# Scalable Algorithms for Association Mining

Mohammed J. Zaki
Computer Science Department
Rensselaer Polytechnic Institute, Troy NY 12180-3590
zaki@cs.rpi.edu
http://www.cs.rpi.edu/~zaki

**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 items are organized into a subset lattice search space, which is decomposed into small independent chunks or sub-lattices, which can be solved in memory. Efficient lattice traversal techniques are presented, which quickly identify all the long frequent itemsets, and their subsets if required. We also present the effect of using different database layout schemes combined with the proposed decomposition and traversal techniques. We experimentally compare the new algorithms against the previous approaches, obtaining improvements of more than an order of magnitude for our test databases.

**Keywords:** Association Rules, Frequent Itemsets, Equivalence Classes, Maximal Cliques, Lattices, Data Mining

## 1 Introduction

The association mining task is to discover a set of attributes shared among a large number of objects in a given database. For example, consider the sales database of a bookstore, where the objects represent customers and the attributes represent books. The discovered patterns are the set of books most frequently bought together by the customers. An example could be that, "40% of the people who buy Jane Austen's *Pride and Prejudice* also buy *Sense and Sensibility*." The store can use this knowledge for promotions, shelf placement, etc. There are many potential application areas for association rule technology, which include catalog design, store layout, customer segmentation, telecommunication alarm diagnosis, and so on.

The task of discovering all frequent associations in very large databases is quite challenging. The search space is exponential in the number of database attributes, and with millions of database objects the problem of I/O minimization becomes paramount. However, most current approaches are iterative in nature, requiring multiple database scans, which is clearly very expensive. Some of the methods, especially those using some form of sampling, can be sensitive to the data-skew, which can adversely affect performance. Furthermore, most approaches use very complicated internal data structures which have poor locality and add additional space and computation overheads. Our goal is to overcome all of these limitations.

In this paper we present new algorithms for discovering the set of frequent attributes (also called itemsets). The key features of our approach are as follows:

1. We use a *vertical tid-list* database format, where we associate with each itemset a list of transactions in which it occurs. We show that all frequent itemsets can be enumerated via simple tid-list intersections.

2. We use a lattice-theoretic approach to decompose the original search space (lattice) into smaller pieces (sub-lattices), which can be processed independently in main-memory. We propose two techniques for achieving the decomposition: prefix-based and maximal-clique-based partition.

3. We decouple the problem decomposition from the pattern search. We propose three new search strategies for enumerating the frequent itemsets within each sub-lattice: bottom-up, top-down and hybrid search.

4. Our approach roughly requires only a a few database scans, minimizing the I/O costs.

We present six new algorithms combining the features listed above, depending on the database format, the decomposition technique, and the search procedure used. These include *Eclat* (**E**quivalence **CLA**ss **T**ransformation), *MaxEclat*, *Clique*, *MaxClique*, *TopDown*, and *AprClique*. Our new algorithms not only minimize I/O costs by making only a small number of database scan, but also minimize computation costs by using efficient search schemes. The algorithms are particularly effective when the discovered frequent itemsets are long. Our tid-list based approach is also insensitive to data-skew. In fact the *MaxEclat* and *MaxClique* algorithms exploit the skew in tid-lists (i.e., the support of the itemsets) to reorder the search, so that the long frequent itemsets are listed first. Furthermore, the use of simple intersection operations makes the new algorithms an attractive option for direct implementation in database systems, using SQL. With the help of an extensive set of experiments, we show that the best new algorithm improves over current methods by over an order of magnitude. At the same time, the proposed techniques retain linear scalability in the number of transactions in the database.

The rest of this paper is organized as follows: In Section 2 we describe the association discovery problem. We look at related work in Section 3. In section 4 we develop our lattice-based approach for problem decomposition and pattern search. Section 5 describes our new algorithms. Some previous methods, used for experimental comparison, are described in more detail in Section 6. An experimental study is presented in Section 7, and we conclude in Section 8. Some mining complexity results for frequent itemsets and their link to graph-theory are highlighted in Appendix A.

## 2 Problem Statement

The association mining task, first introduced in [1], can be stated as follows: Let $\mathcal{I}$ be a set of items, and $\mathcal{D}$ a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. A set of items is also called an *itemset*. An itemset with $k$ items is called a $k$-itemset. The *support* of an itemset $X$, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset. A $k$ length subset of an itemset is called a $k$-subset. An itemset is maximal if it is not a subset of any other itemset. An itemset is *frequent* if its support is more than a user-specified *minimum support (min_sup)* value. The set of frequent $k$-itemsets is denoted $\mathcal{F}_k$.

An *association rule* is an expression $A \Rightarrow B$, where $A$ and $B$ are itemsets. The support of the rule is given as $\sigma(A \cup B)$, and the *confidence* as $\sigma(A \cup B)/\sigma(A)$ (i.e., the conditional probability that a transaction contains $B$, given that it contains $A$). A rule is *confident* if its confidence is more than a user-specified *minimum confidence (min_conf)*.

The data mining task is to generate all association rules in the database, which have a support greater than *min_sup*, i.e., the rules are frequent. The rules must also have confidence greater than *min_conf*, i.e., the rules are confident. This task can be broken into two steps [2]:

1. Find all frequent itemsets. This step is computationally and I/O intensive. Given $m$ items, there can be potentially $2^m$ frequent itemsets. Efficient methods are needed to traverse this exponential itemset search space to enumerate all the frequent itemsets. Thus frequent itemset discovery is the main focus of this paper.

2. Generate confident rules. This step is relatively straightforward; rules of the form $X \backslash Y \Rightarrow Y$, where $Y \subset X$, are generated for all frequent itemsets $X$, provided the rules have at least minimum confidence.

Consider an example bookstore sales database shown in Figure 1. There are five different items (names of authors the bookstore carries), i.e., $\mathcal{I} = \{A, C, D, T, W\}$, and the database consists of six customers who bought books by these authors. Figure 1 shows all the frequent itemsets that are contained in at least three customer transactions, i.e., *min_sup* = 50%. It also shows the set of all association rules with *min_conf* = 100%. The itemsets $ACTW$ and $CDW$ are the maximal frequent itemsets. Since all other frequent

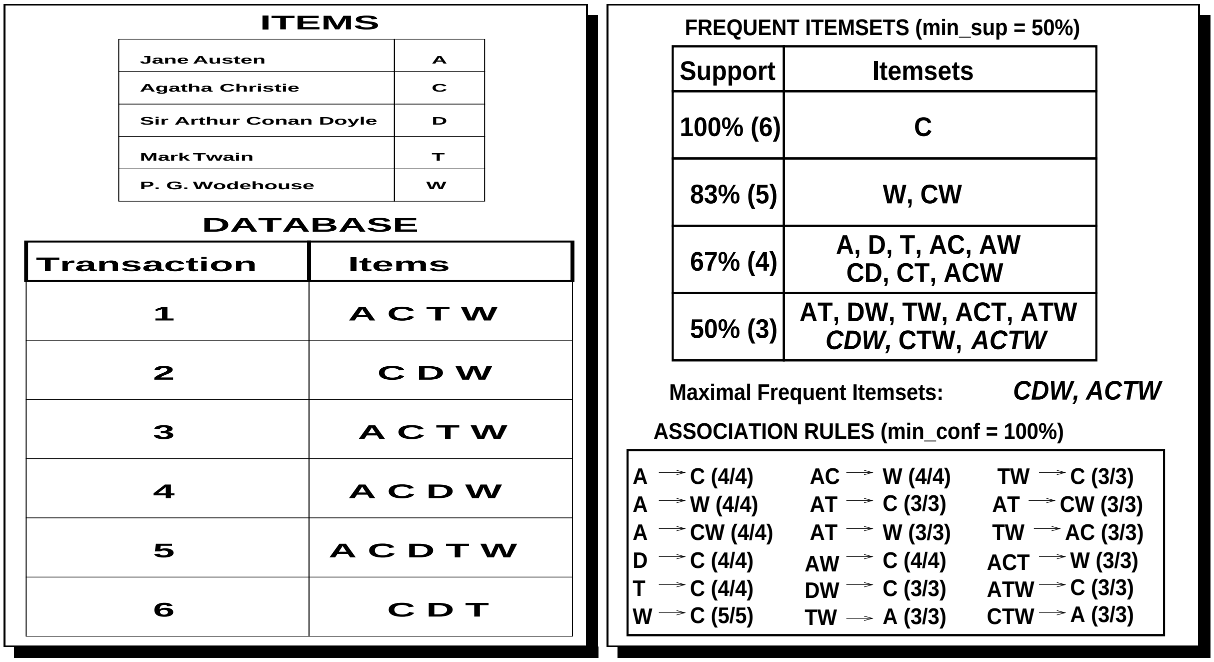


Figure 1: a) Bookstore Database, b) Frequent Itemsets and Confident Rules

itemsets are subsets of one of these two maximal itemsets, we can reduce the frequent itemset search problem to the task of enumerating only the maximal frequent itemsets. On the other hand, for generating all the confident rules, we need the support of all frequent itemsets. This can be easily accomplished once the maximal elements have been identified, by making an additional database pass, and gathering the support of all uncounted subsets.

## 3 Related Work

Several algorithms for mining associations have been proposed in the literature [1, 2, 6, 15, 19, 20, 21, 23, 26, 27]. The *Apriori* algorithm [2] is the best known previous algorithm, and it uses an efficient candidate generation procedure, such that only the frequent itemsets at a level are used to construct candidates at the next level. However, it requires multiple database scans, as many as the longest frequent itemset. The DHP algorithm [23] tries to reduce the number of candidates by collecting approximate counts in the previous level. Like *Apriori* it requires as many database passes as the longest itemset. The *Partition* algorithm [26] 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 or locally frequent itemsets, and in the second pass it counts their global support. Partition may enumerate too many false positives in the first pass, i.e., itemsets locally frequent in some partition but not globally frequent. If this local frequent set doesn't fit in memory, then additional database scans will be required. The DLG [28] algorithm uses a bit-vector per item, noting the tids where the item occurred. It generates frequent itemsets via logical AND operations on the bit-vectors. However, DLG assumes that the bit vectors fit in memory, and thus scalability could be a problem for databases with millions of transactions. The DIC algorithm [6] dynamically counts candidates of varying length as the database scan progresses, and thus is able to reduce the number of scans over Apriori. 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 [29], and [27] presents an exact algorithm that finds all rules using sampling. The AS-CPA algorithm and its sampling versions [20] build on top of *Partition* and produce a much smaller set of potentially frequent candidates. It requires at most two database scans. Approaches using only general-purpose DBMS systems and relational algebra operations have also been studied [14, 15]. Detailed architectural alternatives in the tight-integration of association mining with DBMS were presented in [25]. They also pointed out the benefits of using the vertical database layout.

All the above algorithms generate all possible frequent itemsets. Methods for finding the maximal elements include *All-MFS* [12], which is a randomized algorithm to discover maximal frequent itemsets. The *Pincer-Search* algorithm [19] not only constructs the candidates in a bottom-up manner like *Apriori*, but also starts a top-down search at the same time. This can help in reducing the number of database scans. *MaxMiner* [5] is another algorithm for finding the maximal elements. It uses efficient pruning techniques to quickly narrow the search space. Our new algorithms range from those that generate all the frequent itemsets, to hybrid schemes that generate some maximal along with the remaining itemsets. It is worth noting that since the enumeration task is computationally challenging a number of parallel algorithms have also been proposed [3, 7, 13, 31]

# 4 Itemset Enumeration: Lattice-Based Approach

Before embarking on the algorithm description, we will briefly review some terminology from lattice theory (see [8] for a good introduction).

**Definition 1** *Let $P$ be a set. A* **partial order** *on $P$ is a binary relation $\leq$, such that for all $X, Y, Z \in P$, the relation is:*

*1) Reflexive: $X \leq X$.*

*2) Anti-Symmetric:* $X \leq Y$ *and* $Y \leq X$*, implies* $X = Y$*.*

*3) Transitive:* $X \leq Y$ *and* $Y \leq Z$*, implies* $X \leq Z$*.*

*The set* $P$ *with the relation* $\leq$ *is called an* **ordered set***.*

**Definition 2** *Let* $P$ *be an ordered set, and let* $X, Z, Y \in P$*. We say* $X$ *is* **covered by** $Y$*, denoted* $X \sqsubset Y$*, if* $X < Y$ *and* $X \leq Z < Y$*, implies* $Z = X$*, i.e., if there is no element* $Z$ *of* $P$ *with* $X < Z < Y$*.*

**Definition 3** *Let* $P$ *be an ordered set, and let* $S \subseteq P$*. An element* $X \in P$ *is an* **upper bound** *(***lower bound***) of* $S$ *if* $s \leq X$ *(*$s \geq X$*) for all* $s \in S$*. The least upper bound, also called* **join***, of* $S$ *is denoted as* $\bigvee S$*, and the greatest lower bound, also called* **meet***, of* $S$ *is denoted as* $\bigwedge S$*. The greatest element of* $P$*, denoted* $\top$*, is called the* **top element***, and the least element of* $P$*, denoted* $\bot$*, is called the* **bottom element***.*

**Definition 4** *Let* $L$ *be an ordered set.* $L$ *is called a* **join (meet) semilattice** *if* $X \vee Y$ *(*$X \wedge Y$*) exists for all* $X, Y \in L$*.* $L$ *is called a* **lattice** *if it is both a join and meet semilattice, i.e., if* $X \vee Y$ *and* $X \wedge Y$ *exist for all pairs of elements* $X, Y \in L$*.* $L$ *is a* **complete lattice** *if* $\bigvee S$ *and* $\bigwedge S$ *exist for all subsets* $S \subseteq L$*. A ordered set* $M \subset L$ *is a* **sublattice** *of* $L$ *if* $X, Y \in M$ *implies* $X \vee Y \in M$ *and* $X \wedge Y \in M$*.*

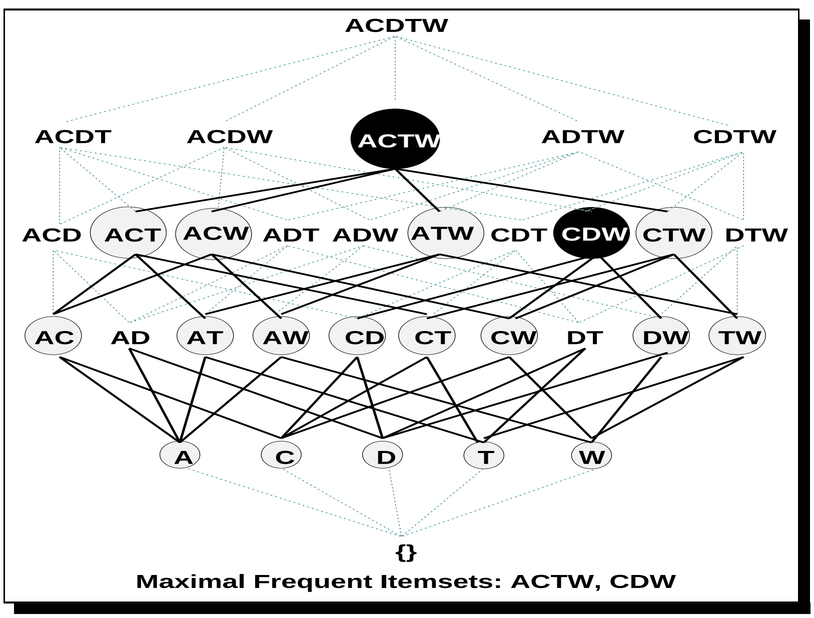


Figure 2: The Complete Powerset Lattice $\mathcal{P}(\mathcal{I})$

For set $S$, the ordered set $\mathcal{P}(S)$, the power set of $S$, is a complete lattice in which join and meet are given by union and intersection, respectively:

$$\bigvee\{A_i \mid i \in I\} = \bigcup_{i \in I} A_i \qquad \bigwedge\{A_i \mid i \in I\} = \bigcap_{i \in I} A_i$$

The top element of $\mathcal{P}(S)$ is $\top = S$, and the bottom element of $\mathcal{P}(S)$ is $\bot = \{\}$. For any $L \subseteq \mathcal{P}(S)$, $L$ is called a *lattice of sets* if it is closed under finite unions and intersections, i.e., $(L; \subseteq)$ is a lattice with the partial order specified by the subset relation $\subseteq$, $X \vee Y = X \cup Y$, and $X \wedge Y = X \cap Y$.

Figure 2 shows the powerset lattice $\mathcal{P}(\mathcal{I})$ of the set of items in our example database $\mathcal{I} = \{A, C, D, T, W\}$. Also shown are the frequent (grey circles) and maximal frequent itemsets (black circles). It can be observed that the set of all frequent itemsets forms a meet semilattice since it is closed under the meet operation, i.e., for any frequent itemsets $X$, and $Y$, $X \cap Y$ is also frequent. On the other hand, it doesn't form a join semilattice, since $X$ and $Y$ frequent, doesn't imply $X \cup Y$ is frequent. It can be mentioned that the infrequent itemsets form a join semilattice.

**Lemma 1** *All subsets of a frequent itemsets are frequent.*

The above lemma is a consequence of the closure under meet operation for the set of frequent itemsets. As a corollary, we get that all supersets of an infrequent itemset are infrequent. This observation forms the basis of a very powerful pruning strategy in a bottom-up search procedure for frequent itemsets, which has been leveraged in many association mining algorithms [2, 23, 26]. Namely, only the itemsets found to be frequent at the previous level need to be extended as candidates for the current level. However, the lattice formulation makes it apparent that we need not restrict ourselves to a purely bottom-up search.

**Lemma 2** *The maximal frequent itemsets uniquely determine all frequent itemsets.*

This observation tells us that our goal should be to devise a search procedure that quickly identifies the maximal frequent itemsets. In the following sections we will see how to do this efficiently.

## 4.1 Support Counting

**Definition 5** *A lattice $L$ is said to be* **distributive** *if for all $X, Y, Z \in L$, $X \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$.*

**Definition 6** *Let $L$ be a lattice with bottom element $\bot$. Then $X \in L$ is called an* **atom** *if $\bot \sqsubset X$, i.e., $X$ covers $\bot$. The set of atoms of $L$ is denoted by $\mathcal{A}(L)$.*

**Definition 7** *A lattice $L$ is called a* **Boolean lattice** *if*

*1) It is distributive.*

*2) It has $\top$ and $\bot$ elements.*

*3) Each member $X$ of the lattice has a complement.*

We begin by noting that the powerset lattice $\mathcal{P}(\mathcal{I})$ on the set of database items $\mathcal{I}$ is a *Boolean* lattice, with the complement of $X \in L$ given as $\mathcal{I} \backslash X$. The set of atoms of the powerset lattice corresponds to the set of items, i.e., $\mathcal{A}(\mathcal{P}(\mathcal{I})) = \mathcal{I}$. We associate with each atom (database item) $X$ its *tid-list*, denoted $\mathcal{L}(X)$, which is a list of all transaction identifiers containing the atom. Figure 3 shows the tid-lists for the atoms in our example database. For example consider atom $A$. Looking at the database in Figure 1, we see that $A$ occurs in transactions 1, 3, 4, and 5. This forms the tid-list for atom $A$.

| A | C | D | T | W |
|---|---|---|---|---|
| 1 | 1 | 2 | 1 | 1 |
| 3 | 2 | 4 | 3 | 2 |
| 4 | 3 | 5 | 5 | 3 |
| 5 | 4 | 6 | 6 | 4 |
| | 5 | | | 5 |
| | 6 | | | |

Figure 3: Tid-List for the Atoms

**Lemma 3** ([8]) *For a finite boolean lattice $L$, with $X \in L$, $X = \bigvee\{Y \in \mathcal{A}(L) \mid Y \leq X\}$.*

In other words every element of a boolean lattice is given as a join of a subset of the set of atoms. Since the powerset lattice $\mathcal{P}(\mathcal{I})$ is a boolean lattice, with the join operation corresponding to set union, we get

**Lemma 4** *For any $X \in \mathcal{P}(\mathcal{I})$, let $J = \{Y \in \mathcal{A}(\mathcal{P}(\mathcal{I})) \mid Y \leq X\}$. Then $X = \bigcup_{Y \in J} Y$, and $\sigma(X) = \mid \bigcap_{Y \in J} \mathcal{L}(Y) \mid$.*

The above lemma states that any itemset can be obtained is a join of some atoms of the lattice, and the support of the itemset can be obtained by intersecting the tid-list of the atoms. We can generalize this lemma to a set of itemsets:

**Lemma 5** *For any $X \in \mathcal{P}(\mathcal{I})$, let $X = \bigcup_{Y \in J} Y$. Then $\sigma(X) = \mid \bigcap_{Y \in J} \mathcal{L}(Y) \mid$.*

This lemma says that if an itemset is given as a union of a set of itemsets in $J$, then its support is given as the intersection of tid-lists of elements in $J$. In particular we can determine the support of any $k$-itemset by simply intersecting the tid-lists of any two of its $(k-1)$ length subsets. A simple check on the cardinality of the resulting tid-list tells us whether the new itemset is frequent or not. Figure 4 shows this process pictorially. It shows the initial database with the tid-list for each item (i.e., the atoms). The intermediate tid-list for $CD$ is obtained by intersecting the lists of $C$ and $D$, i.e., $\mathcal{L}(CD) = \mathcal{L}(C) \cap \mathcal{L}(D)$. Similarly, $\mathcal{L}(CDW) = \mathcal{L}(CD) \cap \mathcal{L}(CW)$, and so on. Thus, only the lexicographically first two subsets at the previous level are required to compute the support of an itemset at any level.

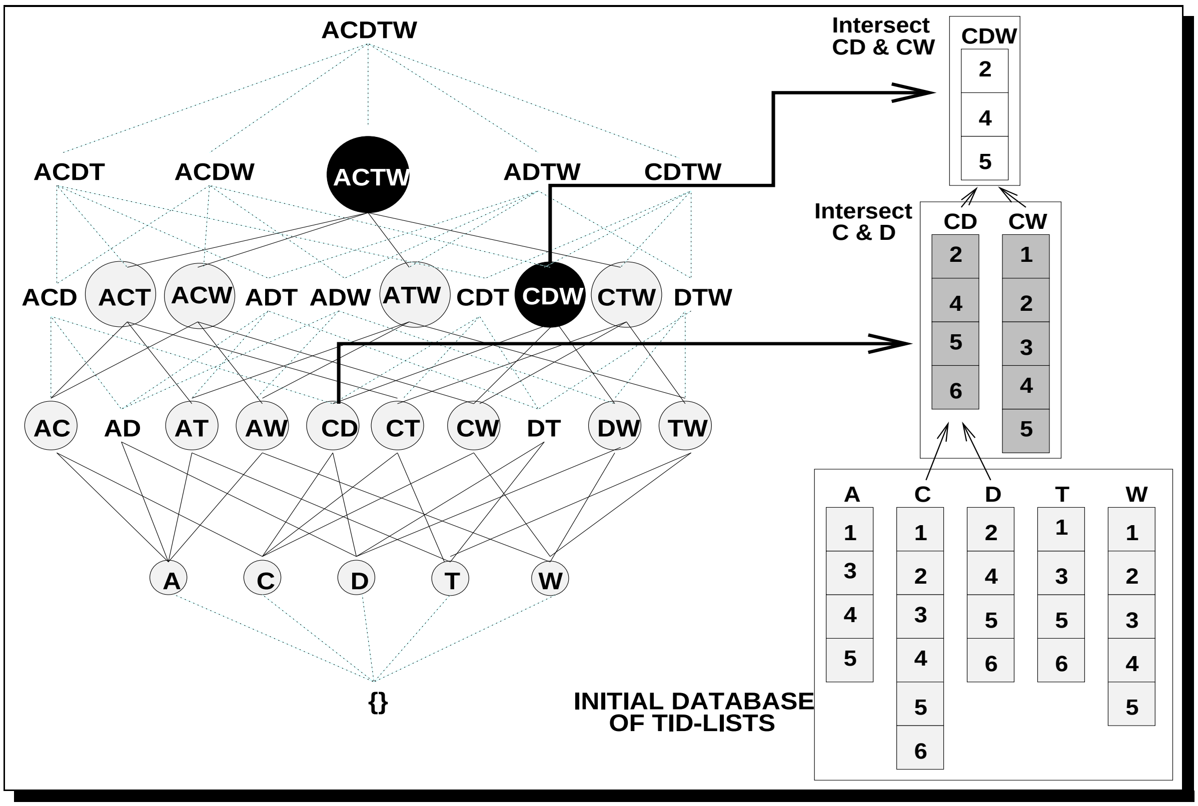


Figure 4: Computing Support of Itemsets via Tid-List Intersections

**Lemma 6** *Let $X$ and $Y$ be two itemsets, with $X \subseteq Y$. Then $\mathcal{L}(X) \supseteq \mathcal{L}(Y)$.*

Proof: Follows from the definition of support. ■

This lemma states that if $X$ is a subset of $Y$, then the cardinality of the tid-list of $Y$ (i.e., its support) must be less than or equal to the cardinality of the tid-list of $X$. A practical and important consequence of the above lemma is that the cardinalities of intermediate tid-lists shrink as we move up the lattice. This results in very fast intersection and support counting.

## 4.2 Lattice Decomposition: Prefix-Based Classes

If we had enough main-memory we could enumerate all the frequent itemsets by traversing the powerset lattice, and performing intersections to obtain itemset supports. In practice, however, we have only a limited amount of main-memory, and all the intermediate tid-lists will not fit in memory. This brings up a natural question: can we decompose the original lattice into smaller pieces such that each portion can be solved independently in main-memory! We address this question below.

**Definition 8** *Let $P$ be a set. An* **equivalence relation** *on $P$ is a binary relation $\equiv$ such that for all $X, Y, Z \in P$, the relation is:*

*1) Reflexive: $X \equiv X$.*

*2) Symmetric: $X \equiv Y$ implies $Y \equiv X$.*

*3) Transitive: $X \equiv Y$ and $Y \equiv Z$, implies $X \equiv Z$.*

*The equivalence relation partitions the set $P$ into disjoint subsets called* **equivalence classes***. The equivalence class of an element $X \in P$ is given as $[X] = \{Y \in P \mid X \equiv Y\}$.*

Define a function $p : \mathcal{P}(\mathcal{I}) \times N \mapsto \mathcal{P}(\mathcal{I})$ where $p(X, k) = X[1 : k]$, the $k$ length prefix of $X$. Define an equivalence relation $\theta_k$ on the lattice $\mathcal{P}(\mathcal{I})$ as follows: $\forall X, Y \in \mathcal{P}(\mathcal{I}),\ X \equiv_{\theta_k} Y \Leftrightarrow p(X, k) = p(Y, k)$. That is, two itemsets are in the same class if they share a common $k$ length prefix. We therefore call $\theta_k$ a *prefix-based* equivalence relation.

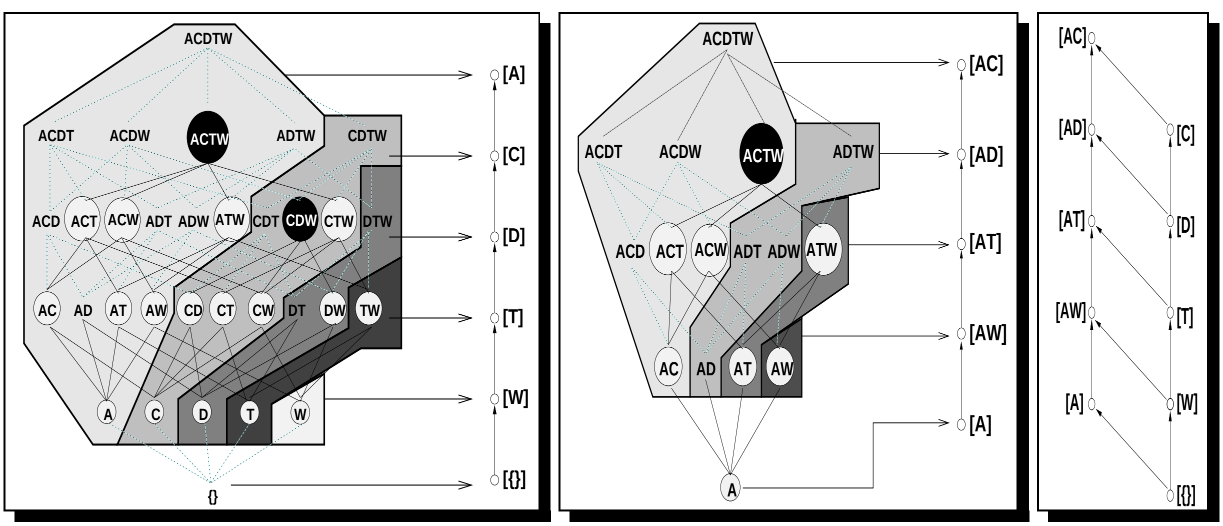


Figure 5: Equivalence Classes of a) $\mathcal{P}(\mathcal{I})$ Induced by $\theta_1$, and b) $[A]_{\theta_1}$ Induced by $\theta_2$; c) Final Lattice of Independent Classes

Figure 5a shows the lattice induced by the equivalence relation $\theta_1$ on $\mathcal{P}(\mathcal{I})$, where we collapse all itemsets with a common 1 length prefix into an equivalence class. The resulting set or lattice of equivalence classes is $\{[A],[C],[D],[T],[W]\}$.

**Lemma 7** *Each equivalence class $[X]_{\theta_k}$ induced by the equivalence relation $\theta_k$ is a sub-lattice of $\mathcal{P}(\mathcal{I})$.*

PROOF: Let $U$ and $V$ be any two elements in the class $[X]$, i.e., $U, V$ share the common prefix $X$. $U \vee V = U \cup V \supseteq X$ implies that $U \vee V \in [X]$, and $U \wedge V = U \cap V \supseteq X$ implies that $U \wedge V \in [X]$. Therefore $[X]_{\theta_k}$ is a sublattice of $\mathcal{P}(\mathcal{I})$. ■

Each $[X]_{\theta_1}$ is itself a boolean lattice with its own set of atoms. For example, the atoms of $[A]_{\theta_1}$ are $\{AC, AD, AT, AW\}$, and the top and bottom elements are $\top = ACDTW$, and $\perp = A$. By the application of Lemmas 4, and 5, we can generate all the supports of the itemsets in each class (sub-lattice) by intersecting the tid-list of atoms or any two subsets at the previous level. If there is enough main-memory to hold temporary tid-lists for each class, then we can solve each $[X]_{\theta_1}$ independently. Another interesting feature of the equivalence classes is that the links between classes denote dependencies. That is to say, if we want to prune an itemset if there exists at least one infrequent subset (see Lemma 1), then we have to process the classes in a specific order. In particular we have to solve the classes from bottom to top, which corresponds to a reverse lexicographic order, i.e., we process $[W]$, then $[T]$, followed by $[D]$, then $[C]$, and finally $[A]$. This guarantees that all subset information is available for pruning.

In practice we have found that the one level decomposition induced by $\theta_1$ is sufficient. However, in some cases, a class may still be too large to be solved in main-memory. In this scenario, we apply recursive class decomposition. Let's assume that $[A]$ is too large to fit in main-memory. Since $[A]$ is itself a boolean lattice, it can be decomposed using $\theta_2$. Figure 5b shows the equivalence class lattice induced by applying $\theta_2$ on $[A]$, where we collapse all itemsets with a common 2 length prefix into an equivalence class. The resulting set of classes are $\{[AC],[AD],[AT],[AW]\}$. Like before, each class can be solved independently, and we can solve them in reverse lexicographic order to enable subset pruning. The final set of independent classes obtained by applying $\theta_1$ on $\mathcal{P}(\mathcal{I})$ and $\theta_2$ on $[A]$ is shown in Figure 5c. As before, the links show the pruning dependencies that exist among the classes. Depending on the amount of main-memory available we can recursively partition large classes into smaller ones, until each class is small enough to be solved independently in main-memory.

## 4.3 Search for Frequent Itemsets

In this section we discuss efficient search strategies for enumerating the frequent itemsets within each class. The actual pseudo-code and implementation details will be discussed in Section 5.

### 4.3.1 Bottom-Up Search

The bottom-up search is based on a recursive decomposition of each class into smaller classes induced by the equivalence relation $\theta_k$. Figure 6 shows the decomposition of $[A]_{\theta_1}$ into smaller classes, and the resulting lattice of equivalence classes. Also shown are the atoms within each class, from which all other elements of a class can be determined. The equivalence class lattice can be traversed in either depth-first or breadth-first manner. In this paper we will only show results for a breadth-first traversal, i.e., we first process the classes $\{[AC], [AT], [AW]\}$, followed by the classes $\{[ACT], [ACW], [ATW]\}$, and finally $[ACTW]$. For computing the support of any itemset, we simply intersect the tid-lists of two of its subsets at the previous level. Since the search is breadth-first, this technique enumerates all frequent itemsets.

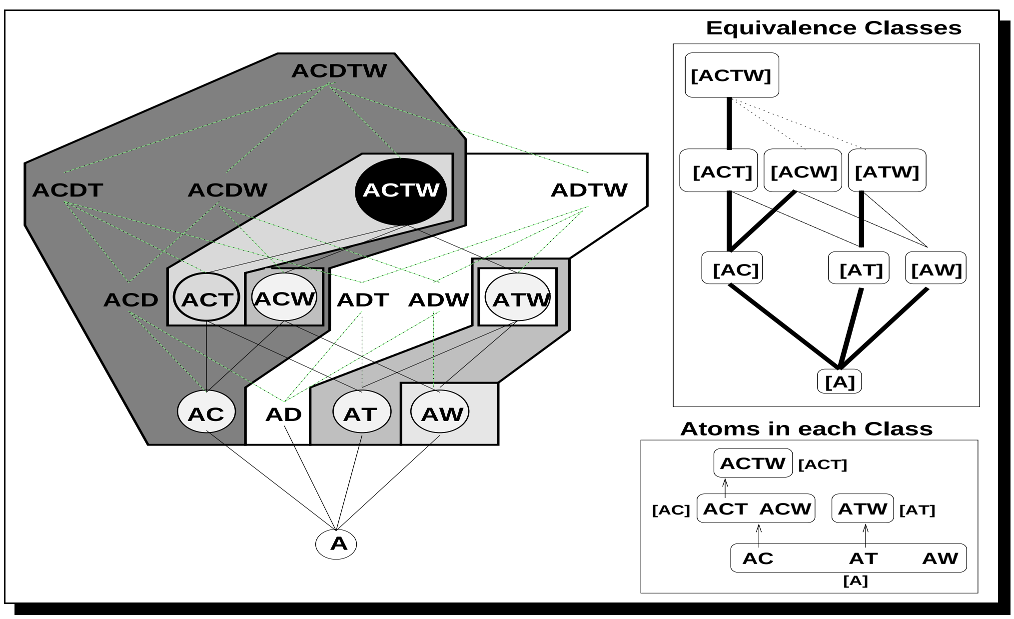


Figure 6: Bottom-Up Search

### 4.3.2 Top-Down Search

The top-down approach starts with the top element of the lattice. Its support is determined by intersecting the tid-lists of the atoms. This requires a $k$-way intersection if the top element is a $k$-itemset. The advantage of this approach is that if the maximal element is fairly large then one can quickly identify it, and one can avoid finding the support of all its subsets. The search starts with the top element. If it is frequent we are done. Otherwise, we check each subset at the next level. This process is repeated until we have identified all minimal infrequent itemsets. Figure 7 depicts the top-down search. This scheme enumerates only the maximal frequent itemsets within each sub-lattice. However, the maximal elements of a sub-lattice may not be globally maximal. It can thus generate some non-maximal itemsets. The search starts with the top element $ACDTW$. Since it is infrequent we have to check each of its four length 4 subsets. Out of these only $ACTW$ is frequent, so we mark all its subsets as frequent as well. We then examine the unmarked length 3 subsets of the three infrequent subsets. The search stops when $AD$, the minimal infrequent itemset has been identified.

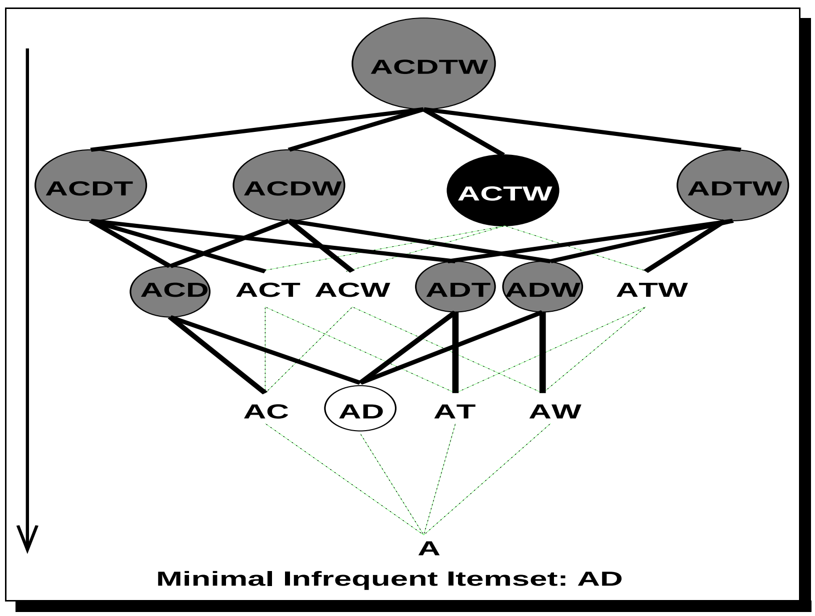


Figure 7: Top-Down Search (gray circles represent infrequent itemsets, black circle the maximal frequent, and white circle the minimal infrequent set)

#### 4.3.3 Hybrid Search

The hybrid scheme is based on the intuition that the greater the support of an frequent itemset the more likely it is to be a part of a longer frequent itemset. There are two main steps in this approach. We begin with the set of atoms of the class sorted in descending order based on their support. The first, hybrid phase starts by intersecting the atoms one at a time, beginning with the atom with the highest support, generating longer and longer frequent itemsets. The process stops when an extension becomes infrequent. We then enter the second, bottom-up phase. The remaining atoms are combined with the atoms in the first set in a breadth-first fashion described above to generate all other frequent itemsets. Figure 8 illustrates this approach (just for this case, to better show the bottom-up phase, we have assumed that $AD$ and $ADW$ are also frequent). The search starts by reordering the 2-itemsets according to support, the most frequent first. We combine $AC$ and $AW$ to obtain the frequent itemset $ACW$. We extend it with the next pair $AT$, to get $ACTW$. Extension by $AD$ fails. This concludes the hybrid phase, having found the maximal set $ACTW$. In the bottom-up phase, $AD$ is combined with all previous pairs to ensure a complete search, producing the equivalence class $[AD]$, which can be solved using a bottom-up search. This hybrid search strategy requires only 2-way intersections. It enumerates the "long" maximal frequent itemsets discovered in the hybrid phase, and also the non-maximal ones found in the bottom-up phase. Another modification of this scheme is to recursively substitute the second bottom-up search with a hybrid search, so that mainly the maximal frequent elements are enumerated.

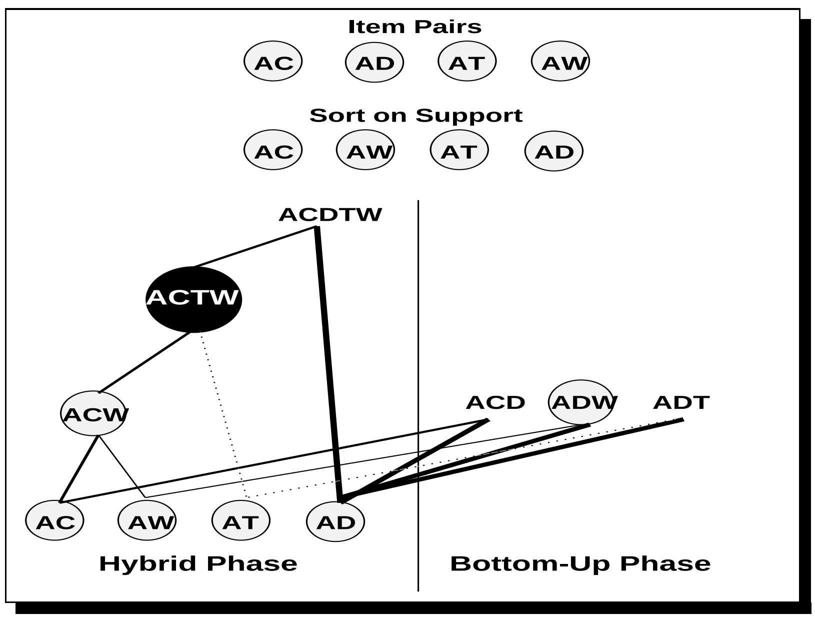


Figure 8: Hybrid Search

## 4.4 Generating Smaller Classes: Maximal Clique Approach

In this section we show how to produce smaller sub-lattices or equivalence classes compared to the pure prefix-based approach, by using additional information. Smaller sub-lattices have fewer atoms and can save unnecessary intersections. For example, if there are $k$ atoms, then we have to perform $\binom{k}{2}$ intersections for the next level in the bottom-up approach. Fewer atoms thus lead to fewer intersections in the bottom-up search. Fewer atoms also reduce the number of intersections in the hybrid scheme, and lead to smaller maximum element size in the top-down search.

**Definition 9** *Let $P$ be a set. A* **pseudo-equivalence relation** *on $P$ is a binary relation $\equiv$ such that for all $X, Y \in P$, the relation is:*

*1) Reflexive: $X \equiv X$.*

*2) Symmetric: $X \equiv Y$ implies $Y \equiv X$.*

*The pseudo-equivalence relation partitions the set $P$ into possibly overlapping subsets called* **pseudo-equivalence classes**.

**Definition 10** *A* **graph** *consists of a set of elements $V$ called* **vertices**, *and a set of lines connecting pairs of vertices, called the* **edges**. *A graph is* **complete** *if there is an edge between all pairs of vertices. A complete subgraph of a graph is called a* **clique**.

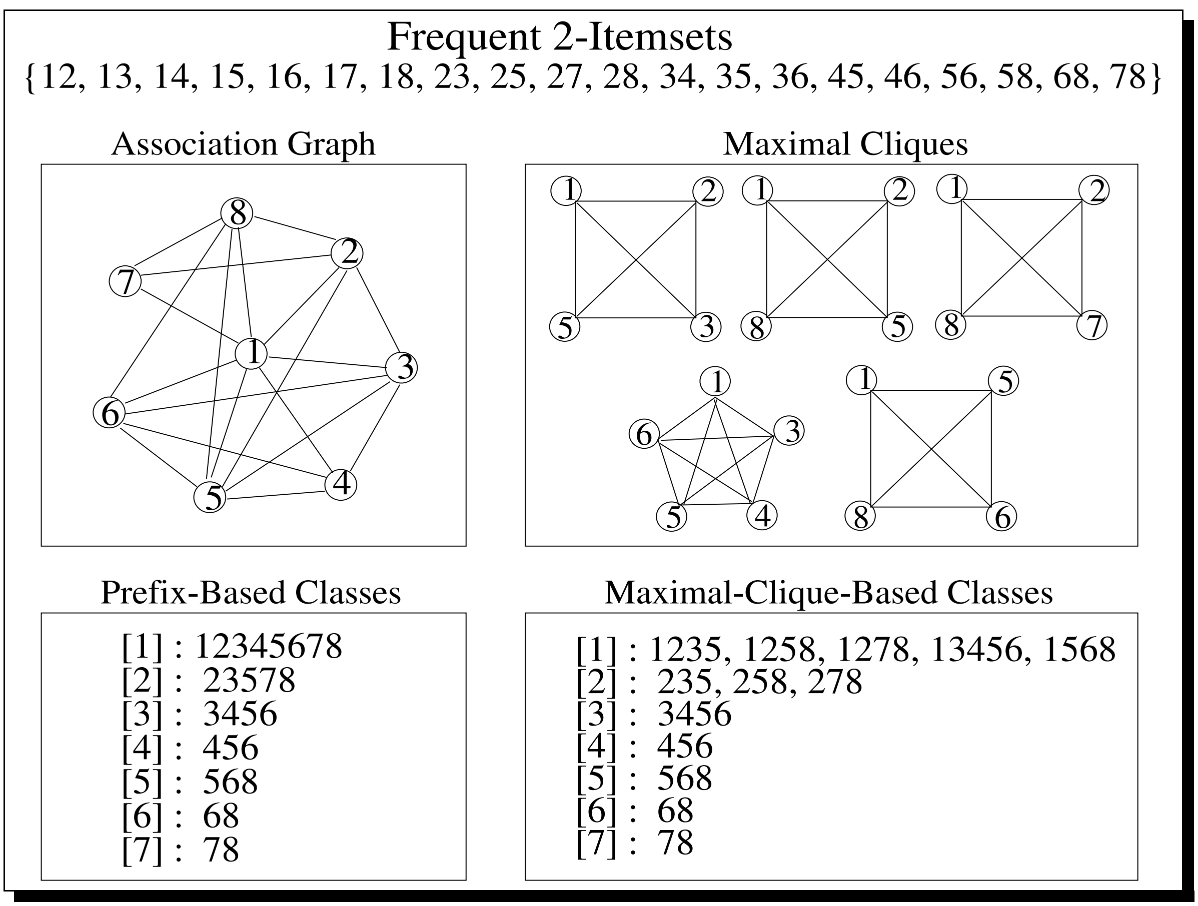


Figure 9: Maximal Cliques of the Association Graph; Prefix-Based and Maximal-Clique-Based Classes

Let $\mathcal{F}_k$ denote the set of frequent $k$-itemsets. Define an $k$*-association graph*, given as $G_k = (V, E)$, with the vertex set $V = \{X \mid X \in \mathcal{F}_1\}$, and edge set $E = \{(X, Y) \mid X, Y \in V \text{ and } \exists\ Z \in \mathcal{F}_{(k+1)}, \text{ such that } X, Y \subset Z\}$. Let $M_k$ denote the set of maximal cliques in $G_k$. Figure 9 shows the association graph $G_1$ for the example $\mathcal{F}_2$ shown. Its maximal clique set $M_1 = \{1235, 1258, 1287, 13456, 1568\}$.

Define a pseudo-equivalence relation $\phi_k$ on the lattice $\mathcal{P}(\mathcal{I})$ as follows: $\forall X, Y \in \mathcal{P}(\mathcal{I}),\ X \equiv_{\phi_k} Y \Leftrightarrow \exists\ C \in M_k$ such that $X, Y \subseteq C$ and $p(X, k) = p(Y, k)$. That is, two itemsets are related, i.e, they are in the same *pseudo-class*, if they are subsets of the same maximal clique and they share a common prefix of length $k$. We therefore call $\phi_k$ a *maximal-clique-based* pseudo-equivalence relation.

**Lemma 8** *Each pseudo-class $[X]_{\phi_k}$ induced by the pseudo-equivalence relation $\phi_k$ is a sub-lattice of $\mathcal{P}(\mathcal{I})$.*

PROOF: Let $U$ and $V$ be any two elements in the class $[X]$, i.e., $U, V$ share the common prefix $X$ and there exists a maximal clique $C \in M_k$ such that $U, V \subseteq C$. Clearly, $U \cup V \subseteq C$, and $U \cap V \subseteq C$. Furthermore, $U \vee V = U \cup V \supseteq X$ implies that $U \vee V \in [X]$, and $U \wedge V = U \cap V \supseteq X$ implies that $U \wedge V \in [X]$. ∎

Thus, each pseudo-class $[X]_{\phi_1}$ is a boolean lattice, and the supports of all elements of the lattice can be generated by applying Lemmas 4, and 5 on the atoms, and using any of the three search strategies described above.

**Lemma 9** *Let $\aleph_k$ denote the set of pseudo-classes of the maximal-clique-based relation $\phi_k$. Each pseudo-class $[Y]_{\phi_k}$ induced by the prefix-based relation $\phi_k$ is a subset of some class $[X]_{\theta_k}$ induced by $\theta_k$. Conversely, each $[X]_{\theta_k}$, is the union of a set of pseudo-classes $\Psi$, given as $[X]_{\theta_k} = \bigcup\{[Z]_{\phi_k} \mid Z \in \Psi \subseteq \aleph_k\}$.*

PROOF: Let $\Gamma(X)$ denote the neighbors of $X$ in the graph $G_k$. Then $[X]_{\theta_k} = \{Z \mid X \subseteq Z \subseteq \{X, \Gamma(X)\}\}$. In other words, $[X]$ consists of elements with the prefix $X$ and extended by all possible subsets of the neighbors of $X$ in the graph $G_k$. Since any clique $Y$ is a subset of $\{Y, \Gamma(Y)\}$, we have that $[Y]_{\phi_k} \subseteq [X]_{\theta_k}$, where $Y$ is a prefix of $X$. On the other hand it is easy to show that $[X]_{\theta_k} = \bigcup\{[Y]_{\phi_k} \mid Y \text{ is a prefix of } X\}$. ∎

This lemma states that each pseudo-class of $\phi_k$ is a refinement of (i.e., is smaller than) some class of $\theta_k$. By using the relation $\phi_k$ instead of $\theta_k$, we can therefore, generate smaller sub-lattices. These sub-lattices require less memory, and can be processed independently using any of the three search strategies described above. Figure 9 contrasts the classes (sub-lattices) generated by $\phi_1$ and $\theta_1$. It is apparent that $\phi_1$ generates smaller classes. For example, the prefix class $[1] = 12345678$ is one big class containing all the elements, while the maximal-clique classes for $[1] = \{1235, 1258, 1278, 13456, 1568\}$. Each of these classes is much smaller than the prefix-based class. The smaller classes of $\phi_k$ come at a cost, since the enumeration of maximal cliques can be computationally expensive. For general graphs the maximal clique decision problem is NP-Complete [10]. However, the $k$-association graph is usually sparse and the maximal cliques can be enumerated efficiently. As the edge density of the association graph increases the clique based approaches may suffer. $\phi_k$ should thus be used only when $G_k$ is not too dense. Some of the factors affecting the edge density include decreasing support and increasing transaction size. The effect of these parameters is studied in the experimental section.

### 4.4.1 Maximal Clique Generation

We modified Bierstone's algorithm [22] for generating maximal cliques in the $k$-association graph. For a class $[x]$, and $y \in [x]$, $y$ is said to *cover* the subset of $[x]$, given by $cov(y) = [y] \cap [x]$. 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 [1], shown in figure 9. $cov(2) = \{3,5,7,8\} = [2]$. Similarly, for our example, $cov(y) = [y]$, for all $y \in [1]$, since each $[y] \subseteq [1]$. The covering set of [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 10 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.

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

Figure 10: The Maximal Clique Generation Algorithm

**Weak Maximal Cliques** For some database parameters, the edge density of the $k$-association graph may be too high, resulting in large cliques with significant overlap among them. In these cases, not only does the clique generation take more time, but redundant frequent itemsets may also be discovered within each sublattice. 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 *$\alpha$-related*, if $\frac{|X \cap Y|}{|X \cup Y|} \geq \alpha$, i.e., the ratio of the common elements to the distinct elements of the cliques is greater than or equal to the threshold $\alpha$. A *weak maximal* clique, $Z = \{X \cup Y\}$, is generated by collapsing the two cliques into one, provided they are $\alpha$-related. During clique generation only weak maximal cliques are generated for some user specified value of $\alpha$. Note that for $\alpha = 1$, we obtain regular maximal cliques, while for $\alpha = 0$, we obtain a single clique. Preliminary experiments indicate that using an appropriate value of $\alpha$, most of the overhead of redundant cliques can be avoided. We found $\alpha = 0.5$ to work well in practice.

# 5 Algorithm Design and Implementation

In this section we describe several new algorithms for efficient enumeration of frequent itemsets. The first step involves the computation of the frequent items and 2-itemsets. The next step generates the sub-lattices (classes) by applying either the prefix-based equivalence relation $\theta_1$, or the maximal-clique-based pseudo-equivalence relation $\phi_1$ on the set of frequent 2-itemsets $\mathcal{F}_2$. The sub-lattices are then processed one at a time in reverse lexicographic order in main-memory using either bottom-up, top-down or hybrid search. We will now describe these steps in some more detail.

## 5.1 Computing Frequent 1-Itemsets and 2-Itemsets

Most of the current association algorithms [2, 6, 20, 23, 26, 27] assume a *horizontal* database layout, such as the one shown in Figure 1, consisting of a list of transactions, where each transaction has an identifier followed by a list of items in that transaction. In contrast our algorithms use the *vertical* database format, such as the one shown in Figure 3, where we maintain a disk-based tid-list for each item. This enables us to check support via simple tid-list intersections.

**Computing $\mathcal{F}_1$** Given the vertical tid-list database, all frequent items can be found in a single database scan. For each item, we simply read its tid-list from disk into memory. We then scan the tid-list, incrementing the item's support for each entry.

**Computing $\mathcal{F}_2$** Let $N = |\mathcal{I}|$ be the number of frequent items, and $A$ the average id-list size in bytes. A naive implementation for computing the frequent 2-itemsets requires $\binom{N}{2}$ id-list intersections for all pairs of items. The amount of data read is $A \cdot N \cdot (N-1)/2$, which corresponds to around $N/2$ data scans. This is clearly inefficient. Instead of the naive method one could use two alternate solutions:

1. Use a preprocessing step to gather the counts of all 2-sequences above a user specified lower bound. Since this information is invariant, it has to be computed once, and the cost can be amortized over the number of times the data is mined.

2. Perform a vertical to horizontal transformation on-the-fly. This can be done quite easily. For each item $i$, we scan its tid-list into memory. We insert item $i$ in an array indexed by tid for each $t \in \mathcal{L}(i)$. For example, consider the id-list for item $A$, shown in Figure 3. We read the first tid 1, and then insert $A$ in the array indexed by transaction 1. We repeat this process for all other items and their tidlists. Figure 11 shows how the inversion process works after the addition of each item and the complete horizontal database recovered from the vertical item tid-lists. This process entails only a trivial amount of overhead. In fact, *Partition* performs the opposite inversion from horizontal to vertical tid-list format on-the-fly, with very little cost. We also implemented appropriate memory management by recovering only a block of database at a time, so that the recovered transactions fit in memory. Finally, we optimize the computation of $\mathcal{F}_2$ by directly updating the counts of candidate pairs in an upper triangular 2D array.

The experiments reported in Section 7 use the horizontal recovery method for computing $\mathcal{F}_2$. As we shall demonstrate, this inversion can be done quite efficiently.

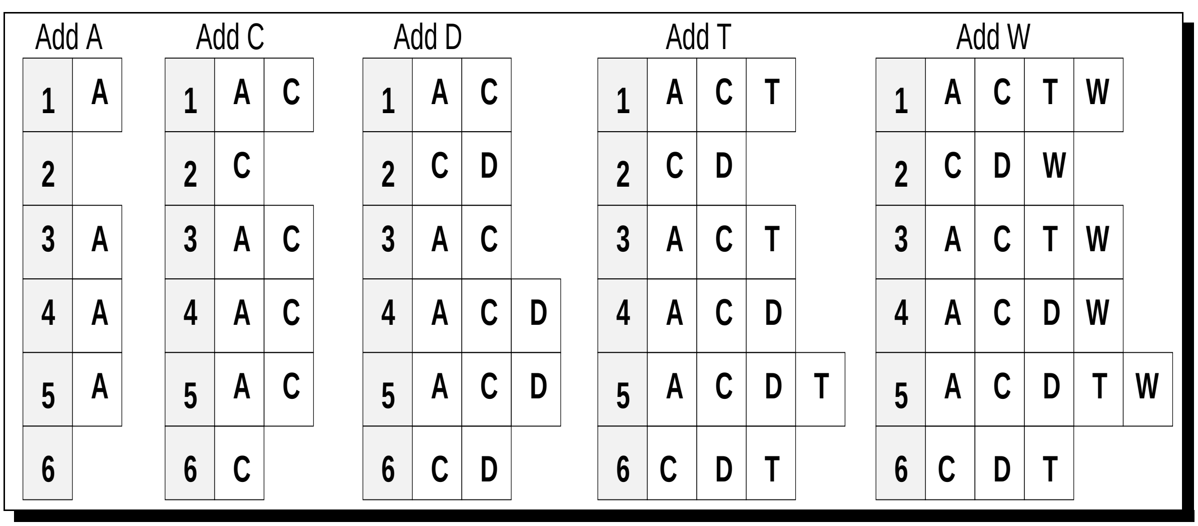


Figure 11: Vertical-to-Horizontal Database Recovery

## 5.2 Search Implementation

**Bottom-Up Search** Figure 12 shows the pseudo-code for the bottom-up search. The input to the procedure is a set of atoms of a sub-lattice $S$. Frequent itemsets are generated by intersecting the tid-lists of all distinct pairs of atoms and checking the cardinality of the resulting tid-list. A recursive procedure call is made with those itemsets found to be frequent at the current level. This process is repeated until all frequent itemsets have been enumerated. In terms of memory management it is easy to see that we need memory to store intermediate tid-lists for at most two consecutive levels. Once all the frequent itemsets for the next level have been generated, the itemsets at the current level can be deleted.

**Bottom-Up($S$):**
**for** all atoms $A_i \in S$ **do**
  $T_i = \emptyset$;
  **for** all atoms $A_j \in S$, with $j > i$ **do**
    $R = A_i \cup A_j$;
    $\mathcal{L}(R) = \mathcal{L}(A_i) \cap \mathcal{L}(A_j)$;
    **if** $\sigma(R) \geq min\_sup$ **then**
      $T_i = T_i \cup \{R\}$; $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
  **end**
**end**
**for** all $T_i \neq \emptyset$ **do** Bottom-Up($T_i$);

Figure 12: Pseudo-code for Bottom-Up Search

Since each sub-lattice is processed in reverse lexicographic order all subset information is available for itemset pruning. For fast subset checking the frequent itemsets can be stored in a hash table. However, in our experiments on synthetic data we found pruning to be of little or no benefit. This is mainly because of Lemma 6, which says that the tid-list intersection is especially efficient for large itemsets. Nevertheless, there may be databases where pruning is crucial for performance, and we can support pruning for those datasets.

**Top-Down Search** The code for top-down search is given in Figure 13. The search begins with the maximum element $R$ of the sub-lattice $S$. A check is made to see if the element is already known to be frequent. If not we perform a $k$-way intersection to determine its support. If it is frequent then we are done. Otherwise, we recursively check the support of each of its $(k-1)$-subsets. We also maintain a hash table $HT$ of itemsets known to be infrequent from previous recursive calls to avoid processing sub-lattices that have already been examined. In terms of memory management the top-down approach requires that only the tid-lists of the atoms of a class be in memory.

```
Top-Down(S):
R = ⋃{A_i ∈ S};
if R ∉ F_|R| then
    L(R) = ⋂{L(A_i) | A_i ∈ S};
    if σ(R) ≥ min_sup then
        F_|R| = F_|R| ∪ {R};
    else
        for all Y ⊂ R, with |Y| = |R| − 1
            if Y ∉ HT then
                Top-Down({A_j | A_j ∈ Y});
                if σ(Y) < min_sup then HT = HT ∪{Y};
        end
```

Figure 13: Pseudo-code for Top-Down Search

**Hybrid Search** Figure 14 shows the pseudo-code for the hybrid search. The input consists of the atom set $S$ sorted in descending order of support. The maximal phase begins by intersecting atoms one at a time until no frequent extension is possible. All the atoms involved in this phase are stored in the set $S_1$. The remaining atoms $S_2 = S \backslash S_1$ enter the bottom-up phase. For each atom in $S_2$, we intersect it with each atom in $S_1$. The frequent itemsets form the atoms of a new sub-lattice and are solved using the bottom-up search. This process is then repeated for the other atoms of $S_2$. The maximal phase requires main-memory only for the atoms, while the bottom-up phase requires memory for at most two consecutive levels.

```
Hybrid(S sorted on support):
R = A_1; S_1 = {A_1};
for all A_i ∈ S, i > 1 do /* Maximal Phase */
    R = R ∪ A_i; L(R) = L(R) ∩ L(A_i);
    if σ(R) ≥ min_sup then
        S_1 = S_1 ∪ {A_i}; F_|R| = F_|R| ∪ {R};
    else break;
end
S_2 = S − S_1;
for all B_i ∈ S_2 do /* Bottom-Up Phase */
    T_i = {X_j | σ(X_j) ≥ min_sup, L(X_j) = L(B_i) ∩ L(A_j), ∀A_j ∈ S_1};
    S_1 = S_1 ∪ {B_i};
    if T_i ≠ ∅ then Bottom-Up(T_i);
end
```

Figure 14: Pseudo-code for Hybrid Search

## 5.3 Number of Database Scans

Before processing each sub-lattice from the initial decomposition all the relevant item tid-lists are scanned into memory. The tid-lists for the atoms (frequent 2-itemsets) of each initial sub-lattice are constructed by intersecting the item tid-lists. All the other frequent itemsets are enumerated by intersecting the tid-lists of the atoms using the different search procedures. If all the initial classes have disjoint set of items, then each item's tid-list is scanned from disk only once during the entire frequent itemset enumeration process over all sub-lattices. In the general case there will be some degree of overlap of items among the different sub-lattices. However only the database portion corresponding to the frequent items will need to be scanned, which can be a lot smaller than the entire database. Furthermore, sub-lattices sharing many common items can be processed in a batch mode to minimize disk access. Thus we claim that our algorithms will usually require a small number of database scans after computing $\mathcal{F}_2$.

## 5.4 New Algorithms

The different algorithms that we propose are listed below. These algorithms differ in the the search strategy used for enumeration and in the relation used for generating independent sub-lattices.

1. **Eclat**: It uses prefix-based equivalence relation $\theta_1$ along with bottom-up search. It enumerates all frequent itemsets.

2. **MaxEclat**: It uses prefix-based equivalence relation $\theta_1$ along with hybrid search. It enumerates the "long" maximal frequent itemsets, and some non-maximal ones.

3. **Clique**: It uses maximal-clique-based pseudo-equivalence relation $\phi_1$ along with bottom-up search. It enumerates all frequent itemsets.

4. **MaxClique**: It uses maximal-clique-based pseudo-equivalence relation $\phi_1$ along with hybrid search. It enumerates the "long" maximal frequent itemsets, and some non-maximal ones.

5. **TopDown**: It uses maximal-clique-based pseudo-equivalence relation $\phi_1$ along with top-down search. It enumerates only the maximal frequent itemsets. Note that for top-down search, using the larger sub-lattices generated by $\theta_1$ is not likely to be efficient.

6. **AprClique**: It uses maximal-clique-based pseudo-equivalence relation $\phi_1$. However, unlike the algorithms described above, it uses horizontal data layout. It has two main steps:

   i) All possible subsets of the maximum element in each sub-lattice are generated and inserted in *hash trees* [2], avoiding duplicates. There is one hash tree for each length, i.e., a $k$-subset is inserted in the tree $C_k$. An internal node of the hash tree at depth $d$ contains a hash table whose cells point to nodes

```
AprClique():
for all sub-lattices S_i induced by φ_1 do
    R = ⋃{A_j ∈ S_i};
    for all k > 2 and k ≤ |R| do
        Insert each k-subset of R in C_k;
end
for all transactions t ∈ D do
    for all k-subsets s of t, with k > 2 and k ≤ |t| do
        if (s ∈ C_k) s.count++;
end
F_k = {c ∈ C_k | c.count ≥ minsup};
Set of all frequent itemsets = ⋃_k F_k;
```

Figure 15: Pseudo-code for AprClique Algorithm

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.

ii) The support counting step is similar to the *Apriori* approach. For each transaction in the database $t \in \mathcal{D}$, we form all possible $k$-subsets. We then search that subset in $C_k$ and update the count if it is found.

The database is thus scanned only once, and all frequent itemset are generated. The pseudo-code is shown in Figure 15.

# 6 The *Apriori* and *Partition* Algorithms

We now discuss *Apriori* and *Partition* in some more detail, since we will experimentally compare our new algorithms against them.

**Apriori Algorithm** *Apriori* [2] is an iterative algorithm that counts itemsets of a specific length in a given database pass. The process starts by scanning all transactions in the database and computing the frequent items. Next, a set of potentially frequent *candidate* 2-itemsets is formed from the frequent items. Another database scan is made to obtain their supports. The frequent 2-itemsets are retained for the next pass, and the process is repeated until all frequent itemsets have been enumerated. The complete algorithm is shown in figure 16. We refer the reader to [2] for additional details.

There are three main steps in the algorithm:

1. Generate candidates of length $k$ from the frequent $(k-1)$ length itemsets, by a self join on $\mathcal{F}_{k-1}$. For ex-

ample, if $\mathcal{F}_2 = \{AB, AC, AD, AE, BC, BD, BE\}$. Then $C_3 = \{ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE\}$.

2. Prune any candidate with at least one infrequent subset. As an example, $ACD$ will be pruned since $CD$ is not frequent. After pruning we get a new set $C_3 = \{ABC, ABD, ABE\}$.

3. Scan all transactions to obtain candidate supports. The candidates are stored in a hash tree to facilitate fast support counting (note: the second iteration is optimized by using an array to count candidate pairs of items, instead of storing them in a hash tree).

```
F_1 = {frequent 1-itemsets };
for (k = 2; F_{k-1} ≠ ∅; k++)
    C_k = Set of New Candidates;
    for all transactions t ∈ D
        for all k-subsets s of t
            if (s ∈ C_k) s.count++;
    F_k = {c ∈ C_k | c.count ≥ min_sup};
Set of all frequent itemsets = ⋃_k F_k;
```

Figure 16: The *Apriori* Algorithm

**Partition Algorithm** *Partition* [26] logically divides the horizontal database into a number of non-overlapping partitions. Each partition is read, and *vertical tid-lists* are formed for each item, i.e., list of all tids where the item appears. Then all locally frequent itemsets are generated via tid-list intersections. All locally frequent 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 size of a partition is chosen so that it can be accommodated in main-memory. *Partition* thus makes only two database scans. The key observation used is that a globally frequent itemset must be locally frequent in at least one partition. Thus all frequent itemsets are guaranteed to be found.

# 7 Experimental Results

Our experiments used a 200MHz Sun Ultra-2 workstation with 384MB main memory. We used different synthetic databases that have been used as benchmark databases for many association rules algorithms [1, 2, 6, 15, 19, 20, 23, 26, 30]. We wrote our own dataset generator, using the procedure described in [2]. Our generator produces longer frequent itemsets for the same parameters (code is available by sending email to the author).

These datasets mimic the transactions in a retailing environment, where people tend to buy sets of items together, the so called potential maximal frequent set. The size of the maximal elements is clustered around a mean, with a few long itemsets. A transaction may contain one or more of such frequent sets. The transaction size is also clustered around a mean, but a few of them may contain many items.

Let $D$ denote the number of transactions, $T$ the average transaction size, $I$ the size of a maximal potentially frequent itemset, $L$ the number of maximal potentially frequent itemsets, and $N$ the number of items. The data is generated using the following procedure. We first generate $L$ maximal itemsets of average size $I$, by choosing from the $N$ items. We next generate $D$ transactions of average size $T$ by choosing from the $L$ maximal itemsets. We refer the reader to [4] for more detail on the database generation. In our experiments we set $N = 1000$ and $L = 2000$. Experiments are conducted on databases with different values of $D$, $T$, and $I$. The database parameters are shown in Table 1.

| Database | $T$ | $I$ | $D$ | Size |
|---|---|---|---|---|
| T5.I2.D100K | 5 | 2 | 100,000 | 2.8MB |
| T20.I6.D100K | 20 | 6 | 100,000 | 8.8MB |
| T10.I8.D400K | 10 | 8 | 400,000 | 19.2MB |
| T20.I4.D400K | 20 | 4 | 400,000 | 35.2MB |
| T20.I8.D400K | 20 | 8 | 400,000 | 35.2MB |
| T20.I12.D400K | 20 | 12 | 400,000 | 35.2MB |
| T30.I8.D400K | 30 | 8 | 400,000 | 51.2MB |
| T30.I16.D400K | 30 | 16 | 400,000 | 51.2MB |
| T40.I8.D400K | 40 | 8 | 400,000 | 67.2MB |
| T10.I4.D250K-5000K | 10 | 4 | 250K-5M | 12MB-240MB |

Table 1: Database Parameter Settings

Figure 17 shows the number of frequent itemsets of different sizes for the databases used in our experiments. The length of the longest frequent itemset and the total number of frequent itemsets for each database are shown in Table 2. For example, $T30.I16.D400K$ has a total of 13480771 frequent itemsets of various lengths. The longest frequent itemset is of size 22 at 0.25% support!

| Database | Longest Freq. Itemset | Number Freq. Itemsets |
|---|---|---|
| T20.I4.D400K | 10 | 12280 |
| T10.I8.D400K | 13 | 19345 |
| T20.I8.D400K | 15 | 121406 |
| T30.I8.D400K | 15 | 186989 |
| T40.I8.D400K | 15 | 310344 |
| T20.I12.D400K | 19 | 1239122 |
| T30.I16.D400K (0.5% minsup) | 22 | 13480771 |

Table 2: Maximum Size and Number of Frequent Sequences (0.25% Support)

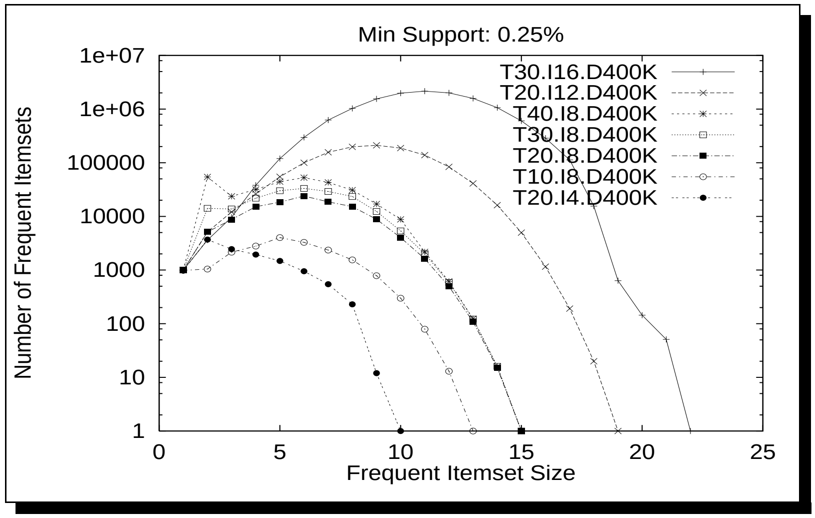


Figure 17: Number of Frequent Itemsets of Different Sizes

**Comparative Performance** In Figure 18 and Figure 19 we compare our new algorithms against *Apriori* and *Partition* (with 3 and 10 database partitions) for decreasing values of minimum support on the different databases. As the support decreases, the size and the number of frequent itemsets increases. *Apriori* thus has to make multiple passes over the database (22 passes for $T30.I16.D400K$), and performs poorly.

*Partition* performs worse than *Apriori* for high support, since the database is scanned only a few times at these points. The overheads associated with inverting the database on-the-fly dominate in *Partition*. However, as the support is lowered, *Partition* wins out over *Apriori*, since it only scans the database twice. These results are in agreement with previous experiments comparing these two algorithms [26]. One problem with *Partition* is that as the number of partitions increases, the number of locally frequent itemsets, which are not globally frequent, increases (this can be reduced somewhat by randomizing the partition selection). *Partition* can thus spend a lot of time in performing these redundant intersections. For example, compare the time for *Partition3* and *Partition10* on all the datasets. *Partition10* typically takes a factor of 1.5 to 2 times more time than *Partition3*. For $T30.I16$ (at 1% support) it takes 13 times more! Figure 20, which shows the number of tid-list intersections for different algorithms on different datasets, makes it clear that *Partition10* is performing four to five times more intersections than *Partition3*.

*AprClique* scans the database only once, and out-performs *Apriori* and *Partition* for higher support values on the $T10$ and $T20$ datasets. *AprClique* is very sensitive to the quality of maximal cliques (sub-lattices) that are generated. For small support, or with increasing transaction size $T$ for fixed $I$, the edge density of the $k$-association graph increases, consequently increasing the size of the maximal cliques. *AprClique* doesn't

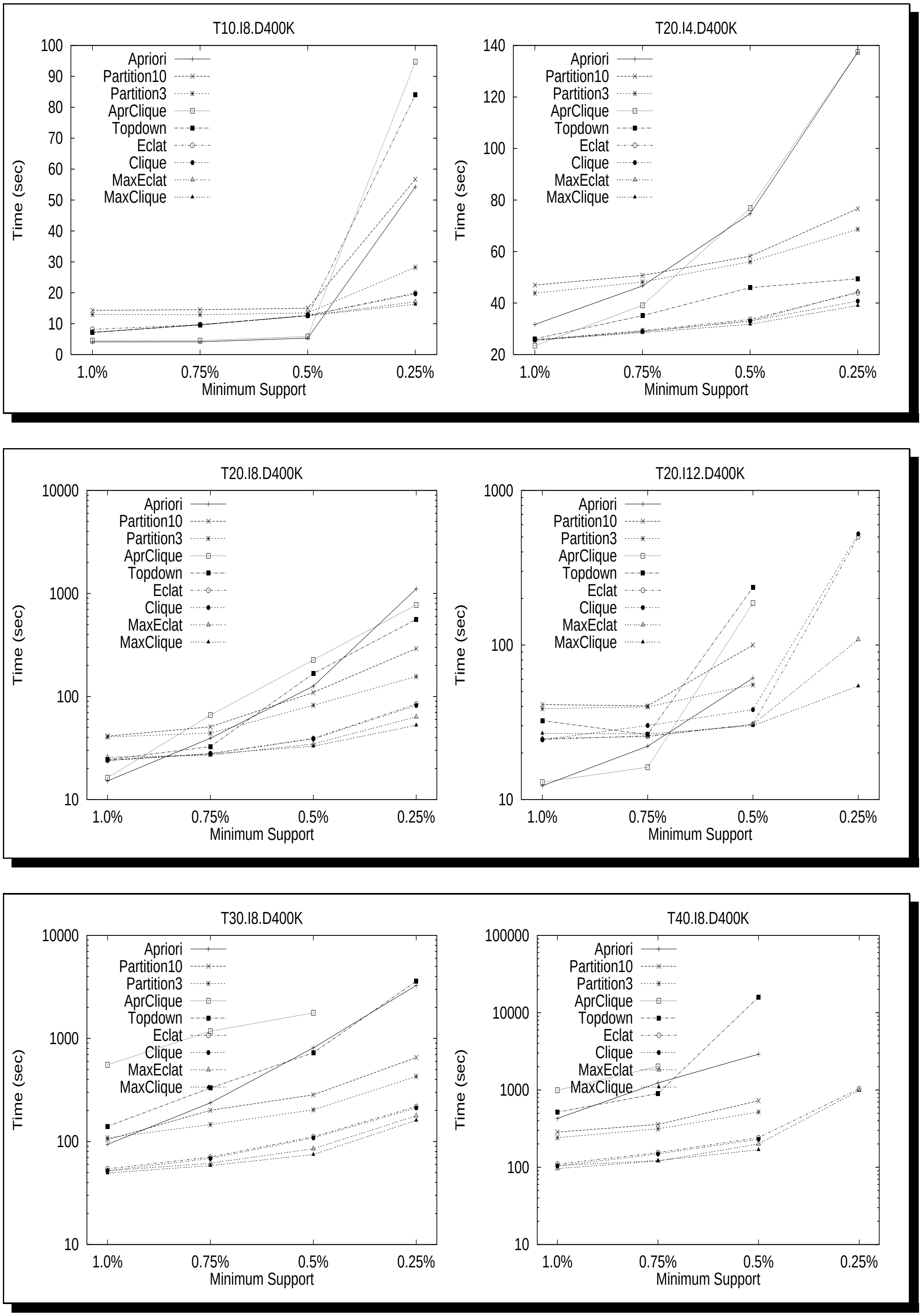


Figure 18: Total Execution Time

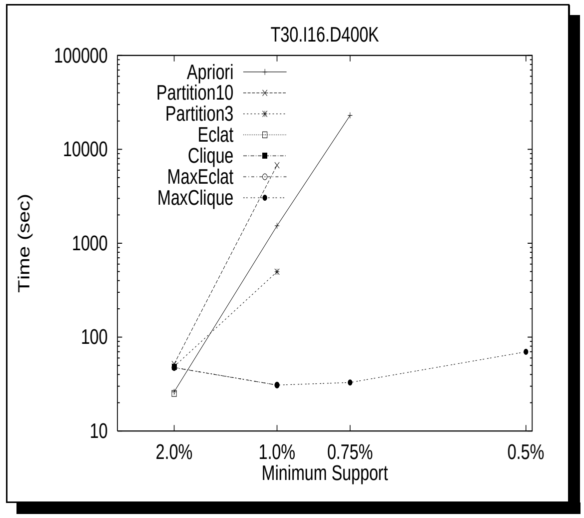


Figure 19: Total Execution Time

perform well under these conditions. *TopDown* usually performs better than *AprClique*, but shares the same characteristics as *AprClique*, i.e., it is better than both *Apriori* and *Partition* for higher support values, except for the $T30$ and $T40$ datasets. At lower support the maximum clique size, in the worst case, can become as large as the number of frequent items, forcing *TopDown* to perform too many k-way intersections to determine the minimal infrequent sets.

*Eclat* performs significantly better than all these algorithms in all cases. It usually out-performs *Apriori* by more than an order of magnitude, *Partition3* by a factor of two, and *Partition10* by a factor of four. *Eclat* makes only a few database scans, requires no hash trees, and uses only simple intersection operations to generate frequent itemsets. Further, *Eclat* is able to handle lower support values in dense datasets (e.g. $T20.I12$ and $T40.I8$), where both *Apriori* and *Partition* run out of virtual memory at 0.25% support.

We now look at the comparison between the remaining four methods, which are the main contributions of this work, i.e., between *Eclat*, *MaxEclat*, *Clique* and *MaxClique*. *Clique* uses the maximal-clique-based decomposition, which generates smaller classes with fewer number of candidates. However, it performs only slightly better than *Eclat*. *Clique* is usually 5-10% better than *Eclat*, since it cuts down on the number of tidlist intersections, as shown in Figure 20. *Clique* performs anywhere from 2% to 46% fewer intersections than *Eclat*. The difference between these methods is not substantial, since the savings in the number of intersections doesn't translate into a similar reduction in execution time.

The graphs for *MaxEclat* and *MaxClique* indicate that the reduction in search space by performing the hybrid search provides significant gains. Both the maximal clique-based strategies outperform their prefix-

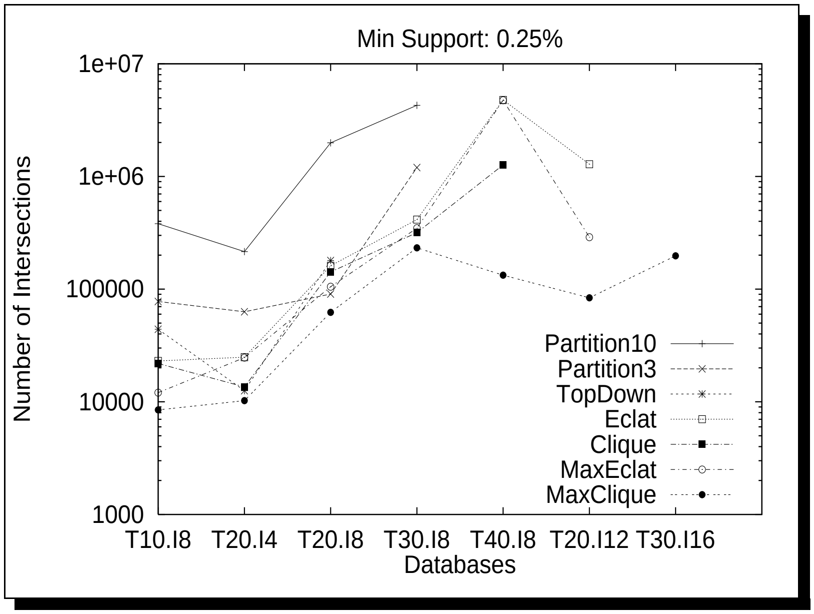


Figure 20: Number of Tid-list Intersections (0.25% Support)

based counterparts. *MaxClique* is always better than *MaxEclat* due to the smaller classes. The biggest difference between these methods is observed for $T20.I12$, where *MaxClique* is twice as fast as *MaxEclat*. An interesting result is that for $T40.I8$ we could not run the clique-based methods on 0.25% support, while the prefix-based methods, *Eclat* and *MaxEclat*, were able to handle this very low support value. The reason why clique-based approaches fail is that whenever the edge density of the association graph increases, the number and size of the cliques becomes large and there is a significant overlap among different cliques. In such cases the clique based schemes start to suffer.

The best scheme for all the databases we considered is *MaxClique* since it benefits from the smaller sub-lattices and the hybrid search scheme. Figure 20 gives the number of intersections performed by *MaxClique* compared against other methods. As one can see, *MaxClique* cuts down the candidate search space drastically, by anywhere from a factor of 3 (for $T20.I4$) to 35 (for $T40.I8$) over *Eclat*. It performs the fewest intersections of any method. In terms of raw performance *MaxClique* outperforms *Apriori* by a factor of 20-30, *Partition10* by a factor of 5, and *Eclat* by as much as a factor of 10 on $T20.I12$. Furthermore, it is the only method that was able to handle support values of 0.5% on $T30.I16$ (see Figure 19) where the longest frequent itemset was of size 22. All bottom-up search methods would have to enumerate at least $2^{22}$ subsets, while *MaxClique* only performed 197601 intersections, even though there were 13480771 total frequent itemsets (see Table 2). *MaxEclat* quickly identifies the 22 sized long itemset and also other long itemsets, and thus avoids enumerating all subsets. At 0.75% support *MaxClique* takes 69 seconds while *Apriori* takes 22963 seconds, a factor of 332, while *Partition10* ran out of virtual memory.

To summarize, there are several reasons why the last four algorithms outperform previous approaches:

1. They use only simple join operation on tid-lists. As the length of a frequent sequence increases, the size of its tid-list decreases, resulting in very fast joins.

2. No complicated hash-tree structure is used, and no overhead of generating and searching of customer subsequences is incurred. These structures typically have very poor locality [24]. On the other hand the new algorithms have excellent locality, since a join requires only a linear scan of two lists.

3. As the minimum support is lowered, more and larger frequent sequences are found. *Apriori* makes a complete dataset scan for each iteration. *Eclat* and the other three methods, on the other hand, restrict themselves to usually only few scan, cutting down the I/O costs.

4. The hybrid search approaches are successful by quickly identifying long itemsets early, and are able to avoid enumerating all subsets. For long itemsets of size 19 or 22, only the hybrid search methods are able to run, while other methods run out of virtual memory.

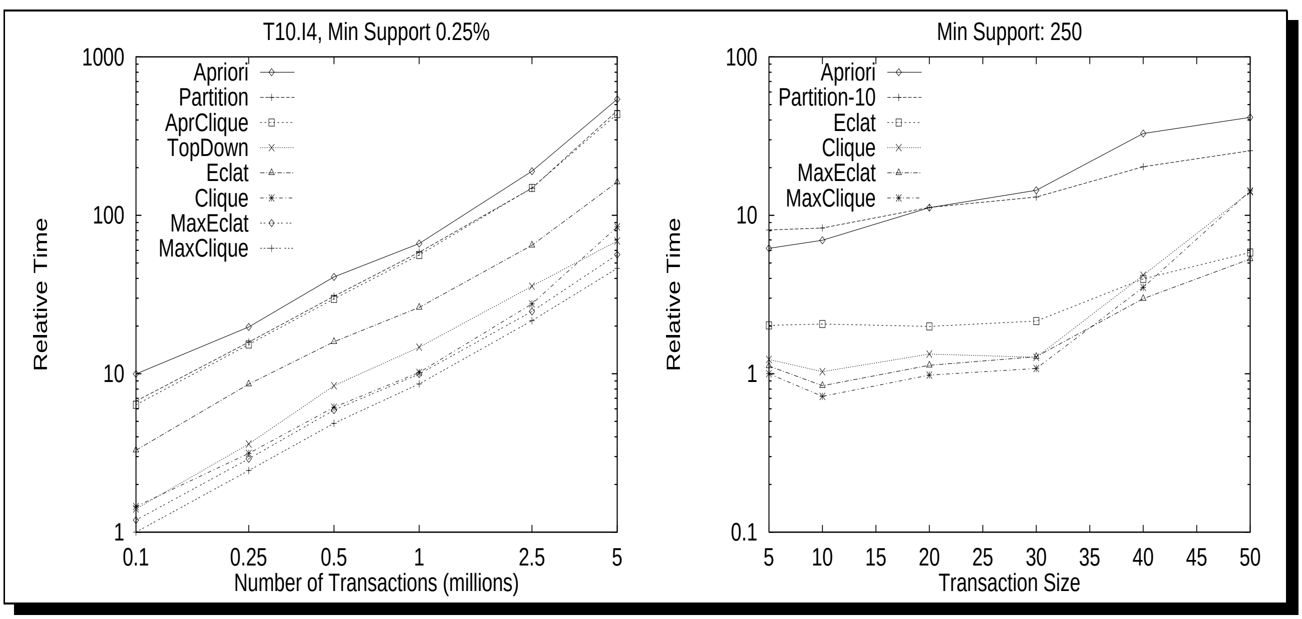


Figure 21: Scale-up Experiments: a) Number of Transactions, b) Transaction Size

**Scalability** The goal of the experiments below is to measure how the new algorithms perform as we increase the number of transactions and average transaction size.

Figure 21 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 the algorithms scale linearly, our new algorithms continue to out-perform *Apriori* and *Partition*.

Figure 21 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 $T = 50$. The goal of this setup is to measure the effect of increasing transaction size while keeping other parameters constant. We can see that 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 than the prefix-based algorithms.

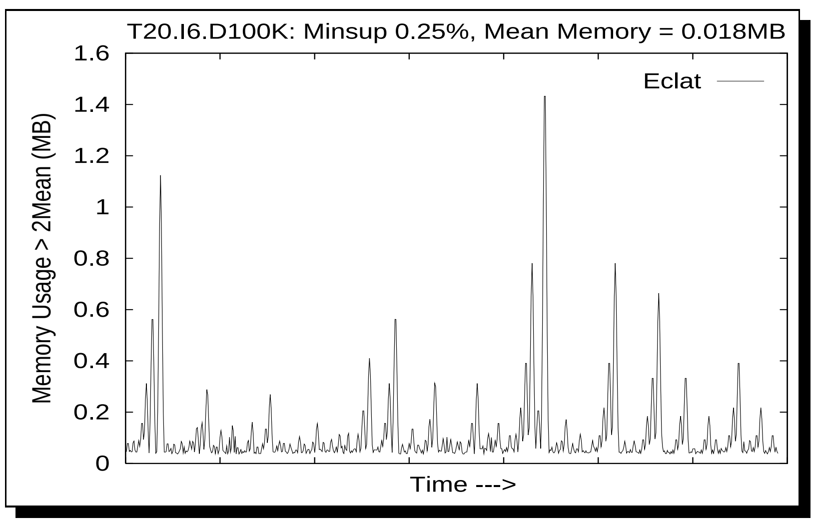


Figure 22: *Eclat* Memory Usage

**Memory Usage** Figure 22 shows the total main-memory used for the tid-lists in *Eclat* as the computation of frequent itemsets progresses on T20.I6.D100K. The mean memory usage is less than 0.018MB, 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) atom tid-lists within each sub-lattice. This figure confirms that the sub-lattices produced by $\theta_1$ and $\phi_1$ are small enough, so that all intermediate tid-lists for a class can be kept in main-memory.

# 8 Conclusions

In this paper we presented new algorithms for efficient enumeration of frequent itemsets. We presented a lattice-theoretic approach to partition the frequent itemset search space into small, independent sub-spaces using either prefix-based or maximal-clique-based methods. Each sub-problem can be solved in main-memory using bottom-up, top-down, or a hybrid search procedure, and the entire process usually takes only a few database scans.

Experimental evaluation showed that the maximal-clique-based decomposition is more precise and leads to smaller classes. When this is combined with the hybrid search, we obtain the best algorithm *MaxClique*, which outperforms current approaches by more than an order of magnitude. We further showed that the new algorithms scale linearly in the number of transactions.

# References


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

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

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

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th VLDB Conference*, September 1994.

[5] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD Conf. Management of Data*, June 1998.

[6] 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.

[7] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.

[8] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[9] D. Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information Processing Letters*, 51:207–211, 1994.

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

[11] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *16th ACM Symp. Principles of Database Systems*, May 1997.

[12] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all the most specific sentences by randomized algorithms. In *Intl. Conf. on Database Theory*, January 1997.

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

[14] 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*, August 1995.

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

[16] T. Kashiwabara, S. Masuda, K. Nakajima, and T. Fujisawa. Generation of maximum independent sets of a bipartite graph and maximum cliques of a circular-arc graph. *J. of Algorithms*, 13:161–174, 1992.

[17] S. O. Kuznetsov. Interpretation on graphs and complexity characteristics of a search for specific patterns. *Nauchn. Tekh. Inf., Ser. 2 (Automatic Document. Math. Linguist.)*, 23(1):23–37, 1989.

[18] L. E. LaForge. Some zarankiewicz numbers. Technical Report SOCS-94.Z, McGill Univ., July 1994.

[19] D-I. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *6th Intl. Conf. Extending Database Technology*, March 1998.

[20] J-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *14th Intl. Conf. on Data Engineering*, February 1998.

[21] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[22] G. D. Mulligan and D. G. Corneil. Corrections to bierstone's algorithm for generating cliques. *J. of the ACM*, 19(2):244–247, 1972.

[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] S. Parthasarathy, M. J. Zaki, and W. Li. Memory placement techniques for parallel association mining. In *4th Intl. Conf. Knowledge Discovery and Data Mining*, August 1998.

[25] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternatives and implications. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

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

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

[28] S.-J. Yen and A. L. P. Chen. An efficient approach to discovering knowledge from large databases. In *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.

[29] 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*, April 1997.

[30] 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*, August 1997.

[31] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343-373, December 1997.

# A Computational Complexity of Mining Frequent Itemsets

In addition to the practical utility of frequent itemsets in real-life data mining problems, they are mathematically beautiful entities. The discussion below highlights the graph-theoretic basis of itemset discovery, and offers some insight into the computational complexity of mining frequent itemsets and related problems.

**Definition 11** *A* **bipartite graph** $G = (U, V, E)$ *has two distinct vertex sets* $U$ *and* $V$*, and an edge set* $E = \{(u, v) \mid u \in U \text{and } v \in V\}$*. A complete bipartite subgraph* $I \times T$ *is called a* **bipartite clique***, and is denoted as* $K_{i,t}$*, where* $|I| = i$*,* $|T| = t$ *and* $I \subseteq U$*,* $T \subseteq V$*.*

The input database for association mining is essentially a very large bipartite graph, with $U$ as the set of items, $V$ as the set of tids, and each (item, tid) pair as an edge. The problem of enumerating all (maximal) frequent itemsets corresponds to the task of enumerating all (maximal) constrained bipartite cliques, $K_{i,t}$, where $t \geq min\_sup$. Due to the one-to-one correspondence between bipartite graphs and binary matrices, one can also view it as the problem of enumerating all (maximal) unit submatrices in a binary matrix, satisfying the support constrains. For connections between mining minimal infrequent itemsets and hypergraph transversals see [11].

Figure 23a shows the database as a bipartite graph, and the maximal bipartite clique $K_{4,3} = ACTW \times 135$ (maximal frequent itemset $ACTW$).

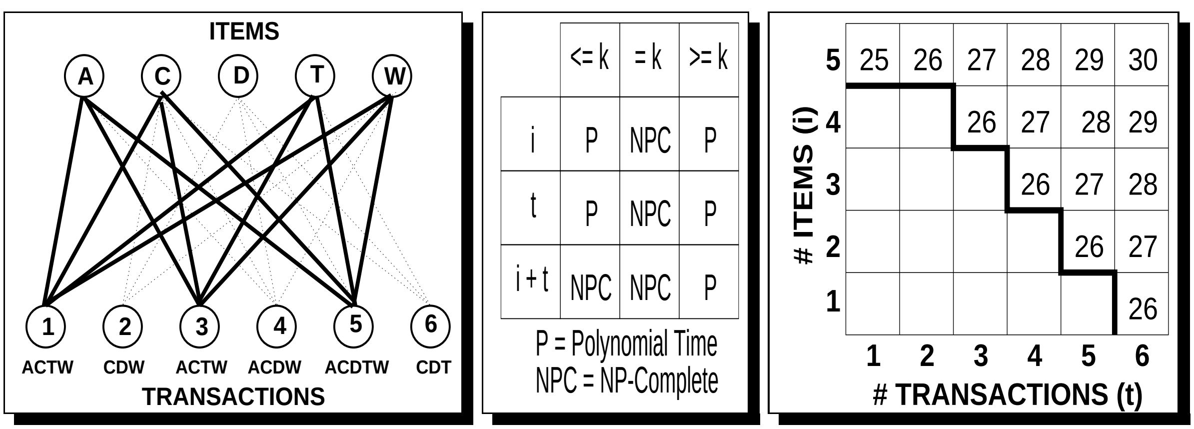


Figure 23: a) Maximal Constrained Bipartite Clique, b) Mining Complexity, c) Zarankiewicz Numbers

**Theorem 1** *Whether a bipartite graph* $G = (U, V, E)$*, with* $|U| = |V| = n$ *contains a balanced bipartite clique* $K_{k,k}$*, is NP-Complete.*

PROOF: If we have a possible solution, then in time $O(k^2)$ we can determine if it is indeed a bipartite clique. The problem is thus in NP. Let's consider the question, "Does the (regular) graph $G' = (V', E')$ contain a clique with $k$ vertices?" which is known to be NP-Complete [10]. We reduce the clique problem in regular graphs to the balanced bipartite clique problem as follows: Let $U = V = V'$, and let $E = \{(u, v) \mid u = v \text{ or } (u, v) \in E'\}$. Then $G'$ has a clique of size $k$ iff $G$ has a balanced bipartite clique of size $k$. This reduction takes $O(E' + V')$ time. This proves the result. ■

**Corollary 1** *Whether a bipartite graph $G = (U, V, E)$, contains a bipartite clique $K_{i,t}$ is NP-Complete. Thus, whether there exists a frequent itemset of a certain size is NP-Complete.*

Figure 23b shows the complexity of decision problems for maximal bipartite cliques (itemsets) with restrictions on the size of $|I| = i$ (items) and $|T| = t$ (support). For example, the problem whether there exists a maximal bipartite clique such that $i + t \geq k$ (with constant $k$) is in P, the class of problems that can be solved in polynomial time. On the other had, the problem whether there exists a maximal bipartite clique such that $i + t = k$ is NP-Complete [17], the class of "hard" problems for which no polynomial time algorithm is known to exist. The last row of the table may seem contradictory. While there is unlikely to exist a polynomial time algorithm for finding a clique with $i + t \leq k$, the largest cliques with $i + t \geq k$ can be found by reducing it to the maximum matching problem [16], which has $O((U + V)^{2.5})$ complexity.

While the class NP asks whether a desired solution exists, the class #P asks how many solutions exist. In the cases known so far, the counting problems that correspond to NP-Complete problems are #P-Complete. The following theorem says that counting the number of maximal cliques in a bipartite graph is extremely hard.

**Theorem 2 ([17])** *Determining the number of maximal bipartite cliques in a bipartite graph is #P-Complete.*

The complexity results shown above are quite pessimistic, and apply to general bipartite graphs. We should therefore focus on special cases where we can find polynomial time solutions. Fortunately for association mining, in practice the bipartite graph (database) is very sparse, and we can in fact obtain linear complexity in the graph size.

The *arboricity* $r(G)$ of a graph is the minimum number of forests into which the edges of G can be partitioned, and is given as $r(G) = \max_{H \subset G} \{e(H)/(n(H) - 1)\}$, where $n(H)$ is the number vertices and $e(H)$ the number of edges of the subgraph $H$. A bound on the arboricity is equivalent to a notion of hereditary sparsity. For a bipartite graph $r(G) = i \cdot t/(i + t - 1)$, where $K_{i,t}$ is a maximum bipartite clique. Furthermore, if we assume $i \ll t$ (as is generally the case in practice, since we want large support), then

$r(G) \approx i$, i.e., the arboricity is given by the maximum sized frequent itemset. The following theorem says that the complexity of finding all maximal bipartite cliques is linear in number of items and transactions:

**Theorem 3 ([9])** *For sparse graphs, of bounded arboricity $i$, all maximal bipartite cliques can be enumerated in time $O(i^3 \cdot 2^{2i} \cdot (U + V))$.*

To explain the intuition behind this theorem we need the following definition:

**Definition 12** *An* **orientation** *of an undirected graph $G$, is obtained by assigning a direction (i.e., "orienting") to each edge. A* **d-bounded** *orientation of $G$ is an orientation in which each edge has out-degree at most $d$.*

It can be shown that if a graph has arboricity $i$, then it has a $i$-bounded orientation, and a $2i$-bounded acyclic orientation [9]. In other words each vertex in the oriented graph can have at most out-degree $2i$. In terms of our database representation, it means that each transaction can have at most $2i$ items, and each item can be part of at most $2i$ transactions. Now for any bipartite clique $I \times T$, either $I$ or $T$ is a subset of the out-neighbors of some vertex. Thus to generate all possible cliques, we can generate all subsets of each vertex's out-neighbors. This costs $O(2^{2i})$ time, and we do this for all vertices, i.e., $|U| + |V|$, giving us $O(2^{2i} \cdot (|U| + |V|))$ time. The remaining $i^3$ factor is the additional overhead of eliminating duplicates, verifying if indeed the enumerated subsets form a clique, and other algorithmic costs. In effect the above theorem says that finding frequent itemsets in databases with bounded transaction size takes time linear in the number of transactions and the number of items.

While it is easy to see, even without the above theorem, that if the largest transaction size is bounded of length $i$, then one can enumerate all itemsets and check if they are frequent in time $O(2^i \cdot V)$, where $V$ is the number of transactions. That is, association mining is linear in the number of transactions (with bounded length) in the worst case. What the above theorem claims is that, at least in theory, the association mining algorithms should also scale linearly in the number of items or attributes, a very important feature if practicable!

It should be noted that even though the complexity of the enumeration for sparse graphs is linear in the number of items and transactions, the bound is not practical for large databases due to the large constant overhead, which is exponential in $i$, which can easily be around 10 to 20 or more in practice.

## A.1 Finding Maximum Frequent Itemsets

**Theorem 4 ([16])** *All maximum independent sets can be listed in* $O((U+V)^{2.5}+\gamma)$ *time, where* $\gamma$ *is the output size.*

The above theorem states that all the maximum (largest) bipartite cliques (independent sets in complementary graph) of a bipartite graph can be found in time polynomial in input, and linear in the output size, i.e., we can find all the largest frequent itemsets in output polynomial time. This result relies on the complexity of maximum bipartite matchings, which is $O((|U|+|V|)^{2.5})$. However, due to the greater than quadratic complexity, it remains to be seen if this algorithm is practical for large databases with millions of transactions.

Finally we would like to point out the connection of frequent itemsets to extremal graph theory. Let $H$ be a fixed graph. The classical problem from which extremal graph theory has originated is to determine the maximum number of edges in a graph on $n$ vertices which does not contain a copy of $H$. This maximum value is called the *Turán number* of H. Specifically, the bipartite case corresponds to the *Zarankiewicz problem*. The *Zarankiewicz number*, $Z$, can tell us the maximum size of a frequent itemset guaranteed to exist given the number of edges present in the database.

**Definition 13** *The* **Zarankiewicz number** $Z(m,n;i,t)$ *is the least number of edges in a bipartite graph* $G=(U,V,E)$, *with* $|U|=m$ *and* $|V|=n$, *such that* $G$ *must contain a* $K_{i,t}$.

**Theorem 5 ([18])** *Let* $L(m,n;i,t)=mn-Z(m,n;i,t)$, $h=m-i$, *and* $k=n-t$. *If* $t>h(\lfloor k/i \rfloor+1)$, *then* $L(m,n;i,t)=h(\lfloor k/i \rfloor+1)+k$. *The dual also holds if we replace* $i$ *with* $t$ *and* $h$ *with* $k$.

This theorem gives exact value for the Zarankiewicz number, but it cannot directly be used to estimate the size since it fills the upper triangle of the matrix indexed by $i$ and $t$, the size of the bipartite clique $K_{i,t}=I\times T$, as shown in Figure 23c. For example, to guarantee the existence of a $K_{5,6}$ we need 30 edges. In association mining, we are usually interested in small $i$ (10 to 20), and $t \geq min\_sup$, where $min\_sup$ is a small fraction of the number of transactions. This point thus lies in the lower half of the matrix, which the above theorem cannot fill. However, since the input database, $G$, is very sparse its complement, $\overline{G}$, must be dense. Furthermore, using the above theorem we can find the size of the guaranteed maximum clique in $\overline{G}$, given as $K_{m-i,n-t}$, which lies in the upper half. This also tells us that $K_{m-i,n-t}$ is an independent set in $G$, and consequently, no frequent itemset can be a subset of $I\times T=K_{m-i,n-t}$.

## Biography

**Mohammed J. Zaki** is currently an assistant professor of Computer Science at Rensselaer Polytechnic Institute (RPI). He received his M.S. and Ph.D. degrees in computer science, both from the University of Rochester in May 1995 and July 1998, respectively.

His primary research interest is in the design of large-scale high-performance data mining systems, including the development of efficient, scalable, and parallel algorithms for various data mining techniques.

He co-chaired the Workshop on Large-Scale Parallel KDD Systems (with KDD'99), and the 3rd Workshop on High Performance Data Mining (with IPPS'00). He has published over 35 papers on data mining and parallel computing. He is a member of ACM and IEEE.