New Algorithms for Fast Discovery of Association Rules *

Mohammed Javeed Zaki, Srinivasan Parthasarathy,
Mitsunori Ogihara, and Wei Li
Computer Science Department
University of Rochester, Rochester NY 14627
{zaki,srini,ogihara,li}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 651

July 1997

Abstract

Association rule discovery has emerged as an important problem in knowledge discovery and data mining. The association mining task consists of identifying the frequent itemsets, and then forming conditional implication rules among them. In this paper we present efficient algorithms for the discovery of frequent itemsets, which forms the compute intensive phase of the task. The algorithms utilize the structural properties of frequent itemsets to facilitate fast discovery. The related database items are grouped together into clusters representing the potential maximal frequent itemsets in the database. Each cluster induces a sub-lattice of the itemset lattice. Efficient lattice traversal techniques are presented, which quickly identify all the true maximal frequent itemsets, and all their subsets if desired. We also present the effect of using different database layout schemes combined with the proposed clustering and traversal techniques. The proposed algorithms scan a (pre-processed) database only once, addressing the open question in association mining, whether all the rules can be efficiently extracted in a single database pass. We experimentally compare the new algorithms against the previous approaches, obtaining improvements of more than an order of magnitude for our test databases.

The University of Rochester Computer Science Department supported this work.

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract (F19628-94-C-0057).

1 Introduction

Knowledge discovery and data mining (KDD) is an emerging field, whose goal is to make sense out of large amounts of collected data, by discovering hitherto unknown patterns. One of the central KDD tasks is the discovery of association rules [1]. The prototypical application is the analysis of supermarket sales or *basket* data [2, 5, 3]. Basket data consists of items bought by a customer along with the transaction date, time, price, etc. The association rule discovery task identifies the group of items most often purchased along with another group of items. For example, we may obtain a rule of the form "80% of customers who buy bread and milk, also buy butter and eggs at the same time". Besides the retail sales example, association rules have been useful in predicting patterns in university enrollment, occurrences of words in text documents, census data, and so on.

The task of mining association rules over basket data was first introduced in [2], which can be formally stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be the set of database *items*. Each transaction, T, in the database, $\mathcal{D}$, has a unique identifier, and *contains* a set of items, called an *itemset*. An itemset with k items is called a *k-itemset*. A subset of k elements is called a k-subset. The *support* of an itemset is the percentage of transactions in $\mathcal{D}$ that contain the itemset. An *association rule* is a conditional implication among itemsets, $A \Rightarrow B$, where itemsets $A, B \subset \mathcal{I}$, and $A \cap B = \emptyset$. The *confidence* of the association rule, given as $support(A \cup B)/support(A)$, is simply the conditional probability that a transaction contains B, given that it contains A. The data mining task for association rules can be broken into two steps. The first step consists of finding all *frequent* itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called *minimum support*. The second step consists of forming the rules among the frequent itemsets. The problem of identifying all frequent itemsets is the computationally intensive step in the algorithm. Given m items, there are potentially 2^m frequent itemsets, which form a *lattice of subsets* over $\mathcal{I}$. However, only a small fraction of the whole lattice space is frequent. This paper presents efficient methods to discover these frequent itemsets. The rule discovery step is relatively easy [3]. Once the support of frequent itemsets is known, rules of the form $X - Y \Rightarrow Y$ (where $Y \subset X$), are generated for all frequent itemsets X, provided the rules meet a desired confidence. Although this step is easy, there remain important research issues in presenting "interesting" rules from the large set of generated rules. If the important issue in the first step is that of "quantity" or performance, due to its compute intensive nature, the dominant concern in the second step is that of "quality" of rules that are generated. See [26, 18] for some approaches to this problem. In this paper we only consider the frequent itemsets discovery step.

Related Work Several algorithms for mining associations have been proposed in the literature [2, 21, 5, 16, 23, 17, 27, 3, 31]. Almost all the algorithms use the *downward closure* property of itemset support to prune the itemset lattice – the property that all subsets of a frequent itemset must themselves be frequent. Thus only the frequent k-itemsets are used to construct *candidate* or *potential frequent* $(k+1)$-itemsets. A pass over the database is made at each level to find the frequent itemsets. The algorithms differ to the extent that they prune the search space using efficient candidate generation procedure. The first algorithm AIS [2] generates candidates on-the-fly. All frequent itemsets from the previous transaction that are contained in the new transaction are extended with other items in that transaction. This results in to many unnecessary candidates. The *Apriori* algorithm [21, 5, 3] which uses a better candidate generation procedure was shown to be superior to earlier approaches [2, 17, 16]. The DHP algorithm [23] collects approximate support of candidates in the previous pass for further pruning, however, this optimization may be detrimental to performance [4]. All these algorithms make multiple passes over the database, once for each iteration k. The *Partition* algorithm [27] minimizes I/O by scanning the database only twice. It partitions the database into small chunks which can be handled in memory. In the first pass it generates the set of all potentially frequent itemsets (any itemset locally frequent in a partition), and in the second pass their global support is obtained. Another way to minimize the I/O overhead is to work with only a small sample of the database. An analysis of the effectiveness of sampling for association mining was presented in [34], and [31] presents an exact algorithm that finds all rules using sampling. The recently proposed DIC algorithm [9] dynamically counts candidates of varying length as the database scan progresses, and thus is able to reduce the number of scans. Approaches using only general-purpose DBMS systems and relational algebra operations have been studied [16, 17], but these don't compare favorably with the specialized approaches. A number of parallel algorithms have also been proposed [24, 4, 32, 11, 14, 33].

All the above solutions are applicable to only binary data, i.e., either an item is present in a transaction or it isn't. Other extensions of association rules include mining over data where the quantity of items is also considered [28], or mining for rules in the presence of a taxonomy on items [6, 15]. There has also been work in finding frequent sequences of itemsets over temporal data [6, 22, 29, 20].

1.1 Contribution

The main limitation of almost all proposed algorithms [2, 5, 21, 23, 3] is that they make repeated passes over the disk-resident database partition, incurring high I/O overheads. Moreover, these algorithms use complicated hash structures which entails additional overhead in maintaining and searching them, and they typically suffer from poor cache locality [25]. The problem with *Partition*, even though it makes only two scans, is that, as the number of partitions is increased, the number of locally frequent itemsets increases. While this can be reduced by randomizing the partition selection, results from sampling experiments [34, 31] indicate that randomized partitions will have a large number of frequent itemsets in common. *Partition* can thus spend a lot of time in performing redundant computation.

Our work contrasts to these approaches in several ways. We present new algorithms for fast discovery of association rules based on our ideas in [33, 35]. The proposed algorithms scan the pre-processed database exactly once greatly reducing I/O costs. The new algorithms are characterized in terms of the clustering information used to group related itemsets, and in terms of the lattice traversal schemes used to search for frequent itemsets. We propose two clustering schemes based on equivalence classes and maximal uniform hypergraph cliques, and we utilize three lattice traversal schemes, based on bottom-up, top-down, and hybrid top-down/bottom-up search. We also present the effect of using different database layouts

– the horizontal and vertical formats. The algorithms using the vertical format use simple intersection operations, making them an attractive option for direct implementation on general purpose database systems. Extensive experimental results are presented contrasting the new algorithms with previous approaches, with gains of over an order of magnitude using the proposed techniques.

The rest of the paper is organized as follows. Section 2 provides details of the itemset clustering techniques, while section 3 describes the lattice traversal techniques. Section 4 gives a brief overview of the KDD process and the data layout alternatives. The *Apriori*, *Partitin* and six new algorithms employing the above techniques are described in section 5. Our experimental study is presented in in section 6, and our conclusions in section 7.

2 Itemset Clustering

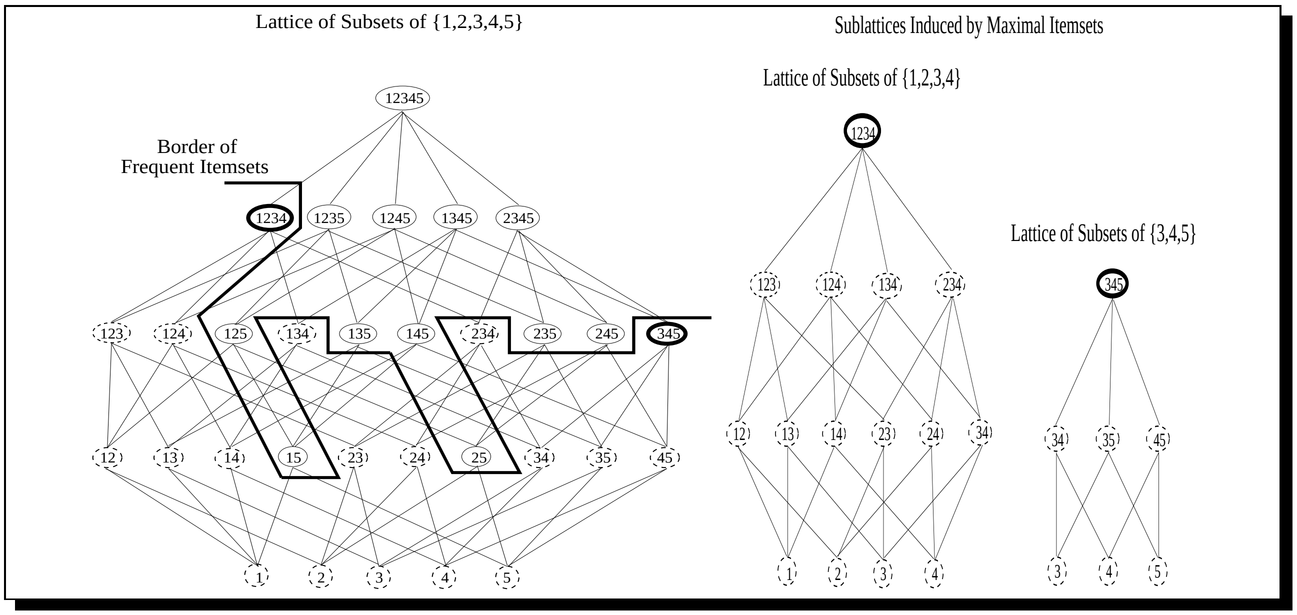

Lattice of Subsets of {1,2,3,4,5}
Border of Frequent Itemsets
12345
1234
1235
1245
1345
2345
123
124
125
134
135
145
234
235
245
345
12
13
14
15
23
24
25
34
35
45
1
2
3
4
5
Sublattices Induced by Maximal Itemsets
Lattice of Subsets of {1,2,3,4}
Lattice of Subsets of {3,4,5}

Figure 1: Lattice of Subsets and Maximal Itemset Induced Sub-lattices

We will motivate the need for itemset clustering by means of an example. Consider the lattice of subsets of the set $\{1, 2, 3, 4, 5\}$, shown in figure 1 (the empty set has been omitted in all figures). The frequent itemsets are shown with dashed circles and the two *maximal* frequent itemsets (a frequent itemset is *maximal* if it is not a proper subset of any other frequent itemset) are shown with the bold circles. Due to the downward closure property of itemset support – the fact that all subsets of a frequent itemset must be frequent – the frequent itemsets form a *border*, such that all frequent itemsets lie below the border, while all infrequent itemsets lie above it. The border of frequent itemsets is shown with a bold line in figure 1. An optimal association mining algorithm will only enumerate and test the frequent itemsets, i.e., the algorithm must efficiently determine the structure of the border. This structure is precisely determined by the maximal frequent itemsets. The border corresponds to the sub-lattices induced by the maximal frequent itemsets. These sub-lattices are shown in figure 1.

Given the knowledge of the maximal frequent itemsets we can design an efficient algorithm that simply gathers their support and the support of all their subsets in just a single database pass. In general we cannot precisely determine the maximal itemsets in the intermediate steps of the algorithm. However we can approximate this set. Our itemset clustering techniques are designed to group items together so that we obtain supersets of the maximal frequent itemsets – the *potential maximal frequent itemsets*. Below we present two schemes to generate the set of potential maximal itemsets based on equivalence classes and maximal uniform hypergraph cliques. These two techniques represent a trade-off in the precision of the potential maximal itemsets generated, and the computation cost. The hypergraph clique approach gives more precise information at higher computation cost, while the equivalence class approach sacrifices quality for a lower computation cost.

2.1 Equivalence Class Clustering

Let's consider the candidate generation step of *Apriori*. The candidates for the k-th pass are generated by joining L_{k-1}, the set of frequent $(k-1)$-itemsets with itself, which can be expressed as: $C_k = \{X = A[1]A[2]...A[k-1]B[k-1]\}$, for all $A, B \in L_{k-1}$, with $A[1:k-2] = B[1:k-2]$, and $A[k-1] < B[k-1]$, where $X[i]$ denotes the i-th item, and $X[i:j]$ denotes items at index i through j in itemset X. Let $L_2 = \{$AB, AC, AD, AE, BC, BD, BE, DE$\}$. Then $C_3 = \{$ ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE$\}$. Assuming that L_{k-1} is lexicographically sorted, we can partition the itemsets in L_{k-1} into *equivalence classes* based on their common $k-2$ length prefixes, i.e., the equivalence class $a \in L_{k-2}$, is given as:

$$S_a = [\mathbf{a}] = \{b[k-1] \in L_1 \mid \ a[1:k-2] = b[1:k-2]\}$$

Candidate k-itemsets can simply be generated from itemsets within a class by joining all $\binom{|S_i|}{2}$ pairs, with the the class identifier as the prefix. For our example L_2 above, we obtain the equivalence classes: $S_A = [\mathbf{A}] = \{$B, C, D, E$\}$, $S_B = [\mathbf{B}] = \{$C, D, E$\}$, and $S_D = [\mathbf{D}] = \{$E$\}$. We observe that itemsets produced by the equivalence class $[\mathbf{A}]$, namely those in the set $\{$ABC, ABD, ABE, ACD, ACE, ADE$\}$, are independent of those produced by the class $[\mathbf{B}]$ (the set $\{$BCD, BCE, BDE$\}$. Any class with only 1 member can be eliminated since no candidates can be generated from it. Thus we can discard the class $[\mathbf{D}]$. This idea of partitioning L_{k-1} into equivalence classes was independently proposed in [4, 32]. The equivalence partitioning was used in [32] to parallelize the candidate generation step. It was also used in [4] to partition the candidates into disjoint sets.

At any intermediate step of the algorithm when the set of frequent itemsets, L_k for $k \geq 2$, has been determined we can generate the set of potential maximal frequent itemsets from L_k. Note that for $k = 1$ we end up with the entire item universe as the maximal itemset. However, For any $k \geq 2$, we can extract more precise knowledge about the association among the items. The larger the value of k the more precise the clustering. For example, figure 2 shows the equivalence classes obtained for the instance where $k = 2$. Each equivalence class is a potential maximal frequent itemset. For example, the class [**1**], generates the maximal itemset 12345678.

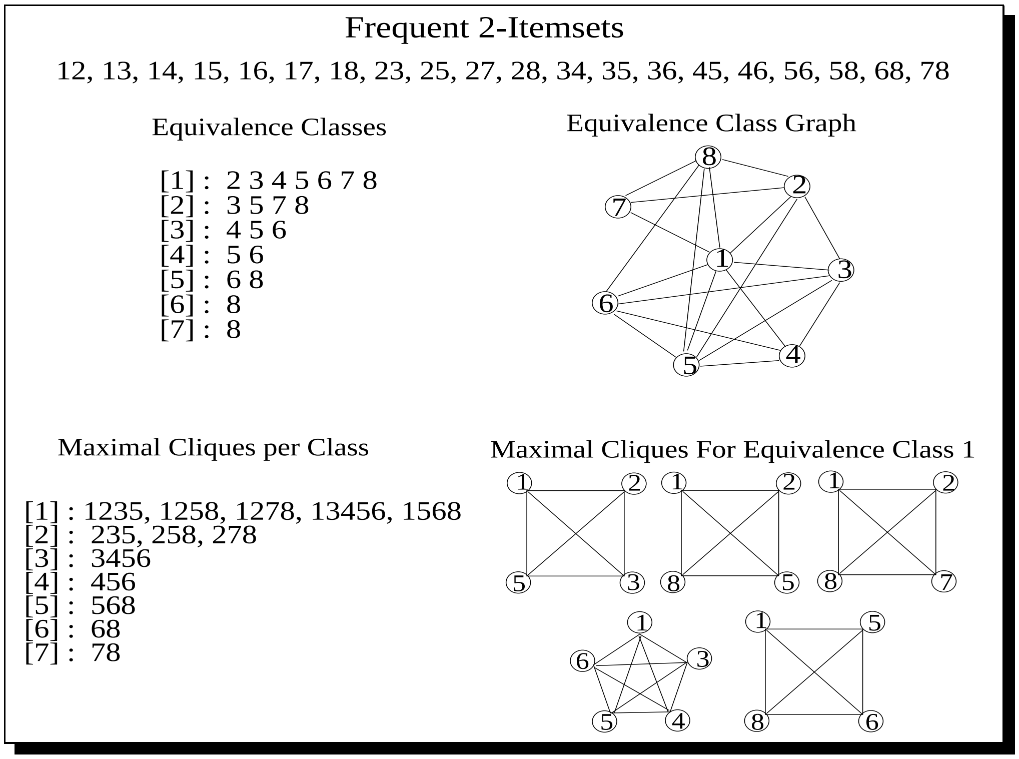


Figure 2: Equivalence Class and Uniform Hypergraph Clique Clustering

2.2 Maximal Uniform Hypergraph Clique Clustering

Let the set of items $\mathcal{I}$ denote the vertex set. A *hypergraph* [7] on $\mathcal{I}$ is a family $H = \{E_1, E_2, ..., E_n\}$ of edges or subsets of $\mathcal{I}$, such that $E_i \neq \emptyset$, and $\cup_{i=1}^{n} E_i = \mathcal{I}$. A *simple hypergraph* is a hypergraph such that, $E_i \subset E_j \Rightarrow i = j$. A simple graph is a simple hypergraph each of whose edges has cardinality 2. The maximum edge cardinality is called the *rank*, $r(H) = max_j|E_j|$. If all edges have the same cardinality, then H is called a *uniform hypergraph.* A simple uniform hypergraph of rank r is called a *r-uniform* hypergraph. For a subset $X \subset \mathcal{I}$, the *sub-hypergraph* induced by X is given as, $H_X = \{E_j \cap X \neq \emptyset | 1 \leq j \leq n\}$. A *$r$-uniform complete hypergraph* with m vertices, denoted as K_m^r, consists of all the r-subsets of $\mathcal{I}$. A r-uniform complete sub-hypergraph is called a *r-uniform hypergraph clique.* A hypergraph clique is *maximal* if it is not contained in any other clique. For hypergraphs of rank 2, this corresponds to the familiar concept of maximal cliques in a graph.

Given the set of frequent itemsets L_k, it is possible to further refine the clustering process producing a smaller set of potentially maximal frequent itemsets. The key observation used is that given any frequent m-itemset, for $m > k$, all its k-subsets must be frequent. In graph-theoretic terms, if each item is a vertex in the hypergraph, and each k-subset an edge, then the frequent m-itemset must form a k-uniform hypergraph clique. Furthermore, the set of maximal hypergraph cliques represents an approximation or upper-bound on the set of maximal potential frequent itemsets. All the "true" maximal frequent itemsets are contained in the vertex set of the maximal cliques, as stated formally in the lemma below.

Lemma 1 *Let H_{L_k} be the k-uniform hypergraph with vertex set $\mathcal{I}$, and edge set L_k. Let C be the set of maximal hypergraph cliques in H, i.e., $C = \{K_m^k | m > k\}$, and let M be the set of vertex sets of the cliques in C. Then for all maximal frequent itemsets f, $\exists t \in M$, such that $f \subseteq t$.*

An example of uniform hypergraph clique clustering is given in figure 2. The example is for the case of L_2, and thus corresponds to an instance of the general clustering technique, which reduces to the case of finding maximal cliques in regular graphs. The figure shows all the equivalence classes, and the maximal cliques within them. It also shows the graph for class 1, and the maximal cliques in it. It can be seen immediately the the clique clustering is more accurate than equivalence class clustering. For example, while equivalence class clustering produced the potential maximal frequent itemset 12345678, the hypergraph clique clustering produces a more refined set $\{1235, 1258, 1278, 13456, 1568\}$ for equivalence class $[\mathbf{1}]$.

Clique Generation

The maximal cliques are discovered using an algorithm similar to the Bierstone's algorithm [19] for generating cliques. For a class $[\mathbf{x}]$, and $y \in [\mathbf{x}]$, y is said to *cover* the subset of $[\mathbf{x}]$, given by $cov(y) = [\mathbf{y}] \cap [\mathbf{x}]$. For each class $\mathcal{C}$, we first identify its *covering set*, given as $\{y \in \mathcal{C} | cov(y) \neq \emptyset$, and $cov(y) \not\subseteq cov(z)$, for any $z \in \mathcal{C}, z < y\}$. For example, consider the class $[\mathbf{1}]$, shown in figure 2. $cov(2) = [\mathbf{2}]$, since $[\mathbf{2}] \subset [\mathbf{1}]$. Similarly, $cov(y) = [\mathbf{y}]$, for all $y \in [\mathbf{1}]$. The covering set of $[\mathbf{1}]$ is given by the set $\{2, 3, 5\}$. The item 4 is not in the covering set since, $cov(4) = \{5, 6\}$ is a subset of $cov(3) = \{4, 5, 6\}$. Figure 3 shows the complete clique generation algorithm. Only the elements in the covering set need to be considered while generating maximal cliques for the current class (step 3). We recursively generate the maximal cliques for elements in the covering set for each class. Each maximal clique from the covering set is prefixed with the class identifier to obtain the maximal cliques for the current class (step 7). Before inserting the new clique, all duplicates or subsets are eliminated. If the new clique is a subset of any clique already in the maximal list, then it is not inserted. The conditions for the above test are shown in line 8.

For general graphs the maximal clique decision problem is NP-Complete [13]. However, the equivalence class graph is usually sparse and the maximal cliques can be enumerated efficiently. As the edge density increases the clique based approaches may suffer. Some of the factors affecting the edge density include decreasing support and increasing transaction size. The effect of these parameters is studied in section 6. A number of other clique generating algorithms were brought to our notice after we had chosen the above algorithm. The algorithm proposed in [8] was shown to have superior performance than the Bierstone algorithm. Some other newer algorithms for this problem are presented in [10, 30]. We plan to incorporate these for the clique generation step of our algorithm, minimizing any overhead due to this step.

```
1:for $i = N; i >= 1; i - -$ do
2:    $[\mathbf{i}].CliqList = \emptyset$;
3:    for all $x \in [\mathbf{i}].CoveringSet$ do
4:        for all $cliq \in [x].CliqList$ do
5:            $M = cliq \cap [\mathbf{i}]$;
6:            if $M \neq \emptyset$ then
7:                insert $(\{i\} \cup M)$ in $[\mathbf{i}].CliqList$ such that
8:                $\nexists X or Y \in [\mathbf{i}]$.CliqList, $X \subseteq Y, or Y \subseteq X$;
```

Figure 3: The Maximal Clique Generation Algorithm

Weak Maximal Cliques

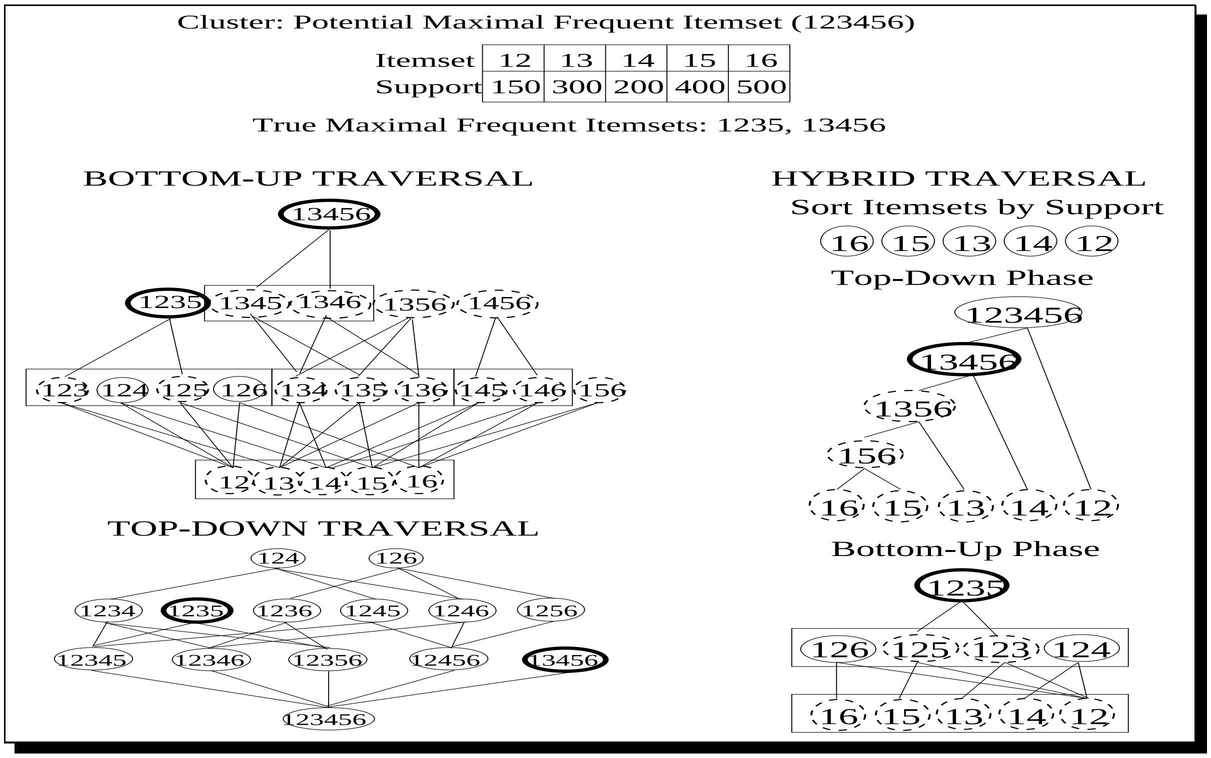
As we shall see in the experimental section, for some database parameters, the edge density of the graph may be too high, resulting in a large number of cliques with significant overlap among them. In these cases, not only the clique generation takes more time, but also redundant frequent itemsets may be discovered within each cluster. To solve this problem we introduce the notion of weak maximality of cliques. Given any two cliques X, and Y, we say that they are α-related, if $\alpha = \frac{|\{X \cap Y\}|}{|\{X \cup Y\}|}$, i.e., the ratio of the common elements between the cliques, to the distinct elements between them. A *weak maximal* clique, $Z = \{X \cup Y\}$, is generated by collapsing the two cliques into one, provided that they are α-related. During clique generation only weak maximal cliques are generated for some user specified value of α. Note that for $\alpha = 1$, we obtain regular maximal cliques, while for $\alpha = 0$, we obtain a single clique, the same as the equivalence class cluster. Preliminary experiments indicate that using an appropriate value of α, all the overhead of redundant cliques can be avoided, yet retaining a better clique quality than the equivalence class clusters.

3 Lattice Traversal

The equivalence class and uniform hyper-graph clique clustering schemes generate the set of potential maximal frequent itemsets. Each such potential maximal itemset induces a sublattice of the lattice of subsets of database items $\mathcal{I}$. We now have to traverse each of these sub-lattices to determine the "true" frequent itemsets. Our goal is to devise efficient schemes to precisely determine the structure of the border of frequent itemsets. Different ways of expanding the frequent itemset border in the lattice space are possible. Below we present three schemes to traverse the sublattices – a pure bottom-up approach, a pure top-down approach, and a hybrid top-down/bottom-up scheme.

3.1 Bottom-up Lattice Traversal

Consider the example shown in figure 4. It shows a particular instance of the clustering schemes which uses L_2 to generate the set of potential maximal itemsets. Let's assume that



Itemset	12	13	14	15	16
Support	150	300	200	400	500

Figure 4: Bottom-Up, Top-Down and Hybrid Lattice Traversal

for equivalence class [**1**], there is only one potential maximal itemset, 123456, while 1235 and 13456 are "true" maximal frequent itemsets. The support of 2-itemsets in this class are also shown. Like figure 1, the dashed circles represent the frequent sets, the bold circles the maximal such itemsets, and the boxes denote equivalence classes. The potential maximal itemset 123456 forms a lattice over the elements of equivalence class $[\mathbf{1}] = \{12, 13, 14, 15, 16\}$. We need to traverse this lattice to determine the "true" frequent itemsets.

A pure bottom-up lattice traversal proceeds in a breadth-first manner generating frequent itemsets of length k, before generating itemsets of level $k+1$, i.e., at each intermediate step we determine the border of frequent k-itemsets. For example, all pairs of elements of [**1**] are joined to produce new equivalence classes of frequent 3-itemsets, namely $[\mathbf{12}] = \{3, 5\}$ (producing the maximal itemset 1235), $[\mathbf{13}] = \{4, 5, 6\}$, and $[\mathbf{14}] = \{5, 6\}$. The next step yields the frequent class, $[\mathbf{134}] = \{5, 6\}$ (producing the maximal itemset 13456). Most current algorithms use this approach. For example, the process of generating C_k from L_{k-1} used in *Apriori* [3], and related algorithms [27, 23], is a pure bottom-up exploration of the lattice space. Since this is a bottom-up approach all the frequent subsets of the maximal frequent itemsets are generated in intermediate steps of the traversal.

3.2 Top-Down Search

The bottom-up approach doesn't make full use of the clustering information. While it uses the cluster to restrict the search space, it may generate spurious candidates in the intermediate steps, since the fact that all subsets of an itemset are frequent doesn't guarantee that the itemset is frequent. For example, the itemsets 124 and 126 in figure 4 are infrequent, even though 12, 14, and 16 are frequent. We can envision other traversal techniques which quickly identify the set of true maximal frequent itemsets. Once this set is known we can either choose to stop at this point if we are interested in only the maximal itemsets, or we can gather the support of all their subsets as well (all subsets are known to be frequent by definition). In this paper we will restrict our attention to only identifying the maximal frequent subsets.

One possible approach is to perform a pure top-down traversal on each cluster or sublattice. This scheme may be thought of as trying to determine the border of infrequent itemsets, by starting at the top element of the lattice and working our way down. For example, consider the potential maximal frequent itemset 123456 in figure 4. If it turns out to be frequent we are done. But in this case it is not frequent, so we then have to check whether each of its 5-subsets is frequent. At any step, if a k-subset turns out to be frequent, we need not check any of its subsets. On the other hand, if it turns out to be infrequent, we recursively test each $(k-1)$-subset. To ensure that infrequent itemsets are not tested multiple times, we also maintain a hash table of infrequent itemsets. Depending on the accuracy of the potential maximal clusters, the top-down approach may save some intersections. In our example the top-down approach performs only 14 intersections against the 16 done by the bottom-up approach. However, this approach doesn't work too well in practice, since the clusters are only an approximation of the maximal frequent itemsets, and a lot of infrequent supersets of the "true" maximal frequent itemsets may be generated. Furthermore, it also uses hash tables, and k-way intersections instead of 2-way intersections, adding extra overhead. We therefore, propose a hybrid top-down and bottom-up approach that works well in practice.

3.3 Hybrid Top-down/Bottom-up Search

The basic idea behind the hybrid approach is to quickly determine the "true" maximal itemsets, by starting with a single element from a cluster of frequent k-itemsets, and extending this by one more itemset till we generate an infrequent itemset. This comprises the top-down phase. In the bottom-up phase, the remaining elements are combined with the elements in the first set to generate all the additional frequent itemsets. An important consideration in the top-down phase is to determine which elements of the cluster should be combined. In our approach we first sort the itemsets in the cluster in descending order of their support. We start with the element with maximum support, and extend it with the next element in the sorted order. This approach is based on the intuition that the larger the support the more the likely is the itemset to be a part of a larger itemset. Figure 4 shows an example of the hybrid scheme on a cluster of 2-itemsets. We sort the 2-itemsets in decreasing order of support, intersecting 16 and 15 to produce 156. This is extended to 1356 by joining 156 with 13, and then to 13456, and finally we find that 123456 is infrequent. The only remaining element is 12. We simply join this with each of the other elements producing the frequent itemset class [**12**], which generates the other maximal itemset 1235. The bottom-up, top-down, and hybrid approaches are contrasted in figure 4, and the pseudo-code for all the schemes is shown in figure 5.

```
Input: $F_k = \{I_1..I_n\}$
cluster of frequent
k-itemsets.

Output: Frequent
$l$-itemsets, $l > k$

Bottom-Up($F_k$):
for all $I_i \in F_k$ do
 $F_{k+1} = \emptyset$;
 for all $I_j \in F_k$, $i < j$ do
  N = ($I_i \cap I_j$);
  if N.sup $\geq$ minsup then
   $F_{k+1} = F_{k+1} \cup \{N\}$;
 end;
 if $F_{k+1} \neq \emptyset$ then
  Bottom-Up($F_{k+1}$);
end;
```

```
$HT$ = HashTable
$X = \{i | i \in I_j, \forall I_j \in F_2\}$;

Top-Down($X$):
$l = |X|$;
if $X \notin F_l$ then
 /* do $l$-way join */
 N = get-intersect(X);

 if N.sup $\geq$ minsup then
  $F_l = F_l \cup \{N\}$;
 else if l > 3 then
  for all $Y \subset X, |Y| = l - 1$
   if $Y \notin$ HT then
    Top-Down($Y$);
    if Y.sup < minsup
     HT = HT $\cup \{Y\}$;
  end;
```

```
Hybrid($F_2$):
/* Top-Down Phase */
$N = I_1$; $S_1 = \{I_1\}$;
for all $I_i \in F_2$, $i > 1$ do
 N = (N $\cap$ $I_i$);
 if N.sup $\geq$ minsup then
  $S_1 = S_1 \cup \{I_i\}$;
 else break;
end;
$S_2 = F_2 - S_1$;
/* Bottom-Up Phase */
for all $I_i \in S_2$,do
 $F_3 = \{Y_j.sup \geq minsup |$
  $Y_j = (I_i \cap X_j), \forall X_j \in S_1\}$;
 $S_1 = S_1 \cup \{I_i\}$;
 if $F_3 \neq \emptyset$ then
  Bottom-Up($F_3$);
end;
```

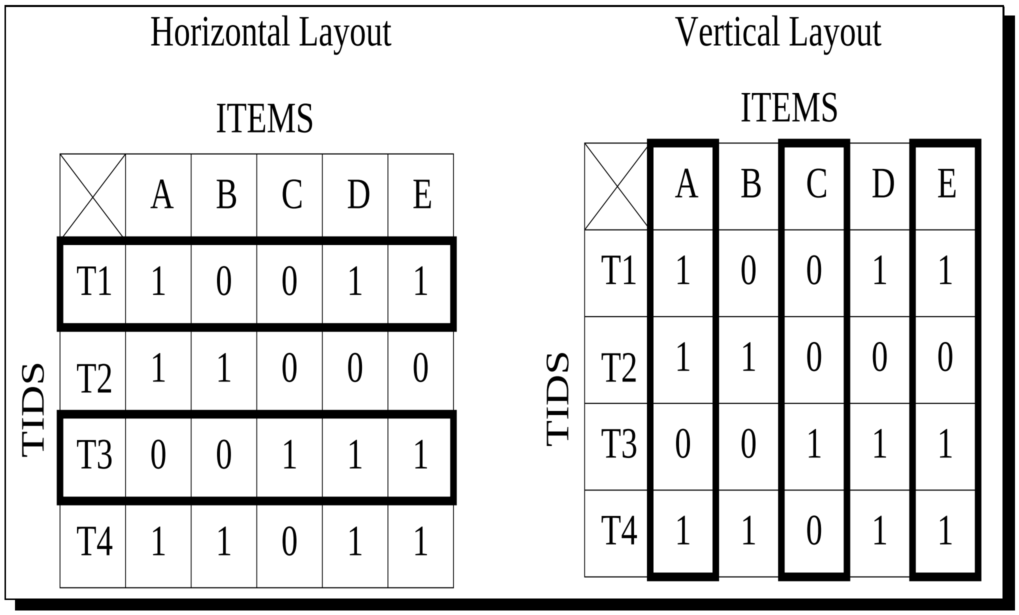
Figure 5: Pseudo-code for Bottom-up and Hybrid Traversal

4 The KDD Process and Database Layout

The KDD process consists of various steps [12]. The initial step consists of creating the target dataset by focusing on certain attributes or via data samples. The database creation may require removing unnecessary information and supplying missing data, and transformation techniques for data reduction and projection. The user must then determine the data mining task and choose a suitable algorithm, for example, the discovery of association rules. The next step involves interpreting the discovered associations, possibly looping back to any of the previous steps, to discover more understandable patterns. An important consideration in the data preprocessing step is the final representation or data layout of the dataset. Another issue is whether some preliminary invariant information can be gleaned during this process. There are two possible layouts of the target dataset for association mining – the horizontal and the vertical layout.

Horizontal Data Layout

This is the format standardly used in the literature (see e.g., [5, 21, 3]). Here a dataset consists of a list of transactions. Each transaction has a transaction identifier (TID) followed by a list of items in that transaction. This format imposes some computation overhead during the support counting step. In particular for each transaction of average length l, during iteration k, we have to generate and test whether all $\binom{l}{k}$ k-subsets of the transaction are contained in C_k. To perform fast subset checking the candidates are stored in a complex hash-tree data structure. Searching for the relevant candidates thus adds additional computation overhead. Furthermore, the horizontal layout forces us to scan the entire database or the local partition once in each iteration. Both *Count* and *Candidate Distribution* must pay the extra overhead entailed by using the horizontal layout. Furthermore, the horizontal layout seems suitable only for the bottom-up exploration of the frequent border. It appears to be extremely complicated to implement the hybrid approach using the horizontal format.


Horizontal Layout

ITEMS

TIDS	A	B	C	D	E
T1	1	0	0	1	1
T2	1	1	0	0	0
T3	0	0	1	1	1
T4	1	1	0	1	1

Vertical Layout

ITEMS

TIDS	A	B	C	D	E
T1	1	0	0	1	1
T2	1	1	0	0	0
T3	0	0	1	1	1
T4	1	1	0	1	1

Figure 6: Horizontal and Vertical Database Layout

Vertical Data Layout

In the vertical (or inverted) layout (also called the *decomposed storage structure* [16]), a dataset consists of a list of items, with each item followed by its *tid-list* — the list of all the transactions identifiers containing the item. An example of successful use of this layout can be found in [16, 27, 33, 35]. The vertical layout doesn't suffer from any of the overheads described for the horizontal layout above due to the following three reasons: First, if the tid-list is sorted in increasing order, then the support of a candidate k-itemset can be computed by simply intersecting the tid-lists of any two $(k-1)$-subsets. No complicated data structures need to be maintained. We don't have to generate all the k-subsets of a transaction or perform the search operations on the hash tree. Second, the tid-lists contain all relevant information about an itemset, and enable us to avoid scanning the whole database to compute the support count of an itemset. This layout can therefore take advantage of the principle of locality. All frequent itemsets from a cluster of itemsets can be generated, before moving on to the next cluster. Third, the larger the itemset, the shorter the tid-lists, which is practically always true. This results in faster intersections. For example, consider figure 6, which contrasts the horizontal and the vertical layout (for simplicity, we have shown the null elements, while in reality sparse storage is used). The tid-list of A, is given as $\mathcal{T}(A) = \{1, 2, 4\}$, and $\mathcal{T}(B) = \{2, 4\}$. Then the tid-list of AB is simply, $\mathcal{T}(AB) = \{2, 4\}$. We can immediately determine the support by counting the number of elements in the tid-list. If it meets the minimum support criterion, we insert AB in L_2. The intersections among the tid-lists can be performed faster by utilizing the minimum support value. For example let's assume that the minimum support is 100, and we are intersecting two itemsets – AB with support 119 and AC with support 200. We can stop the intersection the moment we have 20 mismatches in AB, since the support of ABC is bounded above by 119. We use this optimization, called *short-circuited intersection*, for fast joins.

The inverted layout, however, has a drawback. Examination of small itemsets tends to be costlier than when the horizontal layout is employed. This is because tid-lists of small itemsets provide little information about the association among items. In particular, no such information is present in the tid-lists for 1-itemsets. For example, a database with 1,000,000 (1M) transactions, 1,000 frequent items, and an average of 10 items per transaction has tid-lists of average size 10,000. To find frequent 2-itemsets we have to intersect each pair of items, which requires $\binom{1,000}{2} \cdot (2 \cdot 10,000) \approx 10^9$ operations. On the other hand, in the horizontal format we simply need to form all pairs of the items appearing in a transaction and increment their count, requiring only $\binom{10}{2} \cdot 1,000,000 = 4.5 \cdot 10^7$ operations.

There are a number of possible solutions to this problem: 1) Use a preprocessing step to gather the occurrence count of all 2-itemsets. Since this information is invariant, it has to be performed once during the lifetime of the database, and the cost can be amortized over the number of times the data is mined. This information can also be incrementally updated as the database changes over time. 2) Instead of storing the support counts of all the 2-itemsets, use a user specified lower bound on the minimum support the user may wish to apply. Then store the counts of only those 2-itemsets that have support greater than the lower bound. The idea is to minimize the storage by keeping the counts of only those itemsets that can be frequent, provided the user always specifies a minimum support greater than the lower bound. 3) Use a small sample that would fit in memory, and determine a superset of the frequent 2-itemsets, L_2, by lowering the minimum support, and using simple intersections on the sampled tid-lists. Sampling experiments [31, 34] indicate that this is a feasible approach. Once the superset has been determined we can easily verify the "true" frequent itemsets among them,

Our current implementation uses the pre-processing approach due to its simplicity. We plan to implement the sampling approach in a later paper. The two solutions represent a trade-off. The sampling approach generates L_2 on-the-fly with an extra database pass, while the pre-processing approach requires extra storage. For m items, count storage requires $\mathcal{O}(m^2)$ disk space, which can be quite large for large values of m. However, for $m = 1000$, used in our experiments this adds only a very small extra storage overhead. Note also that the database itself requires the same amount of memory in both the horizontal and vertical formats (this is obvious from figure 6).

5 Algorithms for Frequent Itemset Discovery

We first give a brief overview of the well known previous algorithms:

5.1 *Apriori* Algorithm

Apriori [5, 21, 3] uses the downward closure property of itemset support that any subset of a frequent itemset must also be frequent. Thus during each iteration of the algorithm only the itemsets found to be frequent in the previous iteration are used to generate a new candidate set, C_k. Before inserting an itemset into C_k, *Apriori* tests whether all its $(k-1)$-subsets are frequent. This *pruning* step can eliminate a lot of unnecessary candidates. The candidates are stored in a hash tree to facilitate fast support counting. An internal node of the hash tree at depth d contains a hash table whose cells point to nodes at depth $d+1$. All the itemsets are stored in the leaves. The insertion procedure starts at the root, and hashing on successive items, inserts a candidate in a leaf. For counting C_k, for each transaction in the database, all k-subsets of the transaction are generated in lexicographical order. Each subset is searched in the hash tree, and the count of the candidate incremented if it matches the subset. This is the most compute intensive step of the algorithm. The last step forms L_k by selecting itemsets meeting the minimum support criterion. The complete algorithm is shown in figure 7.

```
$L_1 = \{$frequent 1-itemsets $\}$;
for $(k = 2; L_{k-1} \neq \emptyset; k++)$
    $C_k =$ Set of New Candidates;
    for all transactions $t \in \mathcal{D}$
        for all $k$-subsets $s$ of $t$
            if $(s \in C_k)$ $s.count++$;
    $L_k = \{c \in C_k | c.count \geq$ minimum support$\}$;
Set of all frequent itemsets $= \bigcup_k L_k$;
```

Figure 7: The *Apriori* algorithm

5.2 *Partition* Algorithm

Partition [27] logically divides the horizontal database into a number of non-overlapping partitions. Each partition is read, transformed into vertical format on-the-fly, and all locally frequent itemsets are generated via tid-list intersections. All such itemsets are merged and a second pass is made through all the partitions. The database is again converted to the vertical layout and the global counts of all the chosen itemsets are obtained. The partition sizes are chosen so that they can be accommodated in memory. The key observation used above is that a globally frequent itemset must be locally frequent in at least one partition.

5.3 New Algorithms

We present six new algorithms, depending on the clustering, lattice traversal and database layout scheme used.

Horizontal Data Layout

• **ClusterApr:** *ClusterApr* uses the maximal hypergraph clique clustering with the horizontal database layout. It consists of two distinct phases. In the first step each cluster and all of its subsets are inserted into hash trees, ensuring that no duplicates are inserted. There are multiple hash trees, one for candidates of a given length. Bit masks are created for each hash tree indicating the items currently used in any of the candidates of that length. The second step consists of gathering the support of all the candidates. This support counting is similar to the one used in *Apriori*. For each transaction, we start by generating all subsets of length k, for each $k > 2$, after applying the appropriate bit mask. We then search each subset in C_k and update the count if it is found. Thus only one database pass is required for this step, instead of the multiple passes used in *Apriori*. The pseudo-code is given in figure 8.

```
for all clusters $M$
    for all $k > 2$ and $k \leq |M|$
        Insert each $k$-subset of $M$ in $C_k$;
for all transactions $t \in \mathcal{D}$
    for all $k > 2$ and $k \leq |t|$
        for all $k$-subsets $s$ of $t$
            if $(s \in C_k)$ $s.count++$;
$L_k = \{c \in C_k | c.count \geq \text{minsup}\}$;
Set of all frequent itemsets $= \bigcup_k L_k$;
```

Figure 8: The *ClusterApr* algorithm

Vertical Data Layout

Each of remaining new algorithms uses the vertical layout, and uses one of the itemset clustering schemes to generate the potential maximal itemsets. Each such cluster induces a sublattice, which is traversed using bottom-up search to generate all frequent itemsets, or using the top-down or hybrid scheme to generate only the maximal frequent itemsets. Each cluster is processed in its entirety before moving on to the next cluster. Since the transactions are clustered using the vertical format, this involves a single database scan, resulting in huge I/O savings. Frequent itemsets are determined using simple tid-list intersections. No complex hash structures need to be built or searched. The algorithms have low memory utilization, since only the frequent k-itemsets within a single cluster need be kept in memory at any point. The use of simple intersection operations also makes the new algorithms an attractive option for direct implementation on general purpose database systems. The new algorithms are:

- *Eclat*: use equivalence class clustering along with bottom-up lattice traversal.
- *MaxEclat*: uses equivalence class clustering along with hybrid traversal.
- *Clique*: uses maximal hypergraph clique clustering along with bottom-up traversal
- *MaxClique*: uses maximal hypergraph clique clustering along with hybrid lattice traversal
- *TopDown*: uses maximal hypergraph clique clustering with top-down traversal.

The pseudo-code for the maximal hypergraph clique scheme was presented in figure 3 (generating equivalence classes is quite straightforward), while the code for the three traversal strategies was shown in figure 5. Theses two steps combined represent the pseudo-code for the new algorithms. We would like to note that our current implementation uses only an instance of the general maximal hypergraph cliques technique, i.e. for the case where $k = 2$. We can easily envision a more dynamic scheme where we refine the hypergraph clustering as the frequent k-itemsets for $k > 2$ become known. For example, when all 3-itemsets have been found within a class, we can try to get a more refined set of maximal 3-uniform hypergraph cliques, and so on. We plan to implement this approach in the future.

6 Experimental Results

Our experiments used a 100MHz MIPS processor with 256MB main memory, 16KB primary cache, 1MB secondary cache and an attached 2GB disk. We used different synthetic databases which mimic the transactions in a retailing environment, and were generated using the procedure described in [5]. These have been used as benchmark databases for many association rules algorithms [5, 16, 23, 27, 3]. The different database parameters varied in our experiments are the number of transactions D, average transaction size T, and the average size of a maximal potentially frequent itemset I. The number of maximal potentially frequent itemsets was set at $L = 2000$, and the number of items at $N = 1000$. We refer the reader to [5] for more detail on the database generation. For fair comparison, all algorithms discover frequent k-itemsets for $k \geq 3$, using the 2-itemset supports from the preprocessing step.

6.1 Performance

In figures 9 and 10, we compare our new algorithms against *Apriori* and *Partition* (with 10 partitions) for decreasing values of minimum support on the different databases. We compare *Eclat* against *Apriori*, *ClusterApr* and *Partition* in the left column, and we compare *Eclat* against the other algorithms in the right column, to highlight the differences among them. As the support decreases, the size and the number of frequent itemsets increases. *Apriori* thus has to make multiple passes over the database, and performs poorly. *Partition* performs worse than *Apriori* on small databases, and for high support, since the database is only scanned once or twice at these points. *Partition* also performs the inversion from horizontal to vertical tid-list format on-the-fly, incurring additional overhead. However, as the support is lowered, *Partition* wins out over *Apriori*, since it only scans the database twice. It also saves some computation overhead of hash trees, since it also uses simple intersections to compute frequent itemsets. These results are in agreement with previous experiments comparing these two algorithms [27]. One major problem with *Partition* is that as the number of partitions increases, the number of locally frequent itemsets, which are not globally frequent, increases. While this can be reduced by randomizing the partition selection, sampling experiments [34, 31] indicate that randomized partitions will have a large number of frequent itemsets in common. *Partition* can thus spend a lot of time in performing these redundant intersections. *ClusterApr* scans the database only once, and out-performs *Apriori* in almost all cases, and generally lies between *Apriori* and *Partition*. More extensive experiments on large out-of-core databases are necessary to fully compare it against *Partition*. *ClusterApr* is very sensitive to the quality of maximal cliques that are generated. For small support, or small average maximal potential frequent itemset size I for fixed T, or with increasing average transaction size T for fixed I, the edge density of the hypergraph increases, consequently increasing the size of the maximal cliques. *ClusterApr* is unlikely to perform well under these conditions, and this is confirmed for the T20.I2.D100K database, where it performs the worst. Like *Apriori*, it also maintains hash trees for support counting. *Eclat* performs significantly better than all these algorithms in all cases. It out-performs *Apriori* by more than an order of magnitude, and *Partition* by more than a factor of five. *Eclat* makes only once database scan, requires no hash trees, uses only simple intersection operations to generate globally frequent itemsets, avoids redundant computations, and since it deals with one cluster at a time, has excellent locality.

	Eclat	Clique	MaxEclat	MaxClique	TopDown	Partition
# Joins	83606	61968	56908	20322	24221	895429
Time (sec)	46.7	42.1	28.5	18.5	48.2	174.7

Table 1: Number of Joins: T20.I6.D100K (0.25%)

The right hand columns in figure 9 and 10 present the comparison among the other new algorithms. Among the clustering techniques, *Clique* provides a finer level of clustering, reducing the number of candidates considered, and therefore performs better than *Eclat* when the number of cliques considered is not too large. The graphs for *MaxEclat* and *MaxClique* indicate that the reduction in search space by performing the hybrid search also provides significant gains. Both the maximal strategies outperform their normal counterparts. *TopDown* generally speaking out-performs both *Eclat* and *MaxEclat*, since it also only generates the maximal frequent itemsets. As with *ClusterApr*, it is very sensitive to the accuracy of the cliques, and it suffers as the cliques become larger. The best scheme for the databases considered is *MaxClique* since it benefits from both the finer clustering, and hybrid search scheme. Table 1 gives the number of joins performed on T20.I2.D100K, while figure 12 provides more detail for the different databases. It can be observed that *MaxClique* cuts down the candidate search space drastically (more than a factor of 4 for T20.I6.D100K). In terms of raw performance *MaxClique* outperforms *Apriori* by a factor of 40 *Partition* by a

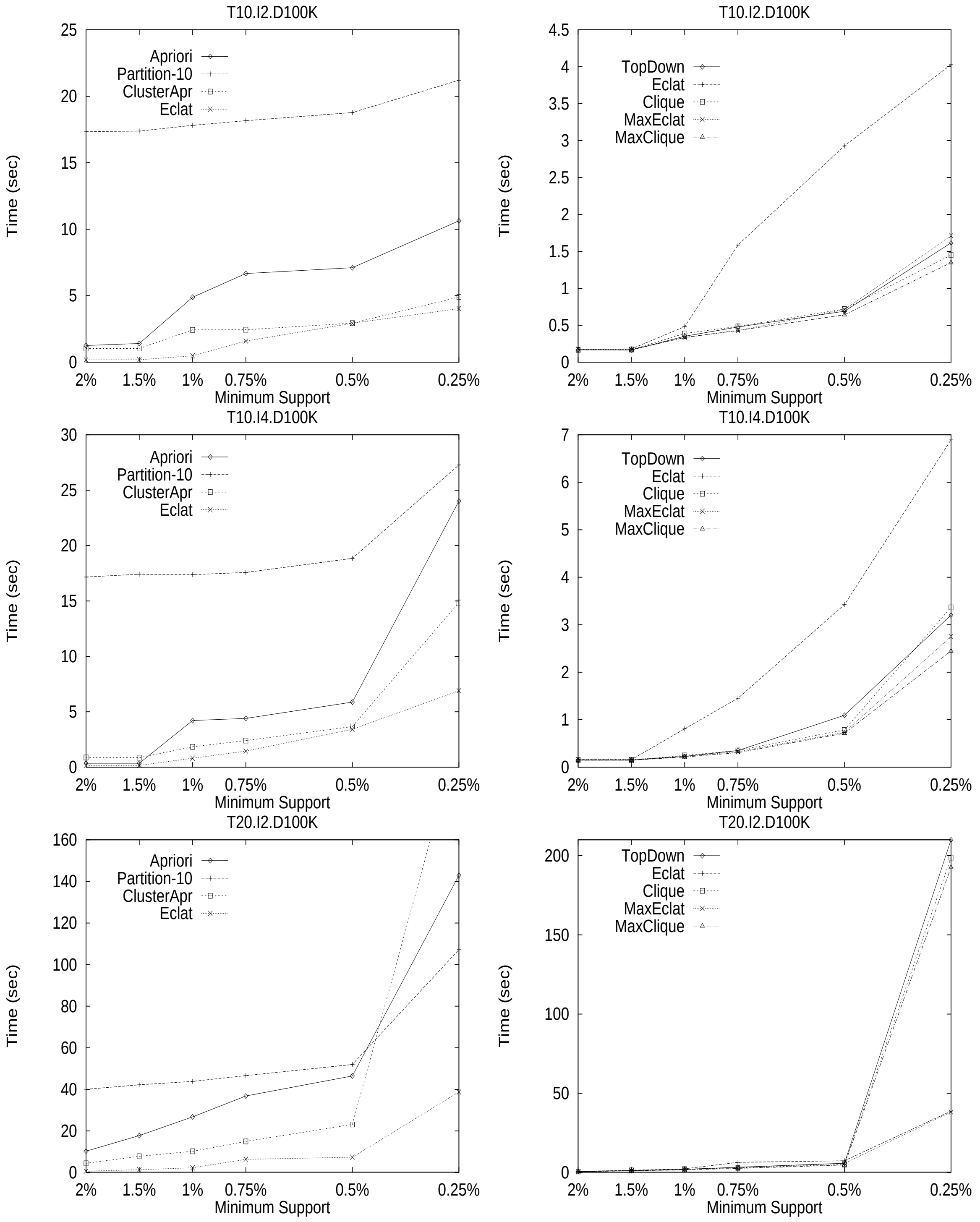


Figure 9: Execution Time

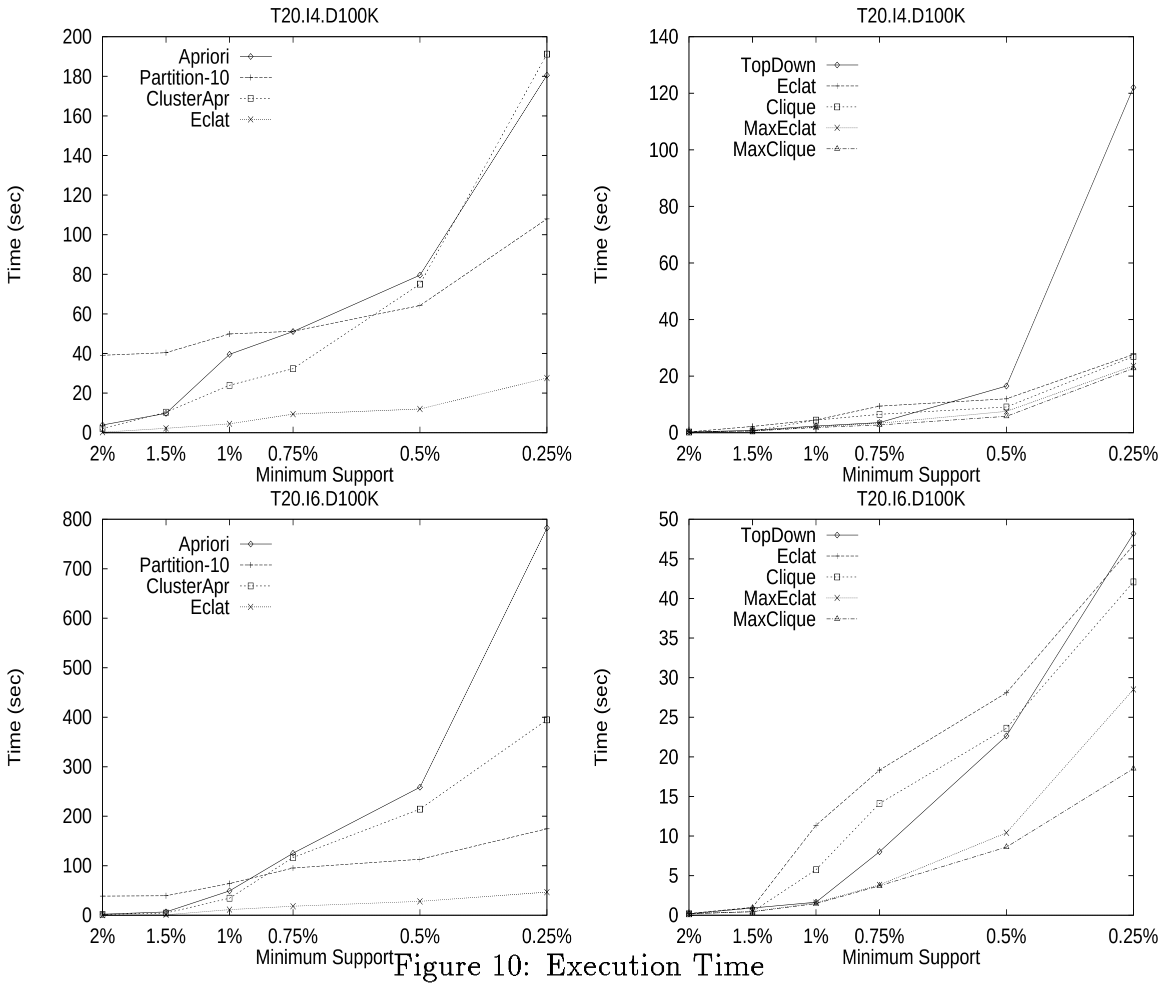
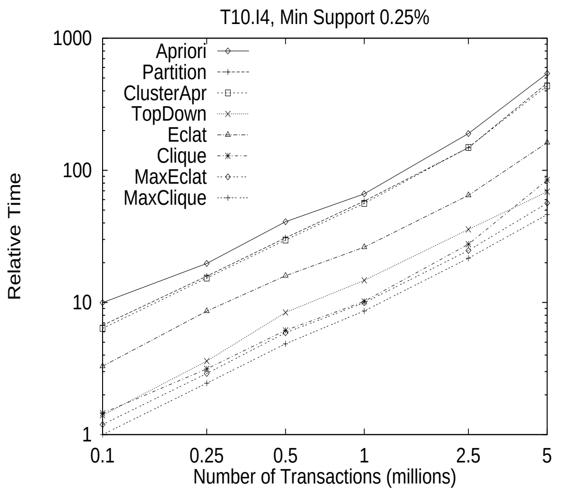
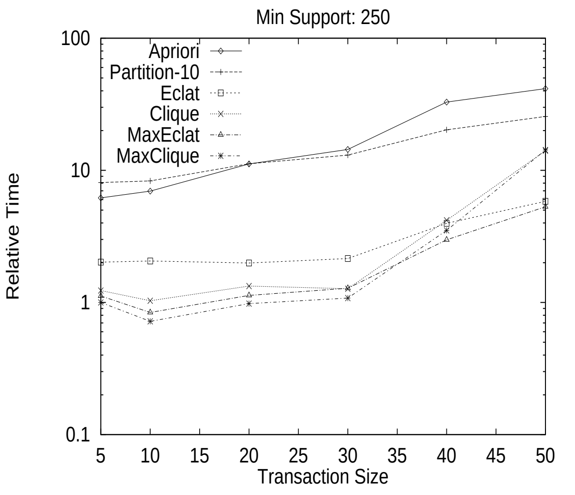


Figure 10: Execution Time



(a) Number of Transactions Scale-up

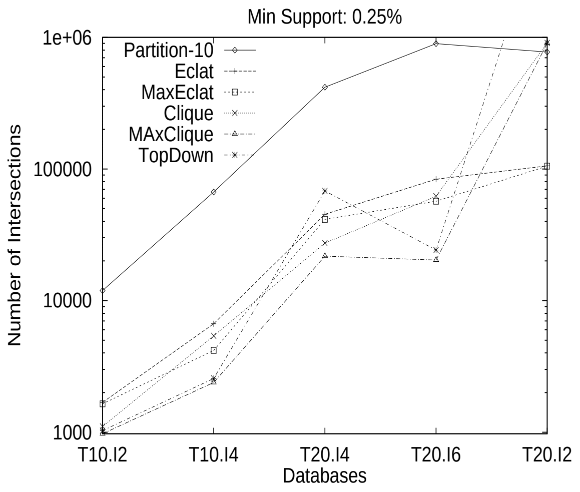


(b) Transaction Size Scale-up

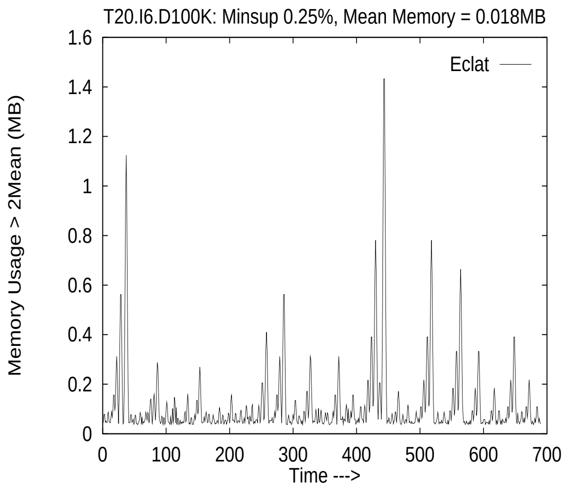
Figure 11: Scale-up Experiments

factor of 20, and *Eclat* by a factor of 2.5 for this case. However, as previously mentioned, whenever the number of cliques becomes too large and there is a significant overlap among different cliques, the clique based schemes will suffer. This is borne out in the graphs for T20.I2.D100K with decreasing support, and in figure 11 b) as the transaction size increases for a fixed support value. We expect to reduce the overhead of the clique generation by implementing the algorithms proposed in [8, 10, 30], which were shown to be superior to the Bierstone algorithm [19], a modification of which is used in the current implementation.

6.2 Join and Memory Statistics



(a) Number of Intersections



(b) Memory Usage

Figure 12: Statistics

Figure 12 a) shows the comparison among the different algorithms in terms of the number of intersections performed. As mentioned above the different clustering and traversal techniques are able to reduce the joins to different extents, providing the key to their performance. The amount of redundant computation for *Partition* is easily seen in the figure in terms of the extra intersections performed by it compared to the other algorithms. Figure 12 b) shows the total memory usage of the *Eclat* algorithm as the computation of frequent itemsets progresses on T20.I6.D100K. The mean memory usage for the tid-lists is less than 0.18MB, or roughly 2% of the total database size. The figure only shows the cases where the memory usage was more than twice the mean. The peaks in the graph are usually due to the initial construction of all the 2-itemset tid-lists within each cluster. For the other algorithms, we expect these peaks to be lower, since the maximal clique clustering is more precise, resulting in smaller clusters, and the hybrid traversal doesn't need the entire cluster 2-itemsets initially.

6.3 Scalability

Figure 11 a) shows how the different algorithms scale up as the number of transactions increases from 100,000 to 5 million. The times are normalized against the execution time for *MaxClique* on 100,000 transactions. A minimum support value of 0.25% was used. The number of partitions for *Partition* was varied from 1 to 50. While all algorithms scale linearly, the slope is much smaller for the new algorithms. This implies that the performance differences for larger databases, across algorithms is likely to increase. Figure 11 b) shows how the different algorithms scale with increasing transaction size. The times are normalized against the execution time for *MaxClique* on $T = 5$ and 200,000 transactions. Instead of a percentage, we used an absolute support of 250. The physical size of the database was kept roughly the same by keeping a constant $T * D$ value. We used $D = 200,000$ for $T = 5$, and $D = 20,000$ for $T = 50$. There is a gradual increase in execution time for all algorithms with increasing transaction size. However the new algorithms again outperform *Apriori* and *Partition*. As the transaction size increases, the number of cliques increases, and the clique based algorithms start performing worse.

7 Conclusions

In this paper we proposed new algorithms for association mining and evaluate their effectiveness. The proposed algorithms scan the preprocessed database exactly once, greatly reducing I/O costs. Three main techniques are employed in these algorithms. We first cluster itemsets using equivalence classes or maximal hypergraph cliques to obtain a set of potential maximal frequent itemsets. We then generate the true frequent itemsets from each cluster sublattice using bottom-up, top-down or hybrid lattice traversal. Two different database layout are studied – the horizontal or the vertical format. Experimental results indicate more than an order of magnitude improvements over previous algorithms.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. In *IEEE Trans. on Knowledge and Data Engg.*, pages 5(6):914–925, 1993.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining.* MIT Press, 1996.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, pages 8(6):962–969, 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th VLDB Conf.*, Sept. 1994.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *Intl. Conf. on Data Engg.*, 1995.

[7] C. Berge. *Hypergraphs: Combinatorics of Finite Sets.* North-Holland, 1989.

[8] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. In *Communications of the ACM*, 16(9):575-577, Sept. 1973.

[9] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[10] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. In *SIAM J. Computing*, 14(1):210-223, Feb. 1985.

[11] D. Cheung, V. Ng, A. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, pages 8(6):911–922, 1996.

[12] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. In *Communications of the ACM – Data Mining and Knowledge Discovery in Databases*, Nov. 1996.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[14] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[15] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *21st VLDB Conf.*, 1995.

[16] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *1st Intl. Conf. Knowledge Discovery and Data Mining*, Aug. 1995.

[17] M. Houtsma and A. Swami. Set-oriented mining of association rules in relational databases. In *11th Intl. Conf. Data Engineering*, 1995.

[18] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, pages 401–407, Nov. 1994.

[19] G. D. Mulligan and D. G. Corneil. Corrections to Bierstone's Algorithm for Generating Cliques. In *J. Association of Computing Machinery*, 19(2):244-247, Apr. 1972.

[20] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurences. In *2nd Intl. Conf. Knowledge Discovery and Data Mining*, 1996.

[21] H. Mannila, H. Toivonen, and I. Verkamo. Efficient algorithms for discovering association rules. In *AAAI Wkshp. Knowledge Discovery in Databases*, July 1994.

[22] H. Mannila, H. Toivonen, and I. Verkamo. Discovering frequent episodes in sequences. In *1st Intl. Conf. Knowledge Discovery and Data Mining*, 1995.

[23] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[24] J. S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *ACM Intl. Conf. Information and Knowledge Management*, Nov. 1995.

[25] S. Parthasarathy, M. J. Zaki, and W. Li. Application driven memory placement for dynamic data structures. Technical Report URCS TR 653, University of Rochester, Apr. 1997.

[26] G. Piatetsky-Shapiro. Discovery, presentation and analysis of strong rules. In G. P.-S. et al, editor, *KDD*. AAAI Press, 1991.

[27] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[28] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *ACM SIGMOD Conf. Management of Data*, June 1996.

[29] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, Mar. 1996.

[30] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. In *SIAM J. Computing*, 6(3):505-517, Sept. 1977.

[31] H. Toivonen. Sampling large databases for association rules. In *22nd VLDB Conf.*, 1996.

[32] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing'96*, Nov. 1996.

[33] M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *9th ACM Symp. Parallel Algorithms and Architectures*, June 1997.

[34] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. In *7th Intl. Wkshp. Research Issues in Data Engg*, Apr. 1997.

[35] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, Aug. 1997.