Chapter 1

Hierarchical Parallel Algorithms for Association Mining

MOHAMMED J. ZAKI

Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180, USA

Email: *zaki@cs.rpi.edu*

1.1 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.

Since the discovery of association rules is a very computational and I/O intensive task, it is crucial to leverage the combined computational power of multiple processors for fast response and scalability. In this paper we present new parallel 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 few database scans (with some pre-processed information), minimizing the I/O costs.

We present four new algorithms combining the features listed above, depending on the database format, the decomposition technique, and the search procedure used. These include *Par-Eclat*, *Par-MaxEclat*, *Par-Clique*, and *Par-MaxClique*. The parallel work is distributed among the processors in such a way that each processor can compute the frequent itemsets independently, using simple intersection operations. These techniques eliminate the need for synchronization after the initial set-up phase, and enable us to scan the database only two times, drastically cutting down the I/O overhead. Our tid-list based approach is also insensitive to data-skew. Furthermore, the use of simple intersection operations makes the new algorithms an attractive option for direct implementation in database systems, using SQL.

Our experimental testbed is a 32-processor DEC Alpha SMP cluster (8 hosts, 4 processors/host) interconnected by the Memory Channel (Gillett 1996) network. The new parallel algorithms are also novel in that they are hierarchical in nature, i.e., they assume a distributed-memory model across the 8 cluster hosts, but assume a shared-memory model for the 4 processors on each host. 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 1.2 we describe the association discovery problem. We look at related work in Section 1.3. In section 1.4 we develop our lattice-based approach for problem decomposition and pattern search. Section 1.5 describes the sequential algorithm, and Section 1.6 presents our new parallel algorithms. An experimental study is presented in Section 1.7, and we conclude in Section 1.8.

1.2 Problem Statement

The association mining task, first introduced in (Agrawal, Imielinski, & Swami 1993), 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 $\mathcal{F}_k$.

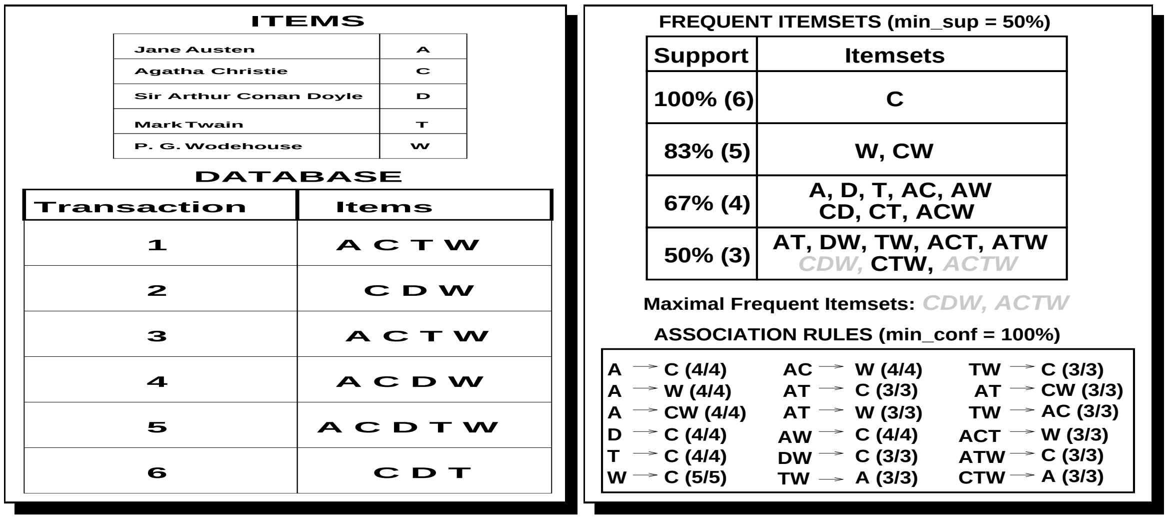
An *association rule* is an expression $A \Rightarrow B$, where A and B are itemsets. The support of the rule is given as $\sigma(A \cup B)$, and the *confidence* as $\sigma(A \cup B)/\sigma(A)$ (i.e., the conditional probability that a transaction contains B, given that it contains A). A rule is *strong* 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 strong. This task can be broken into two steps (Agrawal *et al.* 1996):

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 strong 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 1.1. 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 1.1 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 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 strong rules, we need the support of all frequent itemsets. This can be easily accomplished once the



ITEMS

Jane Austen	A
Agatha Christie	C
Sir Arthur Conan Doyle	D
Mark Twain	T
P. G. Wodehouse	W

DATABASE

Transaction	Items
1	A C T W
2	C D W
3	A C T W
4	A C D W
5	A C D T W
6	C D T

FREQUENT ITEMSETS (min_sup = 50%)

Support	Itemsets
100% (6)	C
83% (5)	W, CW
67% (4)	A, D, T, AC, AW CD, CT, ACW
50% (3)	AT, DW, TW, ACT, ATW CDW, CTW, ACTW

Maximal Frequent Itemsets: CDW, ACTW

ASSOCIATION RULES (min_conf = 100%)

A $\rightarrow$ C (4/4)	AC $\rightarrow$ W (4/4)	TW $\rightarrow$ C (3/3)
A $\rightarrow$ W (4/4)	AT $\rightarrow$ C (3/3)	AT $\rightarrow$ CW (3/3)
A $\rightarrow$ CW (4/4)	AT $\rightarrow$ W (3/3)	TW $\rightarrow$ AC (3/3)
D $\rightarrow$ C (4/4)	AW $\rightarrow$ C (4/4)	ACT $\rightarrow$ W (3/3)
T $\rightarrow$ C (4/4)	DW $\rightarrow$ C (3/3)	ATW $\rightarrow$ C (3/3)
W $\rightarrow$ C (5/5)	TW $\rightarrow$ A (3/3)	CTW $\rightarrow$ A (3/3)

Figure 1.1: a) Bookstore Database, b) Frequent Itemsets and Strong Rules

maximal elements have been identified, by making an additional database pass, and gathering the support of all uncounted subsets.

1.2.1 Computational Complexity

The search space for enumeration of all frequent itemsets is 2^m, which is exponential in m, the number of items. One can prove that the problem of finding a frequent set of a certain size is NP-Complete, by reducing it to the balanced bipartite clique problem, which is known to be NP-Complete (Zaki & Ogihara 1998). However, if we assume that there is a bound on the transaction length, we can prove that frequent itemset enumeration is essentially linear in the database size (Zaki & Ogihara 1998).

Once the frequent itemsets are known, they can be used to obtain rules that describe the relationship between different itemsets. We generate and test the confidence of all rules of the form $X \backslash Y \Rightarrow Y$, where $Y \subset X$, and X is frequent. For example, the itemset CDW generates the following rules $\{CD \Rightarrow W, CW \Rightarrow D, DW \Rightarrow C, C \Rightarrow DW, D \Rightarrow CW, W \Rightarrow CD\}$. For an itemset of size k there are $2^k - 2$ potentially strong rules that can be generated. This follows from the fact that we must consider each subset of the itemset as an antecedent, except for the empty and the full itemset. The complexity of the rule generation step is thus $O(r \cdot 2^l)$, where r is the number of frequent itemsets, and l is the longest frequent itemset.

1.3 Related Work

1.3.1 Sequential Algorithms

Several algorithms for mining associations have been proposed in the literature (Agrawal, Imielinski, & Swami 1993; Agrawal *et al.* 1996; Brin *et al.* 1997; Houtsma & Swami 1995; Lin & Kedem 1998; Lin & Dunham 1998; Mueller 1995; Park, Chen, & Yu 1995a; Savasere, Omiecinski, & Navathe 1995; Toivonen 1996). The *Apriori* algorithm (Agrawal *et al.* 1996) 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. The DHP algorithm (Park, Chen, & Yu 1995a) 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 (Savasere, Omiecinski, & Navathe 1995) 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, and in the second pass it counts their global support. The DLG (Yen & Chen 1996) 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 (Brin *et al.* 1997) dynamically counts candidates of varying length as the database scan progresses, and thus is able to reduce the number of scans. 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 (Zaki *et al.* 1997a), and (Toivonen 1996) presents an exact algorithm that finds all rules using sampling. The AS-CPA algorithm and its sampling versions (Lin & Dunham 1998) build on top of *Partition* and produce a much smaller set of potentially frequent candidates. It requires at most two database scans. Also, sampling may be used to eliminate the second pass altogether. Approaches using only general-purpose DBMS systems and relational algebra operations have also been studied (Holsheimer *et al.* 1995; Houtsma & Swami 1995). We proposed new algorithms (Zaki *et al.* 1997b; 1997c) that were shown to outperform previous approaches.

All the above algorithms generate all possible frequent itemsets. Methods for finding the maximal elements include *All-MFS* (Gunopulos, Mannila, & Saluja 1997), which is a randomized algorithm to discover maximal frequent itemsets. The *Pincer-Search* algorithm (Lin & Kedem 1998) 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* (Bayardo 1998) is another algorithm for finding the maximal elements. It uses efficient pruning techniques to quickly narrow the search space.

1.3.2 Parallel Algorithms

Distributed-Memory Machines Three different parallelizations of *Apriori* on IBM-SP2, a distributed memory machine, were presented in (Agrawal & Shafer 1996). The *Count Distribution* algorithm is a straight-forward parallelization of *Apriori*. Each processor generates the partial support of all candidate itemsets from its local database partition. At the end of each iteration the global supports are generated by exchanging the partial supports among all the processors. The *Data Distribution* algorithm partitions the candidates into disjoint sets, which are assigned to different processors. However to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from huge communication overhead. The *Candidate Distribution* algorithm also partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. The local database portion is still scanned in every iteration. *Count Distribution* was shown to have superior performance among these three algorithms (Agrawal & Shafer 1996). Other parallel algorithms improving upon these ideas in terms of communication efficiency, or aggregate memory utilization have also been proposed (Cheung *et al.* 1996b; 1996a; Han, Karypis, & Kumar 1997). The PDM algorithm (Park, Chen, & Yu 1995b) presents a parallelization of the DHP algorithm (Park, Chen, & Yu 1995a). The hash based parallel algorithms NPA, SPA, HPA, and HPA-ELD, proposed in (Shintani & Kitsuregawa 1996) are similar to those in (Agrawal & Shafer 1996). Essentially NPA corresponds to Count Distribution, SPA to Data Distribution, and HPA to Candidate Distribution. The HPA-ELD algorithm is the best among NPA, SPA, and HPA, since it eliminates the effect of data skew, and reduces communication by replicating candidates with high support on all processors. We also presented a new parallel algorithm *Eclat* (Zaki, Parthasarathy, & Li 1997; Zaki *et al.* 1997d) on a DEC Alpha Cluster. *Eclat* uses the equivalence class decomposition scheme along with a bottom-up lattice traversal. It was shown to outperform *Count Distribution* by more than an order of magnitude.

Shared-Memory Machines In recent work we presented the CCPD parallel algorithm for shared-memory machines (Zaki *et al.* 1996). It is similar in spirit to *Count Distribution*. The candidate itemsets are generated in parallel and are stored in a hash tree which is shared among all the processors. Each processor then scans its logical partition of the database and atomically updates the counts of candidates in the shared hash tree. CCPD uses additional optimization such as candidate balancing, hash-tree balancing and short-circuited subset counting to speed up performance (Zaki *et al.* 1996). APM (Cheung, Hu, & Xia 1998) is an asynchronous parallel algorithm for shared-memory machines based on the DIC algorithm (Brin *et al.* 1997).

1.3.3 The *Apriori* and *Count Distribution* Algorithms

We now describe the *Apriori* algorithm (Agrawal *et al.* 1996) in more detail, since it forms the basis of almost all parallel algorithms proposed to-date. For comparison with the methods we will describe in this paper, we also look at the Count Distribution algorithm (Agrawal & Shafer 1996) which is one of the current best parallel methods.

Apriori Algorithm *Apriori* (Agrawal *et al.* 1996) 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 1.2. We refer the reader to (Agrawal *et al.* 1996) 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 example, 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 for fast support counting.

```
$\mathcal{F}_1 = \{$frequent 1-itemsets $\}$;
for ($k = 2; \mathcal{F}_{k-1} \neq \emptyset; k++$)
    $C_k$ = Set of New Candidates;
    for all transactions $t \in \mathcal{D}$
        for all $k$-subsets $s$ of $t$
            if ($s \in C_k$) $s.count++$;
    $\mathcal{F}_k = \{c \in C_k | c.count \geq min\_sup\}$;
Set of all frequent itemsets = $\bigcup_k \mathcal{F}_k$;
```

Figure 1.2: The *Apriori* Algorithm

The *Count Distribution* Algorithm The *Count Distribution* algorithm (Agrawal & Shafer 1996) is a simple but effective parallelization of *Apriori*. All processors generate the entire candidate hash tree from $\mathcal{F}_{k-1}$. Each processor can thus independently get partial supports of the candidates from its local database partition. This is followed by a sum-reduction to obtain the global counts. Note that only the partial counts need to be communicated, rather than merging different hash trees, since each processor has a copy of the entire tree. Once the global $\mathcal{F}_k$ has been determined each processor builds C_{k+1} in parallel, and repeats the process until all frequent itemsets are found. This simple algorithm minimizes communication since only the counts are exchanged among the processors. However, since the entire hash tree is replicated on each processor, it doesn't utilize the aggregate memory efficiently.

1.4 Itemset Enumeration: Lattice-Based Approach

Before embarking on the algorithm description, we will briefly review some terminology from lattice theory (see (Davey & Priestley 1990) 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 $X, Z, Y \in P$. 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 a join and meet semilattice, i.e., if $X \vee Y$ and $X \wedge Y$ exist of all $X, Y \in L$. L is a* **complete lattice** *if $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq L$. A ordered set $M \subset L$ is a* **sublattice** *of L if $X, Y \in M$ implies $X \vee Y \in M$ and $X \wedge Y \in M$.*

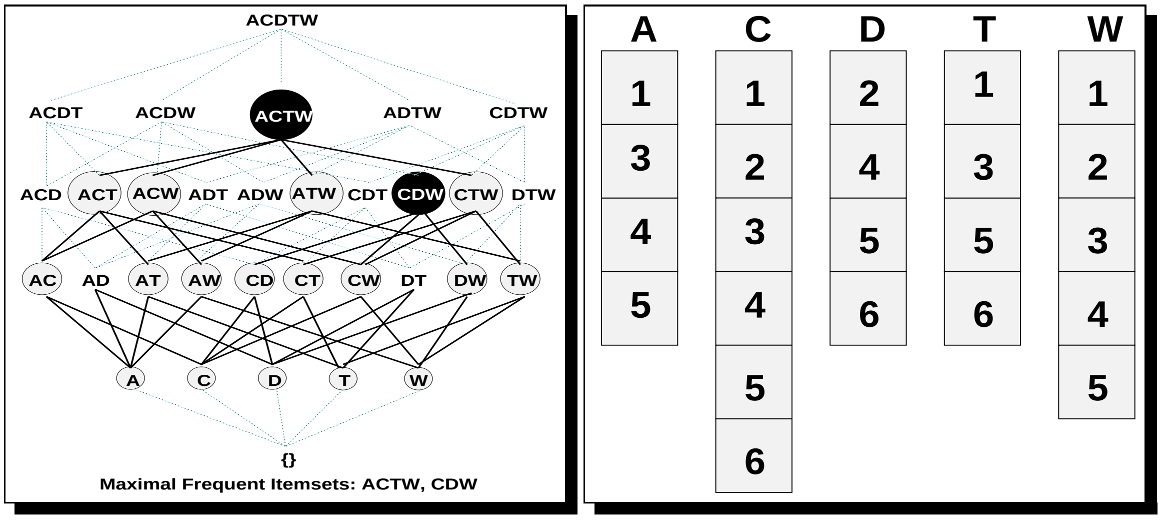


Figure 1.3: a) The Complete Powerset Lattice $\mathcal{P}(\mathcal{I})$, b) Tid-List for the Atoms

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 1.3 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 (Agrawal *et al.* 1996; Park, Chen, & Yu 1995a; Savasere, Omiecinski, & Navathe 1995). 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, i.e., if we view the lattice as a collection of (multiple) paths leading to the maximal frequent itemsets, then one can formulate alternate ways of reaching the maximal elements. We need not look at all intermediate levels as in a bottom-up approach. For example, in a depth-first search, once can reach $ACTW$ in just four steps using the path A, AC, ACT, and $ACTW$. Later, we will describe practical implementations of such alternate enumeration schemes.

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.

1.4.1 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 $\perp$. Then $X \in L$ is called an* **atom** *if $\perp \sqsubset X$, i.e., X covers $\perp$. 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 $\perp$ 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 1.3 shows the tid-lists for the atoms in our example database. For example consider atom A. Looking at the database in Figure 1.1, we see that A occurs in transactions 1, 3, 4, and 5. This forms the tid-list for atom A.

Lemma 3 ((Davey & Priestley 1990)) *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 as 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 $X = \bigcup_{Y \in J} J$. 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 1.4 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.

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.

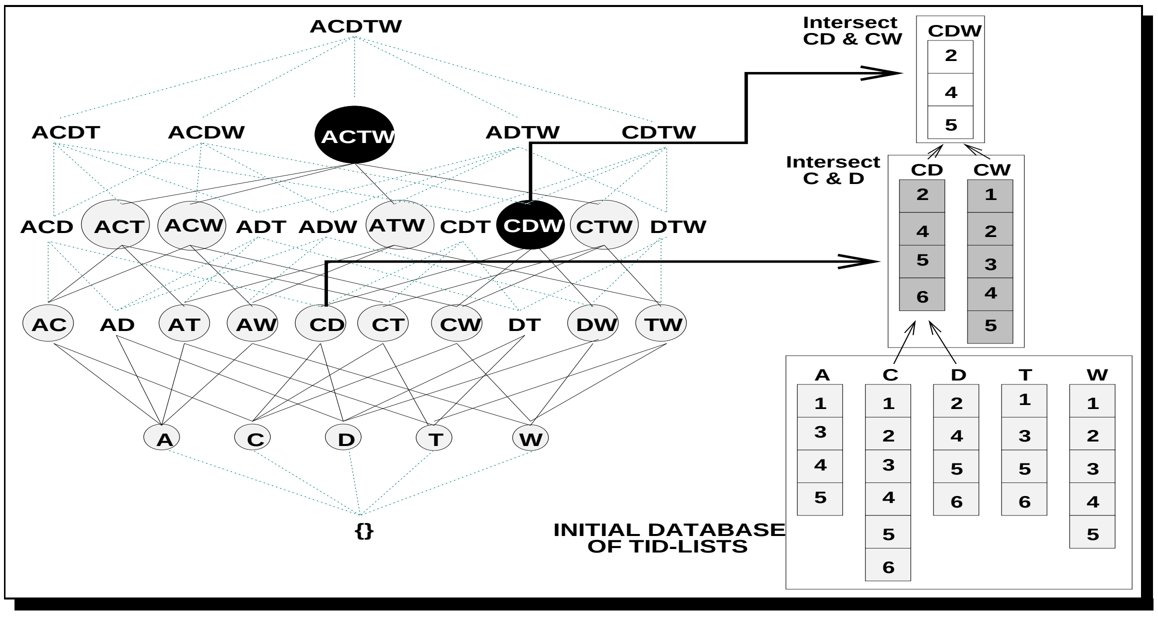


Figure 1.4: Computing Support of Itemsets via Tid-List Intersections

1.4.2 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 $p : \mathcal{P}(\mathcal{I}) \mapsto \mathcal{P}(\mathcal{I})$ where $p(X, k) = X[1 : k]$, the k length prefix of X. Define an equivalence relation θ_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 θ_k a *prefix-based* equivalence relation.

Figure 1.5a shows the lattice induced by the equivalence relation θ_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 relation θ_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 $\bot = A$. 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

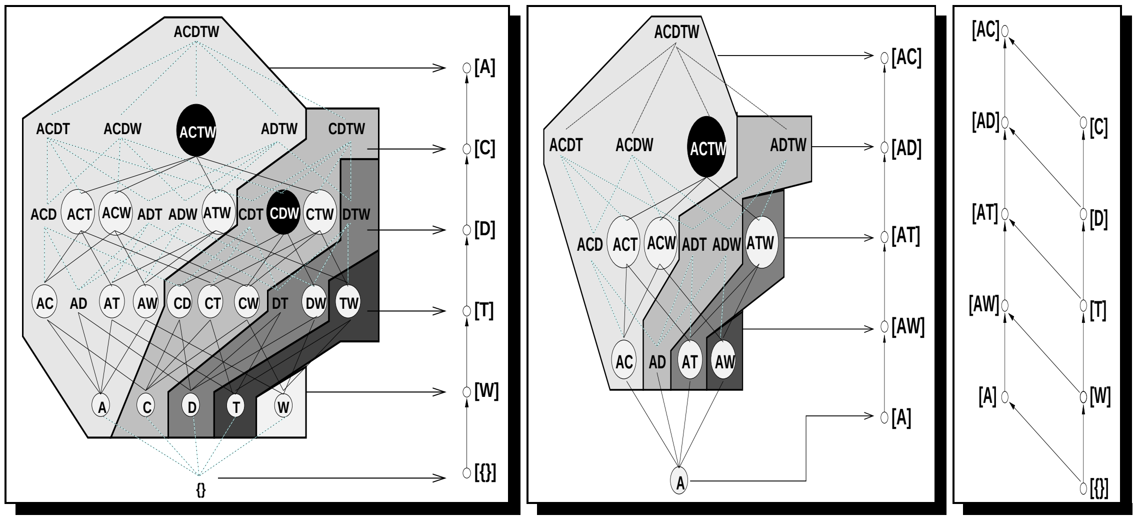


Figure 1.5: Equivalence Classes of a) $\mathcal{P}(\mathcal{I})$ Induced by θ_1, and b) $[A]_{\theta_1}$ Induced by θ_2; c) Final Lattice of Independent Classes

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. For example, assume that we have already enumerated all frequent itemsets of $[W]$, $[T]$, $[D]$, and $[C]$. When we are processing $[A]$, we have full pruning information for itemsets that do not belong to $[A]$. For example we know that CTW is frequent, thus $ACTW$ is a possible candidate. If we further label classes in decreasing order of their size (number of atoms), then we solve small classes first, and the large ones later, allowing us to fully use all pruning information.

In practice we have found that the one level decomposition induced by θ_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 θ_2. Figure 1.5b shows the equivalence class lattice induced by applying θ_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 θ_1 on $\mathcal{P}(\mathcal{I})$ and θ_2 on $[A]$ is shown in Figure 1.5c. 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.

1.4.3 Search for Frequent Itemsets

In this section we discuss efficient search strategies for enumerating the frequent itemsets within each class.

Bottom-Up Search

The bottom-up search is based on a recursive decomposition of each class into smaller classes induced by the equivalence relation θ_k. Figure 1.6 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.

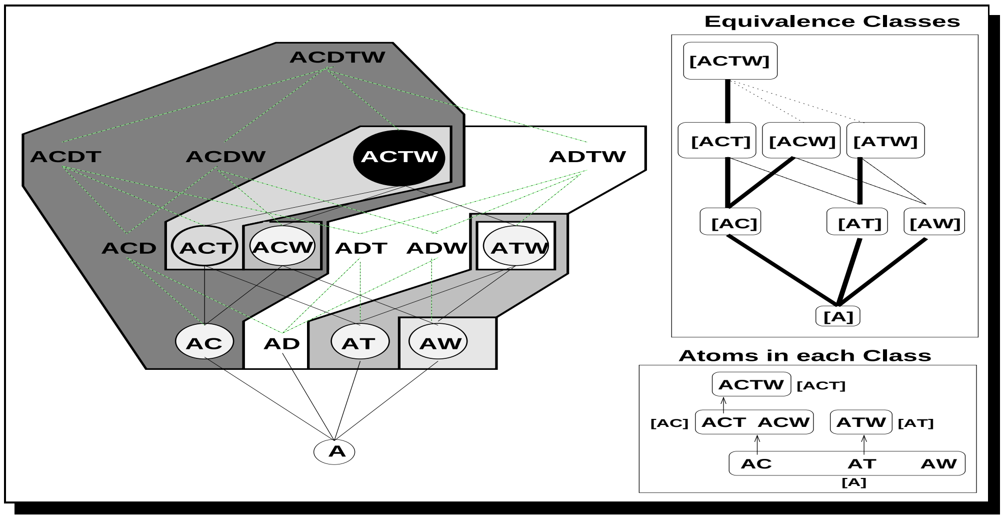


Figure 1.6: Bottom-Up Search

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 1.7 depicts the top-down search. We start with a 5-way tid-list intersection to obtain the support for $ACDTW$, which turns out to be infrequent. We next have to try its subsets at the next level. Out of the 4 subsets, only $ACTW$ is frequent. This means we don't check any of its subsets, since they all must be frequent. For the other three itemsets $ACDT$, $ACDW$, and $ADTW$, we check their subsets that are not known to be frequent, i.e., we check ACD, ADT and ADW, all of which are infrequent. Finally, at the next level, we find the minimal infrequent itemset AD, and the process stops. As it turns out, we had to perform 9 intersections here, the same as in bottom-up search. But if $ACDTW$ had been frequent, we would have saved a lot of computation.

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.

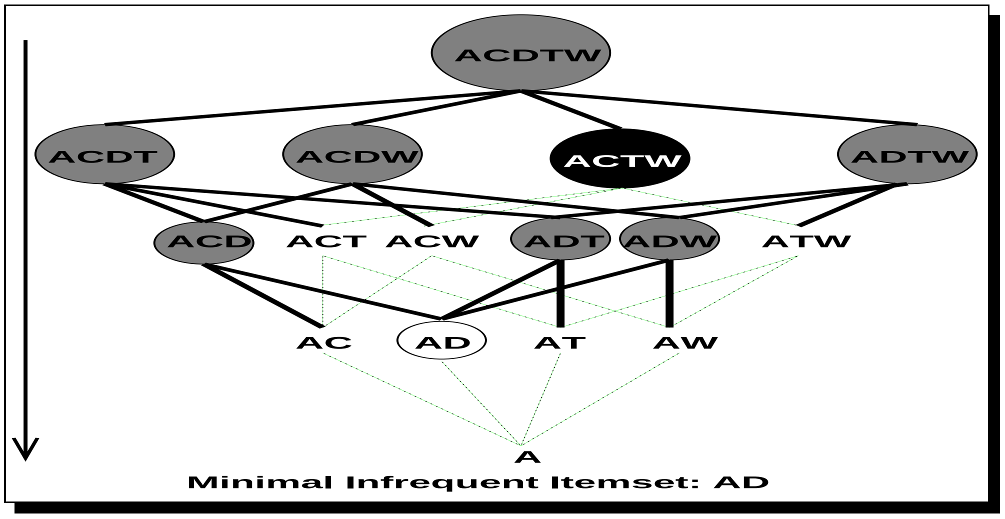


Figure 1.7: Top-Down Search

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 1.8 illustrates this approach (just for this case, to better show the bottom-up phase, we have assumed that AD and ADW are also frequent). Initially, we have four 2-itemsets (or atoms) in the sub-lattice. We sort them in decreasing order of support to obtain the atom list AC, AW, AT, and AD. We now start the hybrid phase. Starting with AC we try to join it with AW, getting ACW which is frequent. We next join ACW with the next atom AT, to get $ACTW$ which is also frequent. Finally we try $ACTW$ with the last atom AD, but $ACDTW$ is infrequent. Note, that if $ACDTW$ were frequent, we would stop the computation at this stage, having found the maximal frequent itemset. The algorithm now shifts into the bottom-up phase. AD is the atom which could not be combined with $ACTW$. With AD as the bottom element, we generate a new sub-lattice, whose atoms are the join of AD with itemsets that precede it in the sorted initial ordering, i.e., the new atoms are ACD, ADW, and ADT. This sublattice can be solved using the bottom-up approach. In fact, if there are many atoms like AD which could not be combined with $ACTW$, then one can even re-apply hybrid search on the newly generated sub-lattice.

Like the bottom-up approach this scheme only requires 2-way intersections. This scheme 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 phase with the hybrid phase. This approach will enumerate some maximal elements (hybrid phase) and the remaining frequent itemset (bottom-phase).

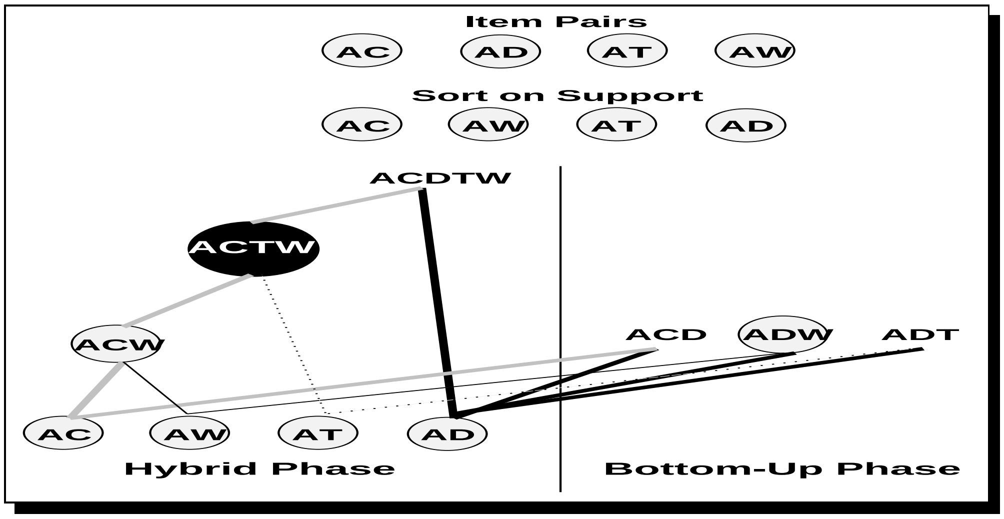


Figure 1.8: Hybrid Search

1.4.4 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. As we shall see later in this section, 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**.

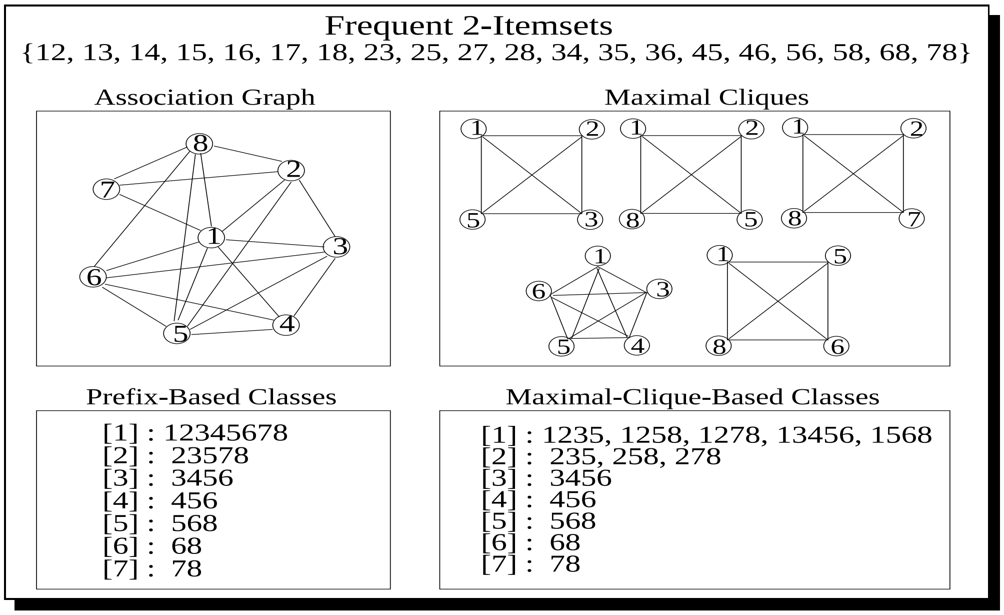


Figure 1.9: Maximal Cliques of the Association Graph; Prefix-Based and Maximal-Clique-Based Classes

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 $E = \{(X, Y) \mid X, Y \in V$ and $\exists\ Z \in \mathcal{F}_{(k+1)},$ such that - $X, Y \subset Z\}$. Let M_k denote the set of maximal cliques in G_k. Figure 1.9 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 ϕ_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 ϕ_k a *maximal-clique-based* pseudo-equivalence relation.

Lemma 8 *Each pseudo-class $[X]_{\phi_k}$ induced by the relation ϕ_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 ϕ_k. Each pseudo-class $[Y]_{\phi_k}$ induced by the prefix-based relation ϕ_k is a subset of some class $[X]_{\theta_k}$ induced by θ_k. Conversely, each $[X]_{\theta_k}$, is the union of a set of pseudo-classes Ψ, 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 $[X]_{\theta_k} = \bigcup\{[Y]_{\phi_k} \mid Y$ is a prefix of $X\}$. ■

This lemma states that each pseudo-class of ϕ_k is a refinement of (i.e., is smaller than) some class of θ_k. By using the relation ϕ_k instead of θ_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 1.9 contrasts the classes (sub-lattices) generated by ϕ_1 and θ_1. It is apparent that ϕ_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 increased refinement of ϕ_k comes at a cost, since the enumeration of maximal cliques can be computationally expensive. For general graphs the maximal clique decision problem is NP-Complete (Garey & Johnson 1979). 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. ϕ_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 lower the support and the longer the transaction size, the greater the edge density of the k-association graph.

Maximal Clique Generation

We used a modified version of the Bierstone's algorithm (Mulligan & Corneil 1972) 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 1.9. $cov(2) = \{3, 5, 7, 8\} = [2]$. 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 1.10 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
2:    [i].$CliqList$ = $\emptyset$;
3:    for all $x \in$ [i].$CoveringSet$ do
4:        for all $cliq \in$ [x].$CliqList$ do
5:            $M = cliq \cap$ [i];
6:            if $M \neq \emptyset$ then
7:                insert $(\{i\} \cup M)$ in [i].$CliqList$ such that
8:                    $\nexists X or Y \in$ [i].CliqList, $X \subseteq Y, or Y \subseteq X$;
```

Figure 1.10: 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 the clique generation takes 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 *α-related*, if $\alpha = \frac{|X \cap Y|}{|X \cup Y|}$, i.e., the ratio of the common elements to the distinct elements of the cliques. 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. Preliminary experiments indicate that using an appropriate value of α, most of the overhead of redundant cliques can be avoided. We found $\alpha = 0.5$ to work well in practice.

1.5 Sequential Algorithm

In this section we describe the sequential algorithm 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 θ_1, or the maximal-clique-based pseudo-equivalence relation ϕ_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.

1.5.1 Computing Frequent 1-Itemsets and 2-Itemsets

Most of the current association algorithms (Agrawal *et al.* 1996; Brin *et al.* 1997; Lin & Dunham 1998; Park, Chen, & Yu 1995a; Savasere, Omiecinski, & Navathe 1995; Toivonen 1996) assume a *horizontal* database layout, such as the one shown in Figure 1.1, 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 1.3, 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 few database scans. 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 tid-list size in bytes. A naive implementation for computing the frequent 2-itemsets requires $\binom{N}{2}$ tid-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 we propose 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 i in an array indexed by tid for each $t \in \mathcal{L}(i)$. This approach can be implemented with little overhead. For example, *Partition* performs the opposite inversion from horizontal to vertical tid-list format on-the-fly, with very little cost. We plan to implement on-the-fly inversion in the future. However, our current implementation uses the first approach due to its simplicity.

1.5.2 Search Implementation

Bottom-Up Search Figure 1.11 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.

One practical implementation note for the bottom-up search using tid-list intersections is that we found the candidate pruning to be of little or no benefit. Recall that in *Apriori*, whenever we generate a new candidate a check is made to see if all its subsets are frequent. If there is any infrequent subset then we can prune the candidate. We can implement a similar step in our approach, since each sub-lattice is processed in reverse lexicographic order, and thus all subset information is available for itemset pruning. Furthermore, 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 no help. 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.

```
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
delete S; //reclaim memory
for all $T_i \neq \emptyset$ do Bottom-Up($T_i$);
```

Figure 1.11: Pseudo-code for Bottom-Up Search

```
Top-Down($S$):
$R = \bigcup\{A_i \in S\}$;
if $R \notin \mathcal{F}_{|R|}$ then
    $\mathcal{L}(R) = \bigcap\{\mathcal{L}(A_i) \mid A_i \in S\}$;
    if $\sigma(R) \geq min\_sup$ then
        $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
    else
        for all $Y \subset R$, with $|Y| = |R| - 1$
            if $Y \notin$ HT then
                Top-Down($\{A_j \mid A_j \in Y\}$);
                if $\sigma(Y) < min\_sup$ then HT = HT $\cup\{Y\}$;
        end
```

Figure 1.12: Pseudo-code for Top-Down Search

Top-Down Search The code for top-down search is given in Figure 1.12. 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.

Hybrid Search Figure 1.13 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.

1.5.3 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

```
**Hybrid**($S$ sorted on support):
$R = A_1$; $S_1 = \{A_1\}$;
**for** all $A_i \in S$, $i > 1$ **do** /* Maximal Phase */
    $R = R \cup A_i$; $\mathcal{L}(R) = \mathcal{L}(R) \cap \mathcal{L}(A_i)$;
    **if** $\sigma(R) \geq min\_sup$ **then**
        $S_1 = S_1 \cup \{A_i\}$; $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
    **else** break;
**end**
$S_2 = S - S_1$;
**for** all $B_i \in S_2$ **do** /* Bottom-Up Phase */
    $T_i = \{X_j \mid \sigma(X_j) \geq min\_sup, \mathcal{L}(X_j) = \mathcal{L}(B_i) \cap \mathcal{L}(A_j), \forall A_j \in S_1\}$;
    $S_1 = S_1 \cup \{B_i\}$;
    **if** $T_i \neq \emptyset$ **then** Bottom-Up($T_i$);
**end**
```

Figure 1.13: Pseudo-code for Hybrid Search

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 few database scans after computing $\mathcal{F}_2$, in contrast to the current approaches which require as many scan as the longest frequent itemset.

There are cases where more concern has to be paid to minimize database scans. For example if there is a large degree of overlap among the atoms of different classes, then it is best to adopt a mixed approach where we simply apply *Apriori* for the initial levels, and then switch to our methods when the overlap is manageable. What this means is that we need to go beyond a simple one level partitioning based on θ_1, instead we might have to use θ_2 or θ_3.

1.6 Parallel Algorithm Design and Implementation

In this section we will discuss the design and implementation of new parallel algorithms for mining frequent itemsets. We present four new parallel algorithms, depending on the decomposition relation used to generate independent classes, and the lattice search scheme used.

- *Par-Eclat*: It uses prefix-based equivalence relation θ_1 along with bottom-up search. It enumerates all frequent itemsets.
- *Par-MaxEclat*: It uses prefix-based equivalence relation θ_1 along with hybrid search. It enumerates the "long" maximal frequent itemsets, and some non-maximal ones.
- *Par-Clique*: It uses maximal-clique-based pseudo-equivalence relation ϕ_1 along with bottom-up search. It enumerates all frequent itemsets.
- *Par-MaxClique*: It uses maximal-clique-based pseudo-equivalence relation ϕ_1 along with hybrid search. It enumerates the "long" maximal frequent itemsets, and some non-maximal ones.

We next present the parallel design and implementation issues, which are applicable to all four algorithms.

1.6.1 Initial Database Partitioning

We assume that the database is in the vertical format, and that we have the support counts of all 2-itemsets available locally on each host. We further assume that the database of tid-lists is initially partitioned among all the hosts. This partitioning is done off-line, similar to the assumption made in *Count Distribution* (Agrawal & Shafer 1996). The tid-lists are partitioned so that the total length of all tid-lists in the local portions on each host are roughly equal. This is achieved using a greedy algorithm. The items are sorted on their support, and the next item is assigned to the least loaded host. Note that the entire tid-list for an item resides on a host. Figure 1.15 shows the original database, and the resultant initial partition on two processors.

```
Begin ParAssociation:
  /* Initialization Phase*/
  $\mathcal{F}_2 = \{$ Set of Frequent 2-Itemsets $\}$
  Generate Independent Classes from $\mathcal{F}_2$ using:
    Prefix-Based or Maximal-Clique-Based Partitioning
  Schedule Classes among the processors $P$
  Scan local database partition
  Transmit relevant tid-lists to other processors
  Receive tid-lists from other processors

  /* Asynchronous Phase */
  for each assigned Class, $C_2$
    Compute Frequent Itemsets: Bottom-Up($C_2$) or Hybrid($C_2$)

  /* Final Reduction Phase*/
  Aggregate Results and Output Associations
End ParAssociation
```

Figure 1.14: Pseudo-code for the New Parallel Algorithms

1.6.2 Parallel Design and Implementation

The new algorithms overcome the shortcomings of the *Count* and *Candidate Distribution* algorithms. They utilize the aggregate memory of the system by partitioning the itemsets into disjoint sets, which are assigned to different processors. The dependence among the processors is decoupled right in the beginning so that the redistribution cost can be amortized by the later iterations. Since each processor can proceed independently, there is no costly synchronization at the end of each iteration. Furthermore the new algorithms use the vertical database layout which clusters all relevant information in an itemset's tid-list. Each processor computes all the frequent itemsets from one class before proceeding to the next. The local database partition is scanned only once. In contrast *Candidate Distribution* must scan it once in each iteration. The new algorithms don't pay the extra computation overhead of building or searching complex data structures, nor do they have to generate all the subsets of each transaction. As the intersection is performed an itemset can immediately be inserted in $\mathcal{F}_k$. Notice that the tid-lists also automatically prune irrelevant transactions. As the itemset size increases, the size of the tid-list decreases, resulting in very fast intersections. There are two distinct phases in the algorithms. The initialization phase, responsible for communicating the tid-lists among the processors, and the asynchronous phase, which generates frequent itemsets. The pseudo-code for the new algorithms is shown in Figure 1.14, and a pictorial representation of the different phases is shown in Figure 1.15.

Initialization Phase

The initialization step consists of three sub-steps. First, the support counts for 2-itemsets from the preprocessing step are read, and the frequent ones are inserted into $\mathcal{F}_2$. Second, applying one of the two decomposition schemes to $\mathcal{F}_2$ – prefix-based or maximal-clique-based – the set of independent classes is generated. These classes are then scheduled among all the processors so that a suitable level of load-balancing can be achieved. Third, the database is repartitioned so that each processor has on its local disk the tid-lists of all 1-itemsets in any class assigned to it.

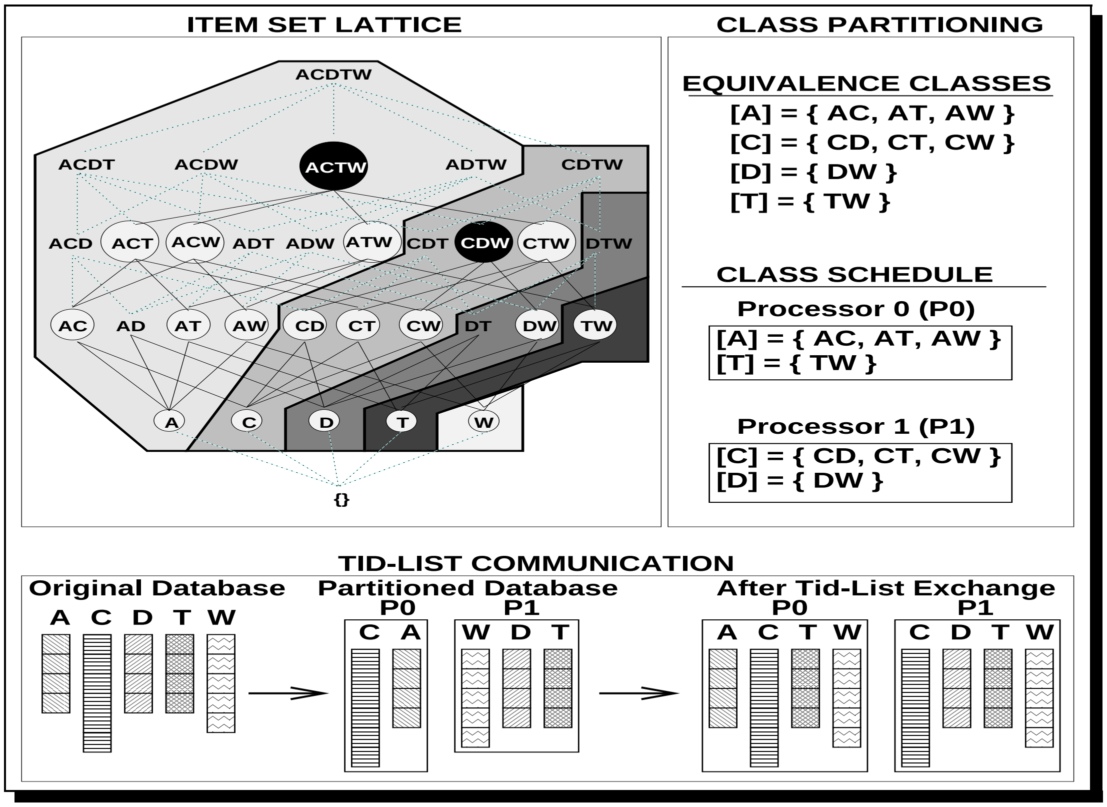


Figure 1.15: Database Partitioning and Class Scheduling

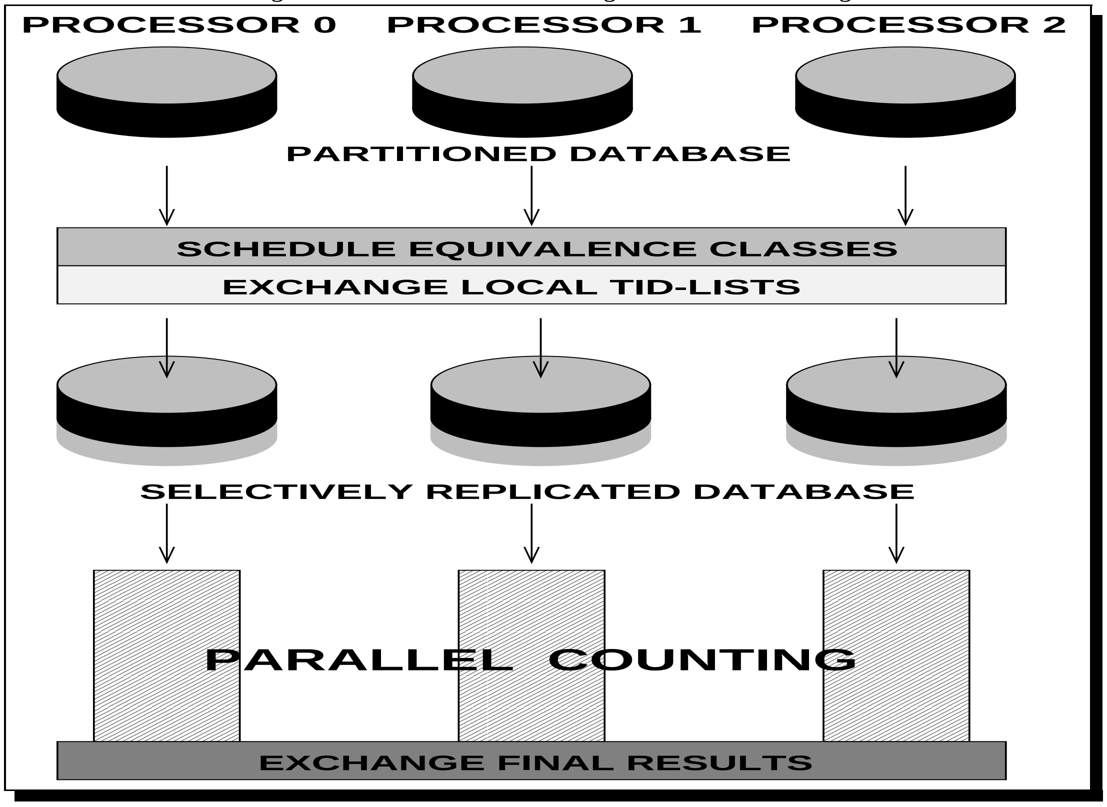


Figure 1.16: The *Par-Eclat* Algorithm

Class Scheduling We first partition $\mathcal{F}_2$ into equivalence classes using prefix-based or maximal-clique-based partitioning. We next generate a schedule of the equivalence classes on the different processors in a manner minimizing the load imbalance and minimizing the inter-process communication required in partially replicating the tid-lists. Note that it may be necessary to sacrifice some amount of load balancing for a better communication efficiency. For this reason, whole equivalence classes are assigned to the same processor. Load balancing is achieved by assigning a weight to each equivalence class based on the number of elements in the class. Since we have to consider all pairs of atoms for the next iteration, we assign the weight $\binom{s}{2}$ to a class with s atoms. Once the weights are assigned we generate a schedule using a greedy heuristic. We sort the classes on the weights, and assign each class in turn to the least loaded processor, i.e., one having the least total weight at that point. Ties are broken by selecting the processor with the smaller identifier. These steps are done concurrently on all the processors since all of them have access to the global $\mathcal{F}_2$. Figure 1.15 shows how the prefix-based classes of our example database (from Figure 1.1) are scheduled on two processors. Notice how an entire class is assigned to a single processor. Although the number of atoms of a class gives a good indication of the amount of work that needs to be done for that class, better heuristics for generating the weights are possible. For example, if we could better estimate the number of frequent itemsets that would be enumerated from a class we could use that as our weight.

Tid-list Communication Once the classes have been scheduled among the processors, each processor has to exchange information with every other processor to read the non-local tid-lists over the Memory Channel network. To minimize communication, and being aware of the fact that in our configuration there is only one local disk per host (recall that our cluster has 8 hosts, with 4 processors per host), only the hosts take part in the tid-list exchange. Additional processes on each of the 8 hosts are spawned only in the asynchronous phase. To accomplish the inter-process tid-list communication, each processor scans the item tid-lists in its local database partition and writes it to a transmit region which is mapped for receive on other processors. The other processors extract the tid-list from the receive region if it belongs to any class assigned to them. For example, Figure 1.15 shows the initial local database on two hosts, and the final local database after the tid-list communication.

Asynchronous Phase

At the end of the initialization step, the relevant tid-lists are available locally on each host, thus each processor can independently generate the frequent itemsets from its assigned classes eliminating the need for synchronization with other processors. Each class is processed in its entirety before moving on to the next class in the schedule. This step involves scanning the local database partition only once (depending on the amount of overlap among the classes). We can thus benefit from huge I/O savings. Since each class induces a sublattice, depending on the algorithm, we either use a bottom-up traversal to generate all frequent itemsets, or we use the hybrid traversal to generate only the "long" maximal and other frequent itemsets. The pseudo-code and implementation of the two lattice search schemes was presented in the last section (see Figure 1.11 and Figure 1.12). As an illustration of the various steps, the *Par-Eclat* algorithm is shown in Figure 1.16. At the end of the asynchronous phase we accumulate all the results from each processor and print them out.

1.6.3 Salient Features of the New Algorithms

In this section we will recapitulate the salient features of our proposed algorithms, contrasting them against *Count Distribution*. Our algorithms differ in the following respect:

- They utilize the aggregate memory of the parallel system by partitioning the candidate itemsets among the processors using the prefix-based and maximal-clique-based decomposition schemes.
- They decouple the processors right in the beginning by repartitioning the database, so that each processor can compute the frequent itemsets independently. This eliminates the need for communicating the frequent itemsets at the end of each iteration.
- They use the vertical database layout which clusters the transactions containing an itemset into tid-lists. Using this layout enables our algorithms to scan the local database partition only a few times on

each processor. It usually takes two scans, the first for communicating the tid-lists, and the second for obtaining the frequent itemsets. In contrast, *Count Distribution* scans the database multiple times – once during each iteration.

- To compute frequent itemsets, they performs simple intersections on two tid-lists. There is no extra overhead associated with building and searching complex hash tree data structures. Such complicated hash structures also suffer from poor cache locality (Parthasarathy, Zaki, & Li 1998). In our algorithms, all the available memory is utilized to keep tid-lists in memory which results in good locality. As larger itemsets are generated the size of tid-lists decreases, resulting in very fast intersections.
- Our algorithms avoid the overhead of generating all the subsets of a transaction and checking them against the candidate hash tree during support counting.

1.7 Experimental Results

All the experiments were performed on a 32-processor (8 hosts, 4 processors/host) Digital Alpha cluster interconnected via the Memory Channel network (Gillett 1996). Each Alpha processor runs at 233MHz. There's a total of 256MB of main memory per host (shared among the 4 processors on that host). Each host also has a 2GB local disk, out of which less than 500MB was available to us.

The Digital Memory Channel Digital's Memory Channel network provides applications with a global address space using memory mapped regions. A region can be mapped into a process' address space for transmit, receive, or both. Virtual addresses for transmit regions map into physical addresses located in I/O space on the Memory Channel's PCI adapter. Virtual addresses for receive regions map into physical RAM. Writes into transmit regions are collected by the source Memory Channel adapter, forwarded to destination Memory Channel adapters through a hub, and transferred via DMA to receive regions with the same global identifier. Figure 1.17 shows the Memory Channel space (The lined region is mapped for both transmit and receive on node 1 and for receive on node 2; The gray region is mapped for receive on node 1 and for transmit on node 2). Regions within a node can be shared across different processors on that node. Writes originating on a given node will be sent to receive regions on that same node only if *loop-back* has been enabled for the region. We do not use the loop-back feature. We use *write-doubling* instead, where each processor writes to its receive region and then to its transmit region, so that processes on a host can see modification made by other processes on the same host. Though we pay the cost of double writing, we reduce the amount of messages to the hub.

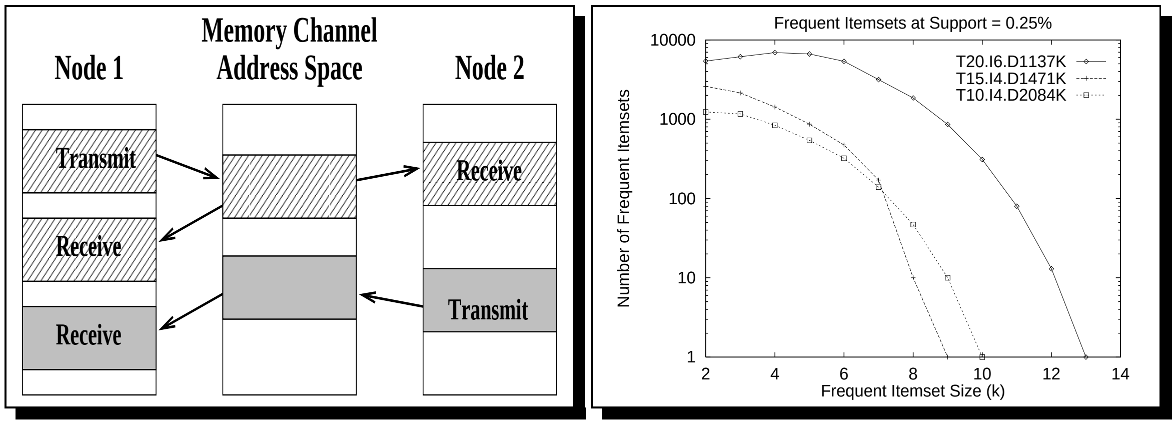


Figure 1.17: a) The Memory Channel Space, b) Number of Frequent Itemsets

In our system unicast and multicast process-to-process writes have a latency of 5.2 μs, with per-link transfer bandwidths of 30 MB/s. Memory Channel peak aggregate bandwidth is also about 32 MB/s. Memory Channel guarantees write ordering and local cache coherence. Two writes issued to the same transmit region (even on different nodes) will appear in the same order in every receive region. When a write appears in a receive region it invalidates any locally cached copies of its line.

Database	T	I	$\mathcal{D}_1$	$\mathcal{D}_1$ Size	$\mathcal{D}_4$	$\mathcal{D}_4$ Size	$\mathcal{D}_6$ Size
T10.I4.D2084K	10	4	2,084,000	91 MB	8,336,000	364MB	546MB
T15.I4.D1471K	15	4	1,471,000	93 MB	5,884,000	372MB	558MB
T20.I6.D1137K	20	6	1,137,000	92 MB	4,548,000	368MB	552MB

Table 1.1: Database Properties

Synthetic Databases All the partitioned databases reside on the local disks of each processor. We used different synthetic databases that have been used as benchmark databases for many association rules algorithms (Agrawal, Imielinski, & Swami 1993; Agrawal *et al.* 1996; Brin *et al.* 1997; Houtsma & Swami 1995; Lin & Kedem 1998; Lin & Dunham 1998; Park, Chen, & Yu 1995a; Savasere, Omiecinski, & Navathe 1995; Zaki *et al.* 1997b). The dataset generation procedure is described in (Agrawal *et al.* 1996), and the code is publicly available from IBM (IBM).

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 (Agrawal & Srikant 1994) for more detail on the database generation.

Table 1.1 shows the databases used and their properties. The number of transactions is denoted as $\mathcal{D}_r$, where r is the replication factor. Using a replication factor allows us to keep the number of frequent itemsets the same for a given minimum support, but it allows us to study larger databases. For $r = 1$, all the databases are roughly 90MB in size. Except for the sizeup experiments, all results shown are on databases with a replication factor of $r = 4$ (≈360MB). We could not go beyond a replication factor of 6 (≈540MB; used in sizeup experiments) since the repartitioned database would become too large to fit on disk. The average transaction size is denoted as T, and the average maximal potentially frequent itemset size as I. The number of maximal potentially frequent itemsets was $L = 2000$, and the number of items was $N = 1000$. All the experiments were performed with a minimum support value of 0.25%. The number of large itemsets discovered are shown in Figure 1.17. For a fair comparison, all algorithms discover frequent k-itemsets for $k \geq 3$, using the supports for the 2-itemsets from the preprocessing step.

1.7.1 Performance Comparison

In this section we will compare the performance of our new algorithms with *Count Distribution* (henceforth referred to as CD), which was shown to be superior to both *Data* and *Candidate Distribution* (Agrawal & Shafer 1996). In all the figures the different parallel configurations are represented as $Hx.Py.Tz$, where $H = x$ denotes the number of hosts, $P = y$ the number of processors per host, and $T = H \cdot P = z$, the total number of processors used in the experiments. Figure 1.18 shows the total execution time for the different databases and on different parallel configurations. The configurations have been arranged in increasing order of T. Configurations with the same T are arranged in increasing order of H. The left column compares *Par-MaxClique*, the best new algorithm, with *Par-Eclat* and CD, while the right column compares only the new algorithms, so that the differences among them become more apparent. It can be clearly seen that *Par-Eclat* out-performs CD for almost all configurations on all the databases, with improvements as high as a factor of 5. If we compare with the best new algorithm *Par-MaxClique*, we see an improvement of upto an order of magnitude over CD. Even more dramatic improvements are possible for lower values of minimum support (Zaki, Parthasarathy, & Li 1997). An interesting trend in the figures is that the performance gap seems

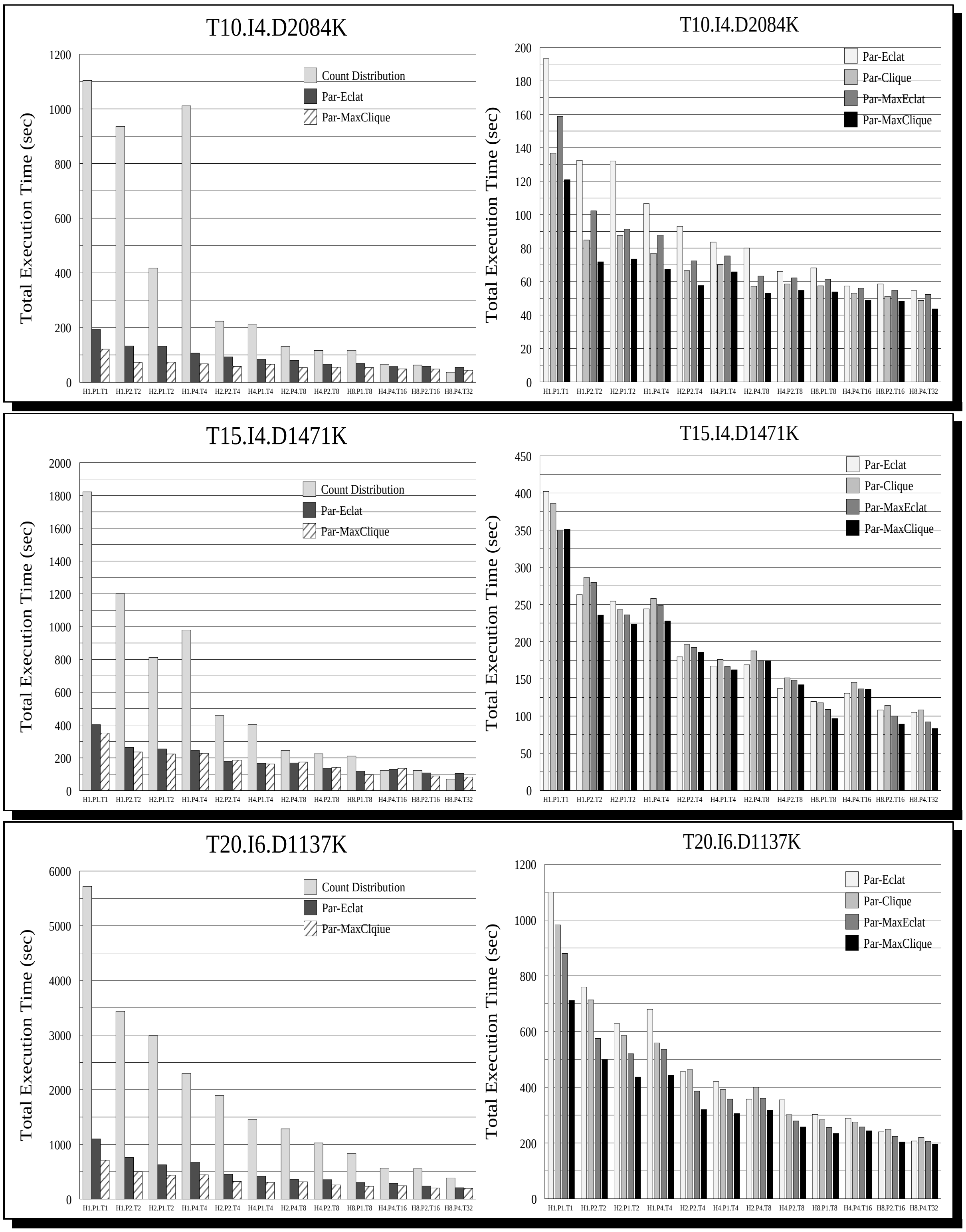


Figure 1.18: Parallel Performance

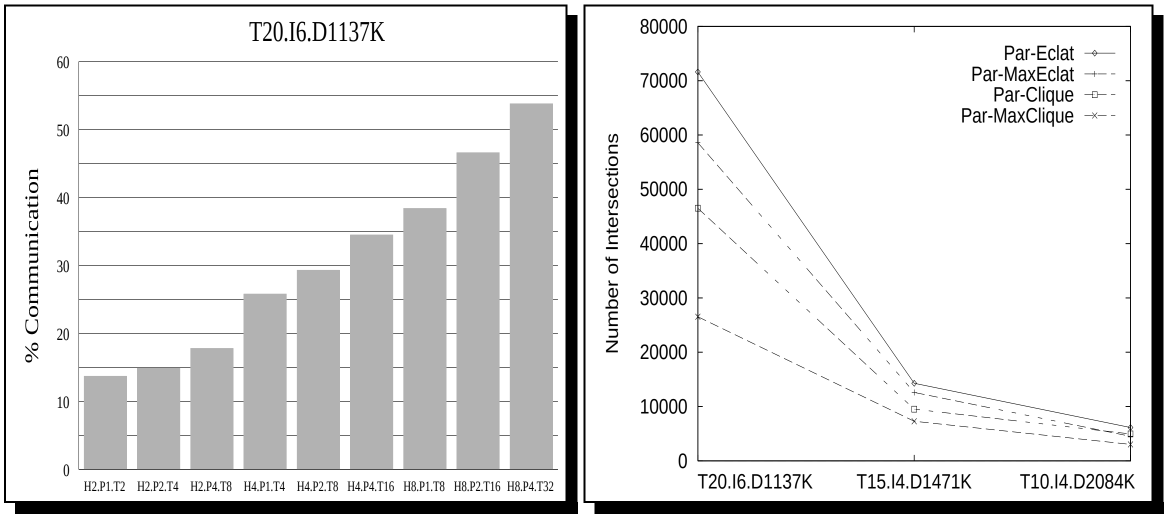


Figure 1.19: a) Communication Cost, b) Number of Intersections

to decrease at larger configurations, with CD actually performing better on the 32 processor configuration H8.P4.T32 for the databases T10.I4.D2084K and T15.I4.D1471K. To see why, consider Figure 1.17, which shows the total number of frequent itemsets of different sizes for the different databases. Also from Figure 1.19, which shows the initial database repartitioning and tid-list communication cost as a percentage of the total execution time of *Par-Eclat*, it becomes clear that there is not enough work for these two databases to sufficiently offset the communication cost, consequently more than 70% of the time is spent in the initialization phase. For T20.I6.D1137K, which has more work, *Par-Eclat* is still about twice as fast as CD on 32 processors. The basic argument falls on the computation versus communication trade-off in parallel computing. Whenever this ratio is high we would expect *Par-Eclat* to out-perform CD. We would also expect the relative improvements of *Par-Eclat* over CD to be better for larger databases. Unfortunately due to disk space constraints we were not able to test the algorithms on larger databases. In all except the $H = 1$ configurations, the local database partition is less than available memory. Thus for CD the entire database is cached after the first scan. The performance of CD is thus a best case scenario for it since the results do not include the "real" hit CD would have taken from multiple disk scans. As mentioned in section 1.6, *Par-Eclat* was designed to scan the database only once during frequent itemset computation, and would thus benefit more with larger database size.

Figure 1.18 (right column) shows the differences among the new algorithms for different databases and parallel configurations. There are several parameters affecting their performance. It can be seen that in general *Par-Clique* and *Par-MaxClique* perform better than *Par-Eclat* and *Par-MaxEclat*, respectively. This is because they use the maximal-clique-based decomposition, which generates more precise classes. On the other axis, in general *Par-MaxClique* and *Par-MaxEclat*, out-perform *Par-Clique* and *Par-Eclat*, respectively. This is because the hybrid lattice search scheme quickly generates the long maximal frequent itemsets, saving on the number of intersections. The results are also dependent on the number of frequent itemsets. The larger the number of frequent itemsets, the more the opportunity for the hybrid approach to save on the joins. For example, consider Figure 1.19, which shows the total number of tid-list intersections performed for the four algorithms on the three databases. For T20.I6.D1137K, which has the largest number of frequent itemsets (see Figure 1.17), *Par-MaxClique* cuts down the number of intersections by more than 60% over *Par-Eclat*. The reduction was about 20% for *Par-MaxEclat*, and 35% for *Par-Clique*. These factors are responsible for the trends indicated above. The winner in terms of the total execution time is clearly *Par-MaxClique*, with improvements over *Par-Eclat* as high as 40%.

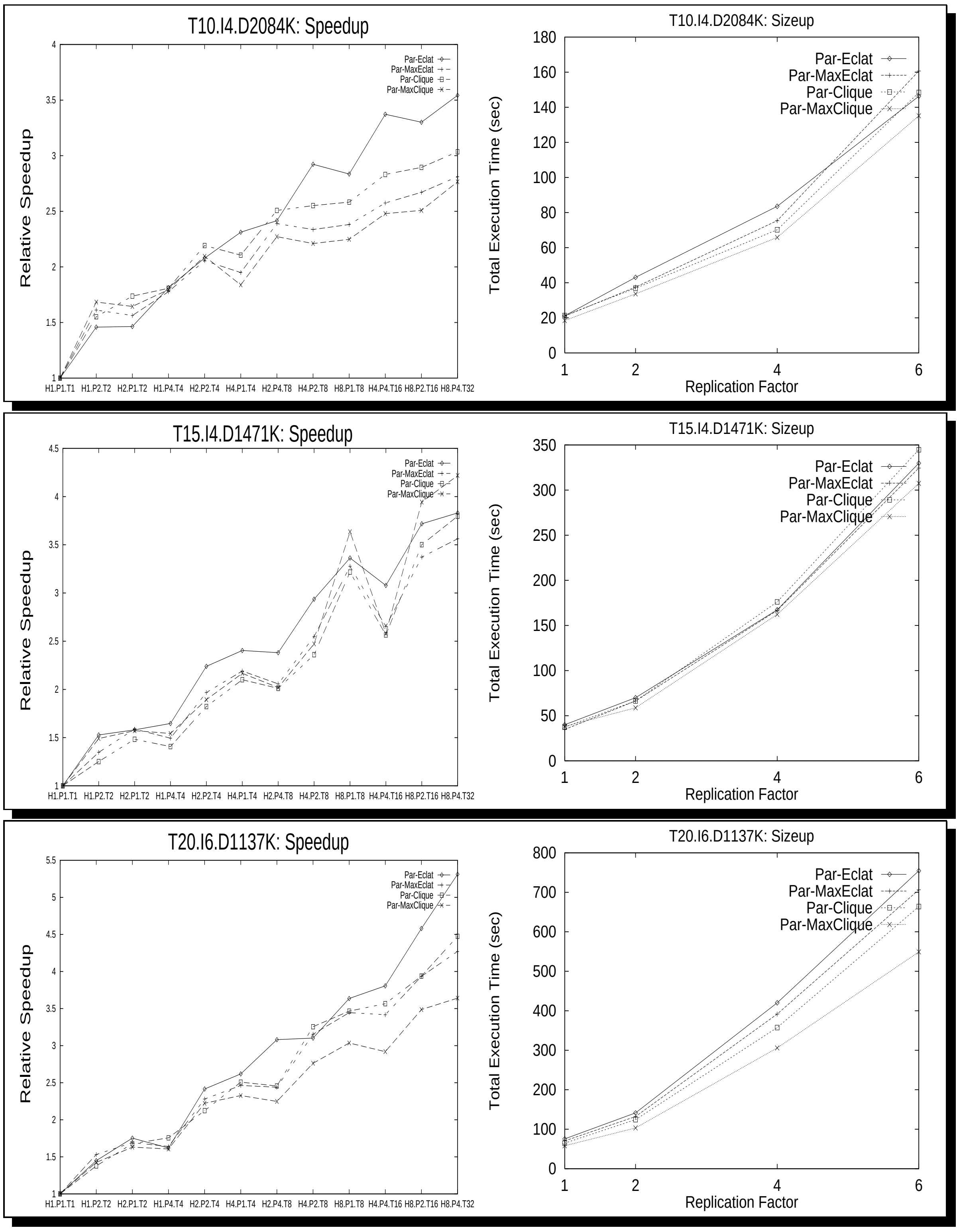


Figure 1.20: Parallel Speedup and Sizeup (H4.P1.T4)

1.7.2 Sensitivity Analysis

Speedup: The goal of the speedup experiments is to see how the new algorithms perform as we increase the number of processors while keeping the data size constant. Figure 1.20, shows the speedup on the different databases and parallel configurations. Due to disk constraints we used a replication factor of 4, for database sizes of approximately 360MB. The speedup numbers are not as impressive at first glance. However, this is not surprising. For example, on the largest configuration H8.P4.T32, there is only about 11MB of data per processor. Combined with the fact that the amount of computation is quite small (see Figure 1.17), and that about 50% to 70% of the time is spent in tid-list communication (see Figure 1.19), we see a maximum speedup of about 5. Another reason is that the communication involves only the 8 hosts. Additional processes on a host are only spawned after the initialization phase, which thus represents a partially-parallel phase, limiting the speedups. If we take out the communication costs we see a maximum speedup of 12 to 16. An interesting trend is the step-effect seen in the speedup graphs. For the configurations which have the same number of total processors, the ones with more hosts perform better. Also, for configurations with more total processors, with $P = 4$, the configurations immediate preceding it, with only 1 processor per host, performs better. In both cases, the reason is that increasing the number of processors on a given host, causes increased memory contention (bus traffic), and increased disk contention, as each processor tries to access the database from the local disk at the same time.

Sizeup: The goal of the sizeup experiments is to see how the new algorithms perform as we increase the size of the database while keeping the number of processors constant. For the sizeup experiments we fixed the parallel configuration to H4.P1.T4, and varied the database replication factor from 1 to 6, with the total database size ranging from about 90MB to 540MB. Figure 1.20 shows the sizeup for the four algorithms on the different databases. The figures indicate an almost linear sizeup. The slightly upward bend is due to the relative computation versus communication cost. The larger the database the more the time spent in communication, while the tid-list intersection cost doesn't increase at the same pace. Moreover, the number of frequent itemsets remains constant (since we use percentages for minimum support, as opposed to absolute counts) for all replication factors.

1.8 Conclusions

In this paper we presented new parallel algorithms for efficient enumeration of frequent itemsets. We p-resented 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.

The set of independent classes is scheduled among the processors, and the database is also selectively replicated so that the portion of the database needed for the computation of associations is local to each processor. After the initial set-up phase the algorithms do not need any further communication or synchronization. The algorithms minimize I/O overheads by scanning the local database portion only two times. Once in the set-up phase, and once when processing all the itemset classes. We implemented the algorithms on a 32 processor Digital cluster interconnected with the Memory Channel network, and compared them against a well known parallel algorithm *Count Distribution* (Agrawal & Shafer 1996). Experimental results indicate that our best parallel algorithm *Par-MaxClique* outperformed *Count Distribution* by upto an order of magnitude.

References

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

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

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

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

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

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

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

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

Cheung, D.; Hu, K.; and Xia, S. 1998. Asynchronous parallel algorithm for mining association rules on shared-memory multi-processors. In *10th ACM Symp. Parallel Algorithms and Architectures*.

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

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

Gillett, R. 1996. Memory channel: An optimized cluster interconnect. In *IEEE Micro, 16(2)*.

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

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

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

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

IBM. *http://www.almaden.ibm.com/cs/quest/syndata.html*. Quest Data Mining Project, IBM Almaden Research Center, San Jose, CA 95120.

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

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

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

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

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

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

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

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

Shintani, T., and Kitsuregawa, M. 1996. Hash based parallel algorithms for mining association rules. In *4th Intl. Conf. Parallel and Distributed Info. Systems*.

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

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

Zaki, M. J., and Ogihara, M. 1998. Theoretical foundations of association rules. In *3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*.

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

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

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

Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997c. New algorithms for fast discovery of association rules. Technical Report URCS TR 651, University of Rochester.

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

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