Towards Long Pattern Generation in Dense Databases

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

ABSTRACT

This paper discusses the problem of long pattern generation in dense databases. In recent years, there has been an increase of interest in techniques for maximal pattern generation. We present a survey of this class of methods for long pattern generation which differ considerably from the level-wise approach of traditional methods. Many of these techniques are rooted in combinatorial tricks which can be applied only when the generation of frequent patterns is not forced to be level wise. We present an overview of the different kinds of methods which can be used in order to improve the counting and search space exploration methods for long patterns.

Keywords

Long Pattern Generation, Association Rules

1. INTRODUCTION

The association rule problem was introduced by Agrawal et al. [3] and has been recognized in the literature as a fundamentally important problem in the field of data mining. The problem of association rules is defined on a database which is composed of a set of binary records called transactions. Each transaction contains information about the presence or absence of items. For example, in a supermarket application, a transaction could consist of the set of items which the customer bought. It is often useful to find the nature of the relationships among the buying patterns of the different items. Such relationships can be modeled as association rules [3]. Applications of association rules extend to finding useful patterns in consumer behavior, target marketing, and electronic commerce. A key step of association rule mining is finding *frequent itemsets* or *large itemsets* [3]. These are sets of items whose support (or fractional presence) is larger than a user-specified threshold. A prominent algorithm for frequent itemset generation is the *Apriori* technique [4], which works quite well when the frequent itemsets are reasonably short. One property of this algorithm is that for each frequent itemset, all subsets of it need to be generated by the algorithm. This can be quite compute-intensive, when the patterns are long. The *Apriori*-method is the outline upon which many frequent itemset generation algorithms in the literature are based. Therefore, such methods suffer from the same shortcomings for dense databases which contain long patterns.

Considerable research has been devoted towards finding faster methods for generating large itemsets [4; 5; 9; 10; 11; 16; 17; 20; 27]. 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; 4]. An interesting analysis of the impact of different kinds of data on access costs has been provided in [9]. 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. 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. For such cases, it becomes important to develop techniques and algorithms which are more focussed on the computational aspect rather than the I/O costs. Such techniques are inherently combinatorial and work well only with main memory databases. In this paper, we also discuss the challenges of extending these ideas to larger databases.

Some of the algorithms in the literature such as *MaxMiner* avoid the problem of generating all possible sub-patterns of frequent itemsets by implementing *lookaheads* [5], in which supersets of frequent patterns are used in order to prune off potential candidates in the search. Other innovative ideas for handling the long pattern case are discussed in [27]. In spite of these advances [5; 9; 27], finding computationally efficient algorithms for generating long patterns continues to be a very difficult problem.

The focus of most algorithms in the literature is on *level-wise* pattern generation: in other words $(k+1)$-itemsets are generated only after all k-itemsets have been generated. The *Apriori* algorithm and its variants satisfy this property. Even the recent look-ahead-based algorithm discussed [5] 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 [7] 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 [5]. 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. In fact, many domains of data such as computational biology have data sets which show this property [18].

2. CLOSED FREQUENT ITEMSETS

An itemset X is closed if there does not exist another itemset X' such that X' is a superset of X, but the transactions which contain X and X' are exactly the same. It is clear that the set of frequent closed itemsets may be orders of magnitude smaller than the set of all itemsets. For example, in a database with a single transaction of 50 items, there are 2^{50} frequent itemsets with 100% support, but there is only one frequent closed itemset, which is the original transaction. A good number of algorithms have been developed for frequent closed itemset generation [12; 13; 26].

CHARM: This is an efficient algorithm [26] for enumerating the set of all closed frequent itemsets. The unique feature of this algorithm is that it simultaneously explores both the itemset space and transaction space. This feature helps it to quickly identify the closed frequent itemsets without having to enumerate many non-closed subsets. The algorithm has been found to scale linearly in terms of number of itemsets and the number of closed itemsets found. Another related method has recently been proposed for finding non-redundant association rules [25]. It has been shown in the same work that the number of non-redundant rules produced by this approach is exponentially smaller than the traditional approach. Therefore, it can often be used in order to mine dense databases.

CLOSET: An interesting class of methods has recently been proposed by Han et al. [11], which find frequent itemsets without candidate generation. This techniques represents the original database in the form of an FP-Tree structure which greatly reduces the time for subsequent mining. The FP-Tree structure can be used in order to find either the set of all itemsets [11] or the set of all frequent closed itemsets [13]. It has been shown in [13], that the CLOSET algorithm outperforms the CHARM algorithm considerably.

We note that even though the set of closed frequent itemsets is much smaller than the set of all itemsets, it cannot compare with the set of *maximal itemsets*. An itemset is defined to be a *frequent maximal itemset*, if it is frequent, and no superset of that itemset is frequent. In many cases, the set of maximal itemsets is orders of magnitude smaller than the set of closed itemsets. In such cases, the frequent closed itemset algorithms spend a lot of time exploring redundant parts of the search space. Often, it may be desirable to use methods which are more restrictive in exploring parts of the search space in which no superset of any generated pattern is also frequent. The FP-Tree method [11] can also be extended to finding maximal itemsets.

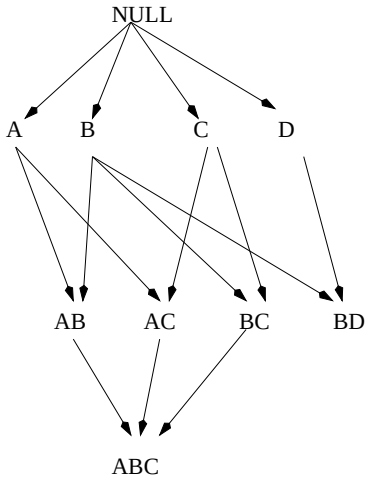

Figure 1: The itemset lattice

3. LATTICE BASED METHODS

Frequent Patterns satisfy an important property which is referred to as the *closure property*. The closure property is that every subset of a frequent pattern is also a frequent pattern in of itself. This property helps us organize the frequent itemsets in the form of a lattice structure. In a lattice of frequent itemsets, a node exists for each frequent itemset. An edge exists between a pair of nodes, when the itemset corresponding to one is smaller than the itemset corresponding to the other by one item. An example of the adjacency lattice is illustrated in Figure 1.

The adjacency lattice induces a natural graph structure on the set of frequent itemsets. This graph structure can be exploited in order to develop more efficient frequent itemset algorithms. Two interesting maximal pattern generation algorithms are *MaxClique* and *MaxEclat* [24; 27] which divide the lattice into cliques and mine them bottom up with a vertical database representation. A variety of strategies are discussed in [24] for exploring the lattice of frequent itemsets. One of the drawbacks of this technique is that it relies on a pre-processing method which restricts future applicability.

Another interesting lattice based method is pincer-search [16]. This method uses Apriori-like candidate generation along with another method for finding long candidate itemsets used for superset frequency pruning. The primary idea in pincer-search is to combine a bottom-up search along with a restricted top-down search methodology. The restricted top-down search is used to prune candidates in the bottom-up search. The long candidate itemset generation procedure of pincer-search is somewhat similar to the *MaxClique* method [27]. This method is however not quite as effective as the *MaxMiner* method [5] which falls in the class of tree based methods.

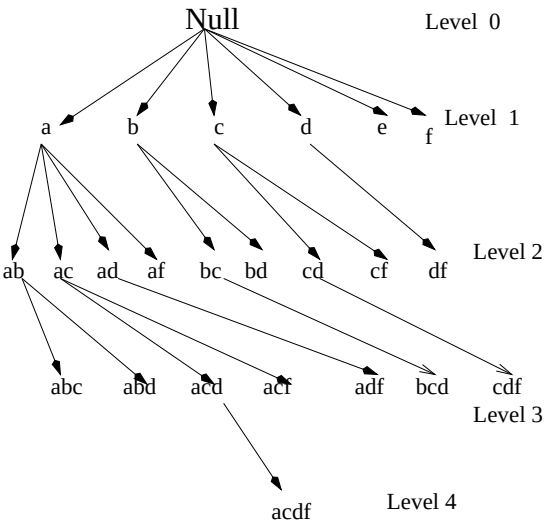

Figure 2: The lexicographic tree

4. TREE BASED METHODS

Just as the frequent itemsets can be arranged in the form of a lattice structure, they can also be arranged in a tree structure by taking into account a particular ordering of the items. This property helps us organize frequent itemsets in the form of a conceptual representation referred to as the lexicographic tree [1]. A set representation of this structure is also referred to as the enumeration tree [5; 19; 23]. Since this conceptual representation is behind the development of a host of recent methods for long pattern generation, we will provide some more understanding of this structure.

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. It is easy to see that the lexicographic tree structure is very closely related to the lattice representation of itemsets (one is a subgraph of the other). An example of the lexicographic tree is illustrated in Figure 2. 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 $E(P)$. In the example illustrated in Figure 2, 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 2, 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.

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. It is important to note that 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 $T(P)$.

We note that the tree representation provides a convenient conceptual understanding of the itemsets which is possible to exploit in order to improve the efficiency of frequent pattern generation. It is possible to explore this tree structure in a variety of ways which leads of different kinds of tradeoffs. The most straightforward generation of the tree structure is by using a simple breadth first approach [5]. This method has the advantage of reducing the number of database passes in the case of a large database. However, when one is focussed on main memory pattern generation, such an advanatge may not be quite as useful.

The FP-Tree structure of [11] should not be confused with the lexicographic tree structure which is quite different. The lexicographic tree structure is a conceptual structure which is tailored towards intelligent candidate exploration, whereas the FP-Tree structure is built on the original set of transactions. In fact, the FP-Tree method finds frequent itemsets without candidate generation.

4.1 Intelligent Tree Exploration

We note that the lexicographic tree can be explored in a variety of ways. The most simplistic one is the breadth-first method which was first proposed in [5]. Another breadth-first technique with efficient counting methods has been discussed in [1]. This technique provides efficient use of cache locality and works well for large databases, but is relevant only for the case of finding all patterns rather than maximal patterns. A significantly more interesting exploration of the tree is in depth-first order which was introduced in [2]. This technique is specifically designed for finding long patterns in dense databases. Another interesting version

of the depth-first approach has been presented in [8]. In depth-first search [2], 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 2 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.

The depth-first exploration technique of [2] is combined with effective transaction projection. Here the idea is to hierarchically project the transactions at all or some of the nodes in the lexicographic tree as it is being generated. 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. An important fact about hierarchical projections is that *we are effectively reusing the information from counting k-itemsets in order to count $(k+1)$-itemsets.*

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. We note that the applicability of the projection methodology extends well beyond frequent pattern mining, and can be used for problems such as mining sequential patterns. A slightly different kind of pattern projection has also been introduced in [11] and has been used in order to efficiently mine sequential patterns [14].

It is important to understand that the use of hierarchical projection in the context of a depth first strategy creates *problem decomposition*; here each node is a *completely independent* itemset generation problem rooted at that node. The independence property is a very desirable one, since we are free to use whatever counting strategy is *most suitable* to the data characteristics at that particular node. For example, a specialized counting method called bucketing [2] can be applied only when the number of frequent extensions at a node P falls below a certain number.

As mentioned earlier, the search space exploration can be significantly reduced with the help of *lookaheads* [5]. However, it turns out that not all strategies have the same level of efficiency in terms of implementation of lookaheads. The advantages of a depth first strategy for effective pruning have been discussed in [2; 8]. 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. 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. Thus, in a depth first strategy, in order to determine whether a frequent itemset should be removed because of non-maximality, we only need to look at itemsets which have been generated *earlier* [8]. Furthermore, it has been shown in [8] that if a superset of the currently generated itemset is frequent, then it is usually present in the last 50 or so itemsets generated. This leads to a very efficient algorithm for pruning away a newly generated itemset.

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 3(a) and (b). It is interesting to note that in one of the cases, the tree structure tends to be much more bushy, whereas in another case the branches separate out quickly. Specifically, in the case of Figure 3(b), the the maximal itemsets eba, dba, and cba separate out at very high level (level 1) of the tree. Therefore, 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 3(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 3(b), the first node which is visited by the depth-first search procedure is the node e, and the process of lookahead $e \cup \{b, a\}$ yields a frequent itemset. This example tends to suggest that *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.* Since we do not know the large itemsets apriori, the strategy of ordering from least support to most support is often a reasonable approximation of the above goal [2; 5; 8]. 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.

Another interesting kind of pruning proposed in [8] is the parent equivalence pruning (PEP). Let P be a node in the lexicographic tree, and $E(P)$ be the set of frequent extensions of P. Let i be an item in $E(P)$. The idea here is that when the node P and $P \cup \{i\}$ have the same set of projected transactions, then we only need to explore only those itemsets rooted at P which necessarily contain the item i. This

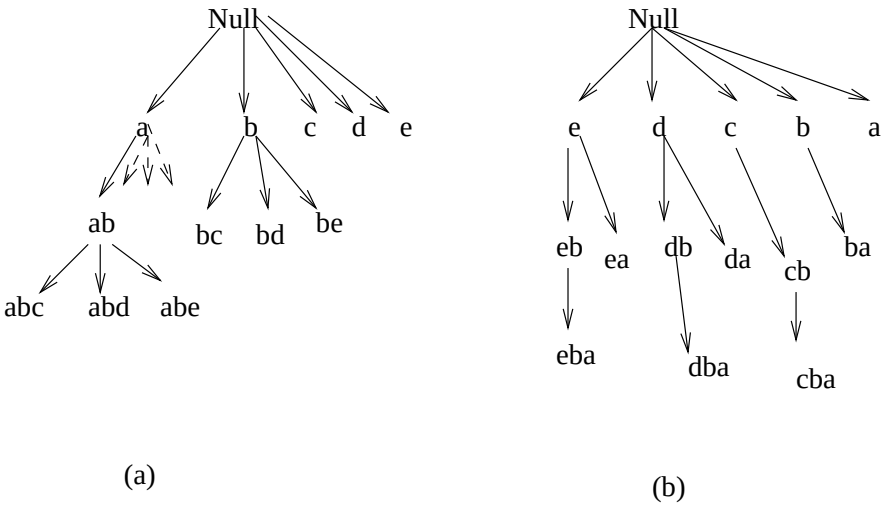

Figure 3: Illustrating the effect of using different orderings

kind of pruning can significantly reduce the search space which needs to be explored.

5. INTELLIGENT COUNTING TECHNIQUES

It turns out that the same strategies and data structures (eg hash tree) [4] which are most useful for counting sparse databases are no longer so useful in the case of dense data bases. In this case, even the initial representation of the data significantly affects the quality of the search. The counting methods can be built around either a horizontal representation of the database or a vertical representation.

5.1 Horizontal Counting Methods

In [2] it has been shown that it was better to use the database in the *bitstring representation* for the dense case. 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 projected 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. This is not the case in dense databases for which most bits take on the value of 1. Furthermore, the density increases even further for lower level nodes and therefore the efficiency of the bit representation increases further.

The projection techniques of tree based methods can be combined eith the bit string representation in order to improve the counting times substantially [2]. Let us consider a node P, at which it is desirable to count the support of each item in $F(P)$. Let T be a transaction in $\mathcal{T}(P)$. 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 the bit vector representation of the transactions [2; 8].

We assume that each transaction $T \in \mathcal{T}(P)$ contains n bits. Therefore, it can 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 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.

5.2 Vertical Counting Techniques

In vertical counting methods the database is represented in a fundamentally different format. Here, for each item in the database, we maintain a vertical list of all the transaction identifiers in which it occurs [9; 21; 27]. This technique can be used to some advantage in order to reduce the counting time. In particular, it has been shown [9], that the disk access times for a counting method with the vertical representation are substantially less for the dense case. However, this observation is not quite as relevant while performing the mining in the context of a main memory database.

However, we note that some advantages of the vertical representation or a mixed representation can be harnessed very effectively in tree based methods as well [2; 8]. For example, in [2], the horizontal transaction projection techniques are combined with a vertical bit vector maintenance in order to improve the counting techniques. At any point in the search,

we maintain the projected transaction sets for only *some* of the nodes on the path from the root to the node which is currently being extended. A pointer is maintained at each node P to the projected transaction set which is available at the nearest ancestor Q of P at which such a set is indeed maintained. We also maintain 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 keep the process of support counting more efficient while reducing the number of times the database needs to be projected.

5.3 Specialized Counting Techniques

An interesting property of the tree based approach is that each node is a completely independent itemset generation problem for a subset of the database and a subset of the items. Therefore, the particular characteristics of the problem at different nodes can be exploited in order to develop more effective counting techniques. 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. *At these levels the database contains only a small number of items.* This is because $E(P)$ is small at the lower level nodes. For these levels, it is possible to use a strategy called bucketing [2] 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:

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. 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. However, it is possible to be more skillful in performing the second phase of this operation.

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 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*. 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.

6. RELATED PROBLEMS

The depth-first techniques discussed in [2; 8] are almost two orders of magnitude faster than the *MaxMiner* algorithm in terms of CPU efficiency. Therefore, there is considerable incentive in leveraging the techniques discussed in [2; 8] even for the case of disk resident databases. We note that a method discussed in [20] 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. However, the method in [20] is developed for the case when the patterns are reasonably small, and is focussed on finding all patterns rather than maximal patterns only. For the case of finding maximal patterns in dense databases, the problem of recombining the itemsets from different partitions becomes significantly more challenging. We note that in the long pattern case, the post processing procedure of [20] becomes the bottleneck and cannot be used directly. This is an interesting open problem which may be explored in future work. It is clear that if each main memory partition is large enough, then the set of maximal itemsets from a given partition are likely to approximate the true maximal itemsets very closely. In such a case, we conjecture that it may be possible to efficiently determine the set of maximal itemsets by using the information gained from the different partitions.

An interesting direction of research for the dense case is to find rules or patterns which satisfy user-specified constraints [6]. It may often be the case that it is significantly more efficient to find frequent patterns in dense databases, when the

constraints are taken into account during the mining process. An interesting approach in this direction for the dense case is discussed in [6]. Another recent technique for constraint based data mining has been discussed in [15]. This method is based on the FP-Tree technique and is likely to work well for dense databases.

7. CONCLUSIONS AND SUMMARY

In this paper, we presented some of the recent tree based algorithms which have been developed for mining long patterns. These methods are combinatorial in nature and are quite different from the level-wise techniques for the short-pattern case. These techniques are based on databases in main memory, but have potential for being extended to larger disk-resident databases as well. We also discussed important variations of frequent pattern generation in which user-specified constraints are allowed.

8. REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad. A Tree Projection Algorithm for generation of frequent itemsets. *Journal on Parallel and Distributed Computing*, Vol. 61, No. 3, pp. 350-371, March 2001.

[2] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad. Depth First Generation of Long Patterns. *Proceedings of the ACM SIGKDD Conference*, 2000.

[3] 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.

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

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

[6] R. J. Bayardo, R. Agrawal, D. Gunopulos. Constraint-Based Rule Mining in Large Dense Databases. *ICDE Conference Proceedings*, 1999.

[7] S. Brin, R. Motwani, J. D. Ullman, S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. *ACM SIGMOD Conference Proceedings*, 1997.

[8] D. Burdick, M. Calimlim, J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. *Proceedings of the ICDE Conference*, 2001.

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

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

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

[12] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. *ICDT Conference Proceedings*, 1999.

[13] J. Pei, J. Han, R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. *DMKD*, 2000.

[14] J. Han, J. Pei, B. Mortazavi, Q. Chen, U. Dayal, M.-C. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. *Proceedings of the ACM KDD Conference*, 2000.

[15] J. Pei, J. Han, L. Lakshmanan. Mining Frequent Itemsets with Convertible Constraints. *Proceedings of the ICDE Conference*, 2001.

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

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

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

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

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

[21] P. Shenoy et al. Turbo-charging Vertical Mining of Large Databases. *ACM SIGMOD Conference Proceedings*, 2000.

[22] H. Toivonen. Sampling Large Databases for Association Rules. *VLDB Conference Proceedings*, pages 134–145, 1996.

[23] G. I. Webb. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:45-83, 1996.

[24] M. J. Zaki. Scalable Algorithms for Association Rule Mining. *IEEE TKDE Journal*, 12(3), pp. 372-390, May/June 2000.

[25] M. J. Zaki. Generating non-redundant association rules. *Proceedings of the ACM SIGKDD Conference*, 2000.

[26] M. J. Zaki, C. Hsiao. CHARM: An Efficient Algorithm for Closed Association Rule Mininf. *Technical Report*, RPI, 1999.

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