*) and subsequently cleaned [6]. The resulting data set had 3174 records with an average number of items equal to 61. The total number of items in the data was 244. The size of the data set was 0.8MB. The Table 2 illustrates the variation of the number of patterns found and length of the longest pattern with support. Most of the patterns were relatively short, and even at very low levels of support (upto 2%) it was found that the length of the longest pattern was 8. The corresponding performance chart is illustrated in Figure 15. As we see from Figure 15, the *TreeProjection* algorithm outperforms the *Apriori* method by more than an order of magnitude. At very low levels of support, the *Apriori* algorithm runs out of memory and is unable to run to completion. This behavior illustrates the memory advantages of using a lexicographic tree projection algorithm over the the hash tree implementation of the *Apriori* algorithm.

6. CONCLUSIONS AND SUMMARY

This paper demonstrated the power of using transaction projection in conjunction with lexicographic tree structures in order to generate frequent itemsets required for association rules. The advantage of visualizing itemset generation in terms of a lexicographic tree is that it provides us with the flexibility of picking the correct strategy during the tree generation phase as well as transaction projection phase. By combining various strategies, a variety of algorithms are possible to provide very high performance in most situations. The depth-first projection technique provides locality of data access, which can exploit multiple levels of cache. We have also demonstrated the parallelizability of the *TreeProjection* technique, and the advantages of its parallel implementation over the parallel implementation of the *Apriori* algorithm. In many situations, the parallel version of the *TreeProjection* algorithm can reduce the communication required by a large factor compared to the parallel version of the *Apriori* algorithm. Our future research will explore the parallel aspects of the *TreeProjection* algorithm in greater detail.

ACKNOWLEDGMENT

We would like to thank Roberto Bayardo and Rakesh Agrawal for providing us with the retail and splice data on which we tested the algorithms. We would also like to thank them for providing us with their latest code for finding frequent itemsets. We thank Anant Jhingran for his comments on an earlier draft of this paper.

REFERENCES

1. Agarwal R. C., Aggarwal C. C., Prasad V. V. V., Crestana V., "A Tree Projection Algorithm for Generation of Large Itemsets For Association Rules." *IBM Research Report, RC 21341.*
2. Agarwal R. C., Aggarwal C. C., Prasad V. V. V., "Depth First Generation of Long Patterns." *Proceedings of the ACM SIGKDD Conference*, 2000.
3. Agrawal R., Imielinski T., Swami A., "Mining association rules between sets of items in very large databases." *Proceedings of the ACM SIGMOD Conference on Management of data,* pages 207-216, 1993.
4. Agrawal R., Mannila H., Srikant R., Toivonen H., Verkamo A. I., "Fast Discovery of Association Rules." *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, Chapter 12, pages 307-328. *Proceedings of the 20th International Conference on Very Large Data Bases,* pages 478-499, 1994.
5. Agrawal R., Shafer J. C., "Parallel Mining of Association Rules." *IEEE Transactions on Knowledge and Data Engineering.* 8(6), pages 962-969, 1996.
6. Bayardo R. J., "Efficiently Mining Long Patterns from Databases." *Proceedings of the ACM SIGMOD*, pages 85-93, 1998.
7. Brin S., Motwani R., Ullman J. D., Tsur S., "Dynamic Itemset Counting and implication rules for Market Basket Data." *Proceedings of the ACM SIGMOD,* pages 255-264, 1997.
8. Han E.-H., Karypis G, Kumar V., "Scalable Parallel Data Mining for Association Rules." *Proceedings of the ACM SIGMOD Conference.* pages 277-288, 1997.
9. Han J., Fu Y., "Discovery of Multi-level Association Rules From Large Databases." *Proceedings of the International Conference on Very Large Databases*, pages 420-431, Zurich, Switzerland, September 1995.
10. Kumar V., Grama A., Gupta A., Karypis G., *Introduction to Parallel Computing: Algorithm Design and Analysis.* Benjamin Cummings/ Addison Wesley, Redword City, 1994.
11. Lin D., Kedem Z. M., "Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Itemset." *EDBT Conference Proceedings*, pages 105-119, 1998.
12. Mannila H., Toivonen H., Verkamo A. I., "Efficient algorithms for discovering association rules." *AAAI Workshop on Knowledge Discovery in Databases,* pages 181-192, 1994.
13. Savasere A., Omiecinski E., Navathe S. B., "An efficient algorithm for mining association rules in large databases." *Proceedings of the 21st International Conference on Very Large Databases,* 1995.
14. Srikant R., Agrawal R., "Mining Generalized Association Rules." *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 407-419, 1995.
15. Srikant R., Agrawal R., "Mining quantitative association rules in large relational tables". *Proceedings of the ACM SIGMOD Conference on Management of Data,* pages 1-12, 1996.
16. Toivonen H.,"Sampling Large Databases for Association Rules". *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, September 1996.
17. Zaki M. J., Parthasarathy S., Ogihara M., Li W., "New Algorithms for Fast Discovery of Association Rules." *KDD Conference Proceedings*, pages 283–286, 1997.