Depth First Generation of Long Patterns

Ramesh C. Agarwal
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
agarwal@watson.ibm.com

Charu C. Aggarwal
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
charu@watson.ibm.com

V. V. V. Prasad
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
vvprasad@watson.ibm.com

ABSTRACT

In this paper we present an algorithm for mining long patterns in databases. The algorithm finds large itemsets by using depth first search on a lexicographic tree of itemsets. The focus of this paper is to develop CPU-efficient algorithms for finding frequent itemsets in the cases when the database contains patterns which are very wide. We refer to this algorithm as *DepthProject*, and it achieves more than one order of magnitude speedup over the recently proposed *MaxMiner* algorithm for finding long patterns. These techniques may be quite useful for applications in areas such as computational biology in which the number of records is relatively small, but the itemsets are very long. This necessitates the discovery of patterns using algorithms which are especially tailored to the nature of such domains.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Association Rules

1. INTRODUCTION

The association rule problem has been recognized in the literature as a fundamentally important problem in the field of data mining. Applications of association rules extend to finding useful patterns in consumer behavior, target marketing, and electronic commerce. The association rule model was introduced by Agrawal, Imielinski, and Swami [5].

Starting with the pioneering work in [5], the association rule problem and its variations have been studied extensively by researchers. Several variations of the association rule problem [4, 8, 10, 19] have been proposed which can provide more interesting rules than the support-confidence framework. In addition, a number of methods have been discussed in the literature which extend the binary association rule problem to related scenarios such as quantitative association rules, generalized association rules, and optimized association rules [12, 13, 22, 23, 27, 28]. Methods for providing ad-hoc query capabilities, and online mining have been discussed in [3, 16, 20, 21].

Let I be a set of items. Each transaction T in the database is a subset of the items occuring in I. A set of items in the database is referred to as an *itemset*. In particular, a set of items with a cardinality of k is referred to as a k-itemset. The measure used to evaluate the level of presence of an itemset in the database is the *support*. The support of itemset X is equal to the fraction of the transactions containing X. A key step of association rule mining is find *frequent itemsets* or *large itemsets* [5]. These are itemsets whose support is larger than a user-specified threshold. A fast algorithm called *Apriori* was proposed in [6], which generates $(k+1)$-candidates using joins over frequent k-itemsets which were already generated. Thus, for each frequent itemset, all subsets of it need to be generated by the algorithm.

Because of the inherent difficulty of the itemset generation problem in terms of computational complexity, considerable research has been devoted towards finding faster methods for generating large itemsets [6, 7, 11, 14, 15, 17, 18, 26, 29]. The large itemset problem is reasonably well solved at least for the case of very sparse sales transaction data, when the pattern lengths are short [1, 6]. An interesting analysis of the impact of different kinds of data on access costs has been provided in [11]. An *Apriori*-style algorithm with improved counting techniques using columnwise data access for databases with a larger number of items has been also been discussed in the same work. We maintain that when the actual frequent patterns are wide, even the CPU-costs of any algorithm which is based on the *Apriori*-framework would be compromised by the investigation of all 2^k subsets of frequent k-patterns. In such cases, the frequent itemset generation algorithms become CPU-bound. Some of the algorithms in the literature such as *MaxMiner* avoid this by implementing *lookaheads* [7], in which supersets of frequent patterns are used in order to prune off potential candidates in the search. Other innovative ideas for handling these problems are discussed in [29]. In spite of these advances [7, 11, 29], finding computationally efficient algorithms for generating long patterns continues to be a very difficult problem. This paper is written with the primary aim of developing an algorithm for long patterns which is CPU-efficient.

This paper is organized as follows. In section 2, we will discuss several formalizations, which will be useful in describing the algorithm. In section 3, we will introduce the depth-first strategy of generating large itemsets. The database representation will be explained in section 4. In section 5, we will discuss the bucketing technique which is a useful method for speeding up the algorithm substantially. An effective pruning and lookahead strategy for the algorithm is discussed in section 6. We will compare the results to the *MaxMiner* algorithm, and present the computational results in section 7. Section 8 discusses the conclusions and summary.

1.1 Contributions and Motivation

This paper proposes a fundamentally different technique for finding large itemsets from most of the algorithms proposed in the recent past. A large number of algorithms in the research literature start the process of generating $(k+1)$ itemsets only after all k-itemsets have been generated. Even the recent look-ahead-based algorithm discussed [7] for mining long patterns guarantees the discovery of all $(k+1)$-itemsets only after generation of all k-itemsets, even though some of the $(k+1)$-itemsets may be discovered earlier than the k-itemsets. The reason for this natural algorithmic design has been motivated by the desire to restrict the number of passes over the database to the length of the longest pattern. This often results in the generation of a large number of subsets of frequent itemsets. A few methods [9] deviate from this natural design in order to reduce the number of I/O passes, but tend to be *Apriori*-like in their overall approach; consequently the combinatorial explosion problem continues to be an issue.

The long pattern problem is so difficult to solve computationally, that even for databases of relatively small sizes, it may be very difficult to find long patterns [7]. In the past decade, memory availability has increased by orders of magnitude. It has recently started becoming increasingly evident that in the near future, many medium to large size databases are likely to be main memory resident. For problems in which the patterns are longer than 15-20 items, and the database is too large to fit under the current memory limitations (which are reaching the Gigabyte order), most of the algorithms which require the generation of all subsets of frequent itemsets are impractical anyway. For the *long pattern* problem, it may perhaps be realistic to design algorithms with much greater focus on CPU requirements for transaction sets of moderate sizes which can fit in main memory.[1]

The long pattern problem has considerable applicability in combinatorial pattern discovery in biological sequences [24]. In this case, we have a small set of biological sequences, and we wish to find all the rigid patterns (motifs) in these sequences. This is an example of a problem in which the number of patterns is small enough to fit in main memory, whereas each individual pattern can be quite long, and it may be too time consuming to use *Apriori*-like algorithms

[1]If desired, the main memory algorithm in this paper can be combined with a method discussed in [26] which divides a disk-resident database into different main-memory partitions and then combines the itemsets from the different partitions in order to generate the final itemsets. This is however not the focus of this paper.

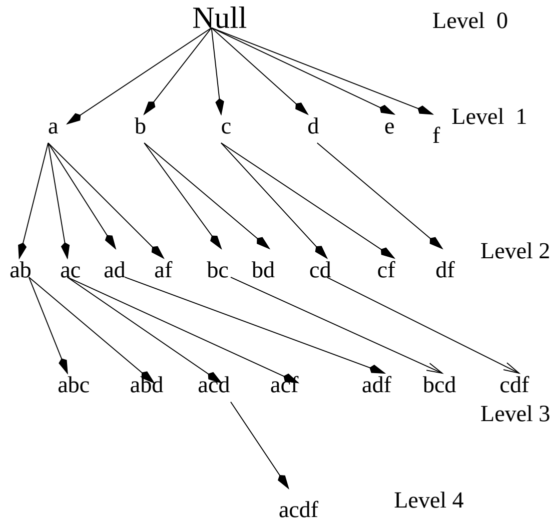


Figure 1: The lexicographic tree

for such problems. Our results in this paper are tailored towards such domains in which the patterns are too long to be handled by *Apriori*-style algorithms.

2. THE LEXICOGRAPHIC TREE

In this section, we will introduce a conceptual string representation of large itemsets which we will refer[2] to as the lexicographic tree [1]. This will provide us with the necessary machinery needed in order to explain the *DepthProject* algorithm. 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 an abstract representation of the large itemsets with respect to this ordering. The lexicographic tree is defined in the following way:

(1) A node exists in the tree corresponding to each large itemset. The root of the tree corresponds to the *null* itemset.

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

This definition of ancestral relationship naturally defines a tree structure on the nodes, which is rooted at the *null* node. The goal in this paper is to use the structure of the lexicographic tree in order to substantially reduce the CPU time for counting large 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 which is the frequent lexicographic tree extension to a node. We will denote the set of frequent lexicographic tree extensions of a node P by

[2]A set representation of this tree is referred to as the enumeration tree [25].

$E(P)$. 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 *prospective 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 $F(P)$. Thus, we have the following relationship: $E(P) \subseteq F(P) \subset E(Q)$. The value of $E(P)$ in Figure 1, when $P = ab$ is $\{c, d\}$. The value of $F(P)$ for $P = ab$ is $\{c, d, f\}$, and for $P = af$, $F(P)$ is empty.

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 an algorithm which constructs the lexicographic tree in depth-first order by starting at the node *null* and successively generating nodes until all nodes have been generated and subsequently examined.

An itemset is defined to be a *frequent maximal itemset*, if it is frequent, and no superset of that itemset is frequent. Thus, in the Figure 1, the itemset *acdf* is maximal.

Let P be a node in the lexicographic tree corresponding to a frequent k-itemset. Then, for a transaction T we define the *projected transaction* $T(P)$ to be equal to $T \cap E(P)$. However, if T does not contain the itemset corresponding to node P then $T(P)$ is null. 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 frequent items $E(P)$ at P.

Consider the transaction $abcdefghk$. Then, for the example A of Figure 1, the projected transaction at node *null* would be $\{a, b, c, d, e, f, g, h, k\} \cap \{a, b, c, d, e, f\} = abcdef$. 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.

We emphasize the following points:

(1) 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 $T(P)$.

(2) The number of items in a projected transaction $T(P)$ is typically much smaller than the original transaction.

(3) 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 P.

In the next section, we will discuss an algorithm which finds the itemsets by depth first creation of the lexicographic tree. The following description will discuss a pure approach in which the entire tree is generated by the algorithm. In a later section, we will show that it is possible to significantly speed up the algorithm by pruning away those subtrees which are guaranteed to contain only non-maximal itemsets. In addition, the discussion in the following section does not include a specialized counting technique (called bucketing) for lower level nodes, which we will also postpone to a later section.

Algorithm *DepthFirst(Itemset Node: N,*
    PointerToDatabase: $\mathcal{T}$, Bitvector : B)
  begin
  $C = GenerateCandidates(N)$;
  $E = Count(N, \mathcal{T}, B, C)$;
  { Let $E = \{i_1, \ldots, i_{|E|}\}$, when expressed in
      lexicographic order }
  Store frequent itemsets $N \cup \{i_r\}$ for $r \in \{1, \ldots, |E|\}$;
  $B' = CreateBitvector(N, B, \mathcal{T})$;
  if ($ProjectionCondition$) **then**
    begin
    $\mathcal{T}' = Project(\mathcal{T}, E, N, B')$;
    Modify B' to be a set of $|\mathcal{T}'|$ ones;
    end;
        else $\mathcal{T}' = \mathcal{T}$;
  for $r := 1$ **to** $|E|$
    begin
    $DepthFirst(N \cup \{i_r\}, \mathcal{T}', B')$;
    end
  end

Subroutine *Project(Database: $\mathcal{T}$,*
    $FrequentExtensions: E$, Bitvector: B)
  begin
  $\mathcal{T}'$ = Empty set of transactions;
  for each transaction $T \in \mathcal{T}$ **do**
    begin
    if corresponding bit in B is 1 **then** add $T \cap E$ to $\mathcal{T}'$;
    end
  return($\mathcal{T}'$);
  end

Subroutine *CreateBitvector($N, B, \mathcal{T}$)*
  begin
  Initialize $B' = B$;
  Let n be the lexicographically largest item in N;
  for each transaction $T \in \mathcal{T}$ **do**
    if $n \notin T$ **then** set the corresponding bit in B' to 0;
  return(B');
end

Figure 2: The Depth First Strategy

At this point, we mention that a very interesting (but different) tree based algorithm *FP-growth* has been proposed in [15]. This method is able to generate itemsets *without* candidate generation. The technique has been compared with the *TreeProjection* algorithm [1] in the empirical results presented in [15]. Unlike *TreeProjection*, the *DepthProject* algorithm is specifically designed for finding long max-patterns and is orders of magnitude faster than the former in these cases.

3. THE DEPTH FIRST STRATEGY

Algorithm *GenerateCandidates(Itemset Node:N)*
begin
  if (N is *null*) **then**
    return all items;
  else
    return frequent extensions of parent of N which
      are lexicographically larger than any item in N;
end

Figure 3: Generating candidate branches of a node

Algorithm *Count(ItemsetNode:N,*
  DatabasePointer: $\mathcal{T}$, Bitvector: B, CandidateSet: C)
begin
  Count all the possible frequent extensions of node N denoted by candidate set C using the database $\mathcal{T}$ and the bitvector B. Details of the counting procedure are provided in Section 4.1;
end

Figure 4: Counting frequent extensions of a node

In depth-first search, the nodes of the lexicographic tree are *examined* in depth-first order. The process of examination of a node refers to the counting of the supports of the candidate extensions of the node. In other words, the support of all descendant itemsets of a node is determined before determining the frequent extensions of other nodes of the lexicographic tree. At a given node, lexicographically lower item-extensions are counted before lexicographically higher ones. Thus, the order in which a depth-first search method would count the extensions of nodes in the Figure 1 is *null*, a, ab, abc, abd, ac, acd, $acdf$, acf, ad, adf, af, bc, bcd, bd, c, cd, cdf, cf, d, df, e, and f. Thus, the depth first strategy quickly tends to find the longer patterns first in the search process. Note that the string representations of the nodes are visited in dictionary order. At any point in the search, we maintain the projected transaction sets for some of the nodes on the path from the root to the node which is currently being extended.[3] A pointer is maintained at each node P to the projected transaction set which is available at the nearest ancestor of P. Since the projected database is substantially smaller than the original database both in terms of the number of transactions, and the number of items, the process of finding the support counts is speeded up substantially. The following information is stored at each node during the process of construction of the lexicographic tree:

(1) The itemset P at that node.

(2) The set of lexicographic tree extensions at that node which are $E(P)$.

(3) A pointer to the projected transaction set $\mathcal{T}(Q)$, where Q is some ancestor of P (including itself). The root of the tree points to the entire transaction database.

[3]In the implementation section, we shall describe in detail which nodes are the ones at which the projected transaction sets are maintained.

(4) A bitvector containing the information about which transactions contain the itemset for node P as a subset. The length of this bitvector is equal to the total number of transactions in $\mathcal{T}(Q)$. The value of a bit for a transaction is equal to 1, if the itemset P is a subset of the transaction. Otherwise it is equal to zero. Thus, the number of 1 bits is equal to the number of transactions in $\mathcal{T}(Q)$ which project to P. The bitvectors are used in order to make the process of support counting more efficient.

Once we have identified all the projected transactions at a given node, then finding the subtree rooted at that node is a completely independent itemset generation problem with a *substantially reduced* transaction set. As was indicated earlier, the number of transactions at a node is proportional to the support at that node. An important fact about projections is the following:

By using hierarchical projections, we are reusing the information from counting k-itemsets in order to count $(k+1)$-itemsets.

Note that such a reuse of information is made possible by the depth first strategy, since we only need to maintain the projected transaction sets on the path of the tree which is currently being explored. Let us consider a k-itemset I at which the database is projected. If a transaction T does not contain this k-itemset I as a subset, then the projection strategy ensures that T will not be used in order to count any of the $(k+1)$-extensions of I. This is important in reducing the running time, since a large fraction of the transactions will not be relevant in counting the support of an itemset. Furthermore, the process of projection reduces the number of fields in the database to a small number so that the counting process becomes more efficient.

The description in Figure 2 shows how the depth first creation of the lexicographic tree is performed. The algorithm is described recursively, so that the call from each node is a completely independent itemset generation problem, which finds all frequent itemsets that are descendants of a node. There are three parameters to the algorithm, a pointer to the database $\mathcal{T}$, the itemset node N, and the bitvector B. The bitvector B contains one bit for each transaction in $T \in \mathcal{T}$, and indicates whether or not the transaction T should be used in finding the frequent extensions of N. A bit for a transaction T is one, if the itemset at that node is a subset of the corresponding transaction. The first call to the algorithm is from the *null* node, the parameter $\mathcal{T}$ is the entire transaction database. Since each transaction in the database is relevant in order to perform the counting, the bitvector B consists of all "one " values.

The first step of the algorithm is to generate all the candidate extensions of N. This is accomplished by the subroutine call $GenerateCandidates(N)$. For the case of the *null* node, this call returns all the items in the entire database. For the case of other nodes, the procedure simply returns the set of items in $F(N)$. As discussed earlier, the set of items in $F(N)$ may be determined by finding all those frequent extensions of the parent of N, which are lexicographically larger than any item in N. The details of the $GenerateCandidates$ procedure are illustrated in Figure 3.

The next step is to count the support of each of these candidate extensions. This is done by the procedure *Count*. The bitvector B and the database $\mathcal{T}$ are used in order to perform the counting efficiently, since it uses information from counting ancestral nodes. The details of how the counting is performed will be discussed in a later section. At the same time, it is desirable to create the bitvectors for node N. The bitvectors for node N may be derived very simply from the bitvector B. This is because the bitvector B corresponds to the parent of node N, which differs from N by exactly one item (the lexicographically largest item). Let n be the lexicographically largest item in N. Thus, the new bitvector for node N may be obtained by changing those bits in B to 0, if the corresponding transactions do not contain the item n.

For some of the nodes, it is desirable to project the database, if there is a substantial reduction in the database size or field width. At this stage, we choose to be ambiguous about this condition by referring to it generally as the *Projection-Condition*. A new database is created using the subroutine *Project* of Figure 2. If the database is indeed projected, then the bitvector at node N also needs to be modified in order to reflect this. Since every transaction in this new database is needed in order to generate the frequent extensions of N, we create a new bitvector B' with as many entries as this newly created database. The value of each bit in this database is 1 because all transactions in this database contain the itemset corresponding to N.

Once the bitvectors and newly projected database has been created, we call the algorithm recursively from all frequent extensions of node N. The calls to the frequent extensions of N are made in lexicographic order. These recursive calls create the subtrees which are rooted at the corresponding nodes of the lexicographic tree.

3.1 Heuristic Rules for selective projection

Various heuristic rules may be used in order to decide the exact nature of the projection condition discussed in Figure 2. For example, it is possible to use the heuristic rule that a projected transaction set may be maintained at a node only when the size of the projected database is less than a certain factor of the previous database size. Alternatively, it is possible to maintain a projected transaction set at a node, only when the number of bits to represent a (projected) transaction at that node is less than a certain number *i.e.* $|E(P)|$ falls below a certain number. The primary motivation is to project only when the database representation becomes "sufficiently more efficient" after performing the process. We will discuss later how we chose the projection condition for our implementation. We will also discuss how to combine the depth first creation of the lexicographic tree with an effective lookahead strategy in order to avoid creation of those subtrees of the lexicographic tree which contain only non-maximal itemsets.

4. DATABASE REPRESENTATION

A number of optimizations were performed in order to improve the performance of the algorithm. Since the algorithm discussed in this paper is geared towards generating long patterns, we found that it was better to use the database in the *bitstring representation*. In the bitstring representation, each item in the database has one bit representing it. Thus, the length of each transaction in bits is equal to the total number of items in the database. Such a representation is inefficient when the number of items in a transaction is significantly less than the total number of items. This is because of the fact that most of the bits take on the value of 0. However, the *DepthProject* algorithm performs counting on the projected transactions which are expressed in terms of $E(P)$. In this representation, most bits take on the value of 1. Furthermore, for problems in which the database contains long patterns, the ratio of the maximum pattern length to the total number of items is relatively high for most lower level nodes.

We have discussed the concept of selective projection slightly earlier. In the implementation of the algorithm over several datasets, we found that the best strategy was to project the database only when $|E(P)|$ is reduced to less than 32. In this case, projected transactions can be represented as 32 bit words. Thus, a projected database with one million transactions can be represented in only 4 MB, when $|E(P)|$ is less than 32. This illustrates the considerable advantages of using the bitstring representation for holding the projected transactions. Our experiments show that most of the time was spent in counting these lower level nodes where the representation and counting ability was most efficient. We will now discuss how the bitstring representation is helpful in reducing the counting times substantially.

4.1 Counting Methods

Carefully designed counting methods are critical in reducing the times for finding support counts. Let us consider a node P, at which it is desirable to count the support of each item in $F(P)$. Let Q be the nearest ancestor of P at which the projected database $\mathcal{T}(Q)$ is maintained. Each projected transaction in $\mathcal{T}(Q)$ contains n bits, where $n = |E(Q)|$. We are assuming that $|F(P)|$ is close to n. Otherwise, we would have projected the database at some intermediate node between P and Q according to the projection condition in order to reduce the database size. Let T be a transaction in $\mathcal{T}(Q)$. A naive method of counting would be to maintain a counter for each item in $F(P)$ and add one to the counters of each of those elements for which the corresponding bit in T takes on the value of 1. However, it is possible to reduce the counting times greatly by using a two phase byte counting method.

The first step in the counting technique is to decide whether the transaction T is relevant in counting the support of candidate extensions of node P. The transaction T is relevant in counting the support of candidate extensions of node P, if P is a subset of T. The bitvector maintained at the immediate parent P' of P provides the information as to whether the transaction T, was relevant in counting the support of the frequent extensions of P'. If this is so, then we need to check additionally whether the lexicographically largest item of P (this item extends P' to P) is present in T. Once it has been determined that the transaction T is relevant in order to perform the counting at node P, we count the support for each item in $F(P)$. Let us assume that each transaction T contains n bits, and can therefore be expressed in the form of $\lceil n/8 \rceil$ bytes. Each byte of the transaction contains the information about the presence or absence of eight items, and the integer value of the corresponding bitstring can take on

```
Algorithm AggregateCounts(Counts:bucket[...])
begin
  { We assume that there are 2^|E(P)| buckets, one
      corresponding to each bitstring of length |E(P)| }
  k = |E(P)|;
  for i := 1 to k do
      begin
      for j := 1 to 2^k do
      if the ith bit of bitstring representation
         of j is 0 then
          begin
          bucket[j] = bucket[j] + bucket[j + 2^(i-1)];
          end
      end
end
```

Figure 5: Aggregating bucket counts

any value from 0 to $2^8 - 1 = 255$. Correspondingly, for each byte of the (projected) transaction at a node, we maintain 256 counters, and we add 1 to the counter corresponding to the integer value of that transaction byte. This process is repeated for each transaction in $\mathcal{T}(P)$. Therefore, at the end of this process, we have $256 * \lceil n/8 \rceil$ counts. We follow up with a postprocessing phase in which we determine the support of an item by adding the counts of the $256/2 = 128$ counters which take on the value of 1 for that bit. Thus, this phase requires $128 * n$ operations only, and is independent of database size. The first phase, (which is the bottleneck) is the improvement over the naive counting method, since it performs only 1 operation for each *byte* in the transaction, which contains 8 items. Thus, the method would be a factor of 8 faster than the naive counting technique, which would need to scan the entire bitstring. The counting times are determined by the time spent at the lower level nodes, at which the projected transactions are inherently dense. At the lower levels in the tree, we use another specialized technique called bucketing in order to perform the counting. We will discuss this method in greater detail in the next section.

5. THE BUCKETING TECHNIQUE

Most of the nodes in the lexicographic tree correspond to the lower levels. Thus, the counting times at these levels account for most of the CPU times of the algorithm. For these levels, we used a strategy called bucketing in order to substantially improve the counting times. The idea is to change the counting technique at a node in the lexicographic tree, if $|E(P)|$ is less than a certain value. In this case, an upper bound on the number of distinct *projected* transactions is $2^{|E(P)|}$. Thus, for example, when $|E(P)|$ is 9, then there are only 512 distinct projected transactions at the node P. Clearly, this is because the projected database contains several repetitions of the same (projected) transaction. The fact that the number of *distinct* transactions in the projected database is small can be exploited in order to yield substantially more efficient counting algorithms. The aim is to count the support for the entire subtree rooted at P with a quick pass through the data, and an additional postprocessing phase which is independent of database size. The process of performing bucket counting consists of two phases:

(1) In the first phase, we count how many of each distinct transaction are present in the projected database. This can be accomplished easily by maintaining $2^{|E(P)|}$ buckets or counters, scanning the transactions one by one, and adding counts to the buckets. The time for performing this set of operations is linear in the number of (projected) database transactions.

(2) In the second phase, we use the $2^{|E(P)|}$ transaction counts in order to determine the aggregate support counts for each itemset. In general, the support count of an itemset may be obtained by adding the counts of all the supersets of that itemset to it. A skillful algorithm (from the efficiency perspective) for performing these operations is illustrated in Figure 5.

Consider a string composed of 0, 1, and $*$, which refers to an itemset in which the positions with 0 and 1 are fixed to those values (corresponding to presence or absence of items), while a position with a $*$ is a "don't care". Thus, all itemsets can be expressed in terms of 1 and $*$, since itemsets are traditionally defined with respect to presence of items. Consider for example, the case when $|E(P)| = 4$, and there are four items, numbered $\{1, 2, 3, 4\}$. An itemset containing items 2 and 4 is denoted by $*1*1$. We start off with the information on $2^4 = 16$ bitstrings which are composed of 0 and 1. These represent all possible distinct transactions. The algorithm aggregates the counts in $|E(P)|$ iterations. The count for a string with a "*" in a particular position may be obtained by adding the counts for the strings with a 0 and 1 in those positions. For example, the count for the string *1*1 may be expressed as the sum of the counts of the strings 01*1 and 11*1.

The procedure in Figure 5 works by starting with the counts of the 0-1 strings, and then converts them to strings with 1 and *. The algorithm requires $|E(P)|$ iterations. In the ith iteration, it increases the counts of all those buckets with a 0 in the ith bit, so that the count now corresponds to a case when that bucket contains a $*$ in that position. This can be achieved by adding the counts of the buckets with a 0 in the ith position to that of the bucket with a 1 in that position, with all other bits having the same value. For example, the count of the string 0*1* is obtained by adding the counts of the buckets 001* and 011*. In Figure 5, the process of adding the count of the bucket j to that of the bucket $j + 2^{i-1}$ achieves this.

The second phase of the bucketing operation requires $|E(P)|$ iterations, and each iteration requires $2^{|E(P)|}$ operations. Therefore, the total time required by the method is proportional to $2^{|E(P)|} \cdot |E(P)|$. When $|E(P)|$ is sufficiently small, the time required by the second phase of postprocessing is small compared to the first phase, whereas the first phase is essentially proportional to reading the database for the current projection.

We have illustrated the second phase of bucketing by an example in which $|E(P)| = 3$. The process illustrated in Figure 6 illustrates how the second phase of bucketing is efficiently performed. The exact strings and the corresponding counts in each of the $|E(P)| = 3$ iterations are illustrated. In the first iteration, all those bits with 0 in the lowest order posi-

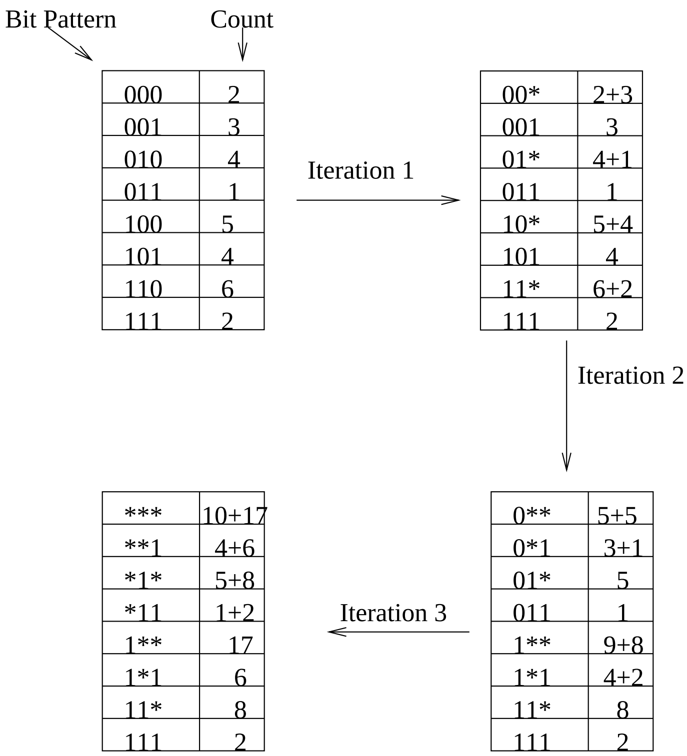


Figure 6: Performing the second phase of bucketing

tion have their counts added with the count of the bitstring with a 1 in that position. Thus, $2^{|E(P)|-1}$ pairwise addition operations take place during this step. The same process is repeated two more times with the second and third order bits. At the end of three passes, each bucket contains the support count for the appropriate itemset, where the '0' for the itemset is replaced by a "don't care" which is represented by a '*'. Note that the number of transactions in this example is 27. This is represented by the entry for the bucket ***. Only 2 transactions contain all three items, which is represented by the bucket 111.

6. TREE PRUNING

In order to speed up the performance of the algorithm, we implement *lookaheads* [7]. Consider a node P with a set of prospective branches $F(P)$. If the node $P \cup F(P)$ is a frequent itemset, then it is not necessary to explore the subtree rooted at P. One way of doing this is to parallelize the process of finding the support count for the itemset $P \cup F(P)$ with that of determining the counts of each of the candidate extensions of P. We choose a simpler alternative by using the following method for performing lookaheads. Let $E = \{i_1, \ldots i_{|E|}\}$ be the set of frequent extensions of a node. Each time after calling the *DepthFirst* procedure from the candidate extension i_{r-1} of a node, we check if the set $\{i_{r-1} \ldots i_{|E|}\}$ was determined to be frequent. If so, the candidate extensions for $\{i_r \ldots i_{|E|}\}$ may be pruned from further consideration. This is a slightly weaker version of the pruning procedure, but it can be implemented much more efficiently. The depth first technique also provides the ability to quickly discover maximal patterns, and thereby prune away all those branches of the tree such that $P \cup E(P)$ is a subset of some itemset which has already been discovered. This kind of lookahead is more effective with a lexicographically branch-ordered depth first strategy, since longer patterns are discovered earlier on. In particular, any *frequent* strict superset Q of the itemset $P \cup E(P)$ contains P and at least one item i which is lexicographically smaller than the largest item in P (otherwise i would be contained in $E(P)$). This means that Q is lexicographically smaller than P. Thus Q (or a superset) would be discovered earlier than P.

The structure of the lexicographic tree is very much dependent upon the lexicographic ordering of the items in the database. For example, consider the case when there are exactly 3 large itemsets: *abc*, *abd*, and *abe*. Let us now consider the cases when the orderings of the items are a, b, c, d, e, and e, d, c, b, a respectively. The lexicographic trees for the two cases are illustrated in Figures 7(a) and (b). As we see, in the case of Figure 7(b), since the branches of the tree at which the maximal itemsets *eba*, *dba*, and *cba* can be found tend to separate out at very high level (level 1), the process of using lookaheads is likely to be more effective in this case. The very first node visited by the depth-first search procedure after the node *null* in the Figure 7(a) is a. In this case the lookahead process $\{a\} \cup \{b, c, d, e\}$ does not yield a frequent itemset. The same is true of the next level-1 node b. On the other hand, in the case of the Figure 7(b), the first node which is visited by the depth-first search proce-

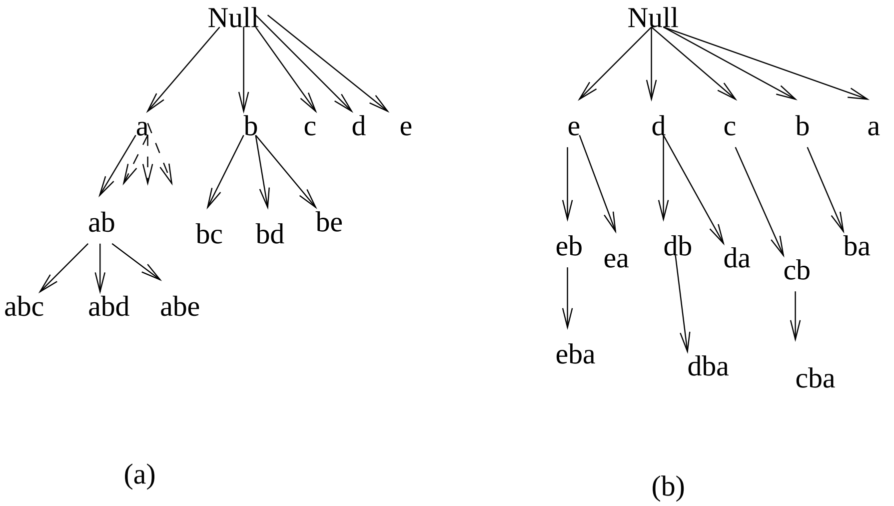


Figure 7: Illustrating the effect of using different orderings

dure is the node e, and the process of lookahead $e \cup \{b, a\}$ yields a frequent itemset. This example tends to suggest the following heuristic rule for choosing good orderings:

The item which occurs in the fewest number of large itemsets hanging at a node should be first and the item occuring in the maximum number of large itemsets should be last.

Unfortunately, we do not know the large itemsets apriori. Consequently, we found the strategy of ordering from least support to most support to be a reasonable approximation of the above goal. This technique has been discussed earlier by other researchers [7] in order to maximize the efficiency of lookaheads. The efficiency of the algorithm is improved further by using a dynamic ordering as opposed to a static ordering. In the case of dynamic orderings, we reorder the items below each node depending upon the support of each lexicographic tree-extension.

All itemsets generated in the *DepthProject* algorithm are created by either lookaheads, or by bucketing at the lower level nodes. Therefore, at the termination of the algorithm, we are left with a set of frequent itemsets which contains all frequent maximal itemsets with *some of the* subsets of maximal itemsets. Therefore, at the termination of the *DepthProject* algorithm, all non maximal itemsets are removed in a pruning phase.

6.1 Space Requirements

The space requirements of the depth first strategy are not significantly more than the size of the transaction set itself. This is because only one path of the lexicographic tree is being explored at any moment of time. The space required for maintaining all the sets of bitvectors is a small fraction of the transaction set size. If the geometric reduction rule for selective projection is used, and the database is projected only if it reduces by a factor of $Q > 1$, then the space requirements for all transaction sets on the current path are no larger than $Q/(Q-1)$ of the original transaction set size. Furthermore, we have already shown that for problems which are amenable to the depth first strategy, the transaction representation can be changed to a much more efficient form. This results in considerable savings.

7. 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.

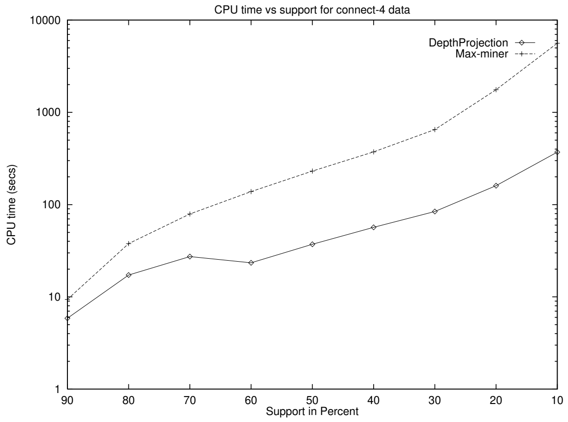
We tested the algorithm on a number of datasets which have been used earlier [7], for testing the algorithm in the case when it is possible to mine long patterns from the data. The algorithm proposed in [7] is very efficient for the case of finding long maximal patterns because of its ability to mine out the max-patterns by using look-ahead techniques.

The characteristics of the data sets are illustrated in the Table 1. The next to last column in this table illustrates the number of large 1-itemsets at the lowest support level. As we can see, the ratio of the average record width to $|L_1|$ in these data sets is quite high compared to typical retail data. This characteristic is true for most datasets which have long patterns in them. This ratio increases significantly at the lower level nodes, when the database is projected. The length of the longest pattern at the lowest support level is in the last column of the Table 1. As we can see, many of these patterns are so long that explicit enumeration of all subsets of these patterns would have been unrealistic.

In Figures 8(a) and 8(b), we have illustrated the performance of our algorithm versus the *MaxMiner* method on the *connect-4* and *mushroom* datasets. All plots are made on a logarithmic scale in order to illustrate the order-of-magnitude performance difference between the different al-

Table 1: Illustrating the features of the Data Sets

Data Set	Attributes	Average Width	Records	Dtbse Size (MB)	$\lvert L_1 \rvert$ (Lowest Support)	Longest pat. (Least Sup.)
connect-4	130	43	67557	12	73	29
mushroom	120	23	8124	0.85	115	21
chess	76	37	3196	0.51	54	23
Pumsb	7117	74	49046	10	147	63



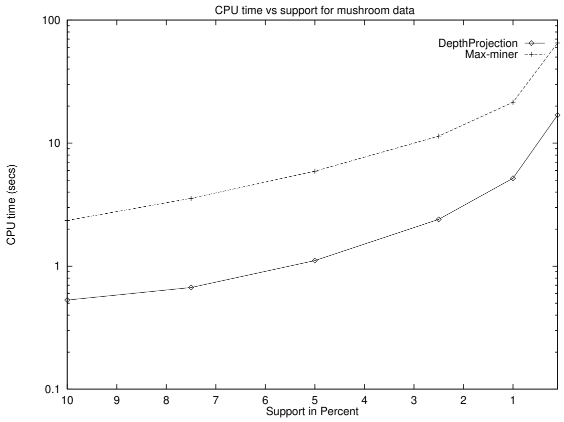
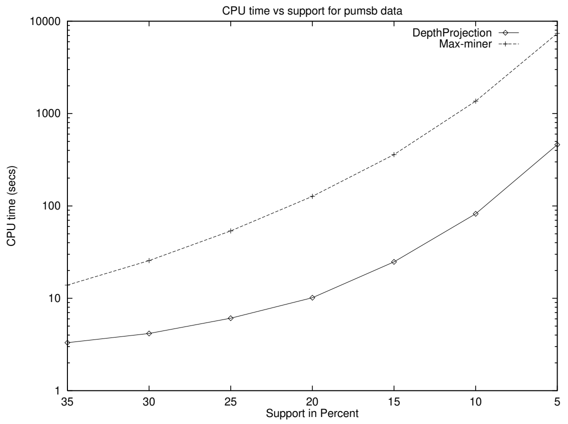


Figure 8: Computational performance on the (a) *connect-4* and (b) *mushroom* data sets



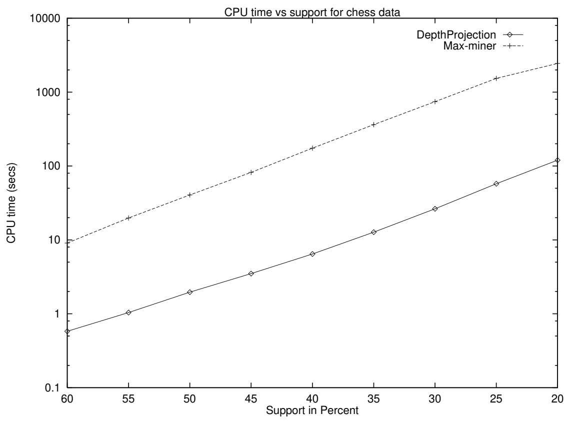


Figure 9: Computational performance on the (a) *pumsb* and (b) *chess* data sets

Table 2: Running Time in various phases of the algorithm at the lowest support level

Data Set	$T(-32)$	$T(+32)$	Counting	Pruning	Itemsets Found	Maximals
connect-4	7.74	347.55	355.29	8.62	131420	130986
mushroom	0.88	10.30	11.18	5.55	70321	27712
chess	0.13	89.84	89.97	30.58	589495	509355
Pumsb	21.98	401.55	423.53	33.48	313268	98140

gorithms. As we see, the *DepthProject* algorithm quickly outstrips the *MaxMiner* method when support values are low. For the *connect-4* dataset, at the highest support value of 90%, the *MaxMiner* algorithm required 9.4 seconds, while the *DepthProject* technique required 5.87 seconds. At the lowest support value of 10%, the *MaxMiner* algorithm required 5624.7 seconds, while the *DepthProject* algorithm required only 370.38 seconds. Thus, the performance gap increased at lower values of support, so that at the lowest support value, the performance gap between the two methods was a factor of 15.2.

The mushroom dataset also exhibits a large performance gap, though the difference is less dramatic in this case. At the highest support level of 10%, the *MaxMiner* algorithm required 2.35 seconds, whereas the *DepthProject* algorithm required 0.53 seconds. At the lowest support level[4] of 0.1%, *MaxMiner* algorithm required 65.19 seconds, whereas the *DepthProject* algorithm required 16.92 seconds. The performance gap between the two methods was a factor of 4 to 5 for all the support levels tested.

The most striking performance improvements were observed in the case of the *chess* and *pumsb* data sets. This is because these data sets correspond to cases when there are a large number of very long patterns. The performance numbers for the *pumsb* dataset are illustrated in Figure 9(a). At the highest support level of 35%, the *MaxMiner* algorithm required 13.91 seconds, whereas the *DepthProject* algorithm required 3.31 seconds. Thus, the performance gap at the highest support level was a factor of 4. At the lowest support level of 5%, *MaxMiner* algorithm required 461.19 seconds, while the *DepthProject* algorithm required 7378.6 seconds. This is a performance gap of approximately 16.

In the case of the *chess* dataset, which is illustrated in Figure 9(b), the most dramatic differences in performance were observed. At the highest support level of 60%, the *MaxMiner* algorithm required 9.08 seconds, whereas the *DepthProject* algorithm required 0.58 seconds. This is a performance gap of approximately 15.655. At the lowest support level of 20%, the *MaxMiner* algorithm required 2442.58 seconds, whereas the *DepthProject* algorithm required 120.06 seconds. Thus, at the lowest support level, the performance gap was a factor of 20.34.

The *DepthProject* algorithm required a postprocessing phase in which the non-maximal itemsets were removed. (The times illustrated include the postprocessing phase.) The time for this postprocessing phase depended upon the number of itemsets which were found by the tree generation and counting phase of the algorithm. As we can see from the the last two columns of Table 2, the number of itemsets found closely matches the maximal itemsets for the *chess* and *connect-4* data sets. This indicates that the process of lookaheads was quite effective in removing the non-maximal patterns quickly. The times for counting and pruning are also illustrated in the same table. (The column corresponding to the counting procedure may be obtained by adding the $T(+32)$ and $T(-32)$ columns, which will be explained below.)

[4]Note that the lowest support level for Figure 8(b) is 0.1%.

The implementation of the *DepthProject* algorithm assumed that the only time that a database was projected at a node was at the highest level of the tree at which the number of frequent extensions of the itemset became less than 32. It is instructive to compare the time taken in order to generate above the level of 32 bit projection to the time taken in order to generate the tree below the level of 32 bit projection. It is apparent from Table 2 that the time required in order to count below the 32 bit level ($T(+32)$) was substantially more than the time required to count above the 32 bit level ($T(-32)$) for all four data sets. Furthermore, since the most efficient counting of the algorithm on a *per node basis* was performed below the 32 bit level, it follows that most of the time was spent in such efficient counting. This also establishes that the efficient representation of the database at the lower levels of the tree contribute substantially to the savings in computational time.

8. CONCLUSIONS AND SUMMARY

This paper investigated a novel technique for mining long patterns in databases, which relies on a depth-first search technique in order to construct the lexicographic tree of itemsets. The use of depth-first search in reducing the number of itemsets is critical in being able to mine very long patterns at low values of support. The technique of depth-first search is optimum from the perspective of node pruning and performing the counting efficiently by using transaction projection. The depth first technique provides the ability to generate long patterns of the tree fast, and prune away the non-maximal patterns earlier on. At the same time, it provides the flexibility for node-specific processing methods such as bucketing. These characteristics explain the excellent computational properties of the *DepthProject* algorithm.

9. ACKNOWLEDGEMENTS

We would like to thank Rakesh Agrawal and Roberto Bayardo for providing us with several datasets for testing our algorithms. Roberto also provided us with access to his *MaxMiner* code for performing experiments.

10. REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad. A Tree Projection Algorithm for finding frequent itemsets. *Journal on Parallel and Distributed Computing*, to appear.

[2] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad. Depth First Generation of Large Itemsets for Association Rules. *IBM Research Report RC 21538*, July 1999.

[3] C. C. Aggarwal, P. S. Yu. Online Generation of Association Rules. *ICDE Conference Proceedings*, pages 402–411, 1998.

[4] C. C. Aggarwal, P. S. Yu. A New Framework for Itemset Generation. *ACM PODS Conference Proceedings*, pages 18–24, 1998.

[5] R. Agrawal, T. Imielinski, A. Swami. Mining Association Rules between Sets of Items in Very Large Databases. *ACM SIGMOD Conference Proceedings*, pages 207–216, 1993.

[6] R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules. *VLDB Conference Proceedings*, pages 487–499, 1994.

[7] R. J. Bayardo. Efficiently Mining Long Patterns from Databases. *ACM SIGMOD Conference Proceedings*, pages 85–93, 1998.

[8] R. J. Bayardo, R. Agrawal. Mining the Most Interesting Rules. *ACM SIGKDD Conference Proceedings*, pages 145–154, 1999.

[9] S. Brin, R. Motwani, J. D. Ullman, S. Tsur. Dynamic Itemset Counting and implication rules for Market Basket Data. *ACM SIGMOD Conference Proceedings*, pages 255–264, 1997.

[10] C. Silverstein, R. Motwani, S. Brin. Beyond Market Baskets: Generalizing Association Rules to Correlations. *ACM SIGMOD Conference Proceedings*, pages 265–276, 1997.

[11] B. Dunkel, N. Soparkar. Data Organization and Access for Efficient Data Mining. *ICDE Conference Proceedings*, pages 522–529, 1999.

[12] T. Fukuda, Y. Morimoto, S. Morishita, T. Tokuyama. Mining Optimized Association Rules for Numeric Attributes. *ACM PODS Conference Proceedings*, pages 182–191, 1996.

[13] T. Fukuda, Y. Morimoto, S. Morishita, T. Tokuyama. Data mining using Two-dimensional Optimized Association Rules for Numeric Attributes: Scheme, Algorithms, Visualization. *ACM SIGMOD Conference Proceedings*, pages 13–23, 1996.

[14] D. Gunopulos, H. Mannila, S. Saluja. Discovering All Most Specific Sentences by Randomized Algorithms. *ICDT Conference Proceedings*, pages 215–229, 1997.

[15] J. Han, J. Pei, Y. Yin. Mining Frequent Patterns without Candidate Generation. *ACM SIGMOD Conference Proceedings*, pages 1–12, 2000.

[16] C. Hidber. Online Association Rule Mining. *ACM SIGMOD Conference Proceedings*, pages 145–156, 1999.

[17] D. Lin, Z. M. Kedem. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Itemset. *EDBT Conference Proceedings*, pages 105–119, 1998.

[18] H. Mannila, H. Toivonen, A. I. Verkamo. Efficient algorithms for discovering association rules. *AAAI Workshop on KDD*, 1994.

[19] M. Klementtinen, H. Mannila, P. Ronkainen, H. Toivonen, A. I. Verkamo. Finding Interesting Rules from Large Sets of discovered association rules. *CIKM Conference Proceedings*, pages 401–407, 1994.

[20] L. V. S. Lakshmanan, R. Ng, J. Han, A. Pang. Optimization of Constrained Frequent Set Queries with 2-variable Constraints. *ACM SIGMOD Conference Proceedings*, pages 157–168, 1999.

[21] B. Nag, P. M. Deshpande, D. J. DeWitt. Using a Knowledge Cache for Interactive Discovery of Association Rules. *ACM SIGKDD Conference Proceedings*, pages 244–253, 1999.

[22] R. Rastogi, K. Shim. Mining Optimized Association Rules for categorical and numeric attributes. *ICDE Conference Proceedings*, pages 503–512, 1998.

[23] R. Rastogi, K. Shim. Mining Optimized Support Rules for Numeric Attributes. *ICDE Conference Proceedings*, pages 126–135, 1999.

[24] I. Rigoutsos, A. Floratos. Combinatorial Pattern Discovery in Biological Sequences. *Bioinformatics*, 14(1): pages 55–67, 1998.

[25] R. Rymon. Search Through Systematic Set Enumeration. *International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[26] A. Savasere, E. Omiecinski, S. B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. *VLDB Conference Proceedings*, pages 432–444, 1995.

[27] R. Srikant, R. Agrawal. Mining Generalized Association Rules. *VLDB Conference Proceedings*, pages 407–419, 1995.

[28] R. Srikant, R. Agrawal. Mining Quantitative Association Rules in Large Relational Tables. *ACM SIGMOD Conference Proceedings*, pages 1–12, 1996.

[29] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li. New Algorithms for Fast Discovery of Association Rules. *KDD Conference Proceedings*, pages 283–286, 1997.